SIGGRAPH 2006 Course 4
State of the Art in Interactive Ray Tracing

Co-Organizers
Peter Shirley
University of Utah

Philipp Slusallek
Universität des Saarlandes

Lecturers
Ingo Wald
University of Utah

William R. Mark
University of Texas

Gordon Stoll
Intel Corporation

Dinesh Minocha
University of North Carolina at Chapel Hill

10

Ray Tracing

Ray tracing is a method to produce realistic images; it determines visible surfaces in an image at the pixel level (Appel, 1968; Kay & Greenberg, 1979; Whitted, 1980). Unlike the z-buffer and BSP tree, ray tracing operates pixel-by-pixel rather than primitive-by-primitive. This tends to make ray tracing relatively slow for scenes with large objects in screen space. However, it has a variety of nice features which often make it the right choice for batch rendering and even for some interactive applications.

Ray tracing's primary benefit is that it is relatively straightforward to compute shadows and reflections. In addition, ray tracing is well suited to "walkthroughs" of extremely large models due to advanced ray tracing's low asymptotic time complexity which makes up for the required preprocessing of the model (Snyder & Barr, 1987; Muuss, 1995; Parker et al., 1999; Wald, Slusallek, & Benthin, 2001).

In an interactive 3D program implemented in a conventional z-buffer environment, it is often useful to be able to select an object using a mouse. The mouse is clicked in pixel (i, j) and the "picked" object is whatever object is "seen" through that pixel. If the rasterization process includes an object identification buffer, this is just a matter of looking up the value in pixel (i, j) of that buffer. However, if that buffer is not available, we can solve the problem of which object is visible via brute force geometrical computation using a "ray intersection test." In this way, ray tracing is useful also to programmers who use only standard graphics APIs.

This chapter also discusses *distribution ray tracing* (Cook, Porter, & Carpenter, 1984), where multiple random rays are sent through each pixel in an image to simultaneously solve the antialiasing, soft shadow, fuzzy reflection, and depth-of-field problems.

9.1 The Basic Ray Tracing Algorithm

The simplest use of ray tracing is to produce images similar to those produced by the z-buffer and BSP-tree algorithms. Fundamentally, those methods make sure the appropriate object is "seen" through each pixel,and that the pixel color is shaded based on that object's material properties, the surface normal seen through that pixel, and the light geometry.

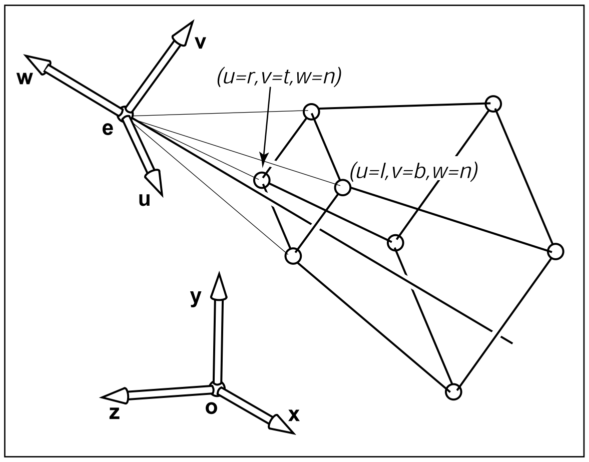


Figure 9.1. The 3D window we look through is the same as in Chapter 6. The borders of the window have simple coordinates in the *uvw* coordinate system with respect to origin **e**.

Figure 9.1 shows the basic viewing geometry for ray tracing, which is the same as we saw earlier in Chapter 6. The geometry is aligned to a *uvw* coordinate system with the origin at the eye location $\mathbf{e}$. The key idea in ray tracing is to identify locations on the view plane at $w = n$ that correspond to pixel centers, as shown in Figure 9.2. A "ray," really just a directed 3D line, is then sent from $\mathbf{e}$ to that point. We then "gaze" in the direction of the ray to see the first object seen in that direction. This is shown in Figure 9.3, where the ray intersects two triangles, but only the first triangle hit, T_2, is returned.

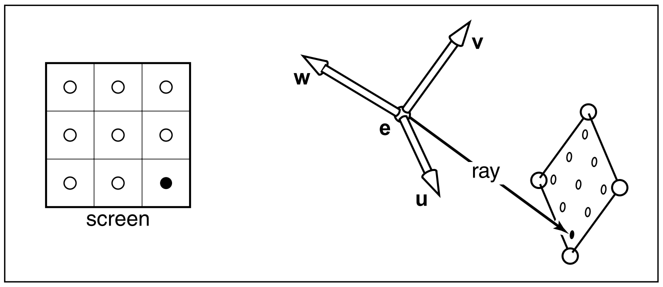


Figure 9.2. The sample points on the screen are mapped to a similar array on the 3D window. A viewing ray is sent to each of these locations.

The structure of the basic ray tracing program is:

```
*Compute* $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ *basis vectors*
***for*** *each pixel* ***do***
    *compute viewing ray*
    *find first object hit by ray and its surface normal* $\mathbf{n}$
    *set pixel color to value based on material, light, and* $\mathbf{n}$
```

The pixel color can be computed using the shading equations of the last chapter.

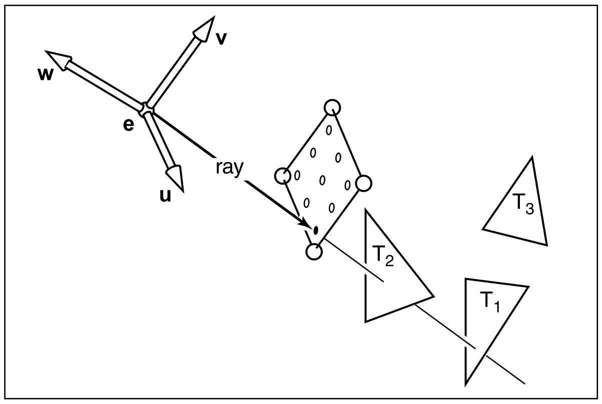


Figure 9.3. The ray is “traced” into the scene and the first object hit is the one seen through the pixel. In this case, the triangle T_2 is returned.

9.2 Computing Viewing Rays

First we need to determine a mathematical representation for a ray. A ray is really just an origin point and a propagation direction; a 3D parametric line is ideal for

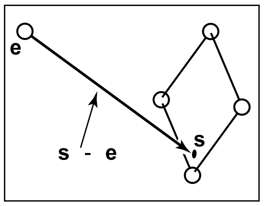


Figure 9.4. The ray from the eye to a point on the screen.

this. As discussed in Section 2.8.1, the 3D parametric line from the eye $\mathbf{e}$ to a point $\mathbf{s}$ on the screen (see Figure 9.4) is given by

$$\mathbf{p}(t) = \mathbf{e} + t(\mathbf{s} - \mathbf{e}).$$

This should be interpreted as, "we advance from $\mathbf{e}$ along the vector $(\mathbf{s} - \mathbf{e})$ a fractional distance t to find the point $\mathbf{p}$." So given t, we can determine a point $\mathbf{p}$. Note that $\mathbf{p}(0) = \mathbf{e}$, and $\mathbf{p}(1) = \mathbf{s}$. Also note that for positive t, if $t_1 < t_2$, then $\mathbf{p}(t_1)$ is closer to the eye than $\mathbf{p}(t_2)$. Also, if $t < 0$, then $\mathbf{p}(t)$ is "behind" the eye. These facts will be useful when we search for the closest object hit by the ray that is not behind the eye. Note that we are overloading the variable t here which is also used for the top of the screen's v-coordinate.

To compute a viewing ray, we need to know $\mathbf{e}$ (which is given) and $\mathbf{s}$. Finding $\mathbf{s}$ may look somewhat difficult. In fact, it is relatively straightforward using the same transform machinery we used for viewing in the context of projecting lines and triangles.

First, we find the coordinates of $\mathbf{s}$ in the uvw-coordinate system with origin $\mathbf{e}$. For all points on the screen, $w_s = n$ as shown in Figure 9.2. The uv-coordinates are found by the windowing transform that takes $[-0.5, n_x-0.5]\times[-0.5, n_y-0.5]$ to $[l, r] \times [b, t]$:

$$u_s = l + (r - l)\frac{i + 0.5}{n_x},$$
$$v_s = b + (t - b)\frac{j + 0.5}{n_y},$$

where (i, j) are the pixel indices. This gives us $\mathbf{s}$ in uvw-coordinates. By definition, we can convert to canonical coordinates:

$$\mathbf{s} = \mathbf{e} + u_s\mathbf{u} + v_s\mathbf{v} + w_s\mathbf{w}. \tag{9.1}$$

Alternatively, we could use the matrix form (Equation 5.8):

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_e \\ 0 & 1 & 0 & y_e \\ 0 & 0 & 1 & z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_s \\ v_s \\ w_s \\ 1 \end{bmatrix}, \tag{9.2}$$

which is just the matrix form of Equation 9.1. We can compose this with the windowing transform in matrix form if we wished, but this is probably not worth doing unless you like the matrix form of equations better.

9.3 Ray-Object Intersection

Given a ray $\mathbf{e} + t\mathbf{d}$, we want to find the first intersection with any object where $t > 0$. It will later prove useful to solve a slightly more general problem of finding the first intersection in the interval $[t_0, t_1]$, and using $[0, \infty)$ for viewing rays. We solve this for both spheres and triangles in this section. In the next section, multiple objects are discussed.

9.3.1 Ray-Sphere Intersection

Given a ray $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$ and an implicit surface $f(\mathbf{p}) = 0$, we'd like to know where they intersect. The intersection points occur when points on the ray satisfy the implicit equation

$$f(\mathbf{p}(t)) = 0.$$

This is just

$$f(\mathbf{e} + t\mathbf{d}) = 0.$$

A sphere with center $\mathbf{c} = (x_c, y_c, z_c)$ and radius R can be represented by the implicit equation

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0.$$

We can write this same equation in vector form:

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0.$$

Any point $\mathbf{p}$ that satisfies this equation is on the sphere. If we plug points on the ray $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$ into this equation, we can solve for the values of t on the ray that yield points on the sphere:

$$(\mathbf{e} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{c}) - R^2 = 0.$$

Rearranging terms yields

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2\mathbf{d} \cdot (\mathbf{e} - \mathbf{c})t + (\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - R^2 = 0.$$

Here, everything is known except the parameter t, so this is a classic quadratic equation in t, meaning it has the form

$$At^2 + Bt + C = 0.$$

The solution to this equation is discussed in Section 2.2. The term under the square root sign in the quadratic solution, $B^2 - 4AC$, is called the *discriminant* and tells us how many real solutions there are. If the discriminant is negative, its square root is imaginary and there are no intersections between the sphere and the line. If the discriminant is positive, there are two solutions: one solution where the ray enters the sphere and one where it leaves. If the discriminant is zero, the ray grazes the sphere touching it at exactly one point. Plugging in the actual terms for the sphere and eliminating the common factors of two, we get

$$t = \frac{-\mathbf{d} \cdot (\mathbf{e} - \mathbf{c}) \pm \sqrt{(\mathbf{d} \cdot (\mathbf{e} - \mathbf{c}))^2 - (\mathbf{d} \cdot \mathbf{d})\left((\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - R^2\right)}}{(\mathbf{d} \cdot \mathbf{d})}.$$

In an actual implementation, you should first check the value of the discriminant before computing other terms. If the sphere is used only as a bounding object for more complex objects, then we need only determine whether we hit it; checking the discriminant suffices.

As discussed in Section 2.7.1, the normal vector at point $\mathbf{p}$ is given by the gradient $\mathbf{n} = 2(\mathbf{p} - \mathbf{c})$. The unit normal is $(\mathbf{p} - \mathbf{c})/R$.

9.3.2 Ray-Triangle Intersection

There are many algorithms for computing ray-triangle intersections. We will use the form that uses barycentric coordinates for the parametric plane containing the triangle, because it requires no long-term storage other than the vertices of the triangle (Snyder & Barr, 1987).

To intersect a ray with a parametric surface, we set up a system of equations where the Cartesian coordinates all match:

$$\begin{aligned} x_e + t x_d &= f(u, v), \\ y_e + t y_d &= g(u, v), \\ z_e + t z_d &= h(u, v). \end{aligned}$$

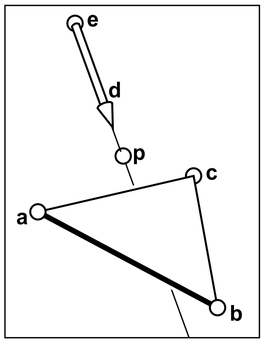


Figure 9.5. The ray hits the plane containing the triangle at point **p**.

Here, we have three equations and three unknowns (t, u, and v), so we can solve numerically for the unknowns. If we are lucky, we can solve for them analytically.

In the case where the parametric surface is a parametric plane, the parametric equation can be written in vector form as discussed in Section 2.11.2. If the vertices of the triangle are $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, then the intersection will occur when

$$\mathbf{e} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}). \tag{9.3}$$

The hitpoint $\mathbf{p}$ will be at $\mathbf{e} + t\mathbf{d}$ as shown in Figure 9.5. Again, from Section 2.11.2, we know the hitpoint is in the triangle if and only if $\beta > 0$, $\gamma > 0$, and $\beta + \gamma < 1$. Otherwise, it hits the plane outside the triangle. If there are no solutions, either the triangle is degenerate or the ray is parallel to the plane containing the triangle.

To solve for t, β, and γ in Equation 9.3, we expand it from its vector form into the three equations for the three coordinates:

$$\begin{aligned} x_e + t x_d &= x_a + \beta(x_b - x_a) + \gamma(x_c - x_a), \\ y_e + t y_d &= y_a + \beta(y_b - y_a) + \gamma(y_c - y_a), \\ z_e + t z_d &= z_a + \beta(z_b - z_a) + \gamma(z_c - z_a). \end{aligned}$$

This can be rewritten as a standard linear equation:

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_e \\ y_a - y_e \\ z_a - z_e \end{bmatrix}.$$

The fastest classic method to solve this 3×3 linear system is *Cramer's Rule*. This gives us the solutions

$$\beta = \frac{\begin{vmatrix} x_a - x_e & x_a - x_c & x_d \\ y_a - y_e & y_a - y_c & y_d \\ z_a - z_e & z_a - z_c & z_d \end{vmatrix}}{|\mathbf{A}|},$$

$$\gamma = \frac{\begin{vmatrix} x_a - x_b & x_a - x_e & x_d \\ y_a - y_b & y_a - y_e & y_d \\ z_a - z_b & z_a - z_e & z_d \end{vmatrix}}{|\mathbf{A}|},$$

$$t = \frac{\begin{vmatrix} x_a - x_b & x_a - x_c & x_a - x_e \\ y_a - y_b & y_a - y_c & y_a - y_e \\ z_a - z_b & z_a - z_c & z_a - z_e \end{vmatrix}}{|\mathbf{A}|},$$

where the matrix $\mathbf{A}$ is

$$\mathbf{A} = \begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix},$$

and $|\mathbf{A}|$ denotes the determinant of $\mathbf{A}$. The 3×3 determinants have common sub-terms that can be exploited. Looking at the linear systems with dummy variables

$$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix},$$

Cramer's rule gives us

$$\beta = \frac{j(ei - hf) + k(gf - di) + l(dh - eg)}{M},$$

$$\gamma = \frac{i(ak - jb) + h(jc - al) + g(bl - kc)}{M},$$

$$t = -\frac{f(ak - jb) + e(jc - al) + d(bl - kc)}{M},$$

where

$$M = a(ei - hf) + b(gf - di) + c(dh - eg).$$

We can reduce the number of operations by reusing numbers such as "*ei-minus-hf*."

The algorithm for the ray-triangle intersection for which we need the linear solution can have some conditions for early termination. Thus, the function should look something like:

```
boolean raytri (ray $\mathbf{r}$, vector3 $\mathbf{a}$, vector3 $\mathbf{b}$, vector3 $\mathbf{c}$, interval $[t_0, t_1]$)
compute $t$
if $(t < t_0)$ or $(t > t_1)$ then
    return false
compute $\gamma$
if $(\gamma < 0)$ or $(\gamma > 1)$ then
    return false
compute $\beta$
if $(\beta < 0)$ or $(\beta > 1 - \gamma)$ then
    return false
return true
```

9.3.3 Ray-Polygon Intersection

Given a polygon with m vertices $\mathbf{p}_1$ through $\mathbf{p}_m$ and surface normal $\mathbf{n}$, we first compute the intersection points between the ray $\mathbf{e} + t\mathbf{d}$ and the plane containing the polygon with implicit equation

$$(\mathbf{p} - \mathbf{p}_1) \cdot \mathbf{n} = 0.$$

We do this by setting $\mathbf{p} = \mathbf{e} + t\mathbf{d}$ and solving for t to get

$$t = \frac{(\mathbf{p}_1 - \mathbf{e}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}.$$

This allows us to compute $\mathbf{p}$. If $\mathbf{p}$ is inside the polygon, then the ray hits it, and otherwise it does not.

We can answer the question of whether $\mathbf{p}$ is inside the polygon by projecting the point and polygon vertices to the xy plane and answering it there. The easiest way to do this is to send any 2D ray out from $\mathbf{p}$ and to count the number of intersections between that ray and the boundary of the polygon (Sutherland et al., 1974; Glassner, 1989). If the number of intersections is odd, then the point is inside the polygon, and otherwise it is not. This is true, because a ray that goes in must go out, thus creating a pair of intersections. Only a ray that starts inside will not create such a pair. To make computation simple, the 2D ray may as well propagate along the x-axis:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \end{bmatrix} + s \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

It is straightforward to compute the intersection of that ray with the edges such as (x_1, y_1, x_2, y_2) for $s \in (0, \infty)$.

A problem arises, however, for polygons whose projection into the xy plane is a line. To get around this, we can choose among the xy, yz, or zx planes for whichever is best. If we implement our points to allow an indexing operation, e.g., $\mathbf{p}(0) = x_p$ then this can be accomplished as follows:

```
if $(abs(z_n) > abs(x_n))$ and $(abs(z_n) > abs(x_n))$ then
    index0 = 0
    index1 = 1
else if $(abs(y_n) > abs\ (x_n))$ then
    index0 = 0
    index1 = 2
else
    index0 = 1
    index1 = 2
```

Now, all computations can use $\mathbf{p}$(index0) rather than x_p, and so on.

9.4 A Ray Tracing Program

We now know how to generate a viewing ray for a given pixel and how to find the intersection with one object. This can be easily extended to a program that produces images similar to the z-buffer or BSP-tree codes of earlier chapters:

```
for each pixel do
    compute viewing ray
    if (ray hits an object with $t \in [0, \infty)$) then
        Compute $\mathbf{n}$
        Evaluate lighting equation and set pixel to that color
    else
        set pixel color to background color
```

Here the statement "if ray hits an object..." can be implemented as a function that tests for hits in the interval $t \in [t_0, t_1]$:

```
hit = false
for each object $\mathbf{o}$ do
    if (object is hit at ray parameter $t$ and $t \in [t_0, t_1]$) then
        hit = true
        hitobject = $\mathbf{o}$
        $t_1 = t$
return hit
```

In an actual implementation, you will need to somehow return either a reference to the object that is hit or at least its normal vector and material properties. This is often done by passing a record/structure with such information. In an object-oriented implementation, it is a good idea to have a class called something like *surface* with derived classes triangle, sphere, surface-list, etc. Anything that a ray can intersect would be under that class. The ray tracing program would then have one reference to a "surface" for the whole model, and new types of objects and efficiency structures can be added transparently.

9.4.1 Object-Oriented Design for a Ray Tracing Program

As mentioned earlier, the key class hierarchy in a ray tracer are the geometric objects that make up the model. These should be subclasses of some geometric object class, and they should support a *hit* function (Kirk & Arvo, 1988). To avoid confusion from use of the word "object," *surface* is the class name often used. With such a class, you can create a ray tracer that has a general interface that assumes little about modeling primitives and debug it using only spheres. An important point is that anything that can be "hit" by a ray should be part of this class hierarchy, e.g., even a collection of surfaces should be considered a subclass of the surface class. This includes efficiency structures, such as bounding volume hierarchies; they can be hit by a ray, so they are in the class.

For example, the "abstract" or "base" class would specify the hit function as well as a bounding box function that will prove useful later:

```
class surface
virtual bool hit(ray $\mathbf{e} + t\mathbf{d}$, real $t_0$, real $t_1$, hit-record rec)
virtual box bounding-box()
```

Here (t_0, t_1) is the interval on the ray where hits will be returned, and *rec* is a record that is passed by reference; it contains data such as the t at intersection when *hit* returns true. The type *box* is a 3D "bounding box", that is two points that define an axis-aligned box that encloses the surface. For example, for a sphere, the function would be implemented by:

```
box sphere::bounding-box()
vector3 min = center - vector3(radius,radius,radius)
vector3 max = center + vector3(radius,radius,radius)
return box(min, max)
```

Another class that is useful is *material*. This allows you to abstract the material behavior and later add materials transparently. A simple way to link objects and materials is to add a pointer to a material in the surface class, although more programmable behavior might be desirable. A big question is what to do with textures; are they part of the material class or do they live outside of the material class? This will be discussed more in Chapter 10.

9.5 Shadows

Once you have a basic ray tracing program, shadows can be added very easily. Recall from Chapter 8 that light comes from some direction $\mathbf{l}$. If we imagine ourselves at a point $\mathbf{p}$ on a surface being shaded, the point is in shadow if we "look" in direction $\mathbf{l}$ and see an object. If there are no objects, then the light is not blocked.

This is shown in Figure 9.6, where the ray $\mathbf{p} + t\mathbf{l}$ does not hit any objects and is thus not in shadow. The point $\mathbf{q}$ is in shadow because the ray $\mathbf{q} + t\mathbf{l}$ does hit an object. The vector $\mathbf{l}$ is the same for both points because the light is "far" away. This assumption will later be relaxed. The rays that determine in or out of shadow are called *shadow rays* to distinguish them from viewing rays.

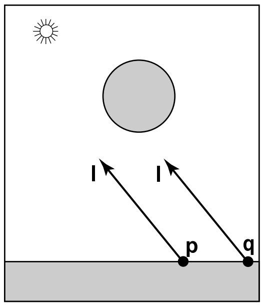


Figure 9.6. The point **p** is not in shadow while the point **q** is in shadow.

To get the algorithm for shading, we add an if statement to determine whether the point is in shadow. In a naive implementation, the shadow ray will check for $t \in [0, \infty)$, but because of numerical imprecision, this can result in an inter-

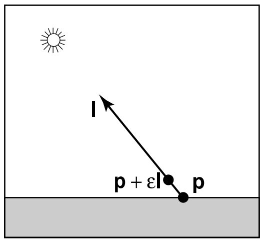


Figure 9.7. By testing in the interval starting at ϵ, we avoid numerical imprecision causing the ray to hit the surface **p** is on.

section with the surface on which $\mathbf{p}$ lies. Instead, the usual adjustment to avoid that problem is to test for $t \in [\epsilon, \infty)$ where ϵ is some small positive constant (Figure 9.7).

If we implement shadow rays for Phong lighting with Equation 8.9 then we have:

```
function raycolor( ray $\mathbf{e} + t\mathbf{d}$, real $t_0$, real $t_1$ )
hit-record rec, srec
if (scene→hit($\mathbf{e} + t\mathbf{d}$, $t_0$, $t_1$, rec)) then
    $\mathbf{p} = \mathbf{e} + rec.t\mathbf{d}$
    color $c = rec.c_r \ rec.c_a$
    if (not scene→hit($\mathbf{p} + s\mathbf{l}$, $\epsilon$, $\infty$, srec)) then
        vector3 $\mathbf{h} = normalized(normalized(\mathbf{l}) + normalized(-\mathbf{d}))$
        $c = c + rec.c_r \, c_l max\,(0, rec.\mathbf{n} \cdot \mathbf{l}) + c_l rec.c_p (\mathbf{h} \cdot rec.\mathbf{n})^{rec.p}$
    return c
else
    return background-color
```

Note that the ambient color is added in either case. If there are multiple light sources, we can send a shadow ray and evaluate the diffuse/phong terms for each light. The code above assumes that $\mathbf{d}$ and $\mathbf{l}$ are not necessarily unit vectors. This is crucial for $\mathbf{d}$, in particular, if we wish to cleanly add *instancing* later.

9.6 Specular Reflection

It is straightforward to add *specular* reflection to a ray tracing program. The key observation is shown in Figure 9.8 where a viewer looking from direction $\mathbf{e}$ sees what is in direction $\mathbf{r}$ as seen from the surface. The vector $\mathbf{r}$ is found using a variant of the Phong lighting reflection Equation 8.6. There are sign changes because the vector $\mathbf{d}$ points toward the surface in this case, so,

$$\mathbf{r} = \mathbf{d} + 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}, \tag{9.4}$$

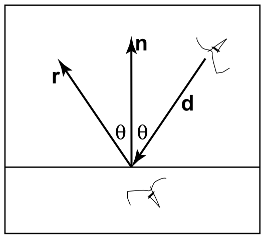


Figure 9.8. When looking into a perfect mirror, the viewer looking in direction **d** will see whatever the viewer "below" the surface would see in direction **r**.

In the real world, some energy is lost when the light reflects from the surface, and this loss can be different for different colors. For example, gold reflects yellow more efficiently than blue, so it shifts the colors of the objects it reflects. This can be implemented by adding a recursive call in raycolor:

```
color $c = c + c_s raycolor(\mathbf{p} + s\mathbf{r}, \epsilon, \infty)$
```

where c_s is the specular RGB color. We need to make sure we test for $s \in [\epsilon, \infty)$

for the same reason as we did with shadow rays; we don't want the reflection ray to hit the object that generates it.

The problem with the recursive call above is that it may never terminate. For example, if a ray starts inside a room, it will bounce forever. This can be fixed by adding a maximum recursion depth. The code will be more efficient if a reflection ray is generated only if c_s is not zero (black).

9.7 Refraction

Another type of specular object is a *dielectric*—a transparent material that refracts light. Diamonds, glass, water, and air are dielectrics. Dielectrics also filter light; some glass filters out more red and blue light than green light, so the glass takes on a green tint. When a ray travels from a medium with refractive index n into one with a refractive index n_t, some of the light is transmitted, and it bends. This is shown for $n_t > n$ in Figure 9.9. Snell's law tells us that

$$n \sin\theta = n_t \sin\phi.$$

Computing the sine of an angle between two vectors is usually not as convenient as computing the cosine which is a simple dot product for the unit vectors such as we have here. Using the trigonometric identity $\sin^2\theta + \cos^2\theta = 1$, we can derive a refraction relationship for cosines:

$$\cos^2\phi = 1 - \frac{n^2\left(1 - \cos^2\theta\right)}{n_t^2}.$$

Note that if n and n_t are reversed, then so are θ and ϕ as shown on the right of Figure 9.9.

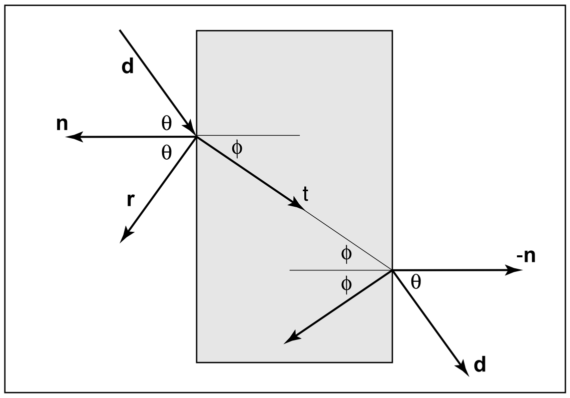


Figure 9.9. Snells' Law describes how the angle ϕ depends on the angle θ and the refractive indices of the object and the surrounding medium.

To convert $\sin\phi$ and $\cos\phi$ into a 3D vector, we can set up a 2D orthonormal basis in the plane of $\mathbf{n}$ and $\mathbf{d}$.

From Figure 9.10, we can see that $\mathbf{n}$ and $\mathbf{b}$ form an orthonormal basis for the plane of refraction. By definition, we can describe $\mathbf{t}$ in terms of this basis:

$$\mathbf{t} = \sin\phi\mathbf{b} - \cos\phi\mathbf{n}.$$

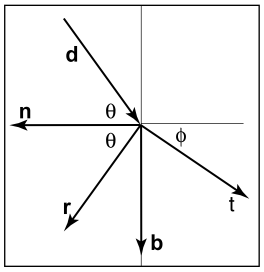


Figure 9.10. The vectors **n** and **b** form a 2D orthonormal basis that is parallel to the transmission vector **t**.

Since we can describe $\mathbf{d}$ in the same basis, and $\mathbf{d}$ is known, we can solve for $\mathbf{b}$:

$$\mathbf{d} = \sin\theta\mathbf{b} - \cos\theta\mathbf{n},$$
$$\mathbf{b} = \frac{\mathbf{d} + \mathbf{n}\cos\theta}{\sin\theta}.$$

This means that we can solve for $\mathbf{t}$ with known variables:

$$\begin{aligned}\mathbf{t} &= \frac{n\left(\mathbf{d} + \mathbf{n}\cos\theta)\right)}{n_t} - \mathbf{n}\cos\phi \\ &= \frac{n\left(\mathbf{d} - \mathbf{n}(\mathbf{d}\cdot\mathbf{n})\right)}{n_t} - \mathbf{n}\sqrt{1 - \frac{n^2\left(1 - (\mathbf{d}\cdot\mathbf{n})^2\right)}{n_t^2}}.\end{aligned}$$

Note that this equation works regardless of which of n and n_t is larger. An immediate question is, "What should you do if the number under the square root is negative?" In this case, there is no refracted ray and all of the energy is reflected. This is known as *total internal reflection*, and it is responsible for much of the rich appearance of glass objects.

The reflectivity of a dielectric varies with the incident angle according to the *Fresnel Equations*. A nice way to implement something close to the Fresnel Equations is to use the *Schlick approximation*,

$$R(\theta) = R_0 + (1 - R_0)\left(1 - \cos\theta\right)^5,$$

where R_0 is the reflectance at normal incidence:

$$R_0 = \left(\frac{n_t - 1}{n_t + 1}\right)^2.$$

Note that the $\cos\theta$ terms above are always for the angle in air (the larger of the internal and external angles relative to the normal).

For homogeneous impurities, as is found in typical glass, a light-carrying ray's intensity will be attenuated according to *Beer's Law*. As the ray travels through the medium it loses intensity according to $dI = -CI\,dx$, where dx is distance. Thus, $dI/dx = -CI$. We can solve this equation and get the exponential $I = k\exp(-Cx) + k'$. The degree of attenuation is described by the RGB attenuation

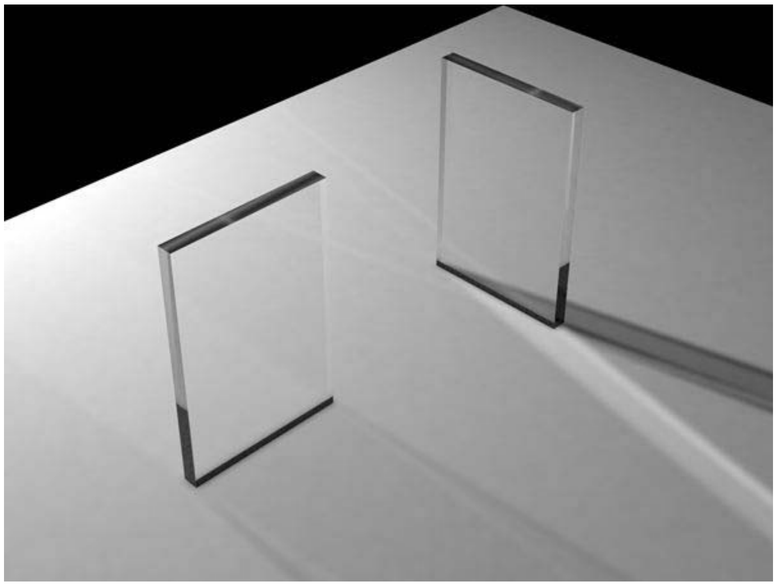


Figure 9.11. The color of the glass is affected by total internal reflection and Beer's Law. The amount of light transmitted and reflected is determined by the Fresnel Equations. The complex lighting on the ground plane was computed using particle tracing as described in Chapter **??**. (See also Plate PLATE.)

constant a, which is the amount of attenuation after one unit of distance. Putting in boundary conditions, we know that $I(0) = I_0$, and $I(1) = aI(0)$. The former implies $I(x) = I_0 \exp(-Cx)$. The latter implies $I_0 a = I_0 \exp(-C)$, so $-C = \ln(a)$. Thus, the final formula is

$$I(s) = I(0)e^{-\ln(a)s},$$

where $I(s)$ is the intensity of the beam at distance s from the interface. In practice, we reverse-engineer a by eye, because such data is rarely easy to find. The effect of Beer's Law can be seen in Figure 9.11, where the glass takes on a green tint.

To add transparent materials to our code, we need a way to determine when a ray is going "into" an object. The simplest way to do this is to assume that all objects are embedded in air with refractive index very close to 1.0, and that surface normals point "out" (toward the air). The code segment for rays and dielectrics with these assumptions is:

```
if ($\mathbf{p}$ is on a dielectric) then
  $\mathbf{r}$ = reflect($\mathbf{d}$, $\mathbf{n}$ )
  if ($\mathbf{d} \cdot \mathbf{n} < 0$) then
    refract($\mathbf{d}$, $\mathbf{n}$,$n$, $\mathbf{t}$ )
```

```
        $c = -\mathbf{d} \cdot \mathbf{n}$
        $k_r = k_g = k_b = 1$
    ***else***
        $k_r = \exp(-a_r t)$
        $k_g = \exp(-a_g t)$
        $k_b = \exp(-a_b t)$
        ***if*** *refract*($\mathbf{d}$, -$\mathbf{n}$, *1/n*, $\mathbf{t}$ ) ***then***
            $c = \mathbf{t} \cdot \mathbf{n}$
        ***else***
            *return* $k*$*color*$(\mathbf{p} + t\mathbf{r})$
    $R_0 = (n-1)^2/(n+1)^2$
    $R = R_0 + (1 - R_0)(1 - c)^5$
    *return* $k(R$ *color*$(\mathbf{p} + t\mathbf{r})$ + $(1 - R)$ *color*$(\mathbf{p} + t\mathbf{t}))$
```

The code above assumes that the natural log has been folded into the constants (a_r, a_g, a_b). The *refract* function returns false if there is total internal reflection, and otherwise it fills in the last argument of the argument list.

9.8 Instancing

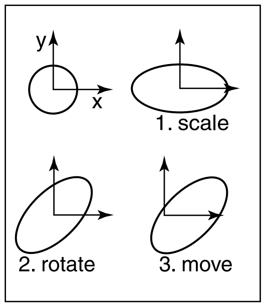


Figure 9.12. An instance of a circle with a series of three transforms is an ellipse.

An elegant property of ray tracing is that it allows very natural *instancing*. The basic idea of instancing is to distort all points on an object by a transformation matrix before the object is displayed. For example, if we transform the unit circle (in 2D) by a scale factor $(2, 1)$ in x and y, respectively, then rotate it by $45°$, and move one unit in the x-direction, the result is an ellipse with an eccentricity of 2 and a long axis along the $x = -y$-direction centered at $(0, 1)$ (Figure 9.12). The key thing that makes that entity an "instance" is that we store the circle and the composite transform matrix. Thus, the explicit construction of the ellipse is left as a future procedure operation at render time.

The advantage of instancing in ray tracing is that we can choose the space in which to do intersection. If the base object is composed of a set of points, one of which is $\mathbf{p}$, then the transformed object is composed of that set of points transformed by matrix $\mathbf{M}$, where the example point is transformed to $\mathbf{Mp}$. If we have a ray $\mathbf{a} + t\mathbf{b}$ which we want to intersect with the transformed object, we can instead intersect an *inverse-transformed ray* with the untransformed object (Figure 9.13). There are two potential advantages to computing in the untransformed space (i.e., the right-hand side of Figure 9.13):

1. the untransformed object may have a simpler intersection routine, e.g., a sphere versus an ellipsoid;

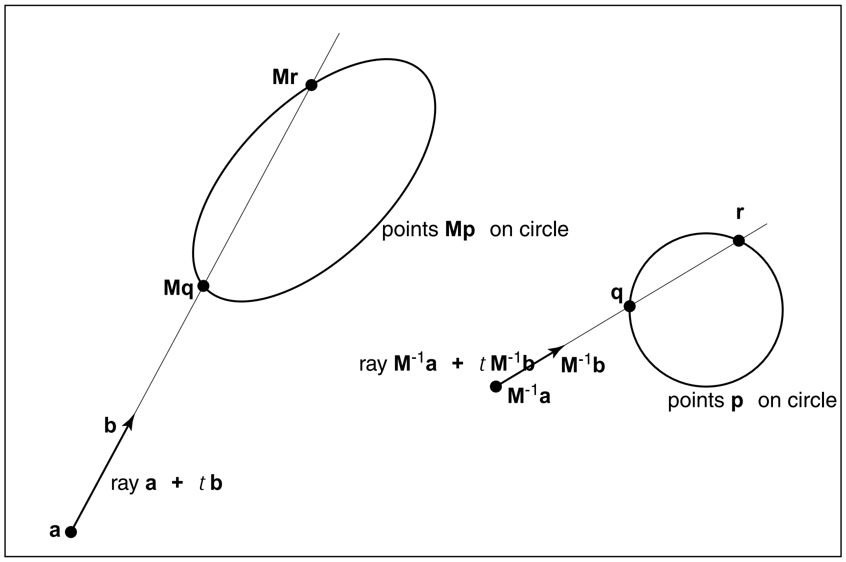

Mr
points **Mp** on circle
Mq
r
q
ray $\mathbf{M}^{-1}\mathbf{a} + t\mathbf{M}^{-1}\mathbf{b}$
$\mathbf{M}^{-1}\mathbf{b}$
$\mathbf{M}^{-1}\mathbf{a}$
points **p** on circle
b
ray **a** + *t* **b**
a

Figure 9.13. The ray intersection problem in the two spaces are just simple transforms of each other. The object is specified as a sphere plus matrix **M**. The ray is specified in the transformed (world) space by location **a** and direction **b**.

2. many transformed objects can share the same untransformed object thus reducing storage, e.g., a traffic jam of cars, where individual cars are just transforms of a few base (untransformed) models.

As discussed in Section 5.2.2, surface normal vectors transform differently. With this in mind and using the concepts illustrated in Figure 9.13, we can determine the intersection of a ray and an object transformed by matrix $\mathbf{M}$. If we create an instance class of type *surface*, we need to create a *hit* function:

```
instance::hit(ray $\mathbf{a} + t\mathbf{b}$, real $t_0$, real $t_1$, hit-record rec)
ray $\mathbf{r}' = \mathbf{M}^{-1}\mathbf{a} + t\mathbf{M}^{-1}\mathbf{b}$
if (base-object→hit($\mathbf{r}'$, $t_0$, $t_1$, rec)) then
    rec.$\mathbf{n} = (\mathbf{M}^{-1})^T$ rec.$\mathbf{n}$
    return true
else
    return false
```

An elegant thing about this function is that the parameter rec.t does not need to be changed, because it is the same in either space. Also note that we need not compute or store the matrix M.

This brings up a very important point: the ray direction $\mathbf{b}$ must *not* be restricted to a unit-length vector, or none of the infrastructure above works. For this reason, it is useful not to restrict ray directions to unit vectors.

For the purpose of solid texturing, you may want to record the local coordinates of the hitpoint and return this in the hit-record. This is just ray $\mathbf{r}'$ advanced by parameter rec.t.

To implement the bounding-box function of class instance, we can just take the eight corners of the bounding box of the base object and transform all of them by $\mathbf{M}$, and then take the bounding box of those eight points. That will not necessarily yield the tightest bounding box, but it is general and straightforward to implement.

9.9 Sub-Linear Ray-Object Intersection

In the earlier ray-object intersection pseudocode, all objects are looped over, checking for intersections. For N objects, this is an $O(N)$ linear search and is thus slow for large values of N. Like most search problems, the ray-object intersection can be computed in sub-linear time using "divide and conquer" techniques, provided we can create an ordered data structure as a preprocess. There are many techniques to do this.

This section discusses three of these techniques in detail: bounding volume hierarchies (Rubin & Whitted, 1980; Whitted, 1980; Goldsmith & Salmon, 1987), uniform spatial subdivision (Cleary, Wyvill, Birtwistle, & Vatti, 1983; Fujimoto, Tanaka, & Iwata, 1986; Amanatides & Woo, 1987), and binary-space partition-

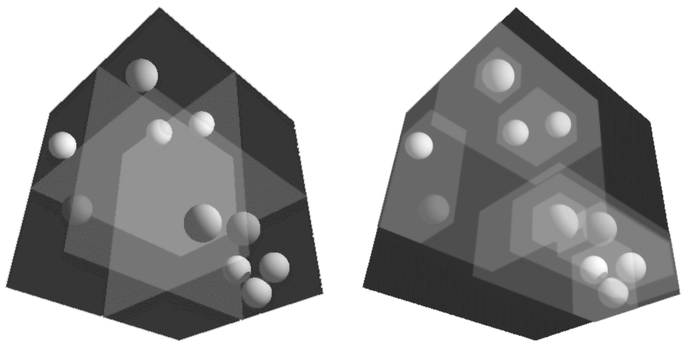


Figure 9.14. Left: a uniform partitioning of space. Right: adaptive bounding-box hierarchy. *Image courtesy David DeMarle.*

ing (Glassner, 1984; Jansen, 1986; Havran, 2000). An example of the first two strategies is shown in Figure 9.14. References for other popular strategies are given in the notes at the end of the chapter.

9.9.1 Bounding Boxes

A key operation in most intersection acceleration schemes is computing the intersection of a ray with a bounding box (Figure 9.15). This differs from conventional intersection tests in that we do not need to know where the ray hits the box; we only need to know whether it hits the box.

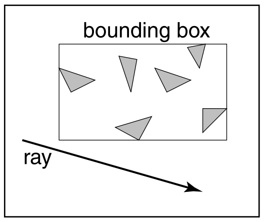


Figure 9.15. The ray is only tested for intersection with the surfaces if it hits the bounding box.

To build an algorithm for ray-box intersection, we begin by considering a 2D ray whose direction vector has positive x and y components. We can generalize this to arbitrary 3D rays later. The 2D bounding box is defined by two horizontal and two vertical lines:

$$\begin{aligned} x &= x_{\text{min}}, \\ x &= x_{\text{max}}, \\ y &= y_{\text{min}}, \\ y &= y_{\text{max}}. \end{aligned}$$

The points bounded by these lines can be described in interval notation:

$$(x, y) \in [x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}].$$

As shown in Figure 9.16, the intersection test can be phrased in terms of these intervals. First, we compute the ray parameter where the ray hits the line $x = x_{\text{min}}$:

$$t_{\text{xmin}} = \frac{x_{\text{min}} - x_e}{x_d}.$$

We then make similar computations for t_{xmax}, t_{ymin}, and t_{ymax}. The ray hits the box if and only if the intervals $[t_{\text{xmin}}, t_{\text{xmax}}]$ and $[t_{\text{ymin}}, t_{\text{ymax}}]$ overlap, i.e., their intersection is non-empty. In pseudocode this algorithm is:

```
$t_{xmin} = (x_{min} - x_e)/x_d$
$t_{xmax} = (x_{max} - x_e)/x_d$
$t_{ymin} = (y_{min} - y_e)/x_d$
$t_{ymax} = (y_{max} - y_e)/x_d$
if $(t_{xmin} > t_{ymax})$ or $(t_{ymin} > t_{xmax})$ then
    return false
else
    return true
```

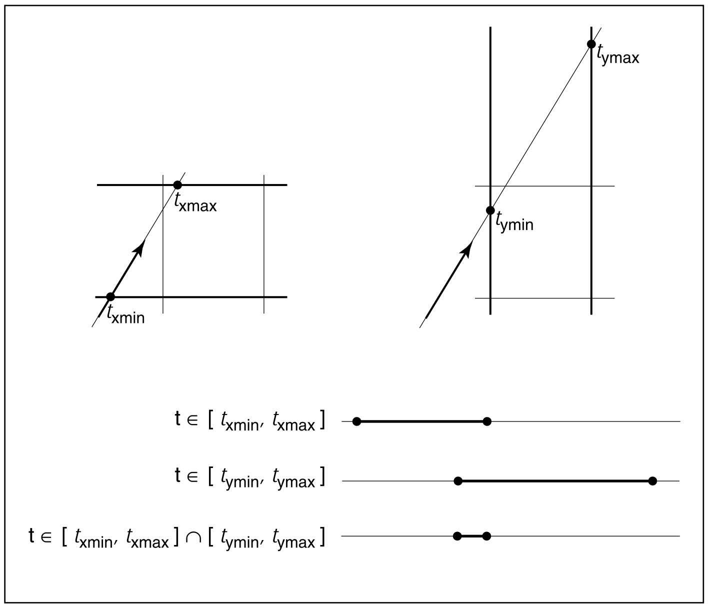

t_{xmax}
t_{xmin}
t_{ymax}
t_{ymin}
$t \in [\, t_{xmin}, t_{xmax} \,]$
$t \in [\, t_{ymin}, t_{ymax} \,]$
$t \in [\, t_{xmin}, t_{xmax} \,] \cap [\, t_{ymin}, t_{ymax} \,]$

Figure 9.16. The ray will be inside the interval $x \in [x_{\min}, x_{\max}]$ for some interval in its parameter space $t \in [t_{x\min}, t_{x\max}]$. A similar interval exists for the y interval. The ray intersects the box if it is in both the x interval and y interval at the same time, i.e., the intersection of the two one-dimensional intervals is not empty.

The if statement may seem non-obvious. To see the logic of it, note that there is no overlap if the first interval is either entirely to the right or entirely to the left of the second interval.

The first thing we must address is the case when x_d or y_d is negative. If x_d is negative, then the ray will hit $x_{\max}$ before it hits $x_{\min}$. Thus the code for computing t_{xmin} and t_{xmax} expands to:

```
if ($x_d \geq 0$) then
    $t_{xmin} = (x_{min} - x_e)/x_d$
    $t_{xmax} = (x_{max} - x_e)/x_d$
else
    $t_{xmin} = (x_{max} - x_e)/x_d$
    $t_{xmax} = (x_{min} - x_e)/x_d$
```

A similar code expansion must be made for the y cases. A major concern is that horizontal and vertical rays have a zero value for y_d and x_d, respectively. This will cause divide by zero which may be a problem. However, before addressing this directly, we check whether IEEE floating point computation handles these cases gracefully for us. Recall from Section 1.6 the rules for divide by zero: for any positive real number a,

$$+a/0 = +\infty;$$
$$-a/0 = -\infty.$$

Consider the case of a vertical ray where $x_d = 0$ and $y_d > 0$. We can then calculate

$$t_{\text{xmin}} = \frac{x_{\text{min}} - x_e}{0};$$
$$t_{\text{xmax}} = \frac{x_{\text{max}} - x_e}{0}.$$

There are three possibilities of interest:

1. $x_e \leq x_{\text{min}}$ (no hit);

2. $x_{\text{min}} < x_e < x_{\text{max}}$ (hit);

3. $x_{\text{max}} \leq x_e$ (no hit).

For the first case we have

$$t_{\text{xmin}} = \frac{\text{positive number}}{0};$$
$$t_{\text{xmax}} = \frac{\text{positive number}}{0}.$$

This yields the interval $(t_{\text{xmin}}, t_{\text{xmin}}) = (\infty, \infty)$. That interval will not overlap with any interval, so there will be no hit, as desired. For the second case, we have

$$t_{\text{xmin}} = \frac{\text{negative number}}{0};$$
$$t_{\text{xmax}} = \frac{\text{positive number}}{0}.$$

This yields the interval $(t_{\text{xmin}}, t_{\text{xmin}}) = (-\infty, \infty)$ which will overlap with all intervals and thus will yield a hit as desired. The third case results in the interval $(-\infty, -\infty)$ which yields no hit, as desired. Because these cases work as desired, we need no special checks for them. As is often the case, IEEE floating point conventions are our ally. However, there is still a problem with this approach.

Consider the code segment:

```
if $(x_d \geq 0)$ then
   $t_{min} = (x_{min} - x_e)/x_d$
   $t_{max} = (x_{max} - x_e)/x_d$
else
   $t_{min} = (x_{max} - x_e)/x_d$
   $t_{max} = (x_{min} - x_e)/x_d$
```

This code breaks down when $x_d = -0$. This can be overcome by testing on the reciprocal of x_d (A. Williams, Barrus, Morley, & Shirley, 2005):

```
$a = 1/x_d$
if $(a \geq 0)$ then
   $t_{min} = a(x_{min} - x_e)$
   $t_{max} = a(x_{max} - x_e)$
else
   $t_{min} = a(x_{max} - x_e)$
   $t_{max} = a(x_{min} - x_e)$
```

9.9.2 Hierarchical Bounding Boxes

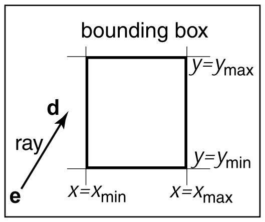


Figure 9.17. A 2D ray **e** + *t***d** is tested against a 2D bounding box.

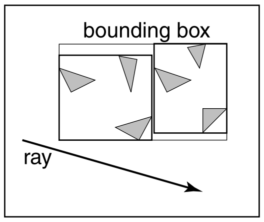


Figure 9.18. The bounding boxes can be nested by creating boxes around subsets of the model.

The basic idea of hierarchical bounding boxes can be seen by the common tactic of placing an axis-aligned 3D bounding box around all the objects as shown in Figure 9.17. Rays that hit the bounding box will actually be more expensive to compute than in a brute force search, because testing for intersection with the box is not free. However, rays that miss the box are cheaper than the brute force search. Such bounding boxes can be made hierarchical by partitioning the set of objects in a box and placing a box around each partition as shown in Figure 9.18. The data structure for the hierarchy shown in Figure 9.19 might be a tree with the large bounding box at the root and the two smaller bounding boxes as left and right subtrees. These would in turn each point to a list of three triangles. The intersection of a ray with this particular hard-coded tree would be:

```
if (ray hits root box) then
   if (ray hits left subtree box) then
      check three triangles for intersection
   if (ray intersects right subtree box) then
      check other three triangles for intersection
   if (an intersections returned from each subtree) then
      return the closest of the two hits
```

```
    else if (a intersection is returned from exactly one subtree) then
        return that intersection
    else
        return false
else
    return false
```

Some observations related to this algorithm are that there is no geometric ordering between the two subtrees, and there is no reason a ray might not hit both subtrees. Indeed, there is no reason that the two subtrees might not overlap.

A key point of such data hierarchies is that a box is guaranteed to bound all objects that are below it in the hierarchy, but they are *not* guaranteed to contain all objects that overlap it spatially, as shown in Figure 9.19. This makes this geometric search somewhat more complicated than a traditional binary search on strictly ordered one-dimensional data. The reader may note that several possible optimizations present themselves. We defer optimizations until we have a full hierarchical algorithm.

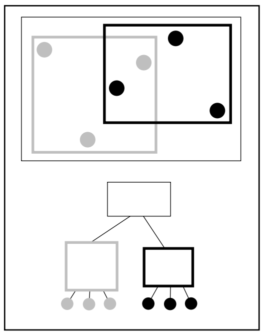


Figure 9.19. The grey box is a tree node that points to the three grey spheres, and the thick black box points to the three black spheres. Note that not all spheres enclosed by the box are guaranteed to be pointed to by the corresponding tree node.

If we restrict the tree to be binary and require that each node in the tree have a bounding box, then this traversal code extends naturally. Further, assume that all nodes are either leaves in the tree and contain a primitive, or that they contain one or two subtrees.

The *bvh-node* class should be of type surface, so it should implement *surface::hit*. The data it contains should be simple:

```
class bvh-node subclass of surface
virtual bool hit(ray $\mathbf{e} + t\mathbf{d}$, real $t_0$, real $t_1$, hit-record rec)
virtual box bounding-box()
surface-pointer left
surface-pointer right
box bbox
```

The traversal code can then be called recursively in an object-oriented style:

```
bool bvh-node::hit(ray $\mathbf{a} + t\mathbf{b}$, real $t_0$, real $t_1$, hit-record rec)
if (bbox.hitbox($\mathbf{a} + t\mathbf{b}$, $t_0$, $t_1$)) then
    hit-record lrec, rrec
    left-hit = (left $\neq$ NULL) and (left $\rightarrow$ hit($\mathbf{a} + t\mathbf{b}$, $t_0$, $t_1$, lrec))
    right-hit = (right $\neq$ NULL) and (right $\rightarrow$ hit($\mathbf{a} + t\mathbf{b}$, $t_0$, $t_1$, rrec))
    if (left-hit and right-hit) then
        if (lrec.t < rrec.t) then
            rec = lrec
        else
            rec = rrec
        return true
    else if (left-hit) then
        rec = lrec
        return true
    else if (right-hit) then
        rec = rrec
        return true
    else
        return false
else
    return false
```

Note that because *left* and *right* point to surfaces rather than bvh-nodes specifically, we can let the virtual functions take care of distinguishing between internal and leaf nodes; the appropriate *hit* function will be called. Note, that if the tree is built properly, we can eliminate the check for *left* being *NULL*. If we want to eliminate the check for *right* being *NULL*, we can replace NULL right pointers with a redundant pointer to left. This will end up checking left twice, but will eliminate the check throughout the tree. Whether that is worth it will depend on the details of tree construction.

There are many ways to build a tree for a bounding volume hierarchy. It is convenient to make the tree binary, roughly balanced, and to have the boxes of sibling subtrees not overlap too much. A heuristic to accomplish this is to sort the surfaces along an axis before dividing them into two sublists. If the axes are defined by an integer with $x = 0$, $y = 1$, and $z = 2$ we have:

```
bvh-node::bvh-node(object-array A, int AXIS)
N = A.length
if (N= 1) then
    left = A[0]
    right = NULL
    bbox = bounding-box(A[0])
else if (N= 2) then
    left-node = A[0]
    right-node = A[1]
    bbox = combine(bounding-box(A[0]), bounding-box(A[1]))
else
    sort A by the object center along AXIS
    left= new bvh-node($A[0..N/2-1]$, $(AXIS+1) \mod 3$)
    right = new bvh-node($A[N/2..N-1]$, $(AXIS+1) \mod 3$)
    bbox = combine(left-node → bbox, right-node → bbox)
```

The quality of the tree can be improved by carefully choosing *AXIS* each time. One way to do this is to choose the axis such that the sum of the volumes of the bounding boxes of the two subtrees is minimized. This change compared to rotating through the axes will make little difference for scenes composed of isotopically distributed small objects, but it may help significantly in less well-behaved scenes. This code can also be made more efficient by doing just a partition rather than a full sort.

Another, and probably better, way to build the tree is to have the subtrees contain about the same amount of space rather than the same number of objects. To do this we partition the list based on space:

```
bvh-node::bvh-node(object-array A, int AXIS)
N = A.length
if ($N = 1$) then
    left = $A[0]$
    right = NULL
    bbox = bounding-box($A[0]$)
else if ($N = 2$) then
    left = $A[0]$
    right = $A[1]$
    bbox = combine(bounding-box($A[0]$), bounding-box($A[1]$))
else
    find the midpoint $m$ of the bounding box of A along AXIS
    partition A into lists with lengths k and (N-k) surrounding $m$
    left = new node($A[0..k]$, $(AXIS+1) \mod 3$)
    right = new node($A[k+1..N-1]$, $(AXIS+1) \mod 3$)
    bbox = combine(left-node → bbox, right-node → bbox)
```

Although this results in an unbalanced tree, it allows for easy traversal of empty space and is cheaper to build because partitioning is cheaper than sorting.

9.9.3 Uniform Spatial Subdivision

Another strategy to reduce intersection tests is to divide space. This is fundamentally different from dividing objects as was done with hierarchical bounding volumes:

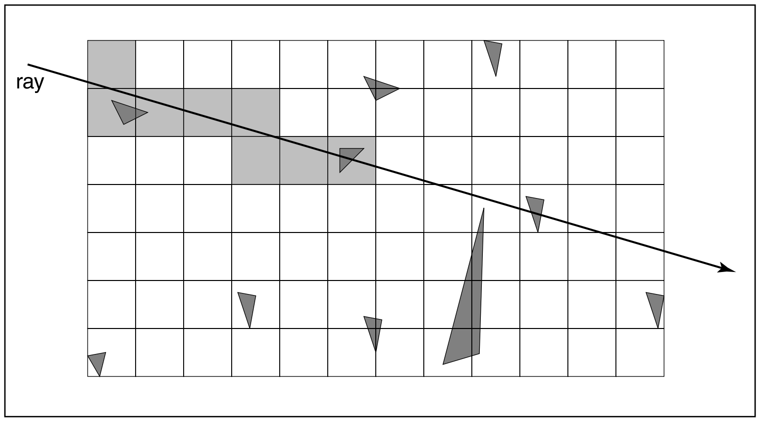


Figure 9.20. In uniform spatial subdivision, the ray is tracked forward through cells until an object in one of those cells is hit. In this example, only objects in the shaded cells are checked.

- In hierarchical bounding volumes, each object belongs to one of two sibling nodes, whereas a point in space may be inside both sibling nodes.
- In spatial subdivision, each point in space belongs to exactly one node, whereas objects may belong to many nodes.

The scene is partitioned into axis-aligned boxes. These boxes are all the same size, although they are not necessarily cubes. The ray traverses these boxes as shown in Figure 9.20. When an object is hit, the traversal ends.

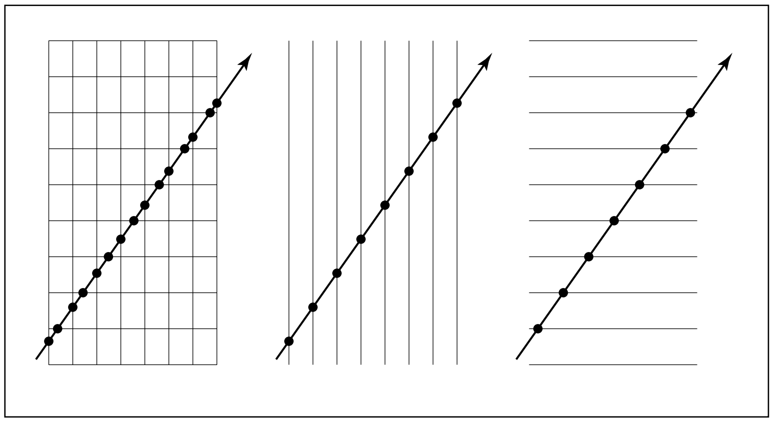


Figure 9.21. Although the pattern of cell hits seems irregular (left), the hits on sets of parallel planes are very even.

The grid itself should be a subclass of surface and should be implemented as a 3D array of pointers to surface. For empty cells these pointers are NULL. For cells with one object, the pointer points to that object. For cells with more than one object, the pointer can point to a list, another grid, or another data structure, such as a bounding volume hierarchy.

This traversal is done in an incremental fashion. The regularity comes from the way that a ray hits each set of parallel planes, as shown in Figure 9.21. To see how this traversal works, first consider the 2D case where the ray direction has positive x and y components and starts outside the grid. Assume the grid is bounded by points $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$. The grid has n_x by n_y cells.

Our first order of business is to find the index (i, j) of the first cell hit by the ray $\mathbf{e} + t\mathbf{d}$. Then, we need to traverse the cells in an appropriate order. The key parts to this algorithm are finding the initial cell (i, j) and deciding whether to increment i or j (Figure 9.22). Note that when we check for an intersection with objects in a cell, we restrict the range of t to be within the cell (Figure 9.23). Most implementations make the 3D array of type "pointer to surface." To improve the locality of the traversal, the array can be tiled as discussed in Section 12.4.

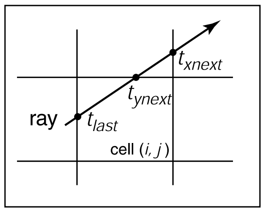


Figure 9.22. To decide whether we advance right or upwards, we keep track of the intersections with the next vertical and horizontal boundary of the cell.

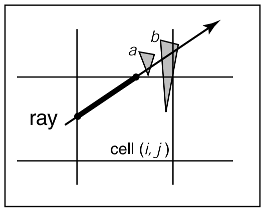


Figure 9.23. Only hits within the cell should be reported. Otherwise the case above would cause us to report hitting object *b* rather than object *a*.

9.9.4 Binary-Space Partitioning

We can also partition space in a hierarchical data structure such as a binary-space-partioning tree (BSP tree). This is similar to the BSP tree used for a painter's algorithm in Chapter 7, but it usually uses axis-aligned cutting planes for easier ray intersection. A node in this structure might contain a single cutting plane and a left and right subtree. These subtrees would contain all objects on either side of the cutting plane. Objects that pass through the plane would be in each subtree. If we assume the cutting plane is parallel to the yz plane at $x = D$, then the node class is:

```
class bsp-node subclass of surface
virtual bool hit(ray $\mathbf{e} + t\mathbf{d}$, real $t_0$, real $t_1$, hit-record rec)
virtual box bounding-box()
surface-pointer left
surface-pointer right
real D
```

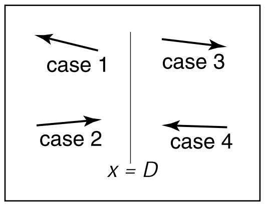


Figure 9.24. The four cases of how a ray relates to the BSP cutting plane *x = D*.

We generalize this to y and z cutting planes later. The intersection code can then be called recursively in an object-oriented style. The code considers the four cases shown in Figure 9.24. For our purposes, the origin of these rays is a point at parameter t_0:

$$\mathbf{p} = \mathbf{a} + t_0\mathbf{b}.$$

The four cases are:

1. The ray only interacts with the left subtree, and we need not test it for intersection with the cutting plane. It occurs for $x_p < D$ and $x_b < 0$.

2. The ray is tested against the left subtree, and if there are no hits, it is then tested against the right subtree. We need to find the ray parameter at $x = D$, so we can make sure we only test for intersections within the subtree. This case occurs for $x_p < D$ and $x_b > 0$.

3. This case is analogous to case 1 and occurs for $x_p > D$ and $x_b > 0$.

4. This case is analogous to case 2 and occurs for $x_p > D$ and $x_b < 0$.

The resulting traversal code handling these cases in order is:

```
bool bsp-node::hit( ray $\mathbf{a} + t\mathbf{b}$, real $t_0$, real $t_1$, hit-record rec)
$x_p = x_a + t_0 x_b$
if ($x_p < D$) then
   if ($x_b < 0$) then
      return (left $\neq$ NULL) and (left$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, $t_0$, $t_1$, rec))
   $t = (D - x_a)/x_b$
   if ($t > t_1$) then
      return (left $\neq$ NULL) and (left$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, $t_0$, $t_1$, rec))
   if (left $\neq$ NULL) and (left$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, $t_0$, t, rec)) then
      return true
   return (right $\neq$ NULL) and (right$\rightarrow$hit($\mathbf{a} + t\mathbf{b}$, t, $t_1$, rec))
else
   analogous code for cases 3 and 4
```

This is very clean code. However, to get it started, we need to hit some root object that includes a bounding box so we can initialize the traversal, t_0 and t_1. An issue we have to address is that the cutting plane may be along any axis. We can add an interger index *axis* to the bsp-node class. If we allow an indexing operator for points, this will result in some simple modifications to the code above, for example,

$x_p = x_a + t_0 x_b$

would become

$u_p = a[\mathit{axis}] + t_0 b[\mathit{axis}]$

which will result in some additional array indexing, but will not generate more branches.

While the processing of a single bsp-node is faster than processing a bvh-node, the fact that a single surface may exist in more than one subtree means there are more nodes and, potentially, a higher memory use. How "well" the trees are built determines which is faster. Building the tree is similar to building the BVH tree. We can pick axes to split in a cycle, and we can split in half each time, or we can try to be more sophisticated in how we divide.

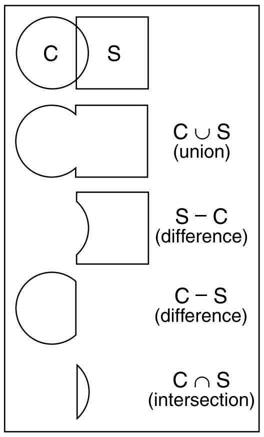


Figure 9.25. The basic CSG operations on a 2D circle and square.

9.10 Constructive Solid Geometry

One nice thing about ray tracing is that any geometric primitive whose intersection with a 3D line can be computed can be seamlessly added to a ray tracer. It turns out to also be straightforward to add *constructive solid geometry* (CSG) to a ray tracer (Roth, 1982). The basic idea of CSG is to use set operations to combine solid shapes. These basic operations are shown in Figure 9.25. The operations can be viewed as *set* operations. For example, we can consider C the set of all points in the circle, and S the set of all points in the square. The intersection operation $C \cap S$ is the set of all points that are both members of C and S. The other operations are analogous.

Although one can do CSG directly on the model, if all that is desired is an image, we do not need to explicitly change the model. Instead, we perform the set operations directly on the rays as they interact with a model. To make this natural, we find all the intersections of a ray with a model rather than just the closest. For example, a ray $\mathbf{a} + t\mathbf{b}$ might hit a sphere at $t = 1$ and $t = 2$. In the context of CSG, we think of this as the ray being inside the sphere for $t \in [1, 2]$. We can compute these "inside intervals" for all of the surfaces and do set operations on those intervals (recall Section 2.1.2). This is illustrated in Figure 9.26, where the hit intervals are processed to indicate that there are two intervals inside the difference object. The first hit for $t > 0$ is what the ray actually intersects.

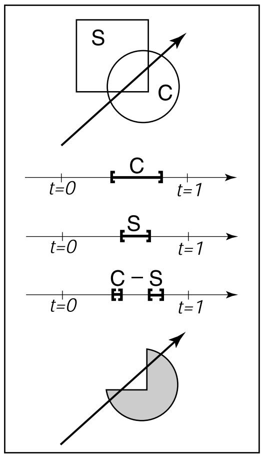


Figure 9.26. Intervals are processed to indicate how the ray hits the composite object.

In practice, the CSG intersection routine must maintain a list of intervals. When the first hitpoint is determined, the material property and surface normal is that associated with the hitpoint. In addition, you must pay attention to precision issues because there is nothing to prevent the user from taking two objects that abut and taking an intersection. This can be made robust by eliminating any interval whose thickness is below a certain tolerance.

9.11 Distribution Ray Tracing

For some applications, ray-traced images are just too "clean." This effect can be mitigated using *distribution ray tracing* (Cook et al., 1984) . The conventionally ray-traced images look clean, because everything is crisp; the shadows are perfectly sharp, the reflections have no fuzziness, and everything is in perfect focus. Sometimes we would like to have the shadows be soft (as they are in real life), the reflections be fuzzy as with brushed metal, and the image have variable degrees of focus as in a photograph with a large aperture. While accomplishing these things from first principles is somewhat involved (as is developed in Chapter **??**), we can get most of the visual impact with some fairly simple changes to the basic ray tracing algorithm. In addition, the framework gives us a relatively simple way to antialias (recall Section 3.7) the image.

9.11.1 Antialiasing

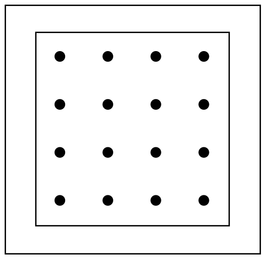


Figure 9.27. Sixteen regular samples for a single pixel.

Recall that a simple way to antialias an image is to compute the average color for the area of the pixel rather than the color at the center point. In ray tracing, our computational primitive is to compute the color at a point on the screen. If we average many of these points across the pixel, we are approximating the true average. If the screen coordinates bounding the pixel are $[i, i+1] \times [j, j+1]$, then we can replace the loop:

```
***for*** *each pixel* $(i, j)$ ***do***
    $c_{ij} = \textit{ray-color}(i + 0.5, j + 0.5)$
```

with code that samples on a regular $n \times n$ grid of samples within each pixel:

```
***for*** *each pixel* $(i, j)$ ***do***
    $c = 0$
    ***for*** $p = 0$ *to* $n - 1$ ***do***
        ***for*** $q = 0$ *to* $n - 1$ ***do***
            $c = c + \textit{ray-color}(i + (p + 0.5)/n, j + (q + 0.5)/n)$
    $c_{ij} = c/n^2$
```

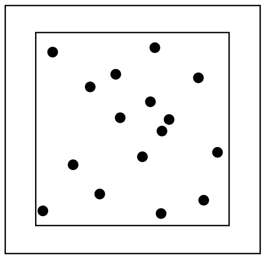


Figure 9.28. Sixteen random samples for a single pixel.

This is usually called *regular sampling*. The 16 sample locations in a pixel for $n = 4$ are shown in Figure 9.27. Note that this produces the same answer as rendering a traditional ray-traced image with one sample per pixel at $n_x n$ by $n_y n$ resolution and then averaging blocks of n by n pixels to get a n_x by n_y image.

One potential problem with taking samples in a regular pattern within a pixel is that regular artifacts such as Moire patterns can arise. These artifacts can be turned into noise by taking samples in a random pattern within each pixel as shown in Figure 9.28. This is usually called *random sampling* and involves just a small change to the code:

```
for each pixel $(i, j)$ do
    $c = 0$
    for $p = 1$ to $n^2$ do
        $c = c$ + ray-color$(i + \xi, j + \xi)$
    $c_{ij} = c/n^2$
```

Here ξ is a call that returns a uniform random number in the range $[0, 1)$. Unfortunately, the noise can be quite objectionable unless many samples are taken. A compromise is to make a hybrid strategy that randomly perturbs a regular grid:

```
for each pixel $(i, j)$ do
    $c = 0$
    for $p = 0$ to $n - 1$ do
        for $q = 0$ to $n - 1$ do
            $c = c$ + ray-color$(i + (p + \xi)/n, j + (q + \xi)/n)$
    $c_{ij} = c/n^2$
```

That method is usually called *jittering* or *stratified sampling* (Figure 9.29).

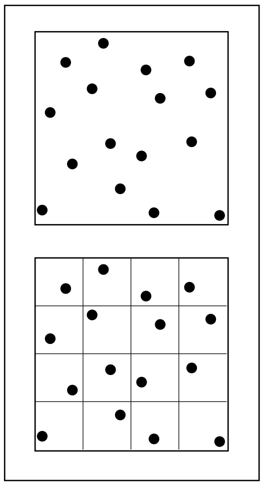


Figure 9.29. Sixteen stratified (jittered) samples for a single pixel shown with and without the bins highlighted. There is exactly one random sample taken within each bin.

9.11.2 Soft Shadows

The reason shadows are hard to handle in standard ray tracing is that lights are infinitesimal points or directions and are thus either visible or invisible. In real life, lights have non-zero area and can thus be partially visible. This idea is shown in 2D in Figure 9.30. The region where the light is entirely invisible is called the *umbra*. The partially visible region is called the *penumbra*. There is not a commonly used term for the region not in shadow, but it is sometimes called the *anti-umbra*.

The key to implementing soft shadows is to somehow account for the light being an area rather than a point. An easy way to do this is to approximate the light with a distributed set of N point lights each with one Nth of the intensity of the base light. This concept is illustrated at the left of Figure 9.31 where nine lights are used. You can do this in a standard ray tracer, and it is a common trick to get soft shadows in an off-the-shelf renderer. There are two potential problems with this technique. First, typically dozens of point lights are needed to achieve visually smooth results, which slows down the program a great deal. The second problem is that the shadows have sharp transitions inside the penumbra.

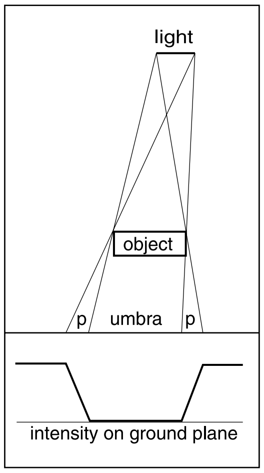


Figure 9.30. A soft shadow has a gradual transition from the unshadowed to shadowed region. The transition zone is the "penumbra" denoted by *p* in the figure.

Distribution ray tracing introduces a small change in the shadowing code. Instead of representing the area light at a discrete number of point sources, we represent it as an infinite number and choose one at random for each viewing ray.

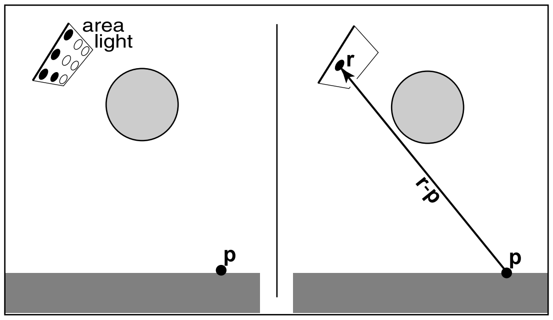


Figure 9.31. Left: an area light can be approximated by some number of point lights; four of the nine points are visible to **p** so it is in the penumbra. Right: a random point on the light is chosen for the shadow ray, and it has some chance of hitting the light or not.

This amounts to choosing a random point on the light for any surface point being lit as is shown at the right of Figure 9.31.

If the light is a parallelogram specified by a corner point $\mathbf{c}$ and two edge vectors $\mathbf{a}$ and $\mathbf{b}$ (Figure 9.32), then choosing a random point $\mathbf{r}$ is straightforward:

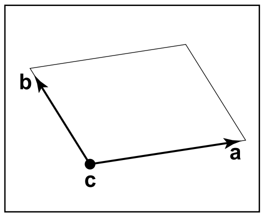


Figure 9.32. The geometry of a parallelogram light specified by a corner point and two edge vectors.

$$\mathbf{r} = \mathbf{c} + \xi_1\mathbf{a} + \xi_2\mathbf{b},$$

where ξ_1 and ξ_2 are uniform random numbers in the range $[0, 1)$.

We then send a shadow ray to this point as shown at the right in Figure 9.31. Note that the direction of this ray is not unit length, which may require some modification to your basic ray tracer depending upon its assumptions.

We would really like to jitter points on the light. However, it can be dangerous to implement this without some thought. We would not want to always have the ray in the upper left-hand corner of the pixel generate a shadow ray to the upper left-hand corner of the light. Instead we would like to scramble the samples, such that the pixel samples and the light samples are each themselves jittered, but so that there is no correlation between pixel samples and light samples. A good way to accomplish this is to generate two distinct sets of n^2 jittered samples and pass samples into the light source routine:

```
for each pixel $(i, j)$ do
  $c = 0$
  generate $N = n^2$ jittered 2D points and store in array $r[\,]$
  generate $N = n^2$ jittered 2D points and store in array $s[\,]$
  shuffle the points in array $s[\,]$
  for $p = 0$ to $N - 1$ do
    $c = c + \text{ray-color}(i + r[p].x(), j + r[p].y(), s[p])$
  $c_{ij} = c/N$
```

This shuffle routine eliminates any coherence between arrays r and s. The shadow routine will just use the 2D random point stored in $s[p]$ rather than calling the random number generator. A shuffle routine for an array indexed from 0 to $N-1$ is:

```
for $i = N - 1$ downto $1$ do
    choose random integer $j$ between $0$ and $i$ inclusive
    swap array elements $i$ and $j$
```

9.11.3 Depth of Field

The soft focus effects seen in most photos can be simulated by collecting light at a non-zero size "lens" rather than at a point. This is called *depth of field*. The lens collects light from a cone of directions that has its apex at a distance where everything is in focus (Figure 9.33). We can place the "window" we are sampling on the plane where everything is in focus (rather than at the $z = n$ plane as we did previously), and the lens at the eye. The distance to the plane where everything is in focus we call the *focus plane*, and the distance to it is set by the user, just as the distance to the focus plane in a real camera is set by the user or range finder.

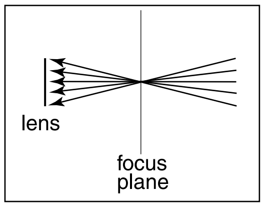


Figure 9.33. The lens averages over a cone of directions that hit the pixel location being sampled.

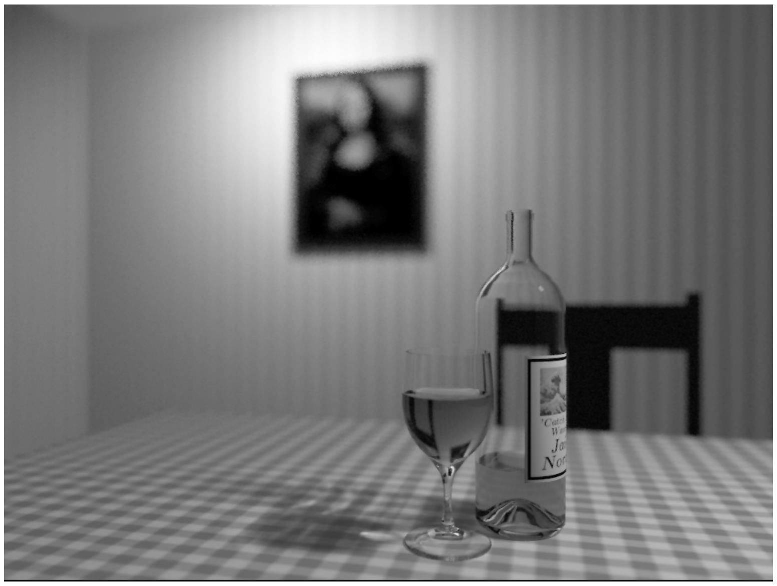


Figure 9.34. An example of depth of field. The caustic in the shadow of the wine glass is computed using particle tracing (Chapter **??**). (See also Plate PLATE.)

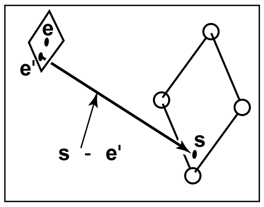


Figure 9.35. To create depth-of-field effects, the eye is randomly selected from a square region.

To be most faithful to a real camera, we should make the lens a disk. However, we will get very similar effects with a square lens (Figure 9.35). So we choose the side-length of the lens and take random samples on it. The origin of the view rays will be these perturbed positions rather than the eye position. Again, a shuffling routine is used to prevent correlation with the pixel sample positions. An example using 25 samples per pixel and a large disk lens is shown in Figure 9.34.

9.11.4 Glossy Reflection

Some surfaces, such as brushed metal, are somewhere between an ideal mirror and a diffuse surface. Some discernible image is visible in the reflection but it is blurred. We can simulate this by randomly perturbing ideal specular reflection rays as shown in Figure 9.36.

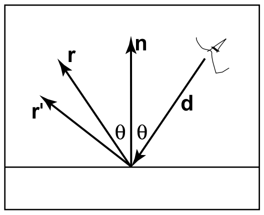


Figure 9.36. The reflection ray is perturbed to a random vector **r**'.

Only two details need to be worked out: how to choose the vector $\mathbf{r}'$, and what to do when the resulting perturbed ray is below the surface from which the ray is reflected. The latter detail is usually settled by returning a zero color when the ray is below the surface.

To choose $\mathbf{r}'$, we again sample a random square. This square is perpendicular to $\mathbf{r}$ and has width a which controls the degree of blur. We can set up the square's orientation by creating an orthonormal basis with $\mathbf{w} = \mathbf{r}$ using the techniques in Section 2.4.6. Then, we create a random point in the 2D square with side length a centered at the origin. If we have 2D sample points $(\xi, \xi') \in [0,1]^2$, then the analogous point on the desired square is

$$u = -\frac{a}{2} + \xi a,$$
$$v = -\frac{a}{2} + \xi' a.$$

Because the square over which we will perturb is parallel to both the $\mathbf{u}$ and $\mathbf{v}$ vectors, the ray $\mathbf{r}'$ is just

$$\mathbf{r}' = \mathbf{r} + u\mathbf{u} + v\mathbf{v}.$$

Note that $\mathbf{r}'$ is not necessarily a unit vector and should be normalized if your code requires that for ray directions.

9.11.5 Motion Blur

We can add a blurred appearance to objects as shown in Figure 9.37. This is called *motion blur* and is the result of the image being formed over a non-zero

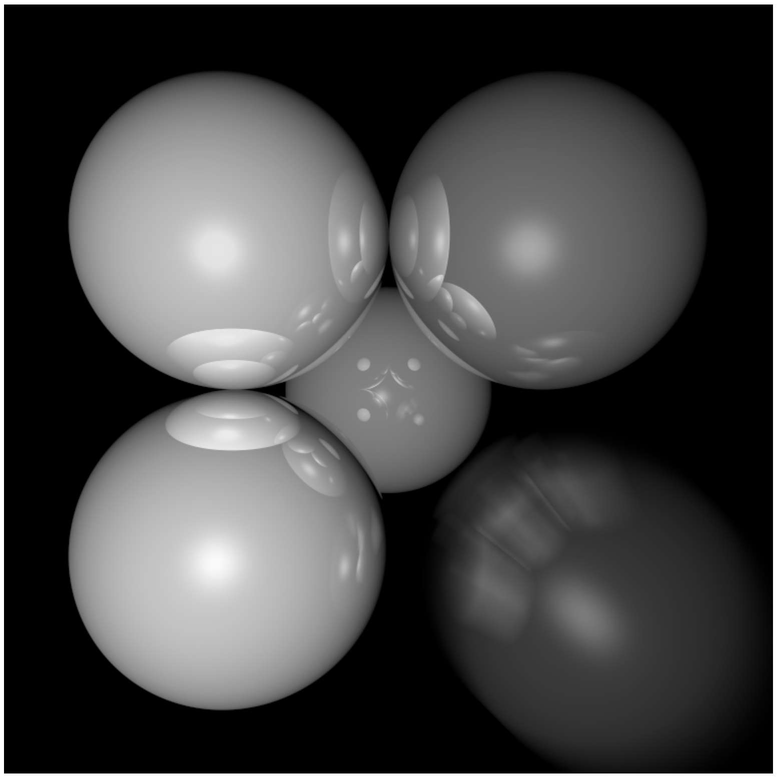


Figure 9.37. The bottom right sphere is in motion and a blurred appearance results. *Image courtesy Chad Barb.*

span of time. In a real camera, the aperture is open for some time interval during which objects move. We can simulate the open aperture by setting a time variable ranging from T_0 to T_1. For each viewing ray we choose a random time,

$$T = T_0 + \xi(T_1 - T_0).$$

We may also need to create some objects to move with time. For example, we might have a moving sphere whose center travels from $\mathbf{c}_0$ to $\mathbf{c}_1$ during the interval. Given T, we could compute the actual center and do a ray–intersection with that sphere. Because each ray is sent at a different time, each will encounter the sphere at a different position, and the final appearance will be blurred. Note that the bounding box for the moving sphere should bound its entire path so an efficiency structure can be built for the whole time interval (Glassner, 1988).

Frequently Asked Questions

- Why is there no perspective matrix in ray tracing?

The perspective matrix in a z-buffer exists so that we can turn the perspective projection into a parallel projection. This is not needed in ray tracing, because it is easy to do the perspective projection implicitly by fanning the rays out from the eye.

- What is the best ray-intersection efficiency structure?

The most popular structures are binary space partitioning trees (BSP trees), uniform subdivision grids, and bounding volume hierarchies. There is no clear-cut answer for which is best, but all are much, much better than brute-force search in practice. If I were to implement only one, it would be the bounding volume hierarchy because of its simplicity and robustness.

- Why do people use bounding boxes rather than spheres or ellipsoids?

Sometimes spheres or ellipsoids are better. However, many models have polygonal elements that are tightly bounded by boxes, but they would be difficult to tightly bind with an ellipsoid.

- Can ray tracing be made interactive?

For sufficiently small models and images, any modern PC is sufficiently powerful for ray tracing to be interactive. In practice, multiple CPUs with a shared frame buffer are required for a full-screen implementation. Computer power is increasing much faster than screen resolution, and it is just a matter of time before conventional PCs can ray trace complex scenes at screen resolution.

- Is ray tracing useful in a hardware graphics program?

Ray tracing is frequently used for *picking*. When the user clicks the mouse on a pixel in a 3D graphics program, the program needs to determine which object is visible within that pixel. Ray tracing is an ideal way to determine that.

Exercises

1. What are the ray parameters of the intersection points between ray $(1, 1, 1) + t(-1, -1, -1)$ and the sphere centered at the origin with radius 1? Note: this is a good debugging case.

2. What are the barycentric coordinates and ray parameter where the ray $(1, 1, 1) + t(-1, -1, -1)$ hits the triangle with vertices $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$? Note: this is a good debugging case.

3. Do a back of the envelope computation of the approximate time complexity of ray tracing on "nice" (non-adversarial) models. Split your analysis into the cases of preprocessing and computing the image, so that you can predict the behavior of ray tracing multiple frames for a static model.

13

Data Structures for Graphics

There are a variety of data structures that seem to pop up repeatedly in graphics applications. This chapter talks about three basic and unrelated data structures that are among the most common and useful. There are many variants of these data structures, but the basic ideas behind them can be conveyed using an example of each.

First the winged-edge data structure for storing tessellated geometric models is discussed (Baumgart, 1974). The winged-edge data structure is useful for managing models where the tessellation changes, such as in subdivision or simplification routines.

Next, the scene-graph data structure is presented. These are rapidly becoming well supported features of all new graphics APIs because they are so useful in managing objects and transformations.

Finally, the tiled multidimensional array is presented. Originally developed to help paging performance, such structures are now crucial for memory locality on machines regardless of whether the array fits in main memory.

12.1 Triangle Meshes

One of the most common model representations is a polygonal mesh as discussed in Section 10.3. When such meshes are unchanging in the program, the simple structure described in that section is usually sufficient. However, when the meshes are to be modified, more complicated data representations are needed to efficiently answer queries such as:

- given a triangle, what are the three adjacent triangles?
- given an edge, which two triangles share it?
- given a vertex, which faces share it?
- given a vertex, which edges share it?

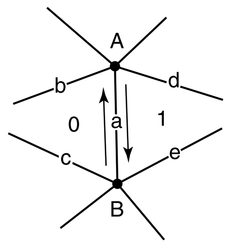
There are many data structures for triangle meshes, polygonal meshes, and polygonal meshes with holes (see the notes at the end of the chapter for references). In many applications the meshes are very large, so an efficient representation can be crucial.

The most straightforward, though bloated, implementation is to have three types: *vertex*, *edge*, and *triangle*. There are a variety of ways to divide the data among these types. While one might be tempted to just store all the relationships, this makes for variable-length data structures that really are not needed. For example, a vertex can have an arbitrarily large number of edges incident to it.

It is best, therefore, to hide the implementation behind a class interface.

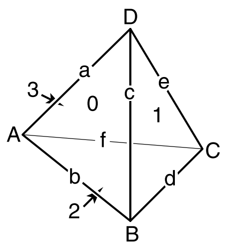
12.2 Winged-Edge Data Structure

We can use the class *winged-edge* data structure. This data structure makes edges the first-class citizen of the data structure. This data structure, a more efficient implementation, is illustrated in Figures 12.1 and 12.2.



edge	*vertex 1*	*vertex 2*	*face left*	*face right*	*pred left*	*succ left*	*pred right*	*succ right*
a	B	A	0	1	c	b	d	e

Figure 12.1. An edge in a winged-edge data structure. Stored with each edge are the face (polygon) to the left of the edge, the face to the right of the edge, and the previous and successor edges in the traversal of each of those faces.



edge	*vertex 1*	*vertex 2*	*face left*	*face right*	*pred left*	*succ left*	*pred right*	*succ right*
a	A	D	3	0	f	e	c	b
b	A	B	0	2	a	c	d	f
c	B	D	0	1	b	a	e	d
d	B	C	1	2	c	e	f	b
e	C	D	1	3	d	c	a	f
f	C	A	3	2	e	a	b	d

vertex	*edge*
A	a
B	d
C	d
D	e

face	*edge*
0	a
1	c
2	d
3	a

Figure 12.2. A tetrahedron and the associated elements for a winged-edge data structure. The two small tables are not unique; each vertex and face stores any one of the edges with which it is associated.

Note that the winged-edge data structure makes the desired queries in constant time. For example, a face can access one of its edges and follow the traversal pointers to find all of its edges. Those edges store the adjoining face.

As with any data structure, the winged-edge data structure makes a variety of time/space trade-offs. For example, we could eliminate the *prev* references. When we need to know the previous edge, we could follow the successor edges in a circle until we get back to the original edge. This would save space, but it would make the computation of the previous edge take longer. This type of issue has led to a proliferation of mesh data structures (see the chapter notes for more information on those structures).

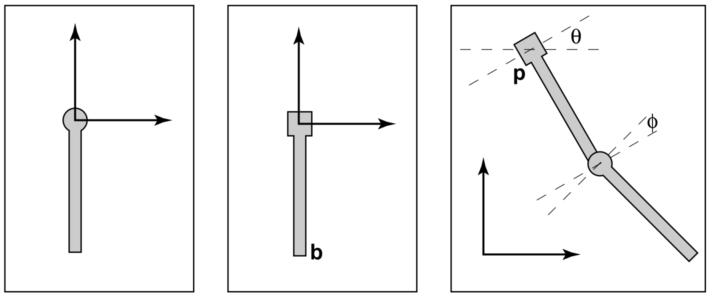


Figure 12.3. A hinged pendulum. On the left are the two pieces in their "local" coordinate systems. The hinge of the top piece is at point **b** and the attachment for the bottom piece is at its local origin. The degrees of freedom for the assembled object are the angles (θ,ϕ) and the location **p** of the top hinge.

12.3 Scene Graphs

To motivate the scene-graph data structure, we will use the hinged pendulum shown in Figure 12.3. Consider how we would draw the top part of the pendulum:

$\mathbf{M}_1 = \textit{rotate}(\theta)$
$\mathbf{M}_2 = \textit{translate}(\mathbf{p})$
$\mathbf{M}_3 = \mathbf{M}_2\mathbf{M}_1$
Apply $\mathbf{M}_3$ *to all points in upper pendulum*

The bottom is more complicated, but we can take advantage of the fact that it is attached to the bottom of the upper pendulum at point $\mathbf{b}$ in the local coordinate system. First, we rotate the lower pendulum so that it is at an angle ϕ relative to its initial position. Then, we move it so that its top hinge is at point $\mathbf{b}$. Now it is at the appropriate position in the local coordinates of the upper pendulum, and it can then be moved along with that coordinate system. The composite transform for the lower pendulum is:

$\mathbf{M}_a = \textit{rotate}(\phi)$
$\mathbf{M}_b = \textit{translate}(\mathbf{b})$
$\mathbf{M}_c = \mathbf{M}_b\mathbf{M}_a$
$\mathbf{M}_d = \mathbf{M}_3\mathbf{M}_c$
Apply $\mathbf{M}_d$ *to all points in lower pendulum*

Thus, we see that the lower pendulum not only lives in its own local coordinate system, but also that coordinate system itself is moved along with that of the upper pendulum.

We can encode the pendulum in a data structure that makes management of these coordinate system issues easier, as shown in Figure 12.4. The appropriate matrix to apply to an object is just the product of all the matrices in the chain from the object to the root of the data structure. For example, consider the model of a ferry that has a car that can move freely on the deck of the ferry, and wheels that each move relative to the car as shown in Figure 12.5.

As with the pendulum, each object should be transformed by the product of the matrices in the path from the root to the object:

ferry transform using M_0

car body transform using M_0M_1

left wheel transform using $M_0M_1M_2$

left wheel transform using $M_0M_1M_3$

An efficient implementation can be achieved using a *matrix stack*, a data structure supported by many APIs. A matrix stack is manipulated using *push* and *pop* operations that add and delete matrices from the right-hand side of a matrix product. For example, calling:

push$(\mathbf{M}_0)$
push$(\mathbf{M}_1)$
push$(\mathbf{M}_2)$

creates the active matrix $\mathbf{M} = \mathbf{M}_0\mathbf{M}_1\mathbf{M}_2$. A subsequent call to *pop()* strips the last matrix added so that the active matrix becomes: $\mathbf{M} = \mathbf{M}_0\mathbf{M}_1$. Combining the matrix stack with a recursive traversal of a scene graph gives us:

```
function traverse(node)
push($\mathbf{M}_{local}$)
draw object using composite matrix from stack
traverse(left child)
traverse(right child)
pop()
```

There are many variations on scene graphs but all follow the basic idea above.

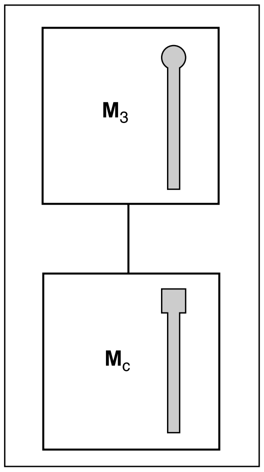


Figure 12.4. The scene graph for the hinged pendulum of Figure 12.3.

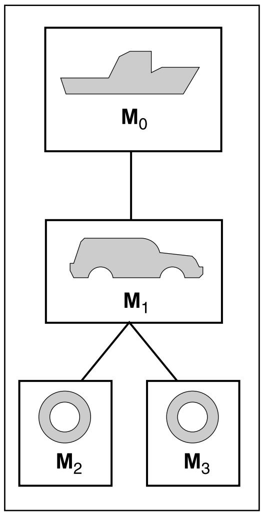


Figure 12.5. A ferry carries a car which has wheels attached (only two shown) are stored in a scene-graph.

12.4 Tiling Multidimensional Arrays

Effectively utilizing the cache hierarchy is a crucial task in designing algorithms for modern architectures. Making sure that multidimensional arrays have data in a "nice" arrangement is accomplished by *tiling*, sometimes also called *bricking*. A traditional 2D array is stored as a 1D array together with an indexing mechanism; for example, an N_x by N_y array is stored in a 1D array of length N_xN_y and the 2D index (x, y) (which runs from $(0, 0)$ to $(N_x - 1, N_y - 1)$) and maps it to the 1D index (running from 0 to $N_xN_y - 1$ using the formula

$$\text{index} = x + N_x y.$$

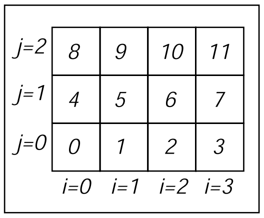


Figure 12.6. The memory layout for an untiled 2D array with $N_x = 4$ and $N_y = 3$.

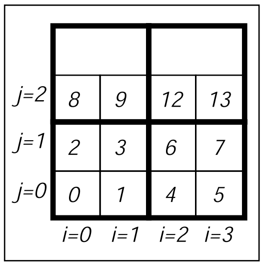


Figure 12.7. The memory layout for a tiled 2D array with $N_x = 4$ and $N_y = 3$ and two by two tiles. Note that padding on the top of the array is needed because N_y is not a multiple of the tile size two.

An example of how that memory lays out is shown in Figure 12.6. A problem with this layout is that although two adjacent array elements that are in the same row are next to each other in memory, two adjacent elements in the same column will be separated by N_x elements in memory. This can cause poor memory locality for large N_x. The standard solution to this is to use *tiles* to make memory locality for rows and columns more equal. An example is shown in Figure 12.7 where two by two tiles are used. The details of indexing such an array are discussed in the next section. A more complicated example with two levels of tiling on a 3D array are covered after that.

A key question is what size to make the tiles. In practice, they should be similar to the memory-unit size on the machine. For example, on a machine with 128-byte cache lines, and using 16-bit data values, n is exactly 8. However, using float (32-bit) datasets, n is closer to 5. Because there are also coarser-sized memory units such as pages, hierarchical tiling with similar logic can be useful.

12.4.1 One-Level Tiling for 2D Arrays

If we assume an N_x by N_y array decomposed into square n by n tiles (Figure 12.8), then the number of tiles required is

$$B_x = N_x/n,$$
$$B_y = N_y/n.$$

Here, we assume that n divides N_x and N_y exactly. When this is not true, the array should be *padded*. For example, if $N_x = 15$ and $n = 4$, then N_x should be changed to 16. To work out a formula for indexing such an array, we first find the tile indices (b_x, b_y) that give the row/column for the tiles (the tiles themselves form a 2D array):

$$b_x = x \div n,$$
$$b_y = y \div n,$$

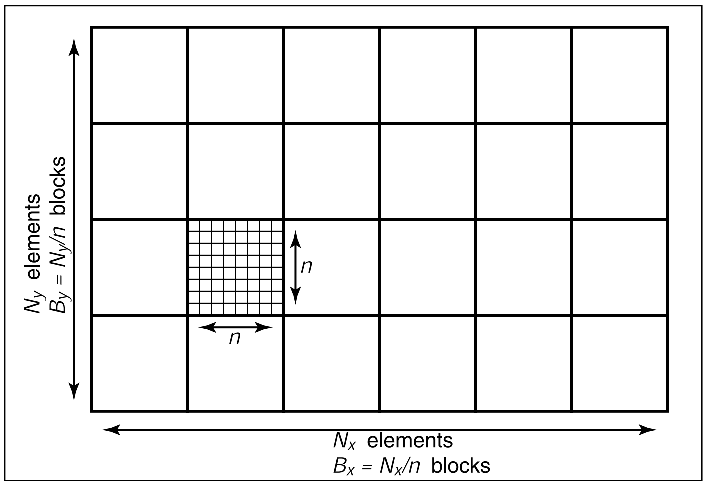


Figure 12.8. A tiled 2D array composed of B_x by B_y tiles each of size n by n.

where $\div$ is integer division, e.g., $12 \div 5 = 2$. If we order the tiles along rows as shown in Figure 12.6, then the index of the first element of the tile (b_x, b_y) is

$$\text{index} = n^2(B_x b_y + b_x).$$

The memory in that tile is arranged like a traditional 2D array as shown in Figure 12.7. The partial offsets (x', y') inside the tile are

$$x' = x \bmod n,$$
$$y' = y \bmod n,$$

where mod is the remainder operator, e.g., $12 \bmod 5 = 2$. Therefore, the offset inside the tile is

$$\text{offset} = y'n + x'.$$

Thus the full formula for finding the 1D index element (x, y) in an N_x by N_y array with n by n tiles is

$$\begin{aligned}\text{index} &= n^2(B_x b_y + b_x) + y'n + x',\\ &= n^2((N_x \div n)(y \div n) + x \div n) + (y \bmod n)n + (x \bmod n).\end{aligned}$$

This expression contains many integer multiplication, divide and modulus operations. On modern processors, these operations are extremely costly. For n that are powers of two, these operations can be converted to bitshifts and bitwise logical operations. However, as noted above, the ideal size is not always a power of two. Some of the multiplications can be converted to shift/add operations, but the divide and modulus operations are more problematic. The indices could be computed incrementally, but this would require tracking counters, with numerous comparisons and poor branch prediction performance.

However, there is a simple solution; note that the index expression can be written as

$$\text{index} = F_x(x) + F_y(y),$$

where

$$\begin{aligned} F_x(x) &= n^2(x \div n) + (x \bmod n), \\ F_y(y) &= n^2(N_x \div n)(y \div n) + (y \bmod n)n. \end{aligned}$$

We tabulate F_x and F_y, and use x and y to find the index into the data array. These tables will consist of N_x and N_y elements, respectively. The total size of the tables will fit in the primary data cache of the processor, even for very large data set sizes.

12.4.2 Example: Two-Level Tiling for 3D Arrays

Effective TLB utilization is also becoming a crucial factor in algorithm performance. The same technique can be used to improve TLB hit rates in a 3D array by creating $m \times m \times m$ bricks of $n \times n \times n$ cells. For example, a $40 \times 20 \times 19$ volume could be decomposed into $4 \times 2 \times 2$ macrobricks of $2 \times 2 \times 2$ bricks of $5 \times 5 \times 5$ cells. This corresponds to $m = 2$ and $n = 5$. Because 19 cannot be factored by $mn = 10$, one level of padding is needed. Empirically useful sizes are $m = 5$ for 16 bit datasets and $m = 6$ for float datasets.

The resulting index into the data array can be computed for any (x, y, z) triple with the expression

$$\begin{aligned} \text{index} \quad = \quad & ((x \div n) \div m)n^3m^3((N_z \div n) \div m)((N_y \div n) \div m) \\ & +((y \div n) \div m)n^3m^3((N_z \div n) \div m) \\ & +((z \div n) \div m)n^3m^3 \\ & +((x \div n) \bmod m)n^3m^2 \\ & +((y \div n) \bmod m)n^3m \\ & +((z \div n) \bmod m)n^3 \\ & +(x \bmod (n^2))n^2 \\ & +(y \bmod n)n \\ & +(z \bmod n), \end{aligned}$$

where N_x, N_y and N_z are the respective sizes of the dataset.

Note that, as in the simpler 2D one-level case, this expression can be written as

$$\text{index} = F_x(x) + F_y(y) + F_z(z),$$

where

$$\begin{aligned} F_x(x) &= ((x \div n) \div m)n^3m^3((N_z \div n) \div m)((N_y \div n) \div m) \\ &\quad +((x \div n) \bmod m)n^3m^2 \\ &\quad +(x \bmod n)n^2, \\ F_y(y) &= ((y \div n) \div m)n^3m^3((N_z \div n) \div m) \\ &\quad +((y \div n) \bmod m)n^3m + \\ &\quad +(y \bmod n)n, \\ F_z(z) &= ((z \div n) \div m)n^3m^3 \\ &\quad +((z \div n) \bmod m)n^3 \\ &\quad +(z \bmod n). \end{aligned}$$

Frequently Asked Questions

• Does tiling really make that much difference in performance?

On some volume rendering applications, a two-level tiling strategy made as much as a factor-of-ten performance difference. When the array does not fit in main memory, it can effectively prevent thrashing in some applications such as image editing.

• How do I store the lists in a winged-edge structure?

For most applications it is feasible to use arrays and indices for the references. However, if many delete operations are to be performed, then it is wise to use linked lists and pointers.

Notes

The discussion of the winged-edge data structure is based on the course notes of *Ching-Kuang Shene*. There are smaller mesh data structures than winged-edge. The trade-offs in using such structures is discussed in *Directed Edges—A Scalable Representation for Triangle Meshes* (Campagna, Kobbelt, & Seidel,

1998) The tiled-array discussion is based on *Interactive Ray Tracing for Volume Visualization* (Parker et al., 1999).

Exercises

1. What is the memory difference for a simple tetrahedron stored as four independent triangles and one stored in a winged-edge data structure?
2. Diagram a scene graph for a bicycle.
3. How many look-up tables are needed for a single-level tiling of an n-dimensional array?

References

Amanatides, J., & Woo, A. (1987). A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics* (pp. 1–10).

Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of the AFIPS spring joint computing conference* (p. 37-49).

Association, I. S. (1985). *IEEE standard for binary floating-point arithmetic.* IEEE Report (New York). (ANSI/IEEE Std 754-1985)

Baumgart, B. (1974, October). *Geometric modeling for computer vision* (Tech. Rep. No. AIM-249). Seattle, WA: Stanford University AI Laboratory.

Bayer, B. E. (1976). *Color imaging array.* (U.S. Patent 3,971,065)

Beck, K., & Andres, C. (2004). *Extreme programming explained : Embrace change* (Second ed.). Reading, MA: Addison-Wesley.

Blinn, J. (1978). Simulation of wrinkled surfaces. In *Proceedings of SIGGRAPH* (pp. 286–292).

Blinn, J. (1996). *Jim blinn's corner*. San Francisco, CA: Morgan Kaufmann.

Blinn, J., & Newell, M. (1978). Clipping using homogeneous coordinates. In *Proceedings of SIGGRAPH* (pp. 245–251).

Blinn, J. F. (1976). Texture and reflection in computer generated images. *Communications of the ACM*, *19*(10), 542-547.

Bresenham, J. E. (1965). Algorithm for computer control of a digital plotter. *IBM Systems Journal*, *4*(1), 25–30.

Campagna, S., Kobbelt, L., & Seidel, H.-P. (1998). Directed edges–a scalable representation for triangle meshes. *Journal of Graphics Tools*, *3*(4), 1–12.

Catmull, E. (1975). Computer display of curved surfaces. In *IEEE conference on computer graphics, pattern recognition and data structures* (pp. 11–17).

Cleary, J., Wyvill, B., Birtwistle, G., & Vatti, R. (1983). A Parallel Ray Tracing Computer. In *Proceedings of the association of simula users conference* (p. 77-80).

Cook, R. L., Carpenter, L., & Catmull, E. (1987). The reyes image rendering architecture. In *Proceedings of SIGGRAPH* (pp. 95–102).

Cook, R. L., Porter, T., & Carpenter, L. (1984). Distributed ray tracing. In *Proceedings of SIGGRAPH* (p. 165-174).

Crow, F. C. (1978). The use of grey scale for improved raster display of vectors and characters. In *Proceedings of SIGGRAPH* (pp. 1–5).

Crowe, M. J. (1994). *A history of vector analysis*. Mineola, NY: Dover.

Cyrus, M., & Beck, J. (1978). Generalized two- and three-dimensional clipping. *Computers and Graphics*, *3*(1), 23–28.

DeRose, T. (1989, September). *A coordinate-free approach to geometric programming* (Tech. Rep. No. 89-09-16). Seattle, WA: Universit of Washington.

Dobkin, D. P., & Mitchell, D. P. (1993). Random–edge discrepancy of supersampling patterns. In *Proceedings of Graphics Interface* (pp. 62–69).

Dooley, D., & Cohen, M. (1990). Automatic illustration of 3d geometric models: Lines. In *Symposium on interactive 3D graphics* (pp. 77–82).

Doran, C., & Lasenby, A. (2003). *Geometric algebra for physicists*. Cambridge: Cambridge University Press.

Eberly, D. (2000). *Game engine design*. San Francisco, CA: Morgan Kaufmann.

Farin, G., & Hansford, D. (2004). *Practical linear algebra: A geometry toolbox*. Wellesley, MA: AK Peters.

Fuchs, H., Kedem, Z. M., & Naylor, B. F. (1980). On visible surface generation by a priori tree structures. In *Proceedings of SIGGRAPH* (pp. 124–133).

Fujimoto, A., Tanaka, T., & Iwata, K. (1986, April). Arts: Accelerated ray-tracing system. *IEEE Computer Graphics & Applications*, 16–26.

Glassner, A. (1984). Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, *4*(10), 15–22.

Glassner, A. (1988). Spacetime ray tracing for animation. *IEEE Computer Graphics & Applications*, *8*(2), 60–70.

Glassner, A. (Ed.). (1989). *An introduction to ray tracing*. London: Academic Press.

Goldman, R. (1985). Illicit expressions in vector algebra. *ACM Transactions on Graphics*, *4*(3), 223–243.

Goldsmith, J., & Salmon, J. (1987, May). Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics & Applications*, 14–20.

Gooch, A., Gooch, B., Shirley, P., & Cohen, E. (1998). A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of SIGGRAPH* (pp. 447–452).

Gouraud, H. (1971). Continuous shading of curved surfaces. *Communications of the ACM*, *18*(6), 623-629.

Hammersley, J., & Handscomb, D. (1964). *Monte-carlo methods*. Methuen, London.

Hanson, A. J. (2005). *Visualizing quaternions*. San Francisco, CA: Morgan Kaufmann.

Hausner, M. (1998). *A vector space approach to geometry*. Mineola, NY: Dover.

Havran, V. (2000). *Heuristic ray shooting algorithms*. Unpublished doctoral dissertation, Czech Technical University in Prague.

Hearn, D., & Baker, M. P. (1986). *Computer graphics*. Englewood Cliffs, N.J.: Prentice-Hall.

Heidrich, W., & Seidel, H.-P. (1998). Ray-tracing procedural displacement shaders. In *Graphics interface* (pp. 8–16).

Hoffmann, B. (1975). *About vectors*. Mineola, NY: Dover.

Hughes, J. F., & Möller, T. (1999). Building an orthonormal basis from a unit vector. *Journal of Graphics Tools*, *4*(4), 33–35.

Jansen, F. W. (1986). Data structures for ray tracing. In *Proceedings of the workshop on data structures for raster graphics* (p. 57-73).

Kalos, M., & Whitlock, P. (1986). *Monte carlo methods, basics*. Wiley-Interscience.

Kay, D. S., & Greenberg, D. P. (1979). Transparency for computer synthesized images. In *Proceedings of SIGGRAPH* (pp. 158–164).

Kernighan, B. W., & Pike, R. (1999). *The practice of programming*. Reading, MA: Addison-Wesley.

Kindlmann, G., Reinhard, E., & Creem, S. (2002). Face-based luminance matching for perceptual colormap generation. In *Proceedings of Visualization* (pp. 299–306).

Kirk, D., & Arvo, J. (1988). The ray tracing kernel. In *Proceedings of Ausgraph.*

Kollig, T., & Keller, A. (2002). Efficient multidimensional sampling. *Computer Graphics Forum*, *21*(3), 557–564.

Lakos, J. (1996). *Large-scale C++ software design*. Reading, MA: Addison-Wesley.

Liang, Y.-D., & Barsky, B. A. (1984). A new concept and method for line clipping. *ACM Transactions on Graphics*, *3*(1), 1–22.

Meyers, S. (1995). *More effective C++: 35 new ways to improve your programs and designs*. Reading, MA: Addison-Wesley.

Meyers, S. (1997). *Effective C++: 50 specific ways to improve your programs and designs* (Second ed.). Reading, MA: Addison-Wesley.

Mitchell, D. P. (1996). Consequences of stratified sampling in graphics. In *Proceedings of SIGGRAPH* (pp. 277–280).

Möller, T., & Haines, E. (1999). *Real-time rendering*. Wellesley, MA: AK Peters.

Möller, T., & Haines, E. (2002). *Real-time rendering* (Second ed.). Wellesley, MA: AK Peters.

Möller, T., & Hughes, J. (1999). Efficiently building a matrix to rotate one vector to another. *Journal of Graphics Tools*, *4*(4), 1–4.

Muuss, M. J. (1995). Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD symposium.*

Paeth, A. W. (1990). A fast algorithm for general raster rotation. In *Graphics gems* (pp. 179–195).

Parker, S., Martin, W., Sloan, P., Shirley, P., Smits, B., & Hansen, C. (1999). Interactive ray tracing. In *ACM symposium on interactive 3D graphics* (pp. 119–126).

Patterson, J., Hoggar, S., & Logie, J. (1991). Inverse displacement mapping. *Computer Graphics Forum*, *10*(2), 129–139.

Peachey, D. (1985). Solid texturing of complex surfaces. In *Proceedings of SIGGRAPH* (pp. 279–286).

Penna, M., & Patterson, R. (1986). *Projective geometry and its applications to computer graphics*. Englewood Cliffs, NJ: Prentice Hall.

Perlin, K. (1985). An image synthesizer. In *Proceedings of SIGGRAPH* (pp. 287–296).

Pharr, M., & Hanrahan, P. (1996). Geometry caching for ray-tracing displacement maps. In *Eurographics rendering workshop* (pp. 31–40).

Pharr, M., Kolb, C., Gershbein, R., & Hanrahan, P. (1997). Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of SIGGRAPH* (pp. 101–108).

Phong, B.-T. (1975). Illumination for computer generated images. *Communications of the ACM*, *18*(6), 311-317.

Pineda, J. (1988). A parallel algorithm for polygon rasterization. In *Proceedings of SIGGRAPH* (pp. 17–20).

Pitteway, M. L. V. (1967). Algorithm for drawing ellipses or hyperbolae with a digital plotter. *Computer Journal*, *10*(3), 282–289.

Plauger, P. J. (1991). *The standard C library*. Englewood Cliffs, NJ: Prentice Hall.

Porter, T., & Duff, T. (1984). Compositing digital images. In *Proceddings of SIGGRAPH* (p. 253-259).

Riesenfeld, R. F. (1981, January). Homogeneous coordinates and projective planes in computer graphics. *IEEE Computer Graphics & Applications*, *1*, 50–55.

Roberts, L. (1965, May). *Homogenous matrix representation and manipulation of n-dimensional constructs* (Tech. Rep. No. MS-1505). Lexington, MA: MIT Lincoln Laboratory.

Roth, S. (1982). Ray casting for modelling solids. *Computer Graphics and Image Processing*, *18*(2), 109–144.

Rubin, S., & Whitted, J. T. (1980). A 3-dimensional representation for fast rendering of complex scenes. In *Proceedings of SIGGRAPH* (pp. 110–116).

Saito, T., & Takahashi, T. (1990). Comprehensible rendering of 3-d shapes. In *Proceedings of SIGGRAPH* (pp. 197–206).

Salomon, D. (1999). *Computer graphics and geometric modeling*. New York, NY: Springer Verlag.

Segal, M., Korobkin, C., Widenfelt, R. van, Foran, J., & Haeberli, P. E. (1992). Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH* (pp. 249–252).

Smits, B., Shirley, P., & Stark, M. M. (2000). Direct ray tracing of displacement mapped triangles. In *Eurographics workshop on rendering* (pp. 307–318).

Snyder, J., & Barr, A. (1987). Ray tracing complex models containing surface tessellations. In *Proceedings of SIGGRAPH* (pp. 119–128).

Sobel, I., Stone, J., & Messer, R. (1975). *The monte carlo method*. Chicago, IL: University of Chicago Press.

Solomon, H. (1978). *Geometric probability*. Philadelphia, PA: SIAM Press.

Strang, G. (1988). *Linear algebra and its applications* (third ed.). Florence, KY: Brooks Cole.

Sutherland, I. E., Sproull, R. F., & Schumacker, R. A. (1974). A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, *6*(1), 1–55.

Turkowski, K. (1990). Properties of surface-normal transformations. In *Graphics gems* (pp. 539–547).

Van Aken, J., & Novak, M. (1985). Curve-drawing algorithms for raster displays. *ACM Transactions on Graphics*, *4*(2), 147–169.

Veach, E., & Guibas, L. J. (1997). Metropolis light transport. In *Proceedings of SIGGRAPH* (pp. 65–76).

Wald, I., Slusallek, P., & Benthin, C. (2001). Interactive distributed ray tracing of highly complex models. In *Proceedings of the Eurographics workshop on rendering* (pp. 277–288).

Warn, D. R. (1983). Lighting controls for synthetic images. In *Proceedings of SIGGRAPH* (pp. 13–21).

Whitted, T. (1980). An improved illumination model for shaded display. *Communications of the ACM*, *23*(6), 343–349.

Williams, A., Barrus, S., Morley, R. K., & Shirley, P. (2005). An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools*, *10*(1).

Williams, L. (1983). Pyramidal parametrics. In *Proceedings of SIGGRAPH* (pp. 1–11).

Williams, L. (1991). Shading in two dimensions. In *Proceedings of Graphics Interface* (pp. 143–151).

Woo, M., Neider, J., Davis, T., & Shreiner, D. (1999). *OpenGL programming guide* (Third ed.). Reading, MA: Addison-Wesley.

Yessios, C. I. (1979). Computer drafting of stones, wood, plant and ground materials. In *Proceedings of SIGGRAPH)* (pp. 190–198).

Graphics and Image Processing — J.D. Foley, Editor

An Improved Illumination Model for Shaded Display

Turner Whitted
Bell Laboratories
Holmdel, New Jersey

To accurately render a two-dimensional image of a three-dimensional scene, global illumination information that affects the intensity of each pixel of the image must be known at the time the intensity is calculated. In a simplified form, this information is stored in a tree of "rays" extending from the viewer to the first surface encountered and from there to other surfaces and to the light sources. A visible surface algorithm creates this tree for each pixel of the display and passes it to the shader. The shader then traverses the tree to determine the intensity of the light received by the viewer. Consideration of all of these factors allows the shader to accurately simulate true reflection, shadows, and refraction, as well as the effects simulated by conventional shaders. Anti-aliasing is included as an integral part of the visibility calculations. Surfaces displayed include curved as well as polygonal surfaces.

Key Words and Phrases: computer graphics, computer animation, visible surface algorithms, shading, raster displays

CR Category: 8.2

Introduction

Since its beginnings, shaded computer graphics has progressed toward greater realism. Even the earliest visible surface algorithms included shaders that simulated such effects as specular reflection [19], shadows [1, 7], and transparency [18]. The importance of illumination models is most vividly demonstrated by the realism produced with newly developed techniques [2, 4, 5, 16, 20].

Author's address: Bell Laboratories, Holmdel, NJ 07733.

The role of the illumination model is to determine how much light is reflected to the viewer from a visible point on a surface as a function of light source direction and strength, viewer position, surface orientation, and surface properties. The shading calculations can be performed on three scales: microscopic, local, and global. Although the exact nature of reflection from surfaces is best explained in terms of microscopic interactions between light rays and the surface [3], most shaders produce excellent results using aggregate local surface data. Unfortunately, these models are usually limited in scope, i.e., they look only at light source and surface orientations, while ignoring the overall setting in which the surface is placed. The reason that shaders tend to operate on local data is that traditional visible surface algorithms cannot provide the necessary global data.

A shading model is presented here that uses global information to calculate intensities. Then, to support this shader, extensions to a ray tracing visible surface algorithm are presented.

1. Conventional Models

The simplest visible surface algorithms use shaders based on Lambert's cosine law. The intensity of the reflected light is proportional to the dot product of the surface normal and the light source direction, simulating a perfect diffuser and yielding a reasonable looking approximation to a dull, matte surface. A more sophisticated model is the one devised by Bui-Tuong Phong [8]. Intensity from Phong's model is given by

$$I = I_a + k_d \sum_{j=1}^{j=ls} (\bar{N} \cdot \bar{L}_j) + k_s \sum_{j=1}^{j=ls} (\bar{N} \cdot \bar{L}_j')^n, \tag{1}$$

where

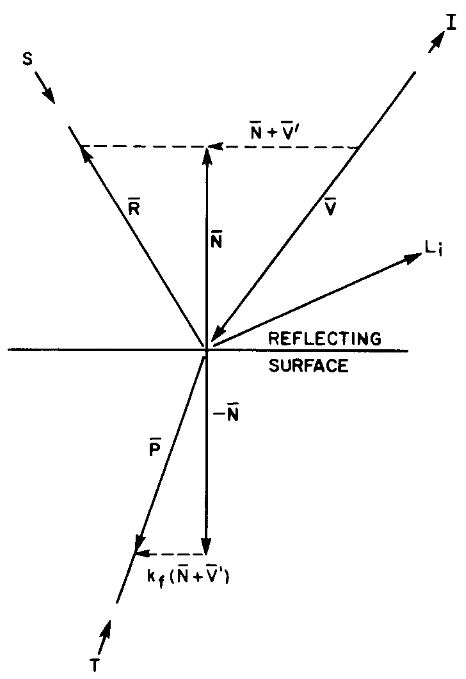
I	=	the reflected intensity,
I_a	=	reflection due to ambient light,
k_d	=	diffuse reflection constant,
$\bar{N}$	=	unit surface normal,
$\bar{L}_j$	=	the vector in the direction of the jth light source,
k_s	=	the specular reflection coefficient,
$\bar{L}_j'$	=	the vector in the direction halfway between the viewer and the jth light source,
n	=	an exponent that depends on the glossiness of the surface.

Phong's model assumes that each light source is located at a point infinitely distant from the objects in the scene. The model does not account for objects within a scene acting as light sources or for light reflected from object to object. As noted in [6], this drawback does not affect the realism of diffuse reflection components very much, but it seriously hurts the quality of specular reflections. A method developed by Blinn and Newell [5] partially solves the problem by modeling an object's environment and mapping it onto a sphere of infinite radius. The technique yields some of the most realistic computer generated pictures ever made, but its limitations preclude its use in the general case.

In addition to the specular reflection, the simulation of shadows is one of the more desirable features of an illumination model. A point on a surface lies in shadow if it is visible to the viewer but not visible to the light source. Some methods [2, 20] invoke the visible surface algorithm twice, once for the light source and once for the viewer. Others [1, 7, 12] use a simplified calculation to determine whether the point is visible to the light source.

Transmission of light through transparent objects has been simulated in algorithms that paint surfaces in reverse depth order [18]. When painting a transparent surface, the background is partially overwritten, allowing previously painted portions of the image to show through. While the technique has produced some impressive pictures, it does not simulate refraction. Kay [17] has improved on this approach with a technique that yields a very realistic approximation to the effects of refraction.

Fig. 1.



2. Improved Model

A simple model for reflection of light from perfectly smooth surfaces is provided by classical ray optics. As shown in Figure 1, the light intensity, I, passed to the viewer from a point on the surface consists primarily of the specular reflection, S, and transmission, T, components. These intensities represent light propagated along the $\bar{V}$, $\bar{R}$, and $\bar{P}$ directions, respectively. Since surfaces displayed are not always perfectly glossy, a term must be added to model the diffuse component as well. Ideally the diffuse reflection should contain components due to reflection of nearby objects as well as predefined light sources, but the computation required to model a distributed light source is overwhelming. Instead, the diffuse term from (1) is retained in the new model. Then the new model is

$$I = I_a + k_d \sum_{j=1}^{j=ls} (\bar{N}\cdot\bar{L}_j) + k_s S + k_t T, \qquad (2)$$

where

S = the intensity of light incident from the $\bar{R}$ direction,
k_t = the transmission coefficient,
T = the intensity of light from the $\bar{P}$ direction.

The coefficients k_s and k_t are held constant for the model used to make pictures in this report, but for the best accuracy they should be functions that incorporate an approximation of the Fresnel reflection law (i.e., the coefficients should vary as a function of incidence angle in a manner that depends on the material's surface properties). In addition, these coefficients must be carefully chosen to correspond to physically reasonable values if realistic pictures are to be generated. The $\bar{R}$ direction is determined by the simple rule that the angle of reflection must equal the angle of incidence. Similarly, the $\bar{P}$ direction of transmitted light must obey Snell's law. Then, $\bar{R}$ and $\bar{P}$ are functions of $\bar{N}$ and $\bar{V}$ given by

$$\bar{V}' = \frac{\bar{V}}{|\bar{V}\cdot\bar{N}|},$$
$$\bar{R} = \bar{V}' + 2\bar{N},$$
$$\bar{P} = k_f(\bar{N} + \bar{V}') - \bar{N},$$

where

$$k_f = (k_n^2|\bar{V}'|^2 - |\bar{V}' + \bar{N}|^2)^{-1/2},$$

and

k_n = the index of refraction.

Since these equations assume that $\bar{V}\cdot\bar{N}$ is less than zero, the intersection processor must adjust the sign of $\bar{N}$ so that it points to the side of the surface from which the intersecting ray is incident. It must likewise adjust the index of refraction to account for the sign change. If the denominator of the expression for k_f is imaginary, T is assumed to be zero because of total internal reflection.

By making k_s smaller and k_d larger, the surface can be made to look less glossy. However, the simple model will not spread the specular term as Phong's model does by reducing the specular exponent n. As pointed out in [3], the specular reflection from a roughened surface is produced by microscopic mirrorlike facets. The intensity of the specular reflection is proportional to the number of these microscopic facets whose normal vector is aligned with the mean surface normal value at the region being sampled. To generate the proper looking specular reflection, a random perturbation is added to the surface normal to simulate the randomly oriented microfacets.

Fig. 2.

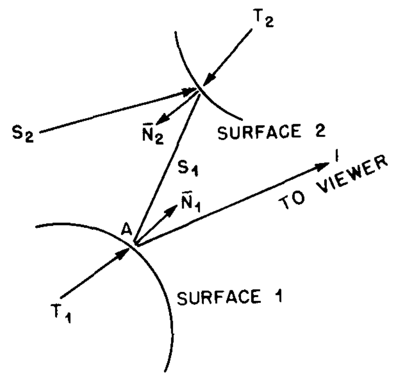
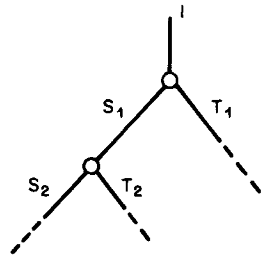


Fig. 3.



(A similar normal perturbation technique is used by Blinn [4] to model texture on curved surfaces.) For a glossy surface, this perturbation has a small variance; with greater variances the surface will begin to look less glossy. This same perturbation will cause a transparent object to look progressively more frosted as the variance is increased. While providing a good model for microscopic surface roughness, this scheme relies on sampled surface normals and will show the effects of aliasing for larger variances. Since this scheme also requires entirely too much additional computing, it is avoided whenever possible. For instance, in the case of specular reflections caused directly by a point light source, Phong's model is used at the point of reflection instead of the perturbation scheme.

The simple model approximates the reflection from a single surface. In a scene of even moderate complexity light will often be reflected from several surfaces before reaching the viewer. For one such case, shown in Figure 2, the components of the light reaching the viewer from point A are represented by the tree in Figure 3. Creating this tree requires calculating the point of intersection of each component ray with the surfaces in the scene. The calculations require that the visible surface algorithm (described in the next section) be called recursively until all branches of the tree are terminated. For the case of surfaces aligned in such a way that a branch of the tree has infinite depth, the branch is truncated at the point where it exceeds the allotted storage. Degradation of the image from this truncation is not noticeable.

In addition to rays in the $\bar{R}$ and $\bar{P}$ direction, rays corresponding to the $\bar{L}_j$ terms in (2) are associated with each node. If one of these rays intersects some surface in the scene before it reaches the light source, the point of intersection represented by the node lies in shadow with respect to that light source. That light source's contribution to the diffuse reflection from the point is then attenuated.

After the tree is created, the shader traverses the tree, applying eq. (2) at each node to calculate intensity. The intensity at each node is then attenuated by a linear function of the distance between intersection points on the ray represented by the node's parent before it is used as an input to the intensity calculation of the parent. (Since one cannot always assume that all the surfaces are planar and all the light sources are point sources, square-law attenuation is not always appropriate. Instead of modeling each unique situation, linear attenuation with distance is used as an approximation.)

3. Visible Surface Processor

Since illumination returned to the viewer is determined by a tree of "rays," a ray tracing algorithm is ideally suited to this model. In an obvious approach to ray tracing, light rays emanating from a source are traced through their paths until they strike the viewer. Since only a few will reach the viewer, this approach is wasteful. In a second approach suggested by Appel [1] and used successfully by MAGI [14], rays are traced in the opposite direction—from the viewer to the objects in the scene, as illustrated in Figure 4.

Unlike previous ray tracing algorithms, the visibility calculations do not end when the nearest intersection of a ray with objects in the scene is found. Instead, each visible intersection of a ray with a surface produces more rays in the $\bar{R}$ direction, the $\bar{P}$ direction, and in the direction of each light source. The intersection process is repeated for each ray until none of the new rays intersects any object.

Because of the nature of the illumination model, some traditional notions must be discarded. Since objects may be visible to the viewer through reflections in other objects, even though some other object lies between it and the viewer, the measure of visible complexity in an image is larger than for a conventionally generated image of the same scene. For the same reason, clipping and eliminating backfacing surface elements are not applicable with this algorithm. Because these normal preprocessor stages that simplify most visible surface algorithms cannot be used, a different approach is taken. Using a technique similar to one described by Clark [11], the object description includes a bounding volume for each item in the scene. If a ray does not intersect the bounding volume of an object, then the object can be eliminated from further processing for that ray. For simplicity of representation and ease of performing the intersection calculation, spheres are used as the bounding volumes.

Since a sphere can serve as its own bounding volume, initial experiments with the shading processor used spheres as test objects. For nonspherical objects, additional intersection processors must be specified whenever a ray does intersect the bounding sphere for that object. For polygonal surfaces the algorithm solves for the point of intersection of the ray and the plane of the polygon and then checks to see if the point is on the interior of the polygon. If the surface consists of bicubic patches, bounding spheres are generated for each patch. If the bounding sphere is pierced by the ray, then the patch is subdivided using a method described by Catmull and Clark [10], and bounding spheres are produced for each subpatch. The subdivision process is repeated until either no bounding spheres are intersected (i.e., the patch is not intersected by the ray) or the intersected bounding sphere is smaller than a predetermined minimum. This scheme was selected for simplicity rather than efficiency.

The visible surface algorithm also contains the mechanism to perform anti-aliasing. Since aliasing is the result of undersampling during the display process, the most straightforward cure is to low-pass filter the entire image before sampling for display [13]. A considerable amount of computing can be saved, however, if a more economical approach is taken. Aliasing in computer generated images is most apparent to the viewer in three cases: (1) at regions of abrupt change in intensity such as the silhouette of a surface, (2) at locations where small objects fall between sampling points and disappear, and (3) whenever a sampled function (such as texture) is mapped onto the surface. The visible surface algorithm looks for these cases and performs the filtering function only in these regions.

For this visible surface algorithm a pixel is defined in the manner described in [9] as the rectangular region whose corners are four sample points as shown in Figure 5(a). If the intensities calculated at the four points have nearly equal values and no small object lies in the region between them, the algorithm assumes that the average of the four values is a good approximation of the intensity over the entire region. If the intensity values are not nearly equal (Figure 5(b)), the algorithm subdivides the sample square and starts over again. This process runs recursively until the computer runs out of resolution or until an adequate amount of information about the detail within the sample square is recovered. The contribution of each single subregion is weighted by its area, and all such weighted intensities are summed to determine the intensity of the pixel. This approach amounts to performing a Warnock-type visibility process for each pixel [19]. In the limit it is equivalent to area sampling, yet it remains a point sampling technique. A better method, currently being investigated, considers volumes defined by each set of four corner rays and applies a containment test for each volume.

To ensure that small objects are not lost, a minimum radius (based on distance from the viewer) is allowed for bounding spheres of objects. This minimum is chosen so

Fig. 4.

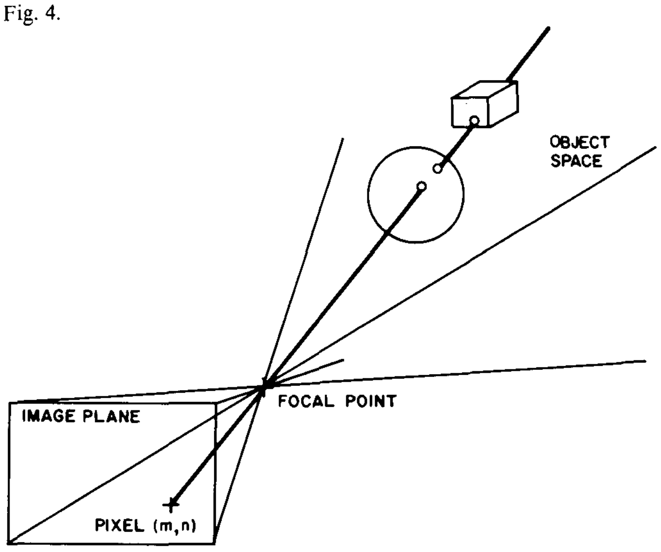
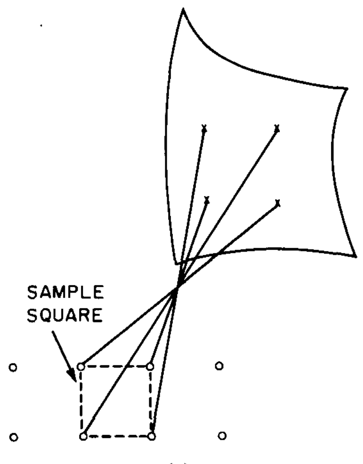
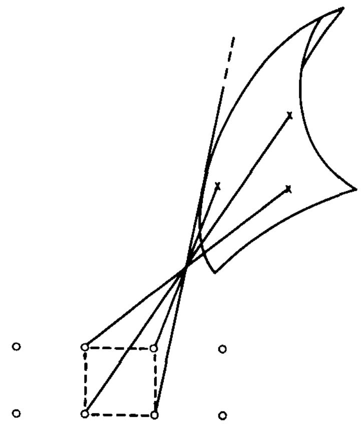


Fig. 5.



(a)



(b)

that no matter how small the object, its bounding sphere will always be intersected by at least one ray. If a ray passes within a minimum radius of a bounding sphere but does not intersect the object, the algorithm will know to subdivide each of the four sample squares that share the ray until the missing object is found. Although

Fig. 6.

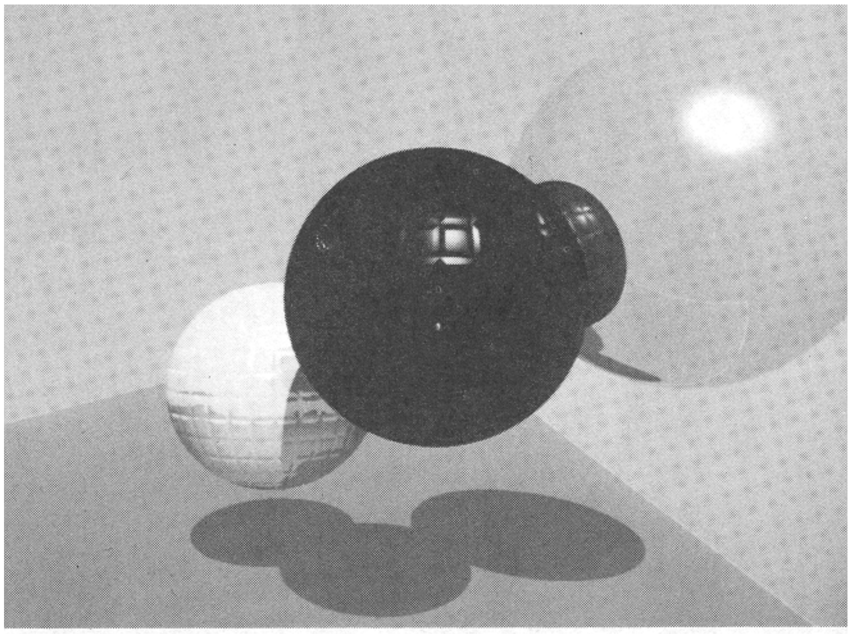


Fig. 7.

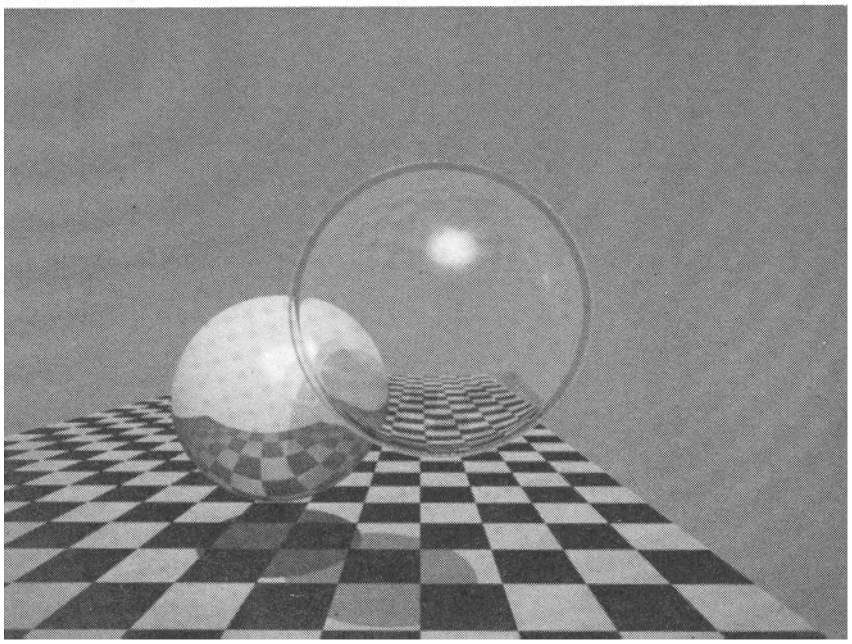


Fig. 8.

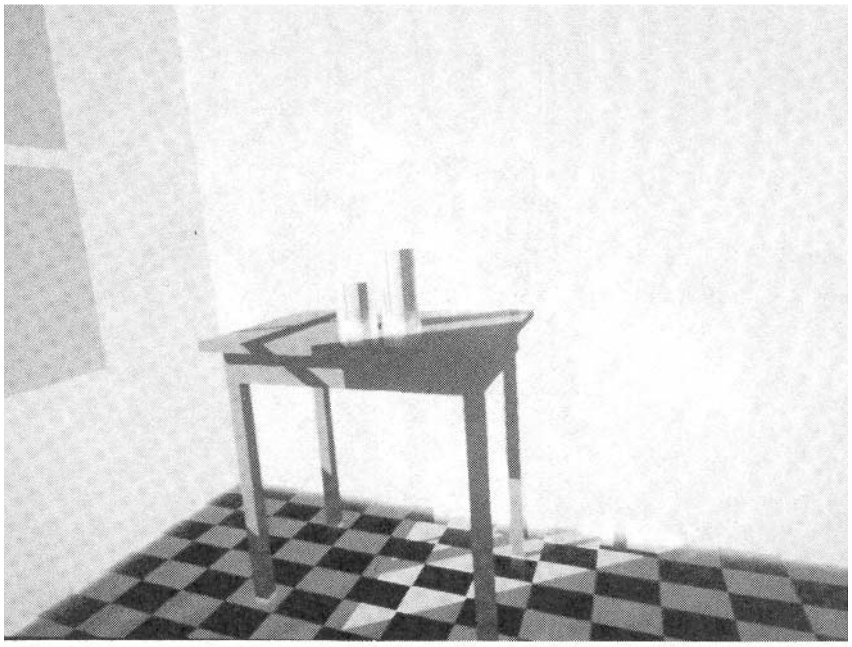
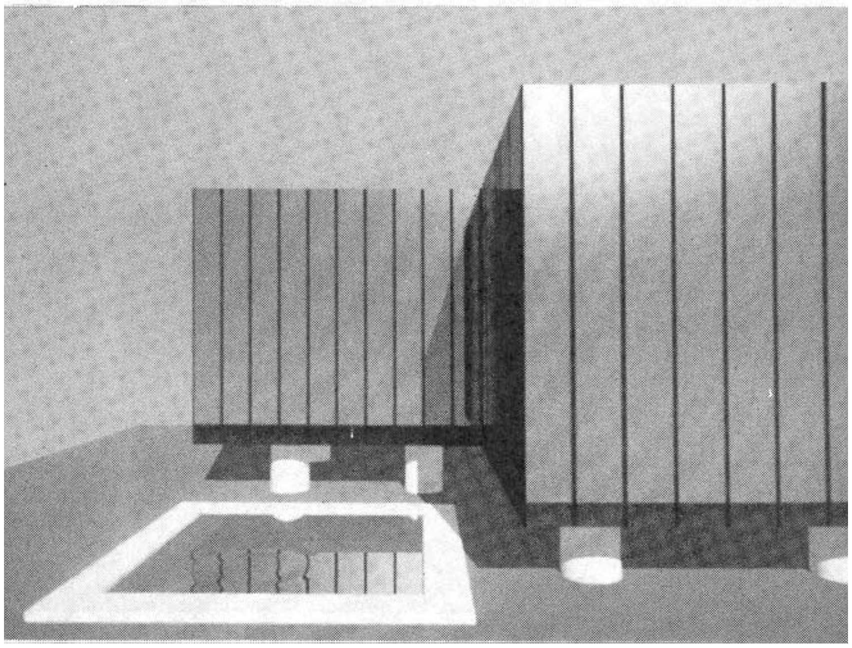


Fig. 9.



adequate for rays that reach the viewer directly, this scheme will not always work for rays being reflected from curved surfaces.

4. Results

A version of this algorithm has been programmed in C, running under UNIX[1] on both a PDP-11/45 and a VAX-11/780. To simplify the programming, all calculations are performed in floating point (at a considerable speed penalty). The pictures are displayed at a resolution of 480 by 640 pixels with 9 bits per pixel. Originally color pictures were photographed from the screen of a color CRT so that only three bits were available for each of the three primary colors. Ordered dither [15] was applied to the image data to produce 111 effective intensity levels per primary. For this report pictures are produced by a high-quality color hardcopy camera that exposes each color separately to provide eight bits of intensity per color.

For the scenes shown in this paper, the image generation times are

Figure 6: 44 minutes,
Figure 7: 74 minutes,
Figure 8: 122 minutes.

All times given are for the VAX, which is nearly three times faster than the PDP-11/45 for this application. The image of Figure 6 shows three glossy objects with shadows and object-to-object reflections. The texturing is added using Blinn's wrinkling technique. Figure 7 illustrates the effect of refraction through a transparent object. The algorithm has also been used to produce a short animated sequence. The enhancements provided by this illumination model are more readily apparent in the animated sequence than in the still photographs.

A breakdown of where the program spends its time for simple scenes is:

Overhead—13 percent,
Intersection—75 percent,
Shading—12 percent.

For more complex scenes the percentage of time required to compute the intersections of rays and surfaces increases to over 95 percent. Since the program makes almost no use of image coherence, these figures are actually quite promising. They indicate that a more efficient intersection processor will greatly improve the algorithm's performance. This distribution of processing times also suggests that a reasonable division of tasks between processors in a multiprocessor system is to have one or more processors dedicated to intersection calculations with ray generation and shading operations performed by the host.

[1] UNIX is a trademark of Bell Laboratories.

5. Summary

This illumination model draws heavily on techniques derived previously by Phong [8] and Blinn [3-5], but it operates recursively to allow the use of global illumination information. The approach used and the results achieved are similar to those presented by Kay [16].

While in many cases the model generates very realistic effects, it leaves considerable room for improvement. Specifically, it does not provide for diffuse reflection from distributed light sources, nor does it gracefully handle specular reflections from less glossy surfaces. It is implemented through a visible surface algorithm that is very slow but which shows some promise of becoming more efficient. When better ways of using picture coherence to speed the display process are found, this algorithm may find use in the generation of realistic animated sequences.

Received 12/78; revised 1/80; accepted 2/80

References

1. Appel, A. Some techniques for shading machine renderings of solids. AFIPS 1968 Spring Joint Comptr. Conf., pp. 37–45.
2. Atherton, P., Weiler, K., and Greenberg, D. Polygon shadow generation. Proc. SIGGRAPH 1978, Atlanta, Ga., pp. 275–281.
3. Blinn, J.F. Models of light reflection for computer synthesized pictures. Proc. SIGGRAPH 1977, San Jose, Calif., pp. 192–198.
4. Blinn, J.F. Simulation of wrinkled surfaces. Proc. SIGGRAPH 1978, Atlanta, Ga., pp. 286–292.
5. Blinn, J.F., and Newell, M.E. Texture and reflection in computer generated images. *Comm. ACM 19*, 10 (Oct. 1976), 542–547.
6. Blinn, J.F., and Newell, M.E. The progression of realism in computer generated images. Proc. of the ACM Ann. Conf., 1977, pp. 444–448.
7. Bouknight, W.K., and Kelley, K.C. An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources. AFIPS 1970 Spring Joint Comptr. Conf., pp. 1–10.
8. Bui-Tuong Phong. Illumination for computer generated images. *Comm. ACM 18*, 6 (June 1975), 311–317.
9. Catmull, E. A subdivision algorithm for computer display of curved surfaces. UTEC CSc-74-133, Comptr. Sci. Dept., Univ. of Utah, 1974.
10. Catmull, E., and Clark, J. Recursively generated B-spline surfaces on arbitrary topological meshes. *Comptr. Aided Design 10*, 6 (Nov. 1978), 350–355.
11. Clark, J.H. Hierarchical geometric models for visible surface algorithms. *Comm. ACM 19*, 10 (Oct. 1976), 547–554.
12. Crow, F.C. Shadow algorithms for computer graphics. Proc. SIGGRAPH 1977, San Jose, Calif., pp. 242–248.
13. Crow, F.C. The aliasing problem in computer-generated shaded images. *Comm. ACM 20*, 11 (Nov. 1977), 799–805.
14. Goldstein, R.A. and Nagel, R. 3-D visual simulation. *Simulation* (Jan. 1971), 25–31.
15. Jarvis, J.F., Judice, C.N., and Ninke, W.H. A survey of techniques for the display of continuous tone pictures on bilevel displays. *Comptr. Graphics and Image Proc. 5* (1976), 13–40.
16. Kay, D.S. Transparency, refraction, and ray tracing for computer synthesized images. Masters thesis, Cornell Univ., Ithaca, N.Y., January 1979.
17. Kay, D.S., and Greenberg, D. Transparency for computer synthesized images. Proc. SIGGRAPH 1979, Chicago, Ill., pp. 158–164.
18. Newell, M.E., Newell, R.G., and Sancha, T.L. A solution to the hidden surface problem. Proc. ACM Ann. Conf., 1972, pp. 443–450.
19. Warnock, J.E. A hidden line algorithm for halftone picture representation. Tech. Rep. TR 4-15, Comptr. Sci. Dept., Univ. of Utah, 1969.
20. Williams, L. Casting curved shadows on curved surfaces. Proc. SIGGRAPH 1978, Atlanta, Ga., pp. 270–274.

Distributed Ray Tracing

Robert L. Cook
Thomas Porter
Loren Carpenter

Computer Division
Lucasfilm Ltd.

Abstract

Ray tracing is one of the most elegant techniques in computer graphics. Many phenomena that are difficult or impossible with other techniques are simple with ray tracing, including shadows, reflections, and refracted light. Ray directions, however, have been determined precisely, and this has limited the capabilities of ray tracing. By distributing the directions of the rays according to the analytic function they sample, ray tracing can incorporate fuzzy phenomena. This provides correct and easy solutions to some previously unsolved or partially solved problems, including motion blur, depth of field, penumbras, translucency, and fuzzy reflections. Motion blur and depth of field calculations can be integrated with the visible surface calculations, avoiding the problems found in previous methods.

CR CATEGORIES AND SUBJECT DESCRIPTORS: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism;

ADDITIONAL KEY WORDS AND PHRASES: camera, constructive solid geometry, depth of field, focus, gloss, motion blur, penumbras, ray tracing, shadows, translucency, transparency

1. Introduction

Ray tracing algorithms are elegant, simple, and powerful. They can render shadows, reflections, and refracted light, phenomena that are difficult or impossible with other techniques[11]. But ray tracing is currently limited to sharp shadows, sharp reflections, and sharp refraction.

Ray traced images are sharp because ray directions are determined precisely from geometry. Fuzzy phenomenon would seem to require large numbers of additional samples per ray. By distributing the rays rather than adding more of them, however, fuzzy phenomena can be rendered with no additional rays beyond those required for spatially oversampled ray tracing. This approach provides correct and easy solutions to some previously unsolved problems.

This approach has not been possible before because of aliasing. Ray tracing is a form of point sampling and, as such, has been subject to aliasing artifacts. This aliasing is not inherent, however, and ray tracing can be filtered as effectively as any analytic method[4]. The filtering does incur the expense of additional rays, but it is not merely oversampling or adaptive oversampling, which in themselves cannot solve the aliasing problem. This antialiasing is based on an approach proposed by Rodney Stock. It is the subject of a forthcoming paper.

Antialiasing opens up new possibilities for ray tracing. Ray tracing need not be restricted to spatial sampling. If done with proper antialiasing, the rays can sample motion, the camera lens, and the entire shading function. This is called *distributed ray tracing.*

Distributed ray tracing is a new approach to image synthesis. The key is that no extra rays are needed beyond those used for oversampling in space. For example, rather than taking multiple time samples at every spatial location, the rays are distributed in time so that rays at different spatial locations are traced at different instants of time. Once we accept the expense of oversampling in space, distributing the rays offers substantial benefits at little additional cost.

- Sampling the reflected ray according to the specular distribution function produces gloss (blurred reflection).
- Sampling the transmitted ray produces translucency (blurred transparency).
- Sampling the solid angle of the light sources produces penumbras.

- Sampling the camera lens area produces depth of field.
- Sampling in time produces motion blur.

2. Shading

The intensity I of the reflected light at a point on a surface is an integral over the hemisphere above the surface of an illumination function L and a reflection function R[1].

$$I(\phi_r,\theta_r) = \int_{\phi_i}\int_{\theta_i} L(\phi_i,\theta_i)R(\phi_i,\theta_i,\phi_r,\theta_r)d\phi_i d\theta_i$$

where

(ϕ_i,θ_i) is the angle of incidence, and

(ϕ_r,θ_r) is the angle of reflection.

The complexity of performing this integration has been avoided by making some simplifying assumptions. The following are some of these simplifications:

- Assume that L is a δ function, i.e., that L is zero except for light source directions and that the light sources can be treated as points. The integral is now replaced by a sum over certain discrete directions. This assumption causes sharp shadows.
- Assume that all of the directions that are not light source directions can be grouped together into an ambient light source. This ambient light is the same in all directions, so that L is independent of ϕ_i and θ_i and may be removed from the integral. The integral of R may then be replaced by an average, or ambient, reflectance.
- Assume that the reflectance function R is a δ function, i.e., that the surface is a mirror and reflects light only from the mirror direction. This assumption causes sharp reflections. A corresponding assumption for transmitted light causes sharp refraction.

The shading function may be too complex to compute analytically, but we can point sample its value by distributing the rays, thus avoiding these simplifying assumptions. Illumination rays are not traced toward a single light direction, but are distributed according to the illumination function L. Reflected rays are not traced in a single mirror direction but are distributed according to the reflectance function R.

2.1. Gloss

Reflections are mirror-like in computer graphics, but in real life reflections are often blurred or hazy. The distinctness with which a surface reflects its environment is called *gloss*[5]. Blurred reflections have been discussed by Whitted[11] and by Cook[2]. Any analytic simulation of these reflections must be based on the integral of the reflectance over some solid angle.

Mirror reflections are determined by tracing rays from the surface in the mirror direction. Gloss can be calculated by distributing these secondary rays about the mirror direction. The distribution is weighted according to the same distribution function that determines the highlights.

This method was originally suggested by Whitted[11], and it replaces the usual specular component. Rays that reflect light sources produce highlights.

2.2. Translucency

Light transmitted through an object is described by an equation similar to that for reflected light, except that the reflectance function R is replaced by a transmittance function T and the integral is performed over the hemisphere behind the surface. The transmitted light can have ambient, diffuse, and specular components[5].

Computer graphics has included transparency, in which T is assumed to be a δ function and the images seen through transparent objects are sharp. Translucency differs from transparency in that the images seen through translucent objects are not distinct. The problem of translucency is analogous to the problem of gloss. Gloss requires an integral of the reflected light, and translucency requires a corresponding integral of the transmitted light.

Translucency is calculated by distributing the secondary rays about the main direction of the transmitted light. Just as the distribution of the reflected rays is defined by the specular reflectance function, the distribution of the transmitted rays is defined by a specular transmittance function.

2.3. Penumbras

Penumbras occur where a light source is partially obscured. The reflected intensity due to such a light is proportional to the solid angle of the visible portion of the light. The solid angle has been explicitly included in a shading model[3], but no algorithms have been suggested for determining this solid angle because of the complexity of the computation involved. The only attempt at penumbras known to the authors seems to solve only a very special case[7].

Shadows can be calculated by tracing rays from the surface to the light sources, and penumbras can be calculated by distributing these secondary rays. The shadow ray can be traced to any point on the light source, not just not to a single light source location. The distribution of the shadow rays must be weighted according the projected area and brightness of different parts of the light source. The number of rays traced to each region should be proportional to the amount of the light's energy that would come from that region if the light was

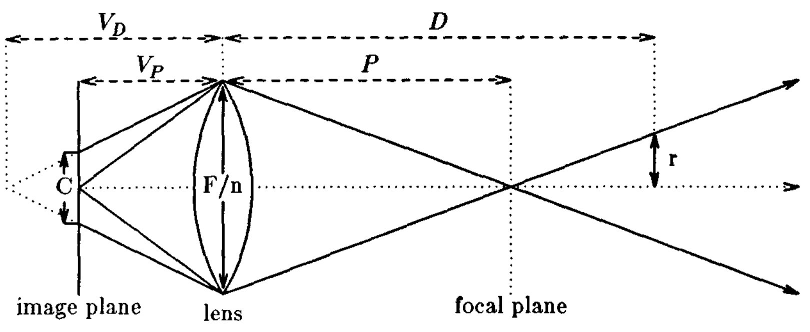


Figure 1. Circle of Confusion.

completely unobscured. The proportion of lighted sample points in a region of the surface is then equal to the proportion of that light's intensity that is visible in that region.

3. Depth of Field

Cameras and the eye have a finite lens aperture, and hence their images have a finite depth of field. Each point in the scene appears as a circle on the image plane. This circle is called the circle of confusion, and its size depends on the distance to the point and on the lens optics. Depth of field can be an unwanted artifact, but it can also be a desirable effect.

Most computer graphics has been based on a pinhole camera model with every object in sharp focus. Potmesil simulated depth of field with a postprocessing technique. Each object is first rendered in sharp focus (i.e., with a pinhole camera model), and later each sharply rendered object is convolved with a filter the size of the circle of confusion[8]. The program spends most of its time in the focus postprocessor, and this time increases dramatically as the aperture decreases.

Such a postprocessing approach can never be completely correct. This is because visibility is calculated from a single point, the center of the lens. The view of the environment is different from different parts of the lens, and the differences include changes in visibility and shading that cannot be accounted for by a postprocessing approach.

For example, consider an object that is extremely out of focus in front of an object that is in focus. Visible surface calculations done with the pinhole model determine the visibility from the center of the lens. Because the front object is not in focus, parts of the focused object that are not visible from the center of the lens will be visible from other parts of the lens. Information about those parts will not available for the postprocessor, so the postprocessor cannot possibly get the correct result.

There is another way to approach the depth of field problem. Depth of field occurs because the lens is a finite size. Each point on the lens "looks" at the same point on the focal plane. The visible surfaces and the shading may be different as seen from different parts of the lens. The depth of field calculations should account for this and be an integral part of the visible surface and shading calculations.

Depth of field can be calculated by starting with the traditional ray from the center of the lens through point p on the focal plane. A point on the surface of the lens is selected and the ray from that point to p is traced. The camera specifications required for this calculation are the focal distance and the diameter of the lens $\frac{F}{n}$, where F is the focal length of the lens and n is the aperture number.

This gives exactly the same circle of confusion as presented by Potmesil[8]. Because it integrates the depth of field calculations with the shading and visible surface calculations, this method gives a more accurate solution to the depth of field problem, with the exception that it does not account for diffraction effects.

Figure 1 shows why this method gives the correct circle of confusion. The lens has a diameter of $\frac{F}{n}$ and is focused at a distance P so that the image plane is at a distance V_P, where

$$V_P = \frac{FP}{P-F} \text{ for } P>F\,.$$

Points on the plane that is a distance D from the lens will focus at

$$V_D = \frac{FD}{D-F} \text{ for } D>F$$

and have a circle of confusion with diameter C of[8]

$$C = |V_D - V_P|\frac{F}{nV_D}$$

For a point I on the image plane, the rays we trace lie inside the cone whose radius at D is

$$r = \frac{1}{2}\frac{F}{n}\frac{|D-P|}{P}$$

The image plane distance from a point on this cone to a point on the axis of the cone is r multiplied by the magnification of the lens.

$$R = r\left(-\frac{V_P}{D}\right).$$

It is easily shown that

$$R = \frac{C}{2}.$$

Hence any points on the cone have a circle of confusion that just touches the image point I. Points outside the cone do not affect the image point and points inside the cone do.

4. Motion Blur

Distributing the rays or sample points in time solves the motion blur problem. Before we discuss this method and how it works, let us first look in more detail at the motion blur problem and at previous attempts to solve it.

The motion blur method described by Potmesil[9] is not only expensive, it also separates the visible surface calculation from the motion blur calculation. This is acceptable in some situations, but in most cases we cannot just calculate a still frame and blur the result. Some object entirely hidden in the still frame might be uncovered for part of the the time sampled by the blur. If we are to blur an object across a background, we have to know what the background is.

Even if we know what the background is, there are problems. For example, consider a biplane viewed from above, so that the lower wing is completely obscured by the upper wing. Because the upper wing is moving, the scenery below it would be seen through its blur, but unfortunately the lower wing would show through too. The lower wing should be hidden completely because it moves with the the upper wing and is obscured by it over the entire time interval.

This particular problem can be solved by rendering the plane and background as separate elements, but not all pictures can easily be separated into elements. This solution also does not allow for changes in visibility within a single object. This is particularly important for rotating objects.

The situation is further complicated by the change in shading within a frame time. Consider a textured top spinning on a table. If we calculate only one shade per frame, the texture would be blurred properly, but unfortunately the highlights and shadows would be blurred too. On a real top, the highlights and shadows are not blurred at all by the spinning. They are blurred, of course, by any lateral motion of the top along the table or by the motion of a light source or the camera. The highlights should be blurred by the motion of the light and the camera, by the travel of the top along the table, and by the precession of the top, but not by the rotation of the top.

Motion blurred shadows are also important and are not rendered correctly if we calculate only one shade per frame. Otherwise, for example, the blades of a fan could be motion blurred, but the shadows of those blades would strobe.

All of this is simply to emphasize the tremendous complexity of the motion blur problem. The prospects for an analytic solution are dim. Such a solution would require solving the visible surface problem as a function of time as well as space. It would also involve integrating the texture and shading function of the visible surfaces over time. Point sampling seems to be the only approach that offers any promise of solving the motion blur problem.

One point sampling solution was proposed by Korein and Badler[6]. Their method, however, point samples only in space, not in time. Changes in shading are not motion blurred. The method involves keeping a list of all objects that cross each sample point during the frame time, a list that could be quite long for a fast moving complex scene. They also impose the unfortunate restriction that both vertices of an edge must move at the same velocity. This creates holes in objects that change perspective severely during one frame, because the vertices move at drastically different rates. Polygons with edges that share these vertices cannot remain adjoining. The algorithm is also limited to linear motion. If the motion is curved or if the vertices are allowed to move independently, the linear intersection equation becomes a higher order equation. The resulting equation is expensive to solve and has multiple roots.

Distributing the sample points in time solves the motion blur problem. The path of motion can be arbitrarily complex. The only requirement is the ability to calculate the position of the object at a specific time. Changes in visibility and shading are correctly accounted for. Shadows (umbras and penumbras), depth of field, reflections and intersections are all correctly motion blurred. By using different distributions of rays, the motion can be blurred with a box filter or a weighted filter or can be strobed.

This distribution of the sample points in time does not involve adding any more sample points. Updating the object positions for each time is the only extra calculation needed for motion blur. Proper antialiasing is required or the picture will look strobed or have holes[4].

5. Other Implications of the Algorithm

Visible surface calculation is straightforward. Since each ray occurs at a single instant of time, the first step is to update the positions of the objects for that instant of time. The next is to construct a ray from the lens to the sample point and find the closest object that the ray intersects. Care must be taken in bounding moving objects. The bound should depend on time so that the number of potentially visible objects does not grow unacceptably with their speed.

Intersecting surfaces are handled trivially because we never have to calculate the line of intersection; we merely have to determine which is in front at a given location and time. At each sample point only one of the surfaces is visible. The intersections can even be motion blurred, a problem that would be terrifying with an analytic method.

The union, intersection, difference problem is easily solved with ray tracing or point sampling[10]. These calculations are also correctly motion blurred.

Transparency is easy even if the transparency is textured or varies with time. Let τ be the transparency of a surface at the time and location it is pierced by the ray, and let R be the reflectance. R and τ are wavelength dependent, and the color of the transparency is not necessarily the same as the color of the reflected light; for example, a red transparent plastic object may have a white highlight. If there are n-1 transparent surfaces in front of the opaque surface, the light reaching the viewer is

$$R_n\prod_{i=1}^{n-1}\tau_i + R_{n-1}\prod_{i=1}^{n-2}\tau_1 + \cdots + R_2\tau_1 + R_1 = \sum_{i=1}^{n} R_i \prod_{j=1}^{i-1} \tau_j.$$

If the surfaces form solid volumes, then each object has a τ, and that τ is scaled by the distance that the transmitted ray travels through that object. The motion blur and depth of field calculations work correctly for these transparency calculations.

The distributed approach can be adapted to a scanline algorithm as well as to ray tracing. The general motion blur and depth of field calculations have been incorporated into a scanline algorithm using distributed sampling for the visible surface calculations. Special cases of penumbras, fuzzy reflections, and translucency have been successfully incorporated for flat surfaces.

6. Summary of the Algorithm

The intensity of a pixel on the screen is an analytic function that involves several nested integrals: integrals over time, over the pixel region, and over the lens area, as well as an integral of reflectance times illumination over the reflected hemisphere and an integral of transmittance times illumination over the transmitted hemisphere. This integral can be tremendously complicated, but we can point sample the function regardless of how complicated it is. If the function depends on n parameters, the function is sampled in the n dimensions defined by those parameters. Rather than adding more rays for each dimension, the existing rays are distributed in each dimension according to the values of the corresponding parameter.

This summary of the distributed ray tracing algorithm is illustrated in Figure 2 for a single ray.

- Choose a time for the ray and move the objects accordingly. The number of rays at a certain time is proportional to the value of the desired temporal filter at that time.

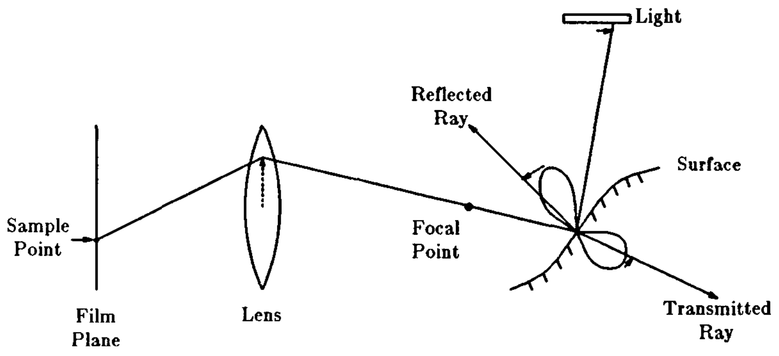


Figure 2. Typical Distributed Ray Path

- Construct a ray from the eye point (center of the lens) to a point on the screen. Choose a location on the lens, and trace a ray from that location to the focal point of the original ray. Determine which object is visible.
- Calculate the shadows. For each light source, choose a location on the light and trace a ray from the visible point to that location. The number of rays traced to a location on the light should be proportional to the intensity and projected area of that location as seen from the surface.
- For reflections, choose a direction around the mirror direction and trace a ray in that direction from the visible point. The number of rays traced in a specific direction should be proportional to the amount of light from that direction that is reflected toward the viewer. This can replace the specular component.
- For transmitted light, choose a direction around the direction of the transmitted light and trace a ray in that direction from the visible point. The number of rays traced in a specific direction should be proportional to the amount of light from that direction that is transmitted toward the viewer.

7. Examples

Figure 3 illustrates motion blurred intersections. The blue beveled cube is stationary, and the green beveled cube is moving in a straight line, perpendicular to one of its faces. Notice that the intersection of the faces is blurred except in in the plane of motion, where it is sharp.

Figures 4 and 5 illustrate depth of field. In figure 4, the camera has a 35 mm lens at f2.8. Notice that the rear sphere, which is out of focus, does not blur over the spheres in front. In figure 5, the camera is focused on the center of the three wooden spheres.

Figure 6 shows a number of moving spheres, with motion blurred shadows and reflections.

Figure 7 illustrates fuzzy shadows and reflections. The paper clip is illuminated by two local light sources which cast shadows with penumbras on the table. Each light is an extended light source (i.e., not a point light source) with a finite solid angle, and the intensity of its shadow at any point on the table is proportional to the amount of light obscured by the paper clip. The table reflects the paper clip, and the reflection blurs according to the specular distribution function of the table top. Note that both the shadows and the reflection blur with distance and are sharper close to the paper clip.

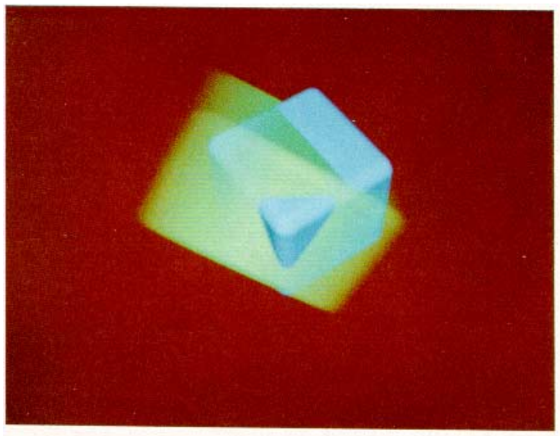


Figure 3. Motion Blurred Intersection.

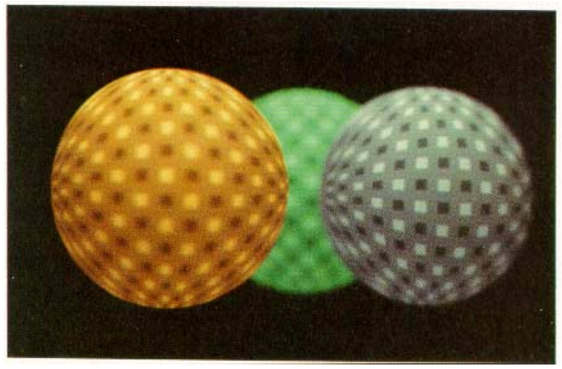


Figure 4. Depth of Field.

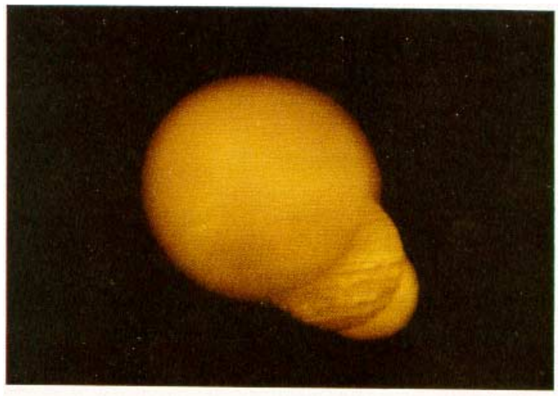


Figure 5. Depth of Field.

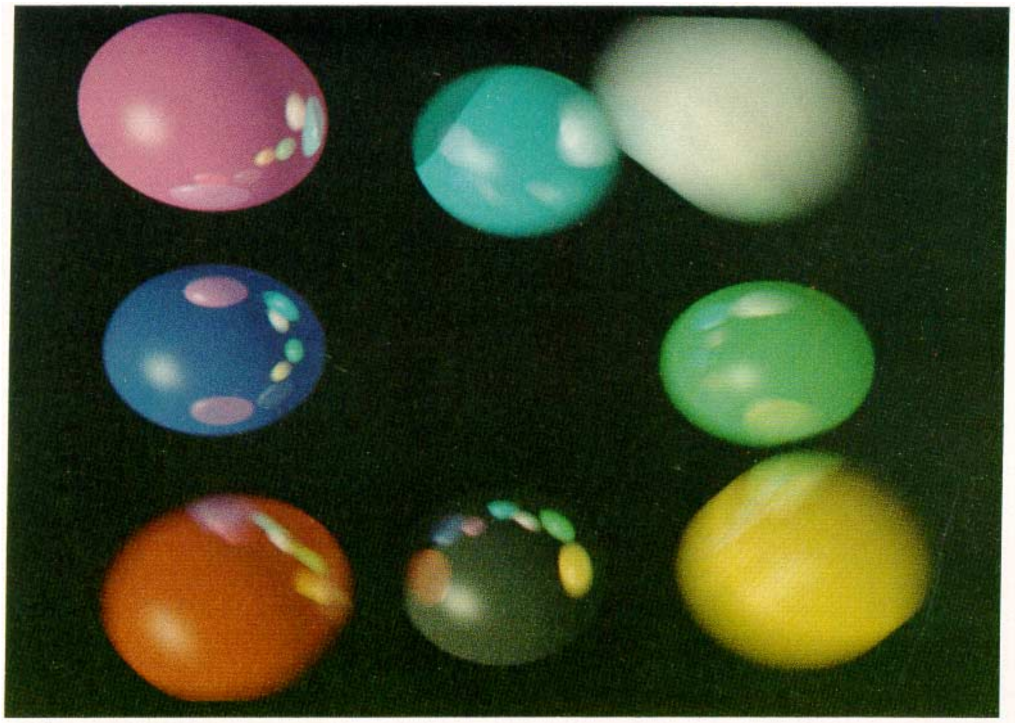


Figure 6. Balls in Motion.

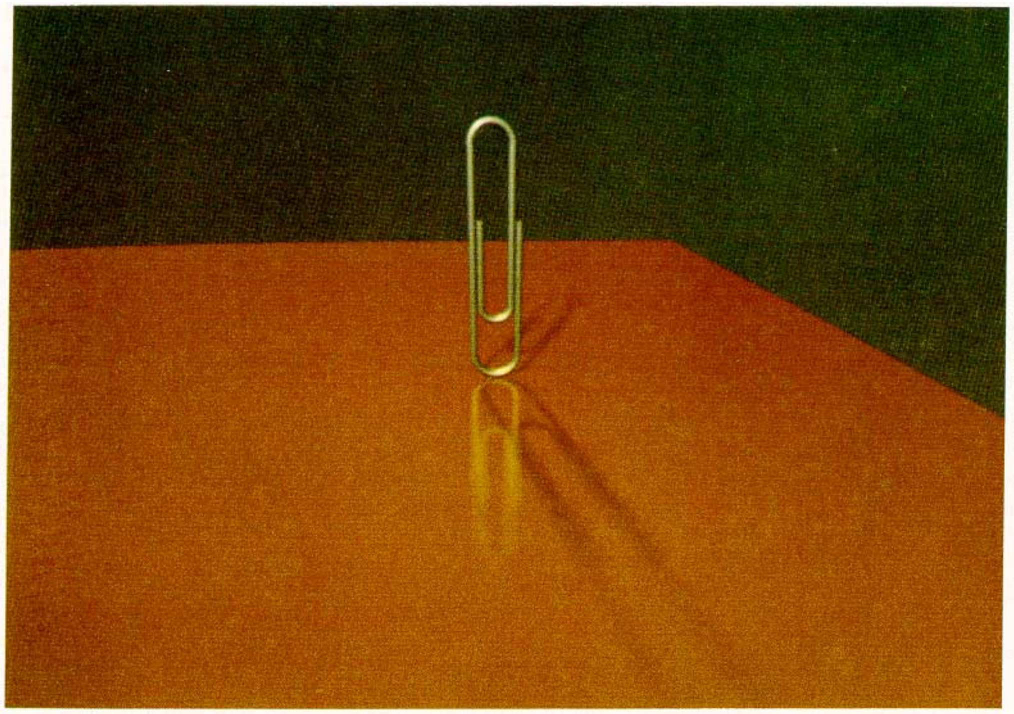


Figure 7. Paper Clip.

Figure 8 shows 5 billiard balls with motion blur and penumbras. Notice that the motion is not linear: the 9 ball changes direction abruptly in the middle of the frame, the 8 ball moves only during the middle of the frame, and the 4 ball only starts to move near the end of the frame. The shadows on the table are sharper where the balls are closer to the table; this most apparent in the stationary 1 ball. The reflections of the billiard balls and the room are motion blurred, as are the penumbras.

Figures 3, 5, and 7 were rendered with a scanline adaptation of this algorithm. Figures 4, 6, and 8 were rendered with ray tracing.

8. Conclusions

Distributed ray tracing a new paradigm for computer graphics which solves a number of hitherto unsolved or partially solved problems. The approach has also been successfully adapted to a scanline algorithm. It incorporates depth of field calculations into the visible surface calculations, eliminating problems in previous methods. It makes possible blurred phenomena such as penumbras, gloss, and translucency. All of the above can be motion blurred by distributing the rays in time.

These are not isolated solutions to isolated problems. This approach to image synthesis is practically no more expensive than standard ray tracing and solves all of these problems at once. The problems could not really be solved separately because they are all interrelated. Differences in shading, in penumbras, and in visibility are accounted for in the depth of field calculations. Changes in the depth of field and in visibility are motion blurred. The penumbra and shading calculations are motion blurred. All of these phenomena are related, and the new approach solves them all together by sampling the multidimensional space they define. The key to this is the ability to antialias point sampling.

9. Acknowledgements

Rodney Stock proposed the approach to antialiased point sampling that formed the basis of the paradigm explored in this paper. John Lasseter drew the environment map of the pool hall for "1984". Ed Catmull worked with us in the image synthesis working group and helped develop and refine these ideas. He and Alvy Ray Smith provided invaluable suggestions along the way. Tom Duff wrote the ray tracing program that we adapted to distributed ray tracing.

References

1. COOK, ROBERT L., TURNER WHITTED, AND DONALD P. GREENBERG, *A Comprehensive Model for Image Synthesis.* unpublished report
2. COOK, ROBERT L., "A Reflection Model for Realistic Image Synthesis," Master's thesis, Cornell University, Ithaca, NY, December 1981.
3. COOK, ROBERT L. AND KENNETH E. TORRANCE, "A Reflection Model for Computer Graphics," *ACM Transactions on Graphics*, vol. 1, no. 1, pp. 7-24, January 1982.
4. COOK, ROBERT L., "Antialiased Point Sampling," Technical Memo #94, Lucasfilm Ltd, San Rafael, CA, October 3, 1983.
5. HUNTER, RICHARD S., *The Measurement of Appearance,* John Wiley & Sons, New York, 1975.
6. KOREIN, JONATHAN AND NORMAN BADLER, "Temporal Anti-Aliasing in Computer Generated Animation," *Computer Graphics*, vol. 17, no. 3, pp. 377-388, July 1983.
7. NISHITA, TOMOYUKI, ISAO OKAMURA, AND EIHACHIRO NAKAMAE, *Siggraph Art Show,* 1982.
8. POTMESIL, MICHAEL AND INDRANIL CHAKRAVARTY, "Synthetic Image Generation with a Lens and Aperture Camera Model," *ACM Transactions on Graphics*, vol. 1, no. 2, pp. 85-108, April 1982.
9. POTMESIL, MICHAEL AND INDRANIL CHAKRAVARTY, "Modeling Motion Blur in Computer-Generated Images," *Computer Graphics*, vol. 17, no. 3, pp. 389-399, July 1983.
10. ROTH, S. D., "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*, no. 18, pp. 109-144, 1982.
11. WHITTED, TURNER, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, pp. 343-349, 1980.

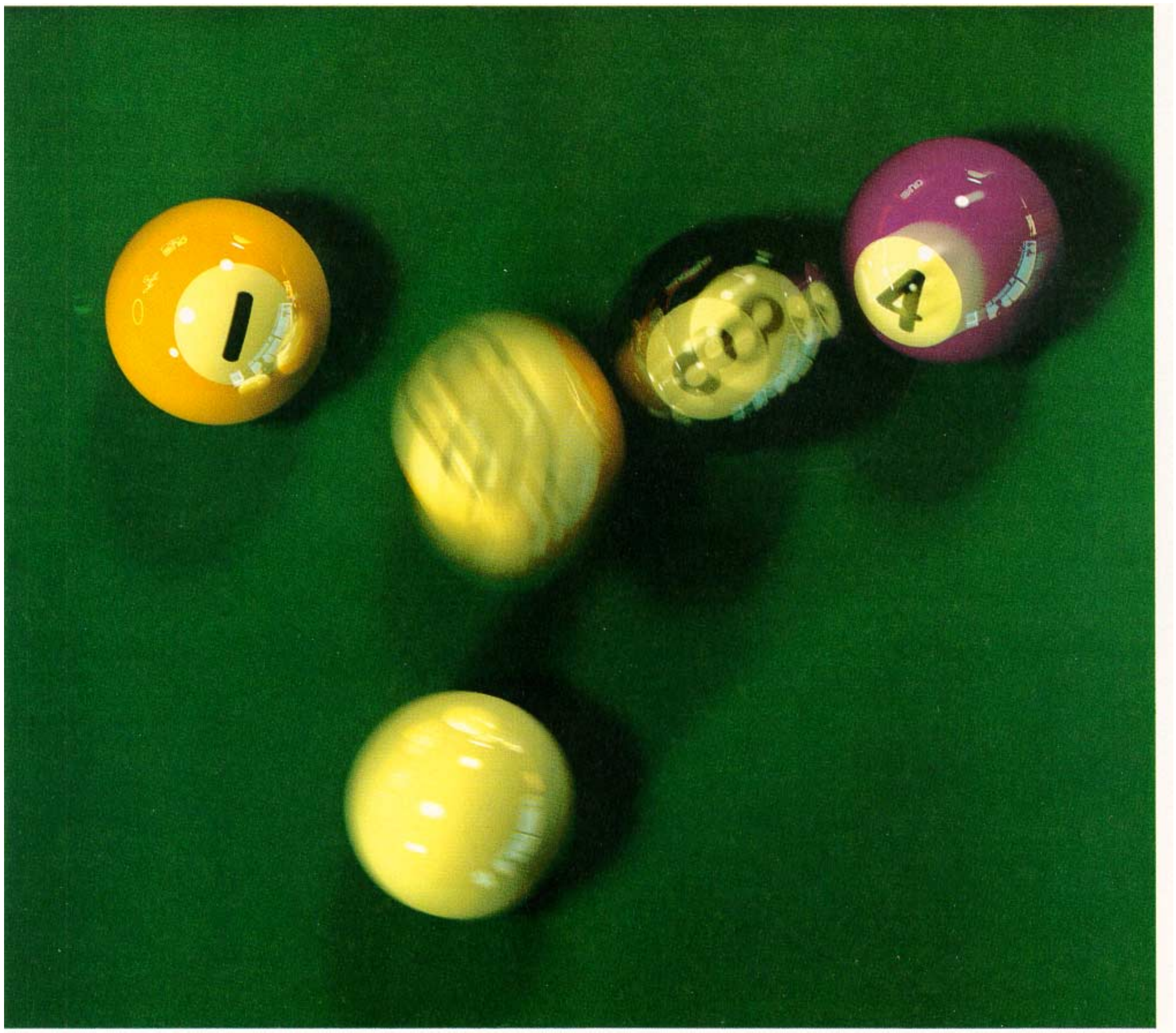


Figure 8. 1984.

Stochastic Sampling in Computer Graphics

ROBERT L. COOK
Pixar

Ray tracing, ray casting, and other forms of point sampling are important techniques in computer graphics, but their usefulness has been undermined by aliasing artifacts. In this paper it is shown that these artifacts are not an inherent part of point sampling, but a consequence of using regularly spaced samples. If the samples occur at appropriate nonuniformly spaced locations, frequencies above the Nyquist limit do not alias, but instead appear as noise of the correct average intensity. This noise is much less objectionable to our visual system than aliasing. In ray tracing, the rays can be stochastically distributed to perform a Monte Carlo evaluation of integrals in the rendering equation. This is called *distributed ray tracing* and can be used to simulate motion blur, depth of field, penumbrae, gloss, and translucency.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism

General Terms: Algorithms

Additional Key Words and Phrases: Antialiasing, filtering, image synthesis, Monte Carlo integration, motion blur, raster graphics, ray tracing, sampling, stochastic sampling

1. INTRODUCTION

Because pixels are discrete, computer graphics is inherently a sampling process. The pixel size determines an upper limit to the frequencies that can be displayed. This limit, one cycle every two pixels, is called the *Nyquist limit.* An attempt to display frequencies greater than the Nyquist limit can produce aliasing artifacts, such as "jaggies" on the edges of objects [6], jagged highlights [26], strobing and other forms of temporal aliasing [19], and Moiré patterns in textures [6]. These artifacts are tolerated in some real-time applications in which speed is more vital than beauty, but they are unacceptable in realistic image synthesis.

Rendering algorithms can be classified as *analytic* or *discrete* according to how they approach the aliasing problem. Analytic algorithms can filter out the high frequencies that cause aliasing before sampling the pixel values. This filtering tends to be complicated and time consuming, but it can eliminate certain types of aliasing very effectively [3, 6, 8, 9, 15]. Discrete algorithms, such as ray tracing,

This research was done when Pixar was the computer division of Lucasfilm Ltd. An earlier version of this paper was prepared as an unpublished Lucasfilm Technical Memo #94, "Antialiased Point Sampling," 1983.

Author's address: Pixar, P.O. Box 13719, San Rafael, CA 94913-3719.

only consider the image at regularly spaced sample points. Since they ignore everything not at these points, they appear by their nature to preclude filtering the image. Thus they are plagued by seemingly inherent aliasing artifacts. This is unfortunate, for these algorithms are much simpler, more elegant, and more amenable to hardware implementation than the analytic methods. They are also capable of many features that are difficult to do analytically, such as shadows, reflection, refraction [13, 24], constructive solid geometry [21], motion blur, and depth of field [5].

There are two existing discrete approaches to alleviating the aliasing problem: *supersampling* and *adaptive sampling*. Supersampling involves using more than one regularly spaced sample per pixel. It reduces aliasing by raising the Nyquist limit, but it does not eliminate aliasing. No matter how many samples are used, there are still frequencies that will alias. In adaptive sampling, additional rays are traced near edges [24]; the additional rays are traced midway between previously traced rays. Unlike supersampling, this approach can antialias edges reliably, but it may require a large number of rays, and it complicates an otherwise simple algorithm.

In this paper a new discrete approach to antialiasing called *stochastic sampling* is presented. Stochastic sampling is a Monte Carlo technique [11] in which the image is sampled at appropriate nonuniformly spaced locations rather than at regularly spaced locations. This approach is inherently different from either supersampling or adaptive sampling, though it can be combined with either of them. Stochastic sampling can eliminate all forms of aliasing, including unruly forms such as highlight aliasing.

With stochastic sampling, aliasing is replaced by noise of the correct average intensity. Frequencies above the Nyquist limit are still inadequately sampled, and they still appear as artifacts in the image. But a highly objectionable artifact (aliasing) is replaced with an artifact that our visual systems tolerate very well (noise).

In addition to providing a solution to the aliasing problem, stochastic sampling also provides new capabilities for discrete algorithms such as ray tracing. The physical equations simulated in the rendering process involve integrals over time, lens area, specular reflection angle, etc. Image-synthesis algorithms have usually avoided performing these integrals by resorting to crude approximations that assume instantaneous shutters, pinhole cameras, mirror or diffuse reflections, etc. But these integrals can be easily evaluated by stochastically sampling them, a process called Monte Carlo integration. In a ray-tracing algorithm, this involves stochastically distributing the rays in time, lens area, reflection angle, etc. This is called *probabilistic* or *distributed ray tracing* [5]. Distributed ray tracing allows the simulation of fuzzy phenomena, such as motion blur, depth of field, penumbrae, gloss, and translucency.

2. UNIFORM POINT SAMPLING

Before discussing stochastic sampling, we first review uniform sampling and the source of aliasing. In a point-sampled picture, frequencies greater than the Nyquist limit are inadequately sampled. If the samples are uniformly spaced, these frequencies can appear as aliases, that is, they can appear falsely as low frequencies [4, 17, 20].

To see how this happens, consider for the moment one-dimensional sampling; we refer to the dimension as time. Let a signal $f(t)$ be sampled at regular intervals of time, that is, at times nT for integer n, where T is the time period between samples, so that $1/T$ is the sampling frequency. The Nyquist limit is half the sampling frequency, or $0.5/T$. This sampling is equivalent to multiplication by the *shah* function $\mathrm{III}(t/T)$, where

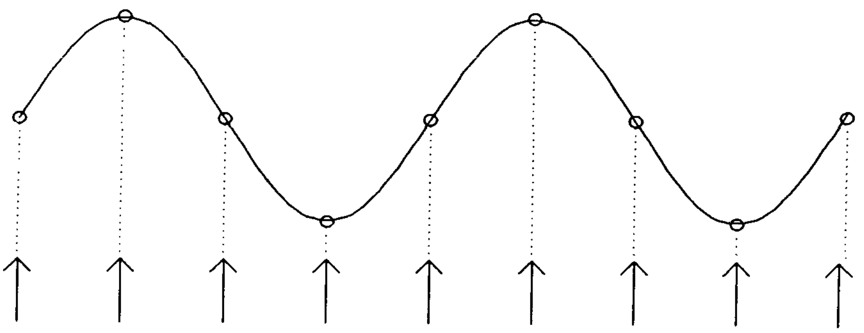
$$\mathrm{III}(x) = \sum_{n=-\infty}^{\infty} \delta(x - n),$$

where δ is the Kronecker delta function. After sampling, information about the original signal $f(t)$ is preserved only at the sample points. The sampling theorem states that, if $f(t)$ contains no frequencies above the Nyquist limit, then sampling followed by an ideal reconstruction filter reproduces the original signal $f(t)$ exactly.

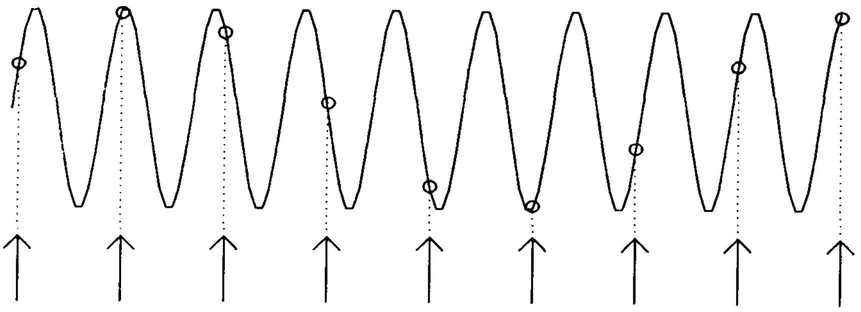
This situation is shown in Figure 1 for a sine wave. In Figure 1a, the frequency of the sine wave is below the Nyquist limit of the samples, and the sampled values accurately represent the function. But, in Figure 1b, the frequency of the sine wave is above the Nyquist limit of the samples. The sampled values do not accurately represent the sampled sine wave; instead they look as if they came from a low-frequency sine wave. The high-frequency sine wave appears incorrectly under the alias of this low-frequency sine wave.

Figure 2 shows this situation in the frequency domain. The Fourier transform of f is denoted by F; the Fourier transform of the shah function $\mathrm{III}(t/T)$ is another shah function $(1/T)\mathrm{III}(tT)$. Figure 2a shows the Fourier transform of the signal in Figure 1a, a single sine wave whose frequency is below the Nyquist limit. Sampling involves convolving the signal with the sampling grid of Figure 2b to produce the spectrum shown in Figure 2c. An ideal reconstruction filter, shown in Figure 2d, would extract the original signal, as in Figure 2e. In Figures 2f–2j, the same process is repeated for the signal in Figure 1b, a single sine wave whose frequency is above the Nyquist limit. In this case, the sampling process can fold the high-frequency sine wave into low frequencies, as shown in Figure 2h. These false frequencies, or aliases, cannot be separated from frequencies that are a part of the original signal. The part of the spectrum extracted by the reconstruction filter contains these aliases, as shown in Figure 2j.

Sampling theory thus predicts that, with a regular sampling grid, frequencies greater than the Nyquist limit can alias. The inability to reproduce those frequencies is inherent in the sampling process, but their appearance as aliases is a consequence of the regularity of the sampling grid. If the sample points are not regularly spaced, the energy in those frequencies can appear as noise, an artifact that is much less objectionable than aliasing. In the case of uniform sampling, aliasing is precisely defined; in the case of nonuniform sampling, we use the term aliasing to mean artifacts with distinct frequency components, as opposed to noise.



(a) Point sampling within the Nyquist limit



(b) Point sampling beyond the Nyquist limit

Fig. 1. Point sampling shown in the spatial domain. The arrows indicate the sample locations, and the circles indicate the sampled values. In (a), the sine wave frequency is within the Nyquist limit, so the sampled values accurately represent the signal. In (b), the sine wave frequency is above the Nyquist limit, and the sampled values incorrectly represent a low-frequency sine wave that is not present in the signal.

3. POISSON DISK SAMPLING

An excellent example of a nonuniform distribution of sample locations is found in the human eye. The eye has a limited number of photoreceptors, and, like any other sampling process, it has a Nyquist limit. Yet our eyes are not normally prone to aliasing [25]. In the fovea, the cells are tightly packed in a hexagonal pattern, and aliasing is avoided because the lens acts as a low-pass filter. Outside of the fovea, however, the cells are further apart and thus the sampling rate is lower, so we might expect to see aliasing artifacts. In this region, aliasing is avoided by a nonuniform distribution of the cells.

The distribution of cones in the eye has been studied by Yellott [27]. Figure 3a is a picture of the distribution of cones in an extrafoveal region of the eye of a rhesus monkey, which has a photoreceptor distribution similar to that in the human eye. Yellott took the optical Fourier transform of this distribution, with the result shown in Figure 3b. This distribution is called a *Poisson disk distribution*, and it is shown schematically in the frequency domain in Figure 4b. There is a spike at the origin (the dc component) and a sea of noise beyond the Nyquist

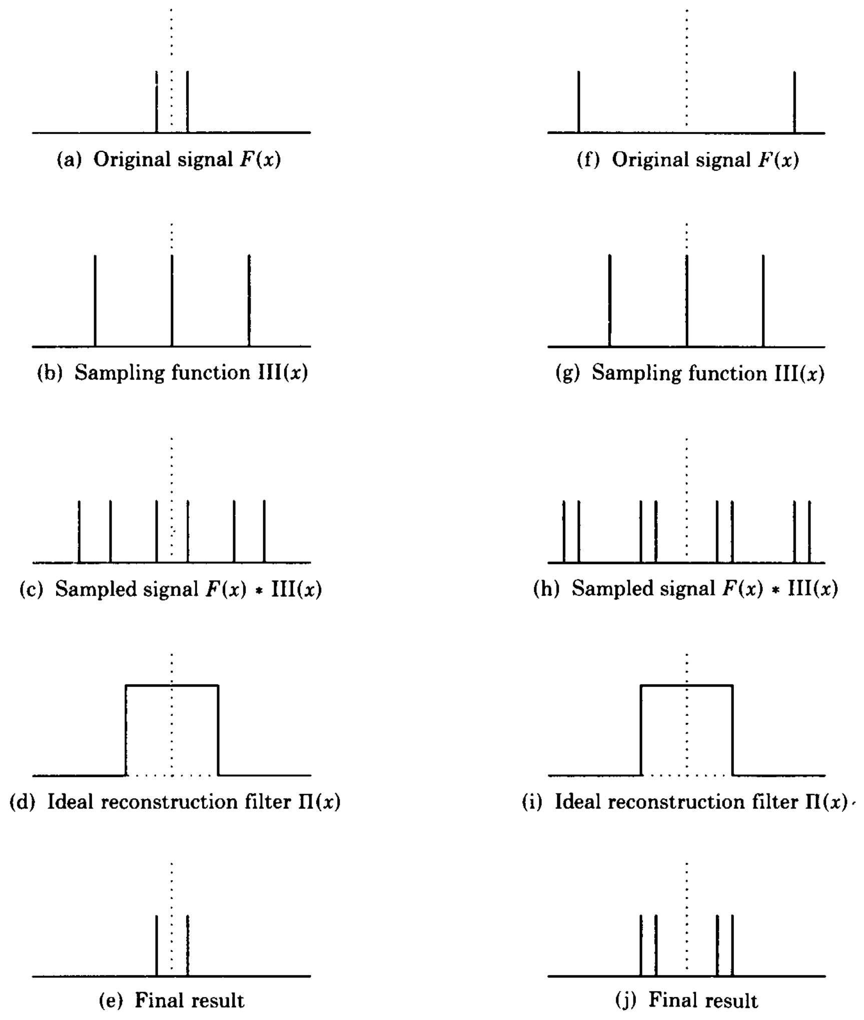

(a) Original signal $F(x)$
(f) Original signal $F(x)$
(b) Sampling function $\mathrm{III}(x)$
(g) Sampling function $\mathrm{III}(x)$
(c) Sampled signal $F(x) * \mathrm{III}(x)$
(h) Sampled signal $F(x) * \mathrm{III}(x)$
(d) Ideal reconstruction filter $\Pi(x)$
(i) Ideal reconstruction filter $\Pi(x)$
(e) Final result
(j) Final result

Fig. 2. Point sampling shown in the frequency domain. The original signal $F(x)$ is convolved with the sampling grid $\mathrm{III}(x)$, and the result is multiplied by an ideal reconstruction filter $\Pi(x)$. The process is shown for a sine wave with a frequency below the Nyquist limit in (a)–(e) and above the Nyquist limit in (f)–(j).

limit. In effect, the samples are randomly placed with the restriction that no two samples are closer together than a certain distance.

Now let us analyze point sampling using a Poisson disk sampling distribution instead of a regular grid. Figure 4a shows a signal that is a single sine wave whose frequency is below the Nyquist limit. Convolution with the Poisson sampling grid of Figure 4b produces the spectrum in Figure 4c. The ideal reconstruction filter of Figure 4d would extract the original signal, Figure 4e. Figure 4f shows a sine wave whose frequency is above the Nyquist limit. Convolution with the Poisson sampling grid produces the spectrum in Figure 4h. An ideal reconstruc-

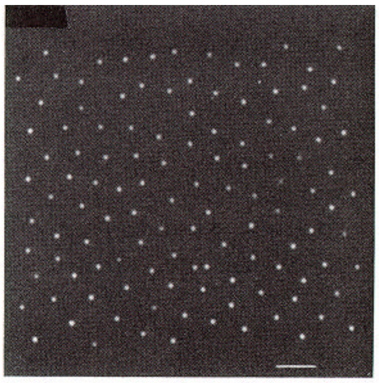


Fig. 3a. Monkey eye photoreceptor distribution.

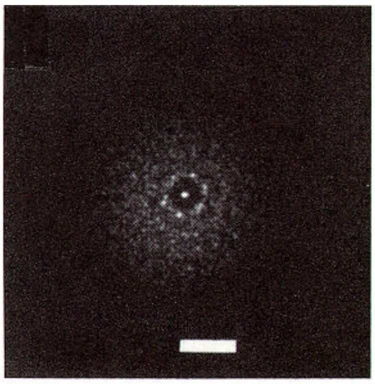


Fig. 3b. Optical transform of monkey eye.

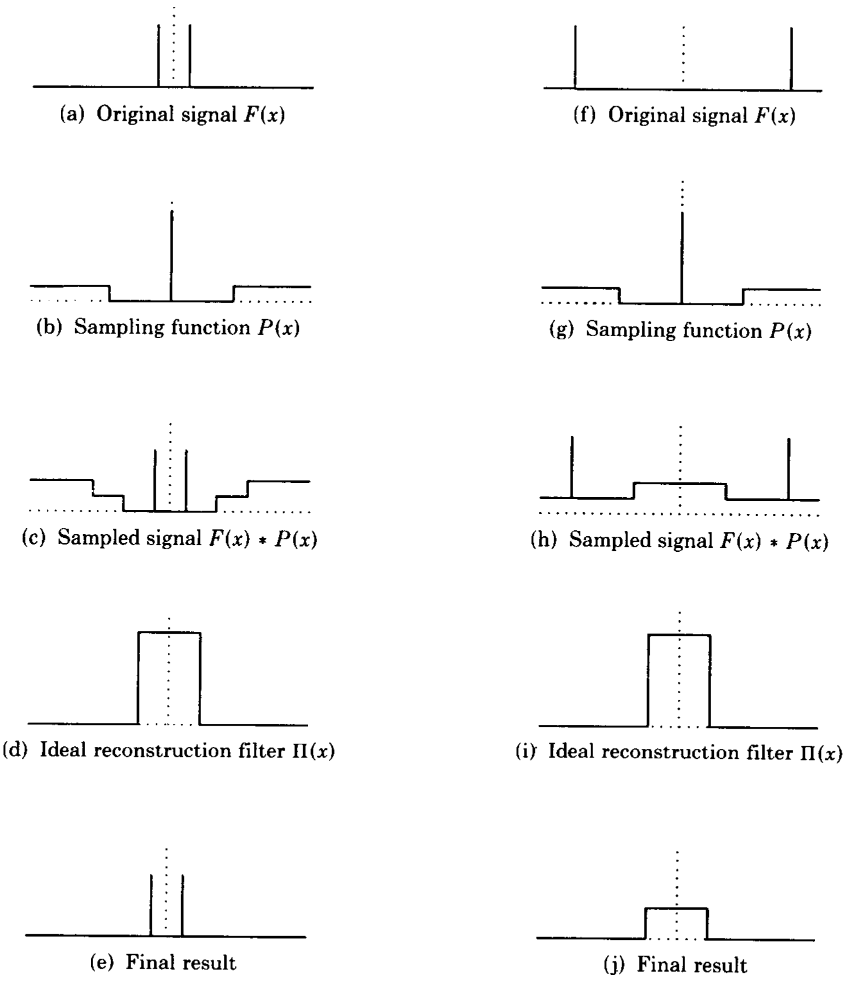

(a) Original signal $F(x)$
(f) Original signal $F(x)$
(b) Sampling function $P(x)$
(g) Sampling function $P(x)$
(c) Sampled signal $F(x) * P(x)$
(h) Sampled signal $F(x) * P(x)$
(d) Ideal reconstruction filter $\Pi(x)$
(i) Ideal reconstruction filter $\Pi(x)$
(e) Final result
(j) Final result

Fig. 4. Poisson disk sampling shown in the frequency domain.

tion filter would extract noise, as shown in Figure 4j. This noise replaces the aliasing of Figure 2j.

The minimum distance restriction decreases the magnitude of the noise. For example, film grain appears to have a random distribution [23], but without the minimum distance restriction of a Poisson disk distribution. With a purely random distribution, the samples tend to bunch up in some places and leave large gaps in other places. Film does not alias, but it is more prone to noise than the eye.

One possible implementation of Poisson disk sampling to image rendering is straightforward, though expensive. A lookup table is created by generating random sample locations and discarding any locations that are closer than a certain distance to any of the locations already chosen. Locations are generated until the sampling region is full. Filter values that describe how each sample affects the neighboring pixels are calculated, and these filter values must be normalized. The locations and filter values are stored in a table. This method would produce good pictures, but it would also require a large lookup table. An alternative method, jittering a regular grid, is discussed in the next section.

4. JITTERING A REGULAR GRID

4.1 Theory

Jittering, or adding noise to sample locations, is a form of stochastic sampling that can be used to approximate a Poisson disk distribution. There are many types of jitter; among these is additive random jitter, which can eliminate aliasing completely [22]. But the discussion in this paper is limited to one particular type of jitter: the jittering of a regular grid. This type of jitter produces good results and is particularly well suited to image-rendering algorithms.

The Fourier transform of a jittered grid (shown later in Figure 11b) is similar to the Fourier transform of a Poisson disk distribution (shown in Figure 4b). An analysis like that in Figures 2 and 4 shows that the results are not quite so good as those obtained with Poisson disk sampling. The images are somewhat noisier and some very small amount of aliasing can remain. We now look at this noise and aliasing quantitatively.

Jitter was analyzed in one dimension (time) by Balakrishnan [2], who calculated the effect of *time jitter*, in which the nth sample is jittered by an amount ζ_n so that it occurs at time $nT + \zeta_n$, where T is the sampling period (see Figure 5a). If the ζ_n are uncorrelated, Balakrishnan reports that jittering has the following effects:

—High frequencies are attenuated.

—The energy lost to the attenuation appears as uniform noise. The intensity of the noise equals the intensity of the attenuated part of the signal.

—The basic composition of the spectrum otherwise does not change.

Sampling by itself cannot be regarded as a filter, because sampling is not a linearly shift-invariant process. Balakrishnan showed, however, that the combination of jittered sampling plus an ideal reconstruction filter is a linearly shift-invariant process, even though the sampling by itself is not [2], so it is in this context that we can talk about frequency attenuation.

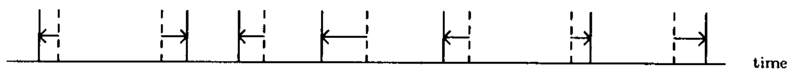


Fig. 5a. Time jitter. Regularly spaced sample times are shown as dashed lines, and the corresponding jittered times are shown as solid lines. Each sample time is jittered by an amount ζ so that the nth sample occurs at time $nT + \zeta_n$ instead of at time nT, where T is the sample period.

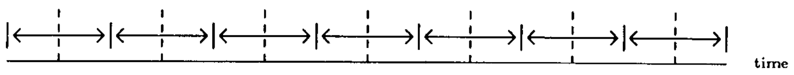


Fig. 5b. White noise jitter for $\gamma = 0.5$. Regularly spaced samples, shown as dashed lines, are jittered so that every time has an equal chance of being sampled.

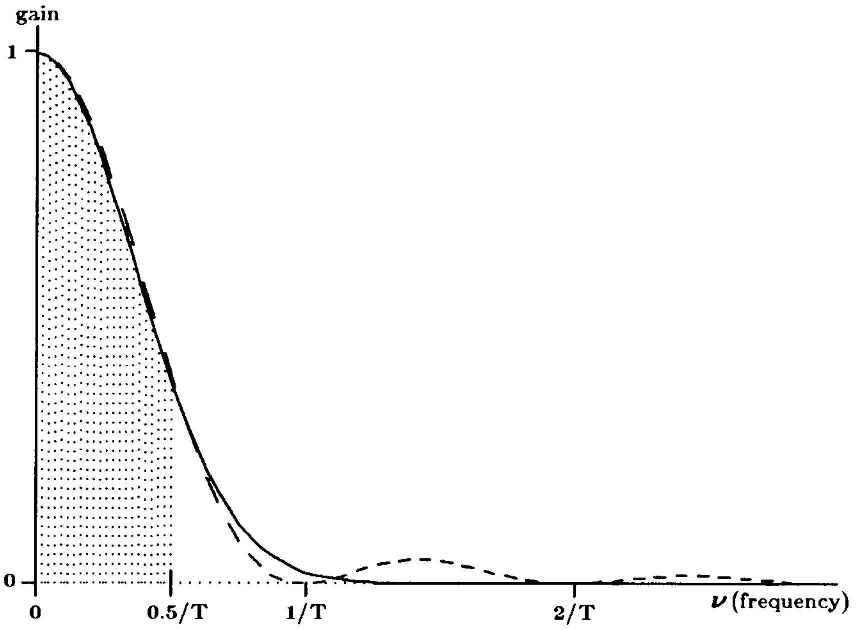


Fig. 6. Attenuation due to jitter. The broken line shows the filter for white noise jitter, the solid line for Gaussian jitter. The shaded area is inside the Nyquist limit.

Uncorrelated jitter is jitter in which any two jitter amounts ζ_n and ζ_m are uncorrelated. Balakrishnan analyzed two types of uncorrelated jitter: *Gaussian jitter* and *white noise jitter.* For Gaussian jitter, the values of ζ are chosen according to a Gaussian distribution with a variance of σ^2. The gain as a function of frequency ν is then

$$e^{-(2\pi\nu\sigma)^2}. \tag{1}$$

This function is plotted with a solid line in Figure 6 for $\sigma = T/6.5$. With white noise jitter, the values of ζ are uniformly distributed between $-\gamma T$ and γT (see

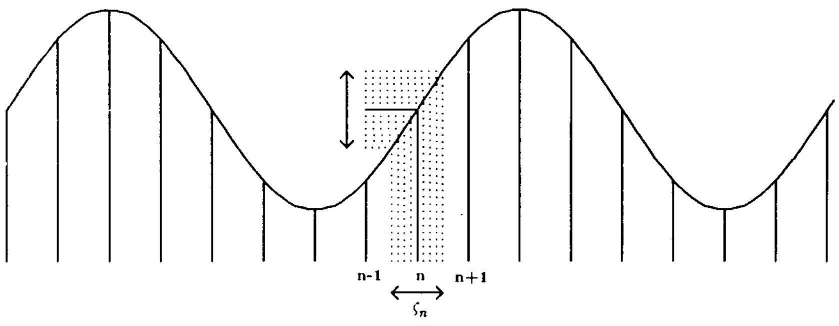


Fig. 7a. The effect of white noise jitter on a sine wave with a frequency below the Nyquist limit. Sample n occurs at a random location in the dotted region. The jitter indicated by the horizontal arrow results in a sampled value that can vary by the amount indicated by the vertical arrow.

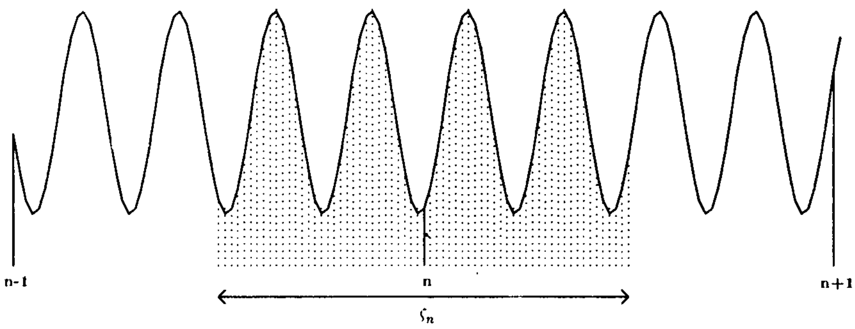


Fig. 7b. The effect of white noise jitter on a sine wave with a frequency above the Nyquist limit. The jitter indicated by the horizontal arrow results in a sampled value that is almost pure noise.

Figure 5b). The gain in this case is

$$\left[\frac{\sin(2\pi\gamma\nu T)}{2\pi\gamma\nu T}\right]^2, \tag{2}$$

as shown with a dashed line in Figure 6 for $\gamma = \frac{1}{2}$.

From this we can see that jittering a regular grid does not eliminate aliasing completely, but it does reduce it substantially. The Nyquist limit of $0.5/T$ is indicated in the figure by the shaded area. Notice that the width of the filter can be scaled by adjusting γ or σ. This gives control of the trade-off between decreased aliasing and increased noise.

For an intuitive explanation of these equations, consider the sine wave shown in Figure 7a, with samples at regularly spaced intervals λ as shown. These samples are inside the Nyquist limit and therefore sample the sine wave properly. Jittering the location of each sample n by some ζ_n in the range $-\lambda/2 < \zeta_n < \lambda/2$ is similar to adding some noise to the amplitude; note that the basic sine wave frequency is not lost. This noise is less for sine waves with a lower frequency relative to the sampling frequency.

Now consider the sine wave shown in Figure 7b. Here the sampling rate is not sufficient for the frequency of the sine wave, so regularly spaced samples can alias. The jittered sample, however, can occur at any amplitude. If there are exactly a whole number of cycles in the range $-\lambda/2 < \zeta_n < \lambda/2$, then the amplitude that we sample is random, since there is an equal probability of sampling each part of the sine wave. In this case none of the energy from the sine wave produces aliasing; it all becomes noise. This corresponds to the zero points of the dashed line in Figure 6. If the sine wave frequency is not an exact multiple of λ, then some parts of the wave will be more likely to be sampled than others. In this case there is some attenuated aliasing and some noise because there is some chance of hitting each part of the wave. This attenuation is greater for higher frequencies because with more cycles of the wave there is less preference for one part of the wave over another. Note also that the average signal level of the noise (the dc component or gray level) is equal to the average signal level of the sine wave. The gray level of the signal is preserved.

4.2 Implementation

The extension of jittering to two dimensions is straightforward. Consider a pixel as a regular grid of one or more rectangular *subpixels*, each with one sample point. Each sample point is placed in the middle of a subpixel, and then noise is added to the x and y locations independently so that each sample point occurs at some random location within its subpixel.

Once the visibility at the sample points is known, the sample values are filtered with a reconstruction filter and resampled on a regular grid of pixel locations to obtain the pixel values. How to do this reconstruction properly is an open problem. The easiest reconstruction filter to compute is a box filter. Each pixel value is obtained by simply averaging the sample values in that pixel. Weighted reconstruction filters with wider filter kernels give better variance reduction. In this case the filter values are a function of the position of each sample point relative to the surrounding pixels. The value of each pixel is the sum of the values of the nearby sample points multiplied by their respective filter values; this total is normalized by dividing by the total of the filter values.

If the random components of the sample locations are small compared with the width of the filter, the effect of the random components on the filter values can usually be ignored. The filter values can then be calculated in advance for the regularly spaced grid locations. These filter values can be prenormalized and stored in a lookup table. Changing filters is simply a matter of changing the lookup table.

5. DISTRIBUTED RAY TRACING

In the previous section, we applied stochastic sampling to the two-dimensional distribution of the sample points used for determining visibility in a z buffer or ray-casting algorithm. But the intensity of a pixel on the screen is an analytic function that may involve several nested integrals: integrals over time, over the pixel region, and over the lens area, as well as an integral of reflectance times illumination over the reflected hemisphere and an integral of transmittance times illumination over the transmitted hemisphere. These integrals can be tremendously complicated.

Image-rendering algorithms have made certain simplifying assumptions in order to avoid the evaluation of these integrals. But the evaluation of these integrals is essential for rendering a whole range of fuzzy phenomena, such as penumbrae, blurry reflections, translucency, depth of field, and motion blur. Thus image rendering has usually been limited to sharp shadows, sharp reflections, sharp refractions, pinhole cameras, and instantaneous shutters. Recent exceptions to this are the radiosity method [10] and cone tracing [1].

The rendering integrals can be evaluated with stochastic sampling. If we regard the variables of integration as additional dimensions, we can perform a Monte Carlo evaluation of the integrals by stochastically distributing the sample points (rays) in those additional dimensions. This is called probabilistic or distributed ray tracing.

—Distributing reflected rays according to the specular distribution function produces gloss (blurry reflection).
—Distributing transmitted rays produces translucency (blurry transparency).
—Distributing shadow rays through the solid angle of each light source produces penumbrae.
—Distributing ray origins over the camera lens area produces depth of field.
—Distributing rays in time produces motion blur.

Distributed ray tracing is discussed in detail in a previous paper [5], and others have extended the results found there [7, 12, 14] (also personal communications from D. Mitchell and from T. Whitted). This section summarizes the distributed ray-tracing algorithm from the viewpoint of stochastic sampling.

5.1 Nonspatial Jittering

One way to distribute the rays in the additional dimensions is with uncorrelated random values. For example, one could pick a random time for each ray or a random point on a light source for each shadow ray. This approach produces pictures that are exceedingly noisy, owing to the bunching up of samples (as illustrated later in Figure 11d). We can reduce the noise level by using a Poisson disk distribution, ensuring that the samples do not bunch up or leave large gaps that are unsampled. As before, we use jittering to approximate a Poisson disk distribution.

To jitter in a nonspatial dimension, we use randomly created prototype patterns in screen space to associate the sample points with a range of that dimension to sample, then jitter to pick the exact location within each range. In the case of sampling in time to produce motion blur, we divide the frame time into slices and randomly assign a slice of time to each sample point. The exact time within each slice is then determined by jittering.

For example, to assign times in a pixel with a 4-by-4 grid of sample points, one could use a random distribution of the numbers 1–16, such as the one shown in

Fig. 8. Example of a prototype time pattern.

7	11	3	14
4	15	13	9
16	1	8	12
6	10	5	2

Figure 8. The sample in the *x*th column and the *y*th row would have a prototype time

$$t_{xy} = \frac{P_{xy} - 0.5}{16},$$

where P_{xy} is the value shown in the *x*th column and the *y*th row of the prototype pattern in Figure 8. A random jitter of $\pm\frac{1}{32}$ is then added to this prototype time to obtain the actual time for a sample. For example, the sample in the upper left subpixel would have a time $\frac{6}{16} \leq t \leq \frac{7}{16}$.

Note that correlation between the spatial locations and the locations in other dimensions can cause aliasing. For example, if the samples on the left side of the pixel are consistently at an earlier time than those on the right side of the pixel, an object moving from right to left might be missed by every sample, whereas an object moving from left to right might be hit by every sample.

5.2 Weighted Distributions

Sometimes we need to weight the samples. For example, we may want to weight the reflected samples according to the specular reflection function, or we may want to use a weighted temporal filter. One approach would be to distribute the samples evenly and then later weight each ray according to the filter. A better approach is *importance sampling* [11], in which the sample points are distributed so that the chance of a location being sampled is proportional to the value of the filter at that location. This avoids the multiplications necessary for the weighting and also puts the samples where they will do the most good.

In order to use jitter to do importance sampling, we divide the filter into regions of equal area, as shown in Figure 9. Each region is sampled by one sample point, with the samples spaced further apart for smaller filter values and closer together for larger filter values. Each sample point is positioned at the center of its region and then jittered to a random location in the region. Note that the size of the jitter varies from sample to sample. If the filter shape is known ahead of time, a list of the centers and jitter magnitudes for each region can be precomputed and stored in a lookup table.

For example, for the reflection ray, we create a lookup table based on the specular reflection function. Given the angle between the surface normal and the incident ray, this lookup table gives a range of reflection angles plus a jitter magnitude for determining an exact reflection angle within that range. For any given reflection ray, the index into this table is determined using its ancestral

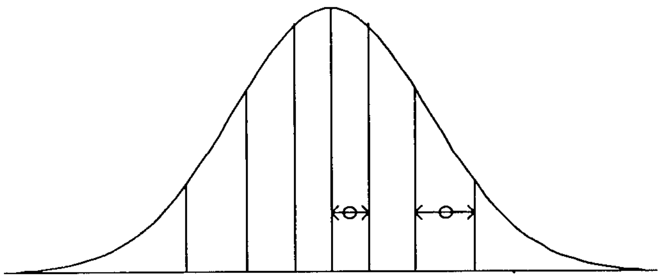


Fig. 9. Importance sampling. The samples are distributed so that they sample regions of equal area under the weighting function. The prototype sample location and jitter range is shown for two of the sampling regions.

primary ray in screen space to associate it with a randomly generated prototype pattern of table indices.

5.3 Summary of Distributed Ray Tracing

The distributed ray-tracing algorithm is illustrated in Figure 10. For each primary ray:

—Determine the spatial location of the ray by jittering.

—Determine the time for the ray from jittered prototype patterns.

—Move the camera and the objects to their location at that time.

—Determine the focal point by constructing a ray from the eye point (center of the lens) through the screen location of the ray. The focal point is located on this ray so that its distance from the eye point is equal to the focal distance.

—Determine the lens location for the ray by jittering a location selected from a prototype pattern of lens locations.

—The primary ray starts at the lens location and goes through the focal point. Determine the visible point for this ray using standard ray-casting or ray-tracing techniques.

—Trace a reflection ray. The direction of the reflection ray is determined by jittering a set of directions that are distributed according to the specular reflection function. This is done with a lookup table; the lookup table index is based on a screen space prototype pattern that assigns indices to primary rays and their descendants. The reflection direction is obtained from the lookup table and then jittered. The range of the jitter is also stored in the table.

—Trace a transparency ray if the visible object is transparent. The direction of the transparency ray is determined by jittering a set of directions that are distributed according to the specular transmission function.

—Trace the shadow rays. For each light source, determine the location on the light for the shadow ray, and trace a ray from the visible point to that location on the light. The chance of tracing the ray to a location on the light should be proportional to the intensity and projected area of that location as seen from the visible point on the surface.

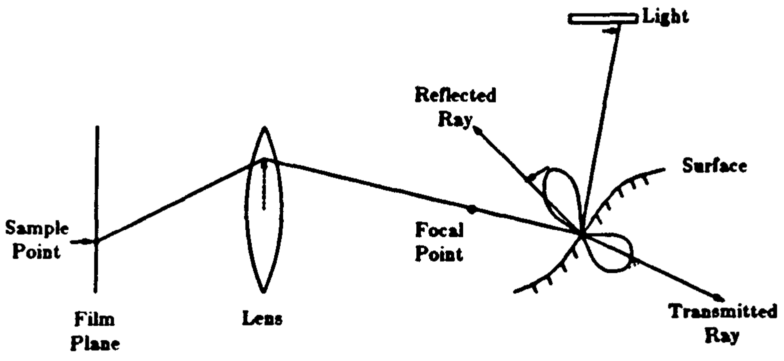


Fig. 10. Distributed ray tracing.

6. EXAMPLES

The jitter used in these examples is white noise jitter with $\gamma = 0.5$. An example of this distribution is shown in Figure 11a, and the Fourier transform of Figure 11a is shown in Figure 11b. Notice how Figure 11b resembles the Fourier transform of a Poisson disk distribution (shown in Figure 4b). By contrast, a pure Poisson distribution of samples with no minimum distance restriction is shown in Figure 11d, and the Fourier transform of Figure 11d is shown in Figure 11e. The C code in Figure 11c was used to generate Figure 11a, and the C code in Figure 11f was used to generate Figure 11d.

In Figures 12 and 13, a box filter was used for a reconstruction filter to accentuate the noise problems. In all of the other examples, the following Gaussian filter was used:

$$e^{-d^2} - e^{-w^2},$$

where d is the distance from the center of the sampling region to the center of the pixel, and $w = 1.5$ is the filter width distance, beyond which the filter was set to zero. The effect of jitter on the filter values was ignored.

Consider the comb of triangular slivers illustrated in Figure 12a. Each triangle is 1.01 pixels wide at the base and 50 pixels high. The triangles are placed in a horizontal row 1.01 pixels apart. If the comb is sampled with a regular grid, aliasing can result as depicted in Figure 12b. A comb containing 200 such triangular slivers is rendered in Figures 12c–f.

In Figure 12c the comb is rendered with a single sample at the center of each pixel. Figure 12d also has one sample per pixel, but the sample location is jittered by $\zeta = \pm\frac{1}{2}$ pixel in x and y. Figure 12c is grossly aliased: there are just a few large triangles spaced 100 pixels apart. This aliasing is replaced by noise in Figure 12d. Because there is only one sample per pixel, each pixel can only be white or black, but in any given region, the percentage of white pixels equals the percentage of that region that is covered by the triangles. Note that the white pixels are denser at the bottom, where the triangles are wider.

In Figure 12e the same comb is rendered with a regular 4-by-4 grid of samples. In Figure 12f the regular 4-by-4 grid is jittered by $\zeta = \pm\frac{1}{8}$ pixel in x and y. Again the regularly spaced samples alias; this time there are a few large overlapping triangles spaced $\frac{100}{4} = 25$ pixels apart. This aliasing is replaced by noise in the

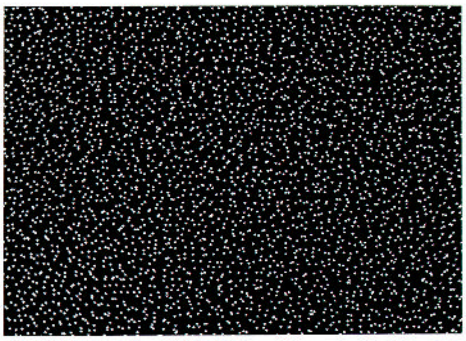


Fig. 11a. Distribution pattern of jittered samples.

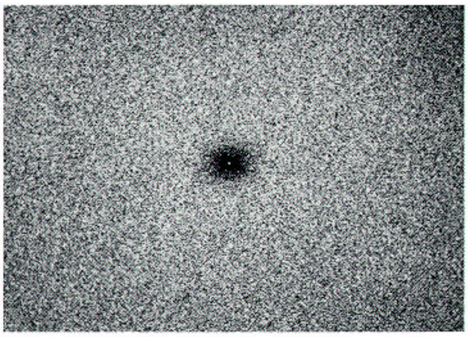


Fig. 11b. Fourier transform of the pattern in Figure 11a.

```
/* Draw a jittered sample pattern in a 512x512 frame buffer.  There is one   */
 * sample in each sample region of 8x8 pixels, for a total of 4096 samples. */
DrawJitterPattern() {
      double Random();                    /* returns a random number in the range 0–1 */
      int x,y;                            /* (x,y) is the corner of the sample region */
      int jx,jy;                          /* (jx,jy) is the jitter */
      for (y=0; y<512; y+=8) {
            for (x=0; x<512; x+=8) {
                  jx = 8*Random();
                  jy = 8*Random();
                  SetPixelToWhite(x+jx,y+jy);
            }
      }
}
```

Fig. 11c. C program that generated the pattern in Fig. 11a.

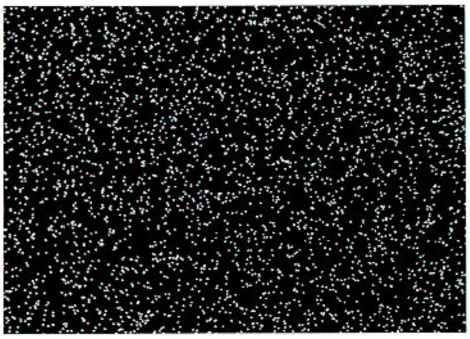


Fig. 11d. Distribution pattern of randomly placed samples.

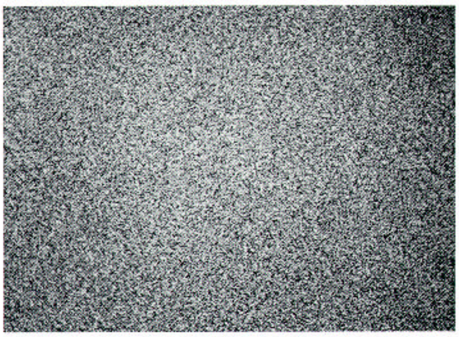


Fig. 11e. Fourier transform of the pattern in Fig. 11d.

```
/* Draw a random sample pattern with 4096 samples. */
DrawPoissonPattern() {
      double Random();                    /* returns a random number in the range 0–1 */
      int n, sx,sy;                       /* (sx,sy) is the sample location */
      for (n=0; n<4096; n++) {
            sx = 512*Random();
            sy = 512*Random();
            SetPixelToWhite(sx,sy);
      }
}
```

Fig. 11f. C program that generated the pattern in Fig. 11d.

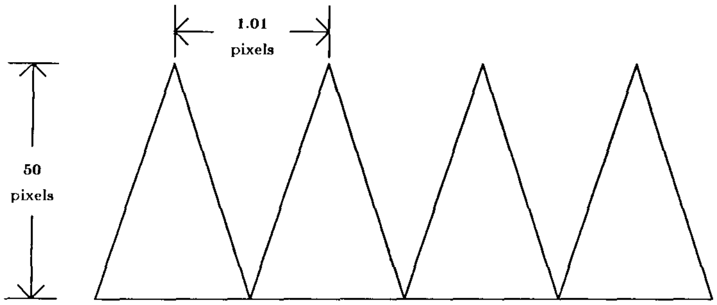


Fig. 12a. Schematic diagram of the comb of triangles example. The triangles are 50 pixels high and 1.01 pixels apart.

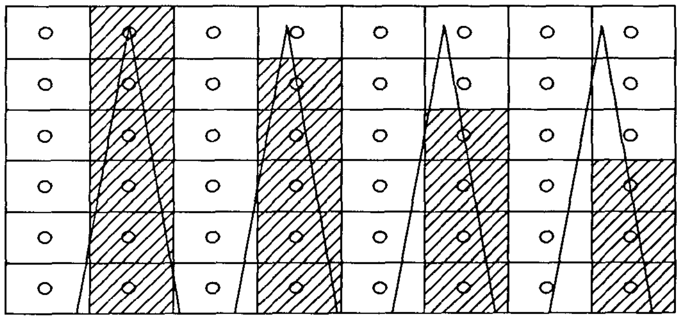


Fig. 12b. The comb of triangles aliases when rendered with a regular grid of sample points in the manner shown here. Samples are shown as circles, and pixels are shown as rectangles. Pixels with samples inside a triangle are shaded.

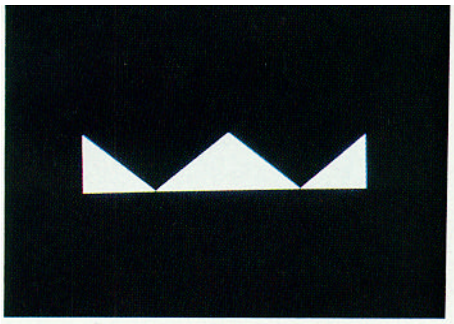


Fig. 12c. Comb rendered with a regular grid, one sample per pixel.

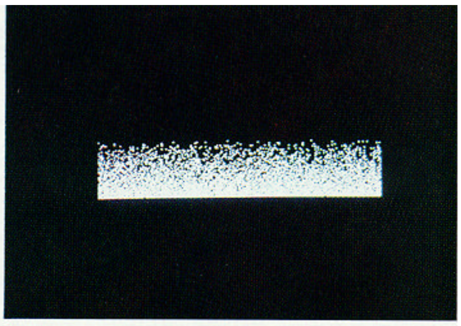


Fig. 12d. Comb rendered with a jittered grid, one sample per pixel.

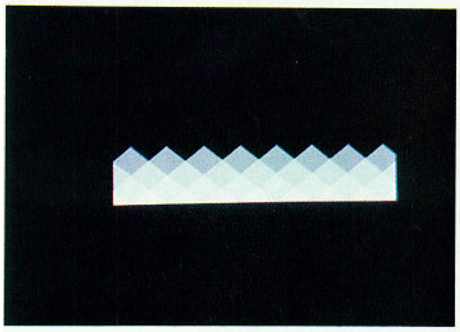


Fig. 12e. Comb rendered with a regular grid, 16 samples per pixel.

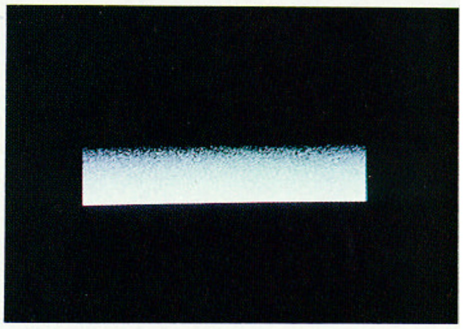
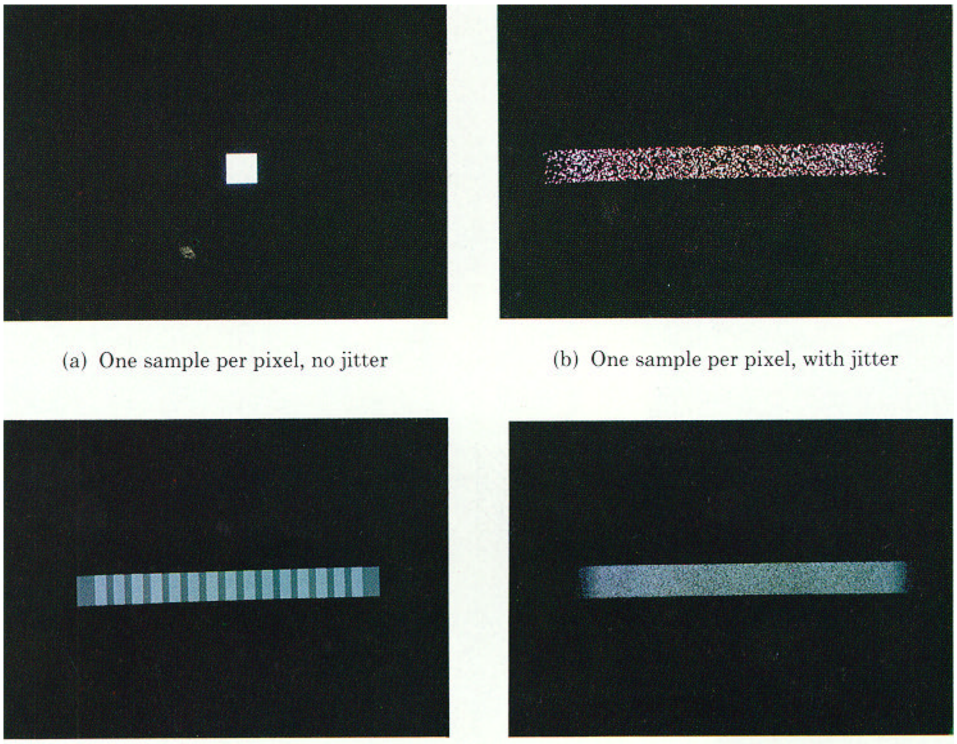


Fig. 12f. Comb rendered with a jittered grid, 16 samples per pixel.



(a) One sample per pixel, no jitter (b) One sample per pixel, with jitter

(c) Sixteen samples per pixel, no jitter (d) Sixteen samples per pixel, with jitter

Fig. 13. Fast-moving polygon.

jittered version, Figure 12f. Notice, though, that the noise is greatly reduced compared with Figure 12d.

Figure 13 shows a small white square moving across the screen. Figure 13a was rendered with no jitter and one sample per pixel, so the image is still. Figure 13b was rendered with jitter and one sample per pixel; the image is now blurred but is extremely noisy because, with only one sample, each pixel can be only one of two colors-the color of the square or the color of the background. Notice, though, that in any given region the number of pixels that are white is proportional to the amount of time the square covered that region; thus the percentage of white pixels is constant in the middle and ramps off at the ends. Figure 13c was rendered with no jitter and 16 samples per pixel, and Figure 13d with jitter and 16 samples per pixel. Notice the reduction in the noise level with the additional samples.

Figure 14a is the ray-traced picture *1984,* with a closeup of the 4-ball shown in Figure 14b. The 4-ball remains stationary for most of the time the shutter is open and moves quickly to the upper right just before the shutter closes. The blur is quite extreme, and yet the image looks noisy instead of aliased. This picture was made with 16 samples per pixel.

Figures 15a and 15b are two frames from the short film *The Adventures of André & Wally B.* [18]. These extreme examples of motion blur were rendered

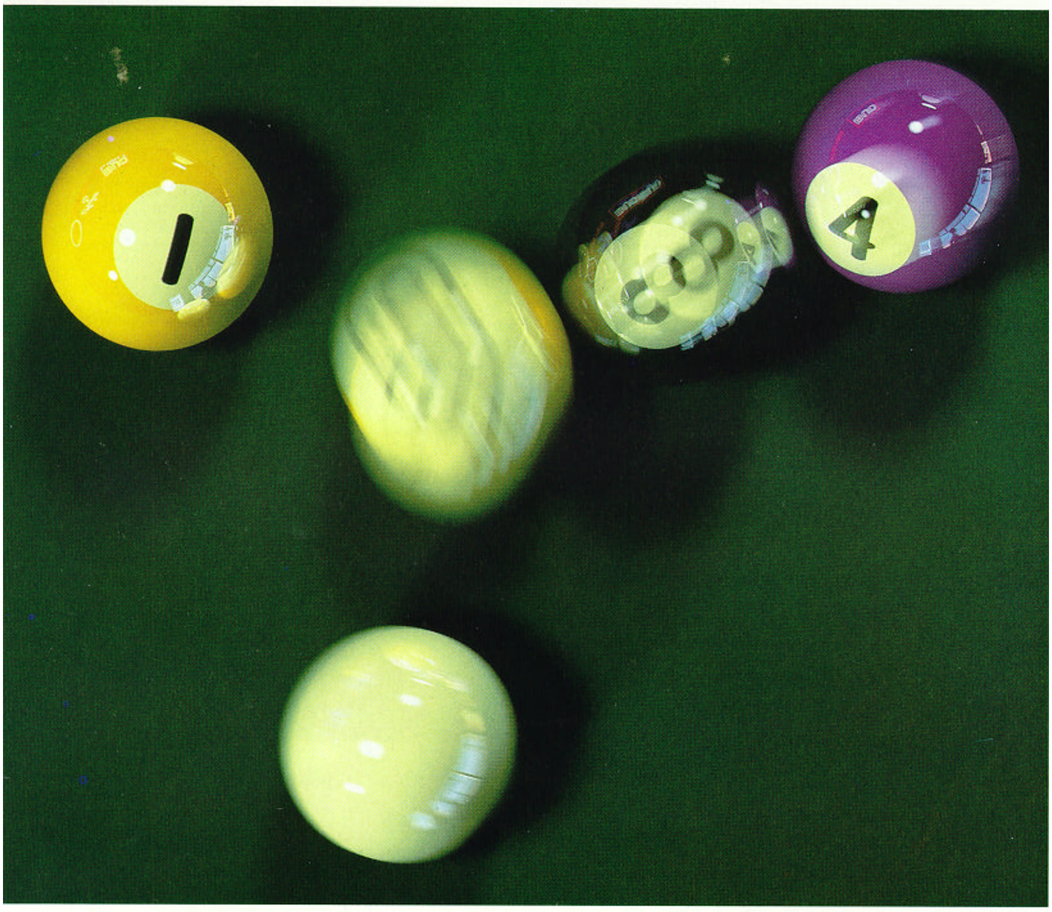


Fig. 14a. *1984*, by Thomas Porter.

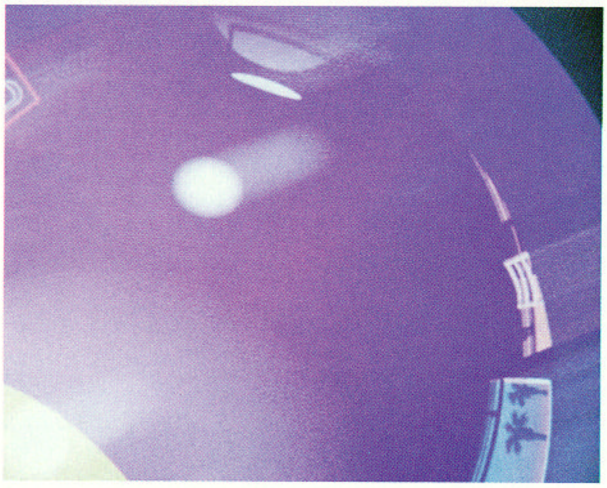


Fig. 14b. Close-up of *1984*.

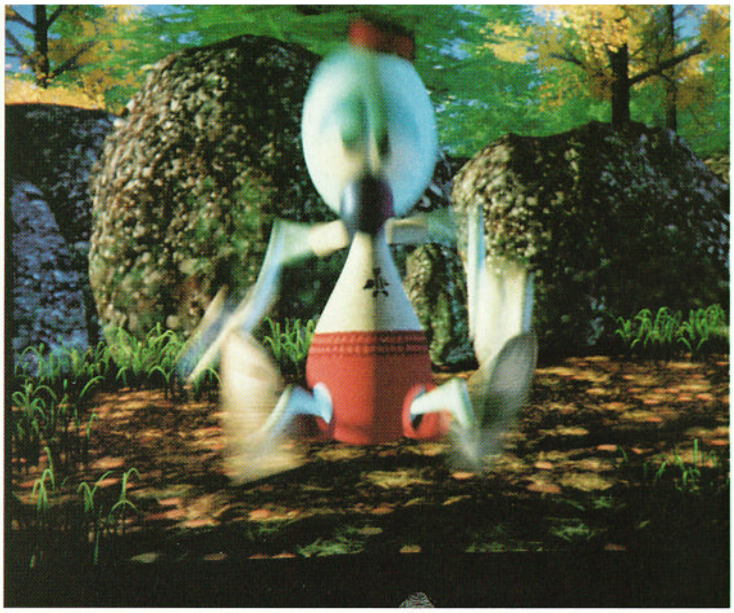


Fig. 15a. Example of motion blur from *The Adventures of André & Wally B.*

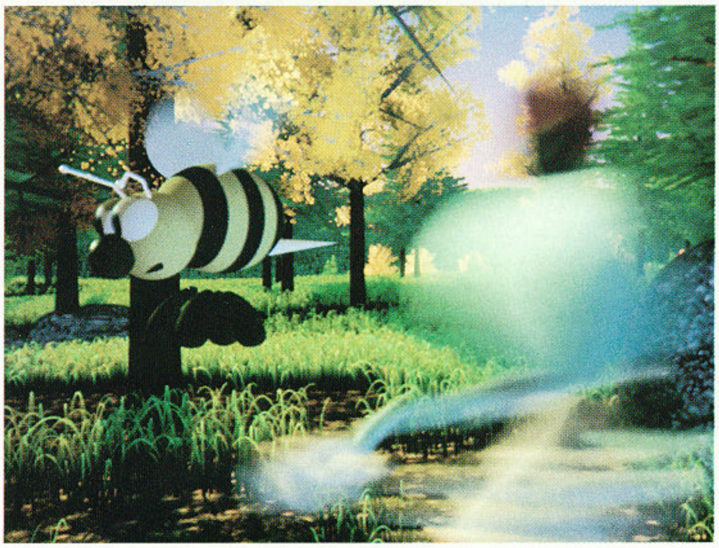


Fig. 15b. Example of motion blur from *The Adventures of André & Wally B.*

with a scan-line algorithm that uses point sampling and a z buffer to determine visibility. In these frames a very simple adaptive method automatically used 16 samples per pixel for most pixels and 64 samples per pixel for pixels that contain objects that move more than 8 pixels in x or y within the frame time.

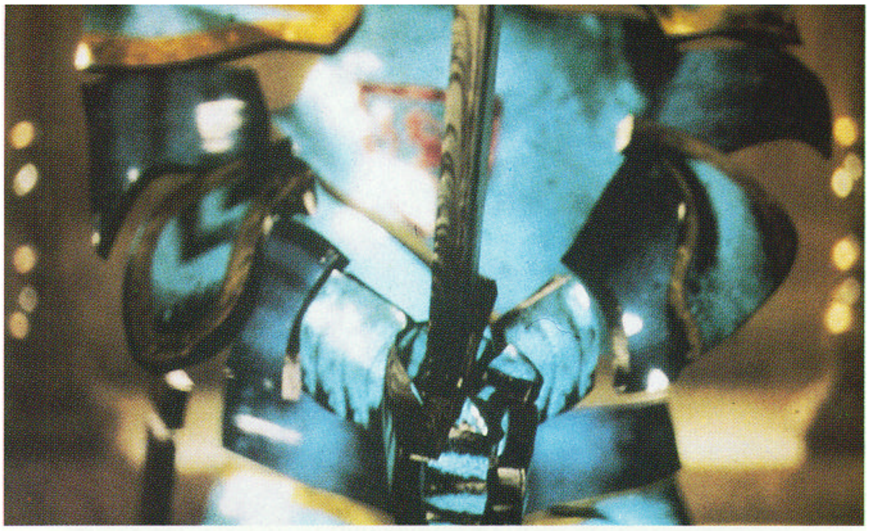


Fig. 16. Example of depth of field from *Young Sherlock Holmes* (Copyright 1985, Paramount Pictures Corp.).

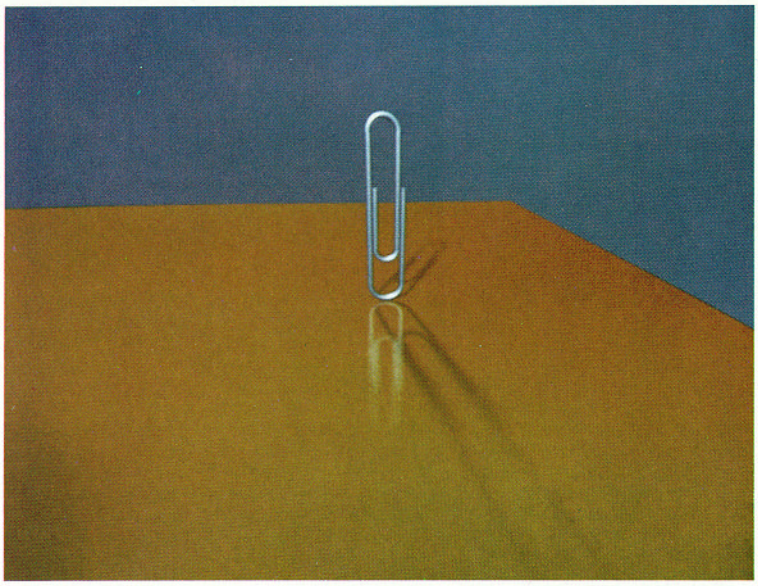


Fig. 17. Example of penumbrae and blurry reflection.

This cuts down considerably on the noise level and helps avoid needless computation. Others have since found ways to add more samples adaptively based on an estimate of the variance of the image in each pixel [12, 14].

Figure 16 shows a frame of a computer-synthesized stained-glass man from *Young Sherlock Holmes* [16]. The camera is focused on the sword, with the body out of focus. This was also rendered with a scan-line algorithm, but in this case, no adaptive method was used to change the number of samples per pixel; instead, there were always 16 samples per pixel. The sequence is also motion blurred.

The paper clip in Figure 17 shows penumbrae and blurry reflection, rendered with 16 samples per pixel. Other examples of distributed ray tracing have appeared in a previous paper [5]. In all cases, areas of extreme blur become noisy instead of aliasing.

7. DISCUSSION AND CONCLUSIONS

With correctly chosen nonuniform sample locations, high frequencies appear as noise instead of aliasing. The magnitude of this noise is determined by the sampling frequency. We have found that using 16 samples per pixel produces an acceptable noise level in most situations, with more needed only for high-frequency situations, such as frames that are extremely motion blurred or out of focus. Stochastic sampling should also work well when integrated with adaptive sampling. This has been the subject of some recent research [12, 14].

The human eye uses a Poisson disk distribution of photoreceptors. A simple and effective approximation to a Poisson disk distribution can be obtained by jittering a regular grid. When this technique is extended to distributed ray tracing, the locations in the nonspatial dimensions can be chosen by jittering randomly generated prototype patterns. Weighted functions can be evaluated using importance sampling.

Stochastic sampling involves some additional computation. Because the samples are not regularly spaced, forward differencing cannot be used to exploit pixel-to-pixel coherence. Compared with standard ray tracing, distributed ray tracing requires additional calculations to move objects to their correct location for each ray. Moving and out-of-focus objects also require a more sophisticated bounding calculation, and these objects must often be intersected with a larger number of rays.

Aliasing has been a major problem for ray-tracing and ray-casting algorithms, and this problem is solved by stochastic sampling. The shading calculations, which have traditionally been point sampled, are automatically antialiased with stochastic sampling, eliminating problems such as highlight aliasing. Another potential application is texture map sampling. Extended to distributed ray tracing, stochastic sampling also provides a solution to motion blur, depth of field, penumbrae, blurry reflections, and translucency.

ACKNOWLEDGMENTS

I would especially like to thank Tom Porter, who made the *1984* picture, suggested the extension of the two-dimensional technique to motion blur, and helped test many of the ideas. Alvy Ray Smith found the article on the distribution of cells in the eye. Andy Moorer and Jim Kajiya helped with the theory, and a number of discussions with Loren Carpenter were invaluable. The idea of dithering sample locations originally came from Rodney Stock, who provided inspiration and motivation for this work. Jack Yellott provided the pictures in Figure 3. Thanks also to the many people at Lucasfilm who made *The Adventures of André & Wally B.* and the stained-glass man sequence from *Young Sherlock Holmes.*

REFERENCES

1. AMANATIDES, J. Ray tracing with cones. *Comput. Graph. 18*, 3 (July 1984), 129–145.
2. BALAKRISHNAN, A. V. On the problem of time jitter in sampling. *IRE Trans. Inf. Theory* (Apr. 1962), 226-236.
3. BLINN, J. F. Computer display of curved surfaces. Ph.D. dissertation, Computer Science Dept., Univ. of Utah, Salt Lake City, 1978.
4. BRACEWELL, R. N. *The Fourier Transform and Its Applications.* McGraw-Hill, New York, 1978.
5. COOK, R. L., PORTER, T., AND CARPENTER, L. Distributed ray tracing. *Comput. Graph. 18*, 3 (July 1984), 137–145.
6. CROW, F. The use of greyscale for improved raster display of vectors and characters. *Comput. Graph. 12*, 3 (Aug. 1978), 1–5.
7. DIPPE, M. A. Z., AND WOLD, E. H. Antialiasing through stochastic sampling. *Comput. Graph. 19*, 3 (July 1985), 69–78.
8. FEIBUSH, E., LEVOY, M., AND COOK, R. L. Synthetic texturing using digital filtering. *Comput. Graph. 14*, 3 (July 1980), 294–301.
9. GARDNER, G. Y. Simulation of natural scenes using textured quadric surfaces. *Comput. Graph. 18*, 3 (July 1984), 11–20.
10. GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. Modeling the interaction of light between diffuse surfaces. *Comput. Graph. 18*, 3 (July 1984), 213–222.
11. HALTON, J. H. A retrospective and prospective survey of the Monte Carlo method. *SIAM Rev. 12*, 1 (Jan. 1970), 1–63.
12. KAJIYA, J. T. The rendering equation. *Comput. Graph. 20*, 4 (Aug. 1986), 143-150.
13. KAY, D. S., AND GREENBERG, D. P. Transparency for computer synthesized images. *Comput. Graph. 13*, 2 (Aug. 1979), 158–164.
14. LEE, M. E., REDNER, R. A., AND USELTON, S. P. Statistically optimized sampling for distributed ray tracing. *Comput. Graph. 19*, 3 (July 1985), 61–67.
15. NORTON, A., ROCKWOOD, A. P., AND SKOLMOSKI, P. T. Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space. *Comput. Graph. 16*, 3 (July 1982), 1–8.
16. PARAMOUNT PICTURES CORP. *Young Sherlock Holmes.* Stained glass man sequence by D. Carson, E. Christiansen, D. Conway, R. Cook, D. DiFrancesco, J. Ellis, L. Ellis, C. Good, J. Lasseter, S. Leffler, D. Muren, T. Noggle, E. Ostby, W. Reeves, D. Salesin, and K. Smith. Pixar and Lucasfilm Ltd., 1985.
17. PEARSON, D. E. *Transmission and Display of Pictorial Information.* Pentech Press, London, 1975.
18. PIXAR. *The Adventures of André & Wally B.* By L. Carpenter, E. Catmull, R. Cook, T. Duff, C. Good, J. Lasseter, S. Leffler, E. Ostby, T. Porter, W. Reeves, D. Salesin, and A. Smith. July 1984.
19. POTMESIL, M., AND CHAKRAVARTY, I. Modeling motion blur in computer-generated images. *Comput. Graph. 17*, 3 (July 1983), 389–399.
20. PRATT, W. K. *Digital Image Processing.* Wiley, New York, 1978.
21. ROTH, S. D. Ray casting for modeling solids. *Comput. Graph. Image Process. 18* (1982), 109–144.
22. SHAPIRO, H. S., AND SILVERMAN, R. A. Alias-free sampling of random noise. *SIAM J. 8*, 2 (June 1960), 225–248.
23. SOCIETY OF PHOTOGRAPHIC SCIENTISTS AND ENGINEERS. *SPSE Handbook of Photographic Science and Engineering.* Wiley, New York, 1973.
24. WHITTED, T. An improved illumination model for shaded display. *Commun. ACM 23*, 6 (June 1980), 343–349.
25. WILLIAMS, D. R., AND COLLIER, R. Consequences of spatial sampling by a human photoreceptor mosaic. *Science 221* (July 22, 1983), 385–387.
26. WILLIAMS, L. Pyramidal parametrics. *Comput. Graph. 17*, 3 (July 1983), 1–11.
27. YELLOTT, J. I., JR. Spectral consequences of photoreceptor sampling in the rhesus retina. *Science 221* (July 22, 1983), 382–385.

Received March 1985; accepted June 1986

Siggraph 2005 Course on Interactive Ray Tracing
The RTRT Core

Ingo Wald

This is an excerpt from
"Realtime Ray Tracing and Interactive Global Illumination",
PhD Thesis, Ingo Wald,
Computer Graphics Group, Saarland University.
Full Version available at `http://www.mpi-sb.mpg.de/~wald/PhD`

"Some argue that in the very long term, rendering may best be solved by some variant of ray tracing, in which huge numbers of rays sample the environment for the eye's view of each frame. And there will also be colonies on Mars, underwater cities, and personal jet packs."

Tomas Möller and Eric Haines,
"Real-Time Rendering", 1st edition (page 391)

The overall design decisions of the RTRT/OpenRT framework are described in detail in [Wald04]. To summarize the most important points, we have chosen to only support triangles, to exploit SIMD extensions in a data-parallel way, to optimize for memory and caches, and to use BSP trees as an acceleration structure. In this chapter, we are now going to discuss the actual algorithms and implementation of these topics in more detail.

1 Fast Triangle Intersection in RTRT

Fast ray triangle intersection code has long been an active field of research in computer graphics and has lead to a large variety of algorithms, e.g. Moeller-Trumbore [Möller97, Möller], Glassner [Glassner89], Badouel [Badouel92], Pluecker [Erickson97, Shoemake98], and many others. The RTRT core uses a modified version of the projection method (see below), which has been specially designed to run as fast as possible with single-ray C code, while still being well suited for SSE code.

Essentially, the task of computing a ray-triangle intersection can be described as follows: Given a ray $R(t) = O + tD; t \in (0, t_{max})$[1] (going from its origin O into direction D), and a triangle with vertices A, B and C, determine whether

[1]In practice, rays usually start at $t_{min} = \epsilon$ in order to avoid "self-intersection".

the ray has a valid hit-point $H = R(t_{hit})$ with the triangle, i.e. whether there exists a t_{hit} with $t_{min} \leq t_{hit} \leq t_{max}$ and $R(t_{hit})$ is inside the triangle.

In case of having found a valid hit point, many ray tracers require that the ray-triangle intersection routine also returns the barycentric coordinates (or local surface coordinates) of the hit-point for shading purposes. As these coordinates are often computed anyway in the process of determining the hit-point, we follow this pattern. Note, however, that this is not the case for shadow rays, for which only the boolean yes/no decision is important, and which can be slightly optimized by not storing these coordinates.

1.1 Barycentric Coordinate Tests

While there are many different methods for computing ray-triangle intersections, many of them are based on computing the barycentric coordinates of the hitpoint and using those for determining whether there is a valid intersection or not[2] (e.g. [Badouel92, Shirley03, Glassner89]). In fact, most ray-triangle intersection algorithms (including the one proposed here) follow this general pattern, and are often only variants and different implementations of the same idea.

In order to use barycentric coordinates for computing ray triangle intersections, one fist computes the signed distance t_{plane} along the ray to the plane embedding the triangle. Given the geometric normal $N = (B - A) \times (C - A)$ and a triangle vertex A, this can be computed as $t_{plane} = -\frac{(O-A).N}{D.N}$. The calculated distance t_{plane} is then tested for whether it lies in the interval in which the ray is actually looking for intersections. If not, no valid intersection can occur, and the triangle test returns "no intersection". The triangle normal N is often computed "on the fly". This minimizes storage requirements, but requires a costly vector product.

If this so-called *distance test* has been passed, one has to check whether the ray actually pierces the triangle. To do this, the actual intersection point with the plane is computed as $H = R(t_{plane}) = O + t_{plane}D$, and is then tested whether it actually lies inside the triangle. The barycentric coordinates of H can then be computed in several ways, e.g. by solving the system of equations $H = \alpha A + \beta B + \gamma C$, or geometrically by considering the relative *signed* (!) areas of the triangles ABC, HBC, AHC and ABH.

Once the barycentric coordinates α, β and γ of H are known, one can determine whether H is inside the triangle by and checking whether the conditions

$$0 \leq \alpha \leq 1, \quad 0 \leq \beta \leq 1, \quad 0 \leq \gamma \leq 1$$

are fulfilled. Note that it is sufficient to check whether $\beta \geq 0$, $\gamma \geq 0$ and $\beta + \gamma \leq 1$, which follows from the properties of barycentric coordinates ($\alpha + \beta + \gamma = 1$).

[2]The barycentric coordinates of H are the values α, β and γ for which $\alpha A + \beta B + \gamma C = H, \alpha + \beta + \gamma = 1$. If H is inside the triangle, both α,β and γ are positive.

1.2 Projection Method

The projection method is an optimization of the barycentric coordinate test. It exploits the fact that projecting both triangle ABC and hit-point H into any other plane (except for the planes that are orthogonal to the plane ABC) does not change the barycentric coordinates of H. The computations for calculating the barycentric coordinates can then be optimized by projecting both triangle and hit-point H into one of the 2D coordinate planes (XY-, XZ- or YZ-plane), in which all further computations can be performed in 2D. For reasons of numerical stability, one should project into the plane in which the triangle has maximum projected area. This so-called "projection dimension" corresponds to the dimension in which the normal N has its maximum absolute component.

After projection, all computations can be performed more efficiently in 2D. For example, projecting into the XY plane (i.e. projection dimension is 'Z') yields

$$H' = \alpha A' + \beta B' + \gamma C',$$

where A', B', C' and H' are the projected points of A, B, C, and H, respectively. Substituting $\alpha = 1 - \beta - \gamma$ and rearranging the terms yields

$$\beta(B' - A') + \gamma(C' - A') = H' - A'.$$

This can be solved (e.g. using the Horner scheme), yielding $\beta = \frac{\det|bh|}{\det|bc|}, \gamma = \frac{\det|hc|}{\det|bc|}$, (where $b = C' - A'$, $c = B' - A'$ and $h = H' - A'$). In 2D, this can be expressed quite efficiently as

$$\beta = \frac{b_x h_y - b_y h_x}{b_x c_y - b_y c_x}, \gamma = \frac{h_x c_y - h_y c_x}{b_x c_y - b_y c_x}. \tag{1}$$

In pseudo-code, the projection method usually looks like the following:

```
// calc edges and normal
b = C-A; c = B-A; N = Cross(c,b);

// distance test
t_plane = - Dot((O-A),N) / Dot(D,N);
if (t_plane < Epsilon || t_plane > t_max) return NO_HIT;

// determine projection dimensiondimensions
if (|N.x| > |N.y|)
  if (|N.x| > |N.z|) k = 0; /* X */ else k=2; /* Z */
else
  if (|N.y| > |N.z|) k = 1; /* Y */ else k=2; /* Z */
u = (k+1) mod 3; v = (k+2) mod 3;

// calc hitpoint
H[u] = O[u] + t_plane * D[u];
H[v] = O[v] + t_plane * D[v];
```

```
beta  = (b[u] * H[v] - b[v] * H[u]) / (b[u] * c[v] - b[v] * c[u]);
if (beta < 0) return NO_HIT;

gamma = (c[v] * H[u] - c[u] * H[v]) / (b[u] * c[v] - b[v] * c[u]);
if (gamma < 0) return NO_HIT;

if (beta+gamma > 1) return NO_HIT;

return HIT(t_plane,beta,gamma);
```

1.3 Optimizing the Projection Method

Taking a closer look at the execution pattern of the above mentioned projection method, it becomes obvious that for different executions on the same triangle many values will be recomputed every time: For example, the edges and normal of a triangle will be recomputed for every intersection test with this triangle, and also the result of determining the projection case will always remain the same. These - and other - computations are thus redundant, and can be saved by precomputing and storing them. This saves the costly computations for the normal, and enables to avoid the branches for determining the projection case. Once the normal is known, the two secondary dimensions ($u = (k+1) mod\, 3$ and $v = (k+2) mod\, 3$) can then be determined by a simple table lookup ($intmodulo[5] = \{0,1,2,0,1\}$), without having to perform the two expensive modulo operations.

Note that we do not have to store the full normal: If k is the projection dimension, $N.k$ can never be zero. As such, we can divide the normal N by $N.k$, yielding $N' = \frac{N}{N.k}$. Then $t = \frac{(A-O).N'}{D.N'} = \frac{A.N' - O_u.N'_u - O_v.N'_v - O_k.N'_k}{D_u.N'_u + D_v.N'_v - D_k.N'_k}$. Obviously the values $d = A.N'$, $N'_u = \frac{N_u}{N_k}$ and $N'_v = \frac{N_v}{N_k}$ are constant for each triangle and thus can be precomputed. By definition, N'_k is equal to one, and thus doesn't have to be stored. Furthermore, knowing that $N'_k = 1$ saves two additional multiplications.

The same idea – simplifying the computations and precomputing as many of the terms as possible – can also be applied to the edges: Rearranging the terms for computing β and γ yields

$$\begin{aligned}
\beta &= \frac{1}{b_x c_y - b_y c_x}(b_x H_y - b_x A_y - b_y H_x + b_y A_x) \\
&= \frac{b_x}{b_x c_y - b_y c_x} H_y + \frac{-b_y}{b_x c_y - b_y c_x} H_x + \frac{b_y A_x - b_x A_y}{b_x c_y - b_y c_x} \\
&= K_{\beta y} H_y + K_{\beta x} H_x + K_{\beta d}.
\end{aligned}$$

This equation now depends only on the projected coordinates H_x and H_y of the hit-point H (which can be calculated entirely from N', O and D). After precomputing and storing the constants $K_{\beta y}$, $K_{\beta x}$, and $K_{\beta,d}$, $\beta = K_{b,nu} H_x +$

$K_{b,nv}H_y + K_{b,d}$[3] can be computed quite efficiently. Note that no other values have to be stored for computing β. Obviously, the same procedure works for the second barycentric coordinate, γ. The last one, α then does not require any further storage space, as $\alpha = 1 - \beta - \gamma$.

With these simplifications and precomputations, only very few operations have to be performed during runtime. In the worst case[4], only 10 multiplies, 1 division, and 11 additions are needed for an intersection. If the ray fails already at the distance test, only 4 muls, 5 adds, and 1 division are needed. Neither geometric normal nor the edge vectors have to be stored or computed during intersection.

1.4 Cache-optimized Data Layout

Obviously, preprocessing can save quite some amount of computations. However, as mentioned above this has to be done quite carefully: Due to the high cost of a cache miss, using additional memory for storing precomputed values carries the chance of actually costing more than the operation itself. On the other hand, careful data layout can even simplify the memory access patterns, and can help in prefetching and in reducing cache misses. Using the just mentioned simplifications, all data needed for a triangle intersection can be expressed in only 10 values: 3 floats (d, N'_u, N'_v) for the scaled plane equation, 3 floats each for the two 2D line equations in the u/v plane, and one int (actually only 2 bits) for storing the projection case k.

Note that these 10 values comprise *all* the data required for the triangle test. In fact, with these precomputed values it is not even necessary any more to know the actual vertex positions of the triangle. Though these are still stored somewhere for potential shading purposes (resulting in an actual *increase* in total memory consumption), they do not have to be accessed at all during traversal and intersection.

Since we know the access pattern of the intersection algorithm, we can even store the 10 values in the order in which they are accessed by the CPU to enable even better data access for the CPU. This leads to a very simple data layout for our triangle acceleration structure:

```
struct TriAccel
{
  // first 16 byte half cache line
  // plane:
  float n_u; //!< == normal.u / normal.k
  float n_v; //!< == normal.v / normal.k
```

[3]It is interesting to note that the same three values can also be derived and explained geometrically. In that case, $K_{b,nu}, K_{b,nv}$ and $K_{b,d}$ correspond to the line equation $L_b(u,v) = K_{b,nu}.u + K_{b,nv}.v + K_{b,d} = 1$ of side $b = C' - A'$ (hence the name of the constants), properly scaled such that inserting the third vertex B' into the line equation yields $L_b(B'_x, B'_y) = 0$.

[4]Note that with a good BSP tree, this worst-case cost (a valid intersection) happens quite frequently, as a good BSP tree already avoids most unsuccessful intersection operations, see Table 5.

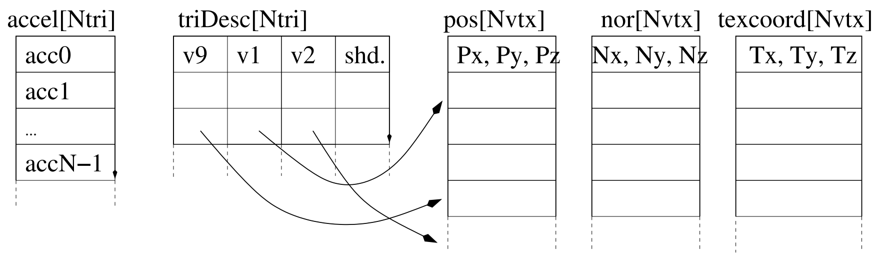


Figure 1: The RTRT core organizes its geometry in the typical "Vertex Array" organization (also called "Indexed Face Sets" in VRML97 terms [Carey97]): Vertices are stored in arrays from where they are referenced by triangles. Each triangle is described by pointers (or IDs in our case) to its three vertices, plus an ID for specifying its shader. The different vertex attributes (e.g. position, normal, texture coordinates etc) are stored in separate lists, thereby allowing to store only those data that have actually be specified by the application. Additionally to this typical data layout, the RTRT core keeps a separate acceleration record for each triangle that stores all data required for an intersection in a preprocessed form. Thus, *neither* ID record nor vertex data is ever touched during traversal and intersection. Whereas typical intersection algorithm require to fetch data from four different, non-cache-aligned memory locations (thereby having to chase the pointers in the ID record), RTRT fetches only this single acceleration data, which lends well to caching and prefetching.

```
  float n_d; //!< constant of plane equation
  int k;    // projection dimension

  // second 16 byte half cache line
  // line equation for line ac
  float b_nu;
  float b_nv;
  float b_d;
  int pad; // pad to next cache line

  // third 16 byte half cache line
  // line equation for line ab
  float c_nu;
  float c_nv;
  float c_d;
  int pad; // pad to 48 bytes for cache alignment purposes
};
```

Though this data layout actually uses *more* memory than other intersection algorithms operating directly on the vertices (like e.g. Moeller-Trumbore [Möller97]), it is likely to use the cache better (see Figure 1): Operating directly on the vertices requires to first access a record that contains the vertex IDs, which require to access at least one cache line. Then accessing the vertices themselves again requires to touch three cache lines, except if the vertices are incidentally stored next to each other. If the index record and/or the vertices straddle cache line boundaries, another four cache lines might be required. In contrast to these up to 8 cache lines, the above structure can be guaranteed to use exactly two cache lines on 32 byte caches, and often only one cache access for 64 byte or 128 byte caches[5].

Furthermore, having all data for the intersection test in one contiguous block also allows for efficient prefetching. Having reached a leaf, prefetching the next triangle before intersecting the current one can guarantee that the next triangle is already in the cache until needed. Finally, having all required data values stored sequentially one after another ideally lends to a streaming-like SIMD implementation.

However, the additional memory overhead can be problematic for extremely complex scenes for which both main memory and address space become quite a limiting factor. For these special cases, the RTRT kernel also contains an efficient implementation of the Moeller-Trumbore algorithm [Möller97] (in both a single-ray C-code as well as in an SSE implementation), which can be used for these cases.

1.5 C Code Implementation for Single-Ray/Triangle Intersection

Writing the code for the just derived intersection algorithm is straightforward, and can be expressed in only a few lines of code:

```
// lookup table for the modulo operation
ALIGN(ALIGN_CACHELINE) static const
unsigned int modulo[] = {0,1,2,0,1};

inline void Intersect(TriAccel &acc,Ray &ray, Hit &hit)
{
#define ku modulo[acc.k+1]
#define ku modulo[acc.k+2]
  // don’t prefetch here, assume data has already been prefetched

  // start high-latency division as early as possible
  const float nd = 1./(ray.dir[acc.k]
     + acc.n_u * ray.dir[ku] + acc.n_v * ray.dir[kv]);
  const float f = (acc.n_d - ray.org[acc.k]
     - acc.n_u * ray.org[ku] - acc.n_v * ray.org[kv]) * nd;

  // check for valid distance.
  if (!(hit.dist > f && f >  EPSILON  )) return;

  // compute hitpoint positions on uv plane
  const float hu = (ray.org[ku] + f * ray.dir[ku]);
```

[5]Intel Pentium-III processors have 32 byte cache lines, whereas AMD Athlon-MPs have 64 bytes, and Intel Pentium IV Xeons have 128 bytes per cache line.

```
  const float hv = (ray.org[kv] + f * ray.dir[kv]);

  // check first barycentric coordinate
  const float lambda = (hu * acc.b_nu + hv * acc.b_nv + acc.b_d);
  if (lambda < 0.0f) return;

  // check second barycentric coordinate
  const float mue = (hu * acc.c_nu + hv * acc.c_nv + acc.c_d);
  if (mue < 0.0f) return;

  // check third barycentric coordinate
  if (lambda+mue > 1.0f) return;

  // have a valid hitpoint here. store it.
  hit.dist = f;
  hit.tri = triNum;
  hit.u = lambda;
  hit.v = mue;
}
```

Note that the costly “modulo 3” operation has been replaced with a precomputed lookup table. The most costly operation in this triangle test is the division at the beginning, which in SSE code can be replaced by a faster reciprocal operation with Newton-Raphson iteration (see e.g. [Intel, AMD]).

Also note that the actual implementation uses a C code “macro” for the intersection code, which (surprisingly) is even faster than an “inline” function as shown above. Instead of the many memory indirections into the origin and direction vectors it is also possible to do a switch-case statement based on *acc.k* at the beginning, and then use hard-coded offset values. The speed difference between these two implementations is small. Depending on the actual CPU used (i.e. Athlon vs. Pentium-III vs. Pentium-IV), sometimes one versions is faster, and sometimes the other.

1.5.1 Single-Ray Intersection Performance

The performance of this optimized implementation is given in Table 1, in which the single-ray C Code version of this triangle test is compared to a fairly optimized implementation of the standard Moeller-Trumbore triangle test [Möller97]. As can be seen, our proposed triangle test in practice is roughly twice as fast as the Moeller-Trumbore code.

1.6 SSE Implementation

By design, the chosen algorithm and data layout naturally lend to SSE implementation. In fact, for our SSE triangle intersection we use exactly the same code and data structures as described above in the previous Section. The only major change is that instead of a single ray, we use a structure that stores four rays together in a SIMD-friendly way (see Figure 2): Opposed to the standard

CPU Cycles	MT	OP	speedup
primary rays	144–172	69–74	2.1–2.3
shadow rays	127–144	68–73	1.9–2.0

Table 1: Performance for the RTRT optimized projection (OP) triangle test algorithm as compared to the Moeller-Trumbore algorithm (MT) [Möller97], measured in CPU cycles on a single 2.5 GHz Pentium-IV notebook. The RTRT code is measured with the single ray C code implementation, *not* with the fast SIMD code described in Section 1.6. Note that these measurements have not been taken with synthetical ray distributions, but correspond to average case performance in typical scenes. The actual cost depends on the probability with which a ray exits at a certain test (e.g. distance test, any of the barycentric coordinate tests, or successful intersection) and as such varies from one scene to another, and also differs for shadow and 'standard' rays (i.e. primary and secondary rays). For the RTRT OP triangle test, these numbers correspond to more than 35 million ray-triangle intersections. Also note that a 2.5GHz notebook CPU is not state of the art any more.

way of storing such four rays as an array of four ray structures (the "AoS" organization), accessing such values efficiently with SSE requires to reorganize such data into a "SoA" (structure of arrays) organization, i.e. first storing the four origin.x values, then the four origin.y values, etc.

Using SSE intrinsics, implementing the above algorithm in SSE is almost straightforward. For example, the line

```
const float hu = (ray.org[ku] + f * ray.dir[ku]);
```

can easily be expressed as

```
const sse_t hu = _mm_add_ps(ray4.org[ku],
                            _mm_mul_ps(f,ray4.dir[ku])).
```

Though converting the whole algorithm in that way is quite simple, the actual code is quite lengthy due to the low-level nature of the SSE operations, and as such is omitted here.

1.6.1 Overhead

A potential source of overhead is that even though some rays may have terminated early, all four rays have to be intersected with a triangle. For coherent rays however this is unlikely. However, not all rays may have found a valid hit, so the hit information may only be updated for rays that actually found an intersection. To achieve this, information on which of the four rays is still active is kept in a bit-field, which can be used to mask out invalid rays in a conditional move instruction when storing the hit point information.

Though this is simple to implement, it results in a considerable overhead, see Table 2: Whereas both shadow rays and 'standard' rays undergo *exactly* the

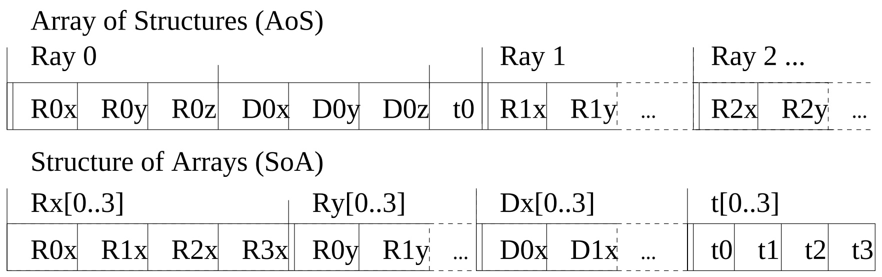

Array of Structures (AoS)
Ray 0
Ray 1
Ray 2 ...
R0x R0y R0z D0x D0y D0z t0 R1x R1y ... R2x R2y ...
Structure of Arrays (SoA)
Rx[0..3]
Ry[0..3]
Dx[0..3]
t[0..3]
R0x R1x R2x R3x R0y R1y ... D0x D1x ... t0 t1 t2 t3

Figure 2: Array-of-structures (AoS) vs. structure-of-arrays (SoA) layout for our ray packets. Each ray consists of origin (R) and direction (D) vectors, as well as its maximum length (t). The same data layout has to be used for the hit point information. While the AoS layout is more natural, efficient SIMD code requires the reorganization to the SIMD-friendly SoA layout. In order to achieve sufficient performance, this layout has to be used during all computations, i.e. already during ray generation.

same floating point computations until the hit/no hit information has been determined, standard rays require several masking operations in order to update the hit information only for those rays that have actually had a valid intersection. Shadow rays have to perform significantly less of these masking operations, as only a single flag has to be stored per ray, in contrast to triangle and instance ID, distance, and barycentric coordinates for normal rays.

CPU Cycles	C Code single ray	SSE 4:1 per packet	SSE 4:1 per ray	speedup	rays per second
primary rays	69–74	101–107	25–27	2.70–2.76	92M–100M
shadow rays	68–73	80–93	20–23	3.17–3.4	108M–125M

Table 2: Cost (in CPU cycles) for our optimized ray-triangle test in a single ray C code implementation and in its data parallel 4:1 SSE implementation. As in Table 1, these numbers correspond to average-case performance in typical scenes. On a 2.5 GHz Pentium IV CPU, 20–27 cycles correspond to 108–125 million ray-triangle intersections per second. Note that the "speedup" is only calculated with respect to the single-ray C code implementation. Comparison to a C code Moeller-Trumbore implementation (see Table 1 would yield a speedup of more than 6.

1.6.2 Performance Results

The overall results of our fast ray-triangle intersection code can be seen in Table 2: Whereas the C Code is already much faster than the Moeller-Trumbore Test (see Table 1), the SSE code achieves an additional, significant speedup: On a 2.5 GHz Pentium-IV CPU, the SSE code for intersecting four rays with a single triangle requires 101–107 CPU cycles, depending on where the code exits.

Amortizing this cost over all four rays results in only 25–27 cycles per intersection. Compared to the C code implementation of the RTRT OP algorithm, this results in a speedup of 2.7–2.8. Compared to the C code Moeller-Trumbore implementation in Table 1, a speedup of more than six can be observed.

As discussed above, shadow rays have significantly less overhead for storing the hit information, and as such are much faster: A shadow ray intersection costs only 80–93 cycles per packet, respectively 20–23 per ray. This once again shows that SSE is extremely efficient for speeding up computations (the actual computations for shadow rays and primary rays are the same), but quickly suffers from any non-computation overhead.

All these measurements have been performed on a 2.5GHz Pentium-IV notebook CPU, on which these numbers correspond to 92–100 million ray triangle intersections for standard rays, and even 108–125 million intersections per second for shadow rays. The overall speedup compared to the single ray C code implementation is around 2.7 for primary and secondary rays, and 3.1–3.4 for shadow rays. This difference clearly shows the impact of the above-discussed overhead for updating the hit information for non shadow rays.

Note that this masking overhead for storing the results might be partially hidden if more than four rays would be intersected in parallel. Generally, operating on larger packet sizes would allow for a more streaming-like approach, in which the latencies of certain operations could be hidden much better. Also note that the application of this data-parallel intersection algorithm is not limited to the RTRT core, but could also be used to accelerate other ray tracing-based rendering algorithms such as memory coherent ray tracing [Pharr97].

2 Fast kd-Tree Traversal

Even before accelerating the triangle test, traversal of the acceleration structure was typically 2-3 times as costly as ray-triangle intersection, as a ray tracer typically performs many more traversal steps than triangle intersections (see Table 5 for statistical traversal data in different scenes[6]). Once the SSE triangle intersection code reduces the intersection cost by more than a factor of three, traversal is the limiting factor in our ray tracing engine. Furthermore, the SSE intersection procedure requires us to always have four rays available anyway. Therefore, we need an algorithm for efficiently traversing four rays through an acceleration structure in parallel.

As already discussed earlier on in this course, a wide variety of ray tracing acceleration schemes have been developed over the last two decades. For example, there are octrees, general BSP-trees, axis-aligned BSP-trees, uniform, non-uniform and hierarchical grids, ray classification, bounding volume hierarchies, and several hybrids of several of these methods. As already discussed in [Wald04], we have chosen to use axis-aligned BSP trees (kd-trees) for the RTRT core. Their traversal code is quite simple, and can very well be implemented in a highly optimized form. Furthermore, BSP trees usually perform at

[6]Similar data hold for different acceleration structures, see [Havran01].

least comparable to other techniques [Havran00, Havran01], and are well-known for their robustness and applicability for a wide range of scenes. However, our main reason for using a BSP tree in the RTRT core is the simplicity of the traversal code, which allows for efficiently traversing packets of rays in parallel: Traversing a node is based on only two binary decisions, one for each child, which can efficiently be done for several rays in parallel using SSE. If any ray needs traversal of a child, all rays will traverse it in parallel.

This is in contrast to algorithms like octrees or hierarchical grids, where each of the rays might take a different decision of which voxel to traverse next. Keeping track of these states is non-trivial and was judged to be too complicated to be implemented efficiently. Bounding Volume Hierarchies have a traversal algorithm that comes close in simplicity to BSP trees, and could also be adapted to a SIMD-traversal method. However, BVHs do not partition *space*, but rather organize the hierarchy. This leads to different parts of the hierarchy overlapping themselves, does not allow for efficiently traversing the voxels in front-to-back order[7], and thus in practice makes BVHs inefficient for complex scenes (for extensive statistical experiments, see [Havran00, Havran01]). Furthermore, algorithms for *building* BVHs that are well-suited for fast traversal are less well investigated than similar algorithms for BSP trees.

2.1 Data Layout of a kd-Tree Node

As mentioned above, the ratio of computation to the amount of accessed memory is very low for scene traversal. This requires us to carefully design the data structure for efficient caching and prefetching.

For a typical BSP node, one has to store

- A flag specifying whether it is a leaf node or an inner node.
- For leaf nodes, an "item list", i.e. a list of integer IDs that specify the triangles in this leaf; consists of a pointer (or index) to the first item in the list, and of the number of items in the list.
- For inner nodes, the addresses of the two children, the dimension of the splitting axis (i.e. x, y, or z), and the location of the split plane.

All these values can be stored in a very compact, unified node layout of only 8 bytes: Obviously, a node can either be a leaf node or an inner node, so they can be stored in the same memory location (a `union` in C code) as long as there is at least one bit reserved for determining the kind of the node.

For inner nodes, we need half the node for storing the float value that specifies the split plane. Addressing the two children can be performed with a single pointer if children of a node are always stored next to each other. Furthermore, if all BSP nodes are stored in one contiguous array (with child nodes always

[7]It is possible to traverse BVHs in front-to-back order by keeping the yet-to-be-traversed parts of the hierarchy organized in a priority queue [Haines91]. This however makes each traversal step considerably more costly than a BSP traversal step

stored after their parent nodes), this single pointer can be expressed as an offset relative to the current node. As this offset is positive, we can use its sign bit for storing the flag that specifies the type of node. Finally, having the nodes stored in an array guarantees that the offset is a multiple of 8 (the node size), so its lower two bits can be safely used for storing the splitting axis.

Leaf nodes can be expressed in quite the same way: The flag that specifies the node type has to remain in place, and the pointer to the start of the item list is stored just like the children pointer, as a relative offset stored in bits 2..30.

This leads to the following simple, compact structure:

```
struct BSPLeaf {
  unsigned int flagDimAndOffset;
  // bits 0..1      : splitting dimension
  // bits 2..30     : offset bits
  // bit  31 (sign) : flag whether node is a leaf
  float splitCoordinate;
};
struct BSPInner {
  unsigned int flagAndOffset;
  // bits 0..30     : offset to first son
  // bit  31 (sign) : flat whether node is a leaf
}
typedef union {
  BSPLeaf  leaf;
  BSPInner inner;
} BSPNode;
```

Note that the exact order and arrangement of the bits has been *very* carefully designed: Each value can be extracted by *exactly* one "bitwise and" operation to mask out the other bits, and does *not* require any costly shift operations for shifting bits to their correct positions.

```
#define ABSP_ISLEAF(n)    (n->flag_k_ofs & (unsigned int)(1<<31))
#define ABSP_DIMENSION(n) (n->flag_k_ofs & 0x3)
#define ABSP_OFFSET(n)    (n->flag_k_ofs & (0x7FFFFFFC))
```

As traversing a BSP node is by far the most common operation in a ray tracer, it has to be implemented with extreme care. For example, an older version of the RTRT kernel originally stored the dimension bits in the *upper* bits of the flag word, from where they could be retrieved by a single shift operation. While this seems comparably cheap, due to this single shift operation (which is quite more costly than a "bitwise and") the old version was roughly 5 percent slower than the current version.

The presented data layout allows for squeezing the whole BSP node description into 8 bytes per node, or 4–16 nodes per cache line[8]. As we always store

[8]Assuming 32 bytes per cache line on a PentiumPro Architecture (Pentium-III), 64 bytes on an AMD Athlon MP, and 128 bytes on an Intel Pentium-IV Xeon. Note that the larger cache sizes on a Xeon CPU might benefit from an improved node packing inside a cache line as discussed in [Havran97, Havran99, Havran01]

both children of a node next to each other, both nodes are stored in the same cache line[9], and are thus always and automatically fetched together.

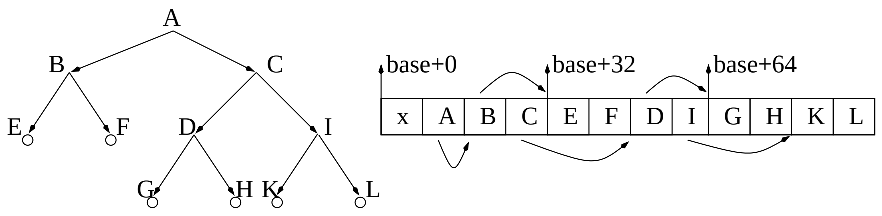


Figure 3: All BSP nodes (inner nodes as well as leaf nodes) in RTRT are stored in one contiguous, cache-aligned array. Depending on cache line size, either 4, 8, or 16 nodes form one cache line. Both children of the same node are always stored next to each other, and thus land in the same cache line. As cache line size is a multiple of node size, node pairs will never overlap a cache line boundary. Both children can be addressed by the same pointer, which is stored as an offset. As this offset is always positive and divisible by four, we can squeeze both node type flag (leaf or inner node) and split dimension (X,Y, or Z) in the sign bit and in the lower two bits, respectively. For leaves, pointers to the item lists (not shown) are stored exactly like pointers to nodes.

Using the same pointer for both node types allows for reducing memory latencies and pipeline stalls by prefetching, as the next data (either a node or the list of triangles) can be prefetched before even processing the current node. Note that though prefetching requires SSE cache control operations, prefetching is also possible for the single-ray, non-SIMD traversal code. Similarly, the benefits of using this optimized node layout, i.e. reduced bandwidth and improved cache utilization, positively affect both the C-code as well as the SSE implementation.

2.2 Fast Single-Ray kd-Tree Traversal

Before describing our algorithm for traversal of four rays in parallel, we first take a look at the traversal of a single ray: In each traversal step, we maintain the *current ray segment* $[t_{near}, t_{far}]$, which is the parameter interval of the ray that actually intersects the current voxel. This ray segment is first initialized to $[0, \infty)$[10], then clipped to the bounding box of the scene, and is updated incrementally during traversal[11]. For each traversed node, we calculate the distance d to the splitting plane defined by that node, and compare that distance to the current ray segment.

[9]In RTRT, all BSP node pairs are aligned to cache line boundaries: All nodes are stored in one consecutive, cache-aligned array, and the cache line size is a multiple of the node size.

[10]In practice, rather to $[\epsilon, t_{max}]$

[11]Instead of clipping to the scene bounding box, it is also possible to not clip at all and rather use six additional BSP planes that represent the bounding box sides. This is typically slower in a software implementation, but can be useful for hardware implementations such as in the SaarCOR architecture

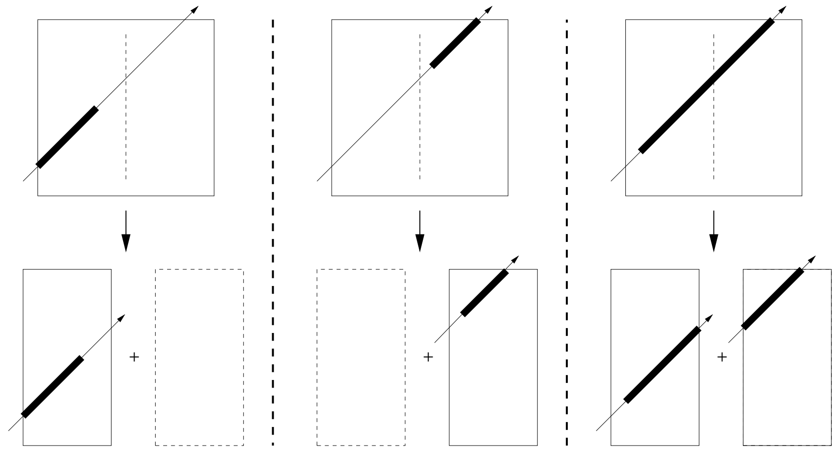


Figure 4: The three traversal cases in a BSP tree: A ray segment is completely in front of the splitting plane (a), completely behind it (b), or intersects both sides (c).

If the ray segment lies completely on one side of the splitting plane (i.e. $d >= t_{far}$ or $d <= t_{near}$), we can "cull" the subtree on the other side and immediately proceed to the corresponding child voxel[12]. If neither side can be culled, one computes the ray parameter at which the plane intersects, and traverses both sides in turn – the first side with ray segment $[t_{near}, d]$, and the second one with $[d, t_{far}]$. This actually leads to three different traversal cases, as depicted in Figure 4.

Basing the traversal entirely on the current ray segments allows for performing all computations in 1D: Only the actual ray parameters for start and end of the segment, as well as distance to the split plane have to be known. Neither the 3D coordinates of the actual entry, exit, or intersection points are required, nor is it necessary to track the current voxel's actual extent[13].

Early ray termination: In the just described implementation, voxels are traversed in front-to-back order, which allows for "early ray termination": If a valid hit point is found *inside* one voxel (i.e. $t_{hit} <= t_{far}$), traversal can be immediately terminated, as all further potential primitive intersections can only be be behind the already found hit point. This early ray termination is actually responsible for the "occlusion culling" feature of ray tracing, and can greatly

[12]Note that using "<=" and ">=" instead of "<" and ">" requires careful programming to correctly handle triangles that lie on the splitting plane. Also not that the exact implementation is quite sensitive to issues such as having rays parallel to the split plane, or rays actually lying inside the split plane. These special cases generate "Infinity"s and "NaN"s during traversal, which need special attention to handle correctly.

[13]This implies that the actual size of the voxel is not known at any time during traversal. Only the current ray segment – i.e. the overlap between the ray and the voxel – is known.

enhance performance. Combined with a high-quality BSP tree (see Section 3), early ray termination can in many scenes lead to an average of *less than two* ray-triangle intersections per ray (see Table 5).

2.2.1 Recursive kd-Tree Traversal

In its most common recursive form, the whole traversal algorithm can be expressed quite simply:

```
void Traverse()
{
  ( t_near,t_far ) = ( Epsilon, ray.t_max );
  ( t_near,t_far ) = Clip(t_near,t_far);
  if (t_near > t_far)
     // ray misses bounding box of object
     return;
  RecTraverse( bspRoot, t_near, t_far );
}

float RecTraverse(node,t_near,t_far)
// returns distance to closest hit point
{
  if (IsLeaf(node)) {
    IntersectAllTrianglesInLeaf(node);
    return ray.t_closest_hit;
      // t_closest_hit initialized to t_max before traversal
  }
  d = (node.split - ray.org[node.dim] / ray.dir[node.dim];
  if (d <= t_near) {
    // case one,  d <= t_near <= t_far -> cull front side
    return RecTraverse(BackSideSon(node),t_near,t_far);
  } else if (d >= t_far) {
    // case two,  t_near <= t_far <= d -> cull back side
    return RecTraverse(FrontSideSon(node),t_near,t_far);
  } else {
    // case three: traverse both sides in turn
    t_hit = RecTraverse(FrontSideSon(node),t_near,d);
    if (t_hit <= d) return t_hit; // early ray termination
    return  RecTraverse(BackSideSon(node),d,t_far);
  }
}
```

2.2.2 Iterative kd-Tree Traversal

Due to the reasons discussed in the previous chapter, a recursive solution is not the best choice for high performance. However, the algorithm can be easily reformulated in an iterative way (see e.g. [Keller98, Havran01]), which in pseudo-code can be written up in only a few lines of code:

```
void Traverse() {
  ( t_near, t_far ) = ( Epsilon, ray.t_max );
  ( t_near, t_far )
      = scene.boundingBox.ClipRaySegment(t_near, t_far);
  node = rootNode;
  if (t_near > t_far)
     // ray misses bounding box of object
     return;
  while (1) {
     while (!node.IsLeaf()) {
        // traverse 'til next leaf
        d = (node.split - ray.org[node.dim]) / ray.dir[node.dim];
        if (d <= t_near) {
           // case one,  d <= t_near <= t_far -> cull front side
           node = BackSideSon(node);
        } else if (d >= t_far) {
           // case two,  t_near <= t_far <= d -> cull back side
           node = FrontSideSon(node);
        } else {
           // case three: traverse both sides in turn
           stack.push(BackSideSon(node),d,t_far);
           ( node, t_far ) = ( FrontSideSon(node), d );
        }
     }
     // have a leaf now
     IntersectAllTrianglesInLeaf(node);
     if (t_far <= ray.t_closesthit)
        return; // early ray termination
     if (stack is empty)
        return; // noting else to traverse any more...
     ( node, t_near, t_far ) = stack.pop();
  }
}
```

Obviously, a realtime kernel requires a very high-performance implementation of this traversal code with many low-level optimizations. For example, this includes precomputation of the “1/ray.dir[dim]” terms, an efficient stack handling, efficient calculation of “FrontSideSon” and “NearSideSon”, careful data layout, and especially efficient handling, organization and ordering of the conditionals. Special emphasis has to be paid on handling all “special cases” – like for example division by zero ray direction (leading to +/- Infinity and NaN values), numerical issues (especially during the comparisons), triangles lying *in* the splitting plane, “flat voxels” leading to zero-length ray segments, etc – in an efficient though nevertheless correct manner. As the discussion of all these implementation details is quite involved, the actual low-level source code is omitted here.

2.3 SIMD Packet Traversal for kd-Trees

As discussed before, efficient use of the SSE instruction set during ray tracing requires to trace packets of several rays in parallel. The algorithm for tracing four different rays is essentially the same as traversing a single one: All four rays are first initialized to $(0, t_{max})$ and clipped to the scene bounding box using fast SSE code. In each traversal step then, SSE operations are used to compute the four distances to the splitting plane and to compare these to the four respective ray segments, all in parallel. If all rays require traversal of the same child, traversal immediately proceeds with this child, without having to change any of the ray segments. Otherwise, we traverse both children, with the ray segments updated accordingly.

As discussed in the previous section, efficient ray tracing requires to traverse the voxel visited by a ray in front-to-back order. However, when tracing several rays at the same time in parallel, the correct traversal order for the packet might be ambiguous, as different rays might demand a different traversal order. In order to get a consistent traversal order for the whole packet, we only allow such rays into the same packet for which the traversal order can be guaranteed to match. This however is easy to guarantee for two common cases, as discussed in more detail below: First, rays starting at the same origin can be shown to never disagree on traversal order, whatever their direction is. Second, rays with the same direction signs in all dimensions will also have the same traversal order at any splitting plane.

2.3.1 Resolving Traversal Order Ambiguities: Same Origin vs. Same Principle Direction

The first case already supports most of the rays in ray tracing, as all primary rays from a pinhole camera, as well as all shadow rays from point light sources fall under this category. However, the computations for determining the traversal order depend on the relation between actual origins of the rays and position of the splitting plane. As such, they have to be performed during each traversal step in the inner loop of the packet traversal code, and as such are quite costly. Furthermore, the operations for computing the respective updated ray segments get relatively complex for this alternative.

The second alternative of only combining rays with matching direction signs on first sight appears more costly: First, each packet of rays has to be checked for matching signs, and rays with non-matching signs either have to split up or require special handling. However, these special cases happen only rarely for coherent rays, which typically have similar directions. Once it is clear that the rays have matching direction signs, the computations in the inner loop get very simple, and can be expressed quite efficiently. In fact, all that is required in the inner loop of the traversal code is a simple XOR with the respective direction sign bit of the first ray. Similar arguments hold for the code computing the respective t_{near}/t_{far} values, which can be expressed quite a bit more efficiently than for the case with common ray origin. As such, the RTRT kernel only supports packets with matching directions signs. Packets are automatically and quickly tested for complying to this rule, and non-complying rays are traced with the fast single-ray traversal code.

2.3.2 Implementation Issues

After restricting the traversal code to packets with matching direction signs, the respective computations get quite simple. The plane distances for all four rays are computed with only one SSE "mult" and one SSE "add", and compared to the four respective t_{near} and t_{far} values with SSE compare instructions[14]. If either *all* ray segments lie in front of the plane, or are all behind the splitting plane (corresponding to cases 'a' and 'b' in Figure 4), the other side is culled, and no special operations have to be performed for the near/far values, nor for the traversal stack. In the case that both sides have to be traversed [15], the respective ray segments get updated to $[t_{min}, min(d, t_{far})]$ for the near side, respectively $[max(d, t_{near}), t_{far}]$ for the far side. The min and max operations are required as not all ray segments may actually have overlapped the splitting plane. These ray segments may obviously not get longer than they have been before.

Note that the "near" and "far" sides of a voxel (with respect to a given ray R) are determined by the order in which a directed infinite line with the same direction as R would cross this line[16]. As such, near and far side are independent of both ray origin and actual BSP plane position, and can be determined once at the start of traversal by the direction signs alone.

Deactivating invalid rays: Rays that get "forced" to traverse a subtree that they would not have traversed had they been traversed alone should obviously not influence any decisions in that subtree. This however can be achieved quite efficiently: Using the SSE min/max for updating the respective ray segments operations as just described, it can be shown easily that rays entering an "invalid" subtree automatically get their ray segments updated to negative length (i.e. $t_{near} > t_{far}$), which can be used to determine which of the rays are still "active" in a subtree. In SSE, this generates hardly any overhead at all: A single SSE compare of t_{near} and t_{far} automatically generates a bit-mask that can be used to mask out any of the latter decision flags in a single operation.

This leads to the following pseudocode for SIMD packet-traversal:

```
void IterativePacketTraverse(ray[4],hit[4]) {
  ( t_near[i], t_far[i] ) = ( Epsilon, ray.t_max );
  // i=0..3 in parallel
```

[14]Note that SSE comparisons are actually not conditionals, but rather generate bit masks that can be used for dependent moves

[15]Note that this case can also happen if *neither* ray wants to traverse both sides, as one ray might want to only traverse the left side, while an other one demands traversal of only the right side.

[16]The "near" side may not be confused with the "first" voxel visited by a ray, as the origin may actually lie on the "far" side.

```
  // t_near[i], t_far[i] are the near/far values for the i'th ray
  ( t_near[i], t_far[i] )
      = scene.boundingBox.ClipRaySegment(t_near[i], t_far[i]);
  node = rootNode;
  while (1) {
    while (!node.IsLeaf()) {
      // traverse 'til next leaf
      d[i] = (node.split - ray[i].org[node.dim])
           / ray[i].dir[node.dim];
      active[i] = (t_near[i] < t_far[i]);
      if for all i=0..3 (d[i] <= t_near[i] || !active[i]) {
        // case one,  d <= t_near <= t_far for all active rays
        //            -> cull front side
        node = BackSideSon(node);
      } else if for all i=0..3 (d[i] >= t_far[i] || !active[i]) {
        // case two,  t_near <= t_far <= d for all active rays
        //             -> cull back side
        node = FrontSideSon(node);
      } else {
        // case three: traverse both sides in turn
        // correctly update all near/far values
        // push all near/far values for entire packet
        stack.push(BackSideSon(node),
                   max(d[i],t_near[i]),t_far[i]);
        ( node, t_far[i] )
           = ( FrontSideSon(node), min(d[i],t_near[i]) );
      }
     }
     // have a leaf now
     IntersectAllTrianglesInLeaf(node);
     if for all i=0..3 (t_far[i] <= ray[i].t_closesthit)
        return; // early ray termination
     if (stack is empty)
        return; // noting else to traverse any more...
     // restore all near/far values for entire packet
     ( node, t_near[i], t_far[i] ) = stack.pop();
  }
}
```

Note that all "x[i]" statements are always executed for all four rays in parallel using a SIMD instruction. While this algorithm only operates on packets of 4 rays, the extension to larger packet sizes is straightforward.

Note that the respective computations for properly computing the near/far values (including marking invalid ray segments) get quite a bit more involved for the alternative case in which the origin coincides but the directions differ.

The actual SSE implementation of this algorithm can be performed quite efficiently. Obviously, the same iterative algorithm as in the single ray code can be used, and many of the single-ray optimizations (such as changing the divisions to multiplies with the precomputed inverse) can be performed as well.

All mathematical computations in the inner loop consist of only one SSE multiply and one SSE add. As SSE does not easily work together with non-SSE conditionals, many of the conditionals can be expressed more efficiently by SSE "conditional moves" (realized via SSE bit operations). Furthermore, all of the min/max operations for traversal case 3 can be expressed with a single SEE instruction each.

2.4 Traversal Overhead

Obviously, traversing packets of rays through the acceleration structure generates some overhead: Even if only a single ray requires traversal of a subtree or intersection with a triangle, the operation is always performed on all four rays. Our experiments have shown that this overhead is relatively small as long as the rays are coherent. Table 3 shows the overhead in additional BSP node traversals for different packet sizes.

As can be seen from this experiment, overhead is in the order of a few percent for 2×2 packets of rays, but goes up for larger packets. On the other hand, increasing screen resolution also increases coherence between primary rays.

Most important is the fact that the effective memory bandwidth has been reduced essentially by a factor of four through the new SIMD traversal and intersection algorithms as triangles and BSP nodes need not be loaded separately for each ray. This effect is particularly important for ray traversal as the computation to bandwidth ratio in relatively low.

Of course one could operate on even larger packets of rays to enhance the effect. However, our results show that we are running almost completely within the processor caches even with only four rays. We have therefore chosen not to use more rays per ray packet, as it would additionally increase the overhead due to redundant traversal and intersection computations, and would make the basic algorithm more complicated again[17]. For the SaarCOR architecture however [Woop05], the same packet traversal principle is used with a significantly larger number of rays per packet.

	2×2	4×4	8×8	256^2	1024^2
ERW6	1.4%	4.4%	11.8%	5.8%	1.4%
Office	2.6%	8.2%	21.6%	10.4%	2.6%
Conference.	3.2%	10.6%	28.2%	12.2%	3.2%

Table 3: Overhead (measured in number of additional node traversals) of tracing entire packets of rays at an image resolution of 1024^2 in the first three columns: As expected, overhead increases with scene complexity (800, 34k, and 280k triangles, respectively) and packet size, but is tolerable for small packet sizes. The two columns on the right show the overhead for 2×2 packets at different screen resolutions.

[17]Larger packets especially suffer from the limited number of registers in the `ia32` architectures. Whereas most values for the single ray code can be kept in registers, larger packets require frequent load/store operations to save and restore certain values into the registers

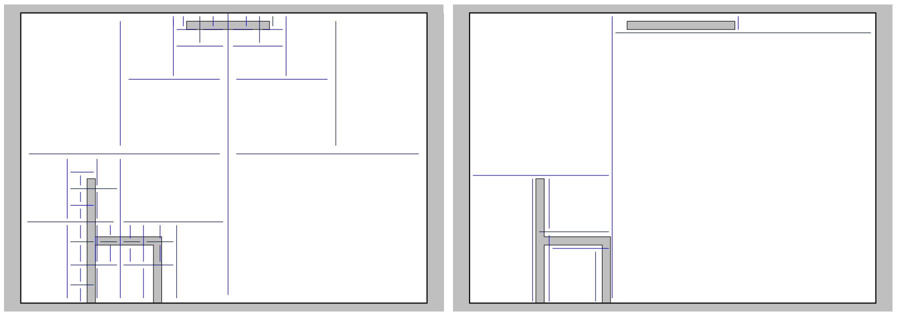


Figure 5: Naive kd-tree vs. high-quality kd-tree in a simple scene consisting of a room with one chair and one light source. Center: The scene with a BSP tree as it would result from a typical naive BSP construction code that always splits the biggest dimension in the middle, until a maximum depth or a minimum number of triangles is reached. Right: The same scene with a high-quality BSP as it results if the planes are placed based a good cost prediction function. Obviously, the BSP with the cost function would be significantly faster to traverse than the BSP with the naive plane placement. The effect of a good BSP tree can be even more pronounced in practical, more complex scenes.

3 High-Quality BSP Construction

Except for efficient traversal and intersection code as just described in Sections 1 and 2, the performance of a ray tracer using a kd-tree to a large degree depends on the algorithms with which the BSP tree has been built. Therefore, it is important to briefly discuss how good BSP trees can be built (for a more in-depth discussion of this topic, see e.g. [Havran01]).

Once the kd-tree has been built – i.e. the location and orientation of the BSP planes, and the decision when to stop subdivision have been fixed – the number of traversal steps and triangle intersections for a given ray and traversal algorithm is predetermined. As such, building a BSP tree that better adapts to the scene complexity directly influences these two critical performance parameters. This can have a *significant* impact on overall performance: For example, since its original publication in [Wald01], the RTRT core has been enhanced with a better BSP construction code which has roughly *doubled* its performance – on top of the already very high performance as originally published. This speedup of two is entirely due to the improved BSP tree, and did not require any other changes to the core[18].

When building BSP trees, the most common approach is to always split each voxel in the middle. In the most naive approach, the splitting dimension is cho-

[18]Note that similar speedups apply for the SaarCOR architecture [Woop05]: As the SaarCOR architecture uses exactly the same data structures as the RTRT kernel (and in fact uses RTRT to generate the binary scene dumps it runs on), any speedups due to better BSPs translate similarly to better SaarCOR performance!

sen in a round-robin fashion, and subdivision proceeds until either a maximum depth has been reached, or voxel contains less than a specified number of triangles[19]. However, it is common knowledge that the BSP tree for non-cube-like scenes can be improved by always splitting the box in the dimension where it has maximum extent[20]. This can be explained by the fact that this approach produces the most cube-like voxels[21]. However, it is also long known that putting the plane into the middle might not be a perfect position, either [Havran01].

Scene	#triangles	absolute performance				speedup	
		RR	ME	PS	SAH	PS	ME/RR
ERW6	804	4.33	4.16	4.53	8.18	80 %	89 %
ERW10	83,600	1.30	2.74	3.03	5.51	81 %	101 %
Office	34,000	2.50	2.32	2.85	4.31	51 %	72 %
Theater	112,306	1.30	1.12	1.47	2.43	65 %	87 %
Conference (sta)	282,801	2.18	1.89	2.47	4.17	69 %	91 %
SodaHall (in)	2,247,879	2.50	2.13	2.87	3.46	20 %	38 %
SodaHall (out)	2,247,879	2.62	2.78	3.63	4.08	12 %	47 %
Cruiser	3,637,101	1.67	1.56	2.03	3.01	48 %	80 %
PowerPlant (in)	12,748,510	0.51	0.50	0.81	1.26	56 %	147 %
PowerPlant (out)	12,748,510	0.72	0.78	0.97	1.44	48 %	84 %

Table 4: Relative performance of rendering with BSPs built by different construction algorithms: Kaplan-BSP with round-robin subdivision (RR), Splitting the voxel in the dimension of maximum extent (ME), "PlaneShifter", i.e. ME with shifting the plane to maximize empty voxels (PS), and a surface area heuristic (SAH). Numbers correspond to million primary rays per second with SSE code on a 2.2GHz Pentium-IV Xeon. Right two columns show the relative SAH speedup as compared to PS, ME and RR. As expected the SAH performs best. Except for Soda Hall, SAH usually performs 50–80 percent faster than the best other method. Note that the effect in practice is even more pronounced: Whereas RR, ME and PS require extensive parameter tuning to achieve the result given in this table, the SAH performs reasonably well already with its default parameters.
The respective scenes can be seen in Figure 7, some statistical data on the generated BSPs is given in Table 5.

Many people assume that placing the split plane towards the object median (i.e. placing it such that both halves contain an equal number of triangles) would be a better choice. Though this appeals to intuition, it is actually a very bad

[19]In practice, 20–25 for maximum depth, and 2–3 for the "triangles per leaf" threshold are usually close to optimal values.

[20]Interestingly, though this is "common knowledge", it is actually a misconception except for extremely "non-cubic" voxels, as can be seen in Table 4 (columns 'RR' vs. 'ME'): For most scenes, splitting in the middle is actually slightly faster.

[21]For cube-like voxels, the ratio of voxel surface to voxel volume reaches its minimum. As the voxel surface influences the probability of a voxel to be hit by a ray[MacDonald89], a voxel of a given volume has the least chance of being traversed.

choice. Splitting at the object median aims at building a balanced tree with equal depth of all leaves. Though this is optimal for binary search trees with equal access probabilities to each leaf node, it is not optimal for ray tracing with a kd-tree: First, the probability of accessing different voxels is certainly not equally distributed, as larger voxels are more likely to be hit than small ones. Furthermore, traversing a kd-tree is actually not the same as a search in a binary search tree (in which traversal always proceeds from the root to the leaf in one straight line), but rather a range searching process in which several leaves have to be accessed, and in which traversal frequently goes up and down in the tree. As such, BSP trees should not be optimized towards having an equal number of traversal steps towards each leaf (i.e. balancing it), but should rather minimize the number of traversal steps for traversing a ray from one location to another. For this kind of traversal, BSP trees behave best if they have large voxels of empty space as close to the root node as possible, as large "empty space" allows for traversing a ray over a large distance at small cost. Splitting at the object median results in empty space being pushed far down the tree into many small voxels, and thus leads to many traversal steps and bad performance.

Some other intuitive improvements to the split plane position lead to more successful heuristics. For example, if one of the half-voxels produced by a split is empty, the argument of empty space being beneficial suggests that the split plane should be "shifted" as far into the non-empty half as possible. This reduces the probability of the ray having to traverse the non-empty leaf, significantly improves the BSP quality, and is easy to implement. This heuristic can also be furtherly refined to yield even more improvements. Though the results of such intuitive approaches are quite limited – in the range of 30–50 percent over the naive construction method (see Table 4) – they are relatively easy to implement, and thus should always be preferred over the naive approach. However, these "simple" heuristics by far cannot match the BSP quality that can be generated with a well-designed cost function (see below).

3.1 Surface Area Heuristic (SAH)

A more successful – though unfortunately also quite more complicated – approach is to optimize the positioning of the splitting plane via cost prediction functions in the spirit of Goldman and Salmon [Goldsmith87], MacDonald and Booth [MacDonald89, MacDonald90], and Subramanian [Subramanian90]. Such a cost prediction function uses certain assumptions for estimating how costly a split would be. This estimate can then be used to place the plane at the position of minimal cost. Furthermore, the cost function provides a much more effective termination criteria for the subdivision than the above-mentioned "maximum depth and triangle threshold": Using a cost-estimate function, subdivision is simply terminated as soon as the estimated traversal cost for a leaf node is less than the cost for the split with minimum estimated cost.

The most famous of these cost prediction functions is the "surface area heuristic" (SAH) as introduced by MacDonald and Booth [MacDonald89, MacDonald90]: The surface area heuristic assumes that rays are equally distributed in space, and are not blocked by objects. Under these (somewhat unrealistic) assumptions, it is possible to calculate the probability with which a ray hitting a voxel also hits any of its sub-voxels. More specifically, having a voxel V that is partitioned into two voxels V_L and V_R, the probability of a ray traversing these two sub-voxels can be calculated as

$$P(V_L|V) = \frac{SA(V_L)}{SA(V)} \text{ and } P(V_R|V) = \frac{SA(V_R)}{SA(V)}$$

where $SA(V) = 2(V_w V_d + V_w V_h + V_d V_h)$ is the surface area of voxel V (with V_w,V_h, and V_d being width, depth and height of the voxel, respectively).

Once these respective probabilities are know, one can estimate the cost of a split: Assuming that a traversal step and a ray triangle intersection have an average cost of C_{trav} and C_{isec} respectively, the average cost of splitting voxel V into V_L and V_R can be estimated as

$$Cost_{split}(V_L, N_L, V_R, N_R) = C_{trav} + C_{isec}(P(V_L|V)N_L + P(V_R|V)N_R)$$

where N_L and N_R are the number of triangles in V_L and V_R, respectively.

3.1.1 Finding the best split positions

This function is continuous except for the split plane positions at which the numbers N_L and N_R change (also see [Havran01]). These are exactly the positions where either a triangle side ends (i.e. at a vertex), or where a triangle side pierces the side of a voxel [Havran01, Hurley02]). These locations form the "potential split positions", from which the position with the minimum cost is chosen. Unfortunately, checking all potential splits can be quite expensive, and requires a carefully designed algorithm to avoid quadratic complexity during each splitting step. Furthermore, finding all potential splits can be quite costly and numerically unstable, especially for those potential splits that are computed by intersecting a triangle side with the voxel surface.

Instead of performing these side-voxel intersections it is also possible to only consider each triangle's bounding box sides as potential split planes. This is much easier to implement, and still performs better than not using the SAH at all. However, "perfect" split positions usually achieve superior performance than only considering the bounding box sides. As such, the RTRT core uses perfect split positions, and uses a carefully designed implementation to avoid all potential numerical inaccuracies without sacrificing performance.

3.1.2 Automatic termination criterion

Using the above assumptions, one can estimate the minimum cost of traversing the split object. Similarly, one can estimate the cost of not splitting a voxel at all, as $Cost_{leaf}(V) = N_V \times C_{isec}$. Simply comparing these two values provides a very simple and efficient termination criterion. Of course, it is still possible to combine the surface area heuristic with other heuristics. For example, it may make sense to still specify a maximum tree depth[22], or to add heuristics for encouraging splits that produce empty space (see e.g. [Havran01]).

3.2 Post-Process Memory Optimizations

The BSP construction process in the RTRT core actually is a two-stage process. While the optimized data layout described in the previous section is quite easy to use during traversal, it would be quite awkward to use while building the BSP. As such, we first build the BSP tree with a more easy-to-use node layout that uses twice as much memory and lots of pointers. Once the build-tree process is finished, RTRT performs several optimizations on the BSP tree (see Figure 6): First, for some build-tree algorithms RTRT first iterates over the whole tree a second time, thereby undoing any splits that have not produced useful results (e.g. a node with two leaves containing the same item lists)[23]. Then, this memory-unfriendly data layout is re-arranged to the more cache-friendly form as described above. Though this data reorganization is quite costly, it is much more convenient than having to program the whole BSP construction code directly on the optimized data layout.

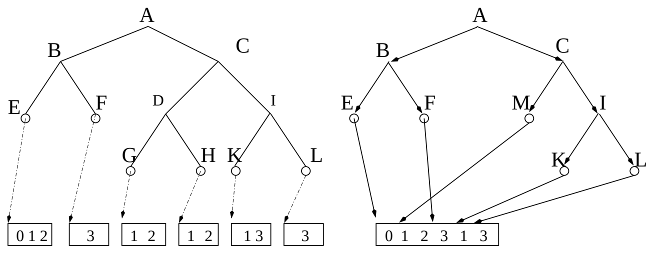


Figure 6: Post-process memory optimizations: After construction, splits that did not produce sensible results get collapsed (e.g. nodes G and H), and the item lists are stored in a compressed form by checking whether the same node list can already be found in the list array. Different item lists can overlap the same memory regions without any problems, as the length of the list is stored in the BSP node anyway. After these collapse operations, the BSP is reformatted to the memory-compressed form as shown in Figure 3.

Finally, it is possible to perform some minor optimizations during the data rearrangement, such as having similar item lists use the same memory space. For example, the item lists “12,13,17” and “13,17” can be stored in the same memory region if the pointer for the second lists points “into” the first list (see Figure 6). Though this can save some memory especially for deeply subdivided BSPs, the performance impact of these final optimizations is quite limited.

[22]Compared to Kaplan-BSPs, a maximum tree depth with the surface area heuristic is more likely to be in the range of 50 or more

[23]Obviously, this could also be done already during BSP construction.

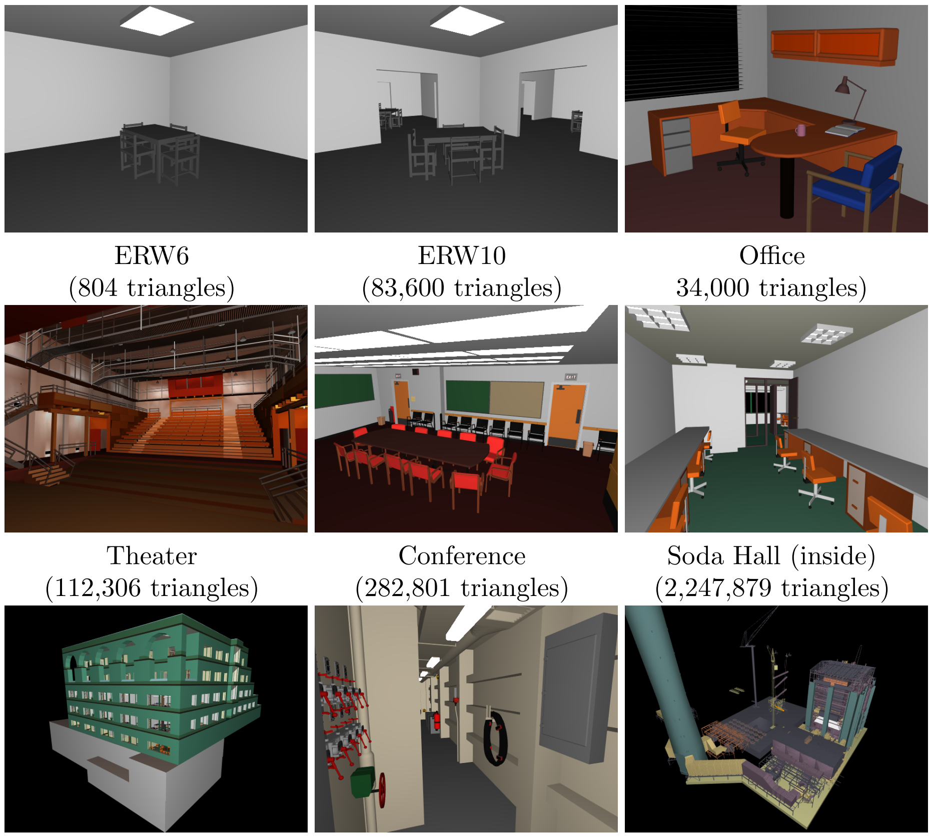


Figure 7: The scenes used for the RTRT benchmarks in Table 6. Including simple SSE shading, these scenes run at 1.3–5.4 frames per second at full-screen (1024×1024) resolutions on a single 2.5GHz Pentium-IV notebook CPU.

3.3 Results of different BSP Construction Strategies

In its current implementation, the surface area heuristic in typical scenes is roughly 50–100 percent faster than a typical Kaplan-type BSP (see Table 4), and is still up to 50 percent faster than the best non-SAH as implemented in RTRT by 2001 (as used in the original 2001 "Coherent Ray Tracing" paper [Wald01]).

Though these results are impressive, the surface area heuristic also has several problems. First of all, it can be quite costly to generate, especially for complex scenes. Second, the SAH – though being already very good – is still not optimal[24]. Following a greedy strategy for picking the split plane can lead to getting stuck in local minima. The same is actually true for the termination

[24]Computing the best BSP tree is known to be NP-complete [Havran01].

criterion: Very easily, it may happen that no split can be found with a cost less than the cost of making a leaf – in which case a leaf will be generated – even though a better configuration might be found if another level of splits were considered (see e.g. Figure 8). This could be fixed by using a global optimization method, which however would probably be far too costly to generate. More importantly, the SAH is quite complicated to implement correctly, and is error-prone both to programming bugs as well as to numerical inaccuracies.

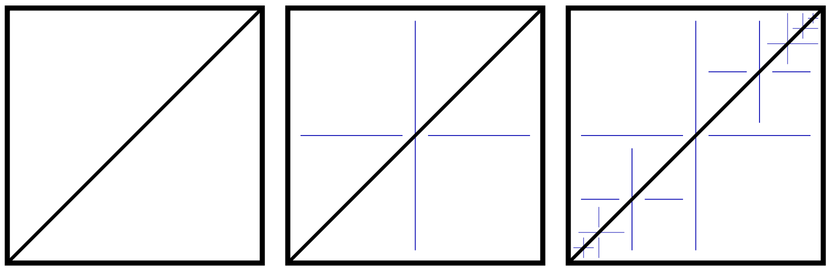


Figure 8: With a greedy method for choosing the split plane, the surface area heuristic can get stuck in local minima. For example, no single split plane can be found that subdivides the left voxel in a way that would have a better cost function than creating a leaf (as each side would have as many triangles as the node itself). If however a "non-optimal" split were allowed in the center, the following split would find a configuration that has less cost than the left one (center image). Right: The same argument can be repeated infinitely, making automatic termination problematic if such splits are allowed. Note that this is a very common configuration for practical scenes, as for example all walls of a room match this setting.

Finally, the SAH requires the ray tracer to work exactly: For example, working on perfect split positions often leads to the generation of "flat" cells with zero width: All triangles that are orthogonal to a coordinate axis (such as walls) will eventually end up in a cell that exactly encloses them, and which thus will be flat[25]. This can easily lead to numerical problems during traversal, as a ray traversing an empty cell actually has a zero-length overlap with this voxel, which may easily be "over-seen" by the traverser. Though this is not exactly a problem of the SAH, it may still lead to problems when using it. Obviously, the RTRT traversal code correctly handles this case.

[25]This case also has to be handled correctly during BSP construction: For example, when further subdividing a flat cell, the construction code has to take care when computing the side-voxel intersections.

Scene (view)	BSP generation strategy	num. trav. steps	number of traversed leaves (total)	(empty)	(full)	Triangle-Isecs mailboxing yes	no
ERW6	Kaplan	32.22	8.05	1.60	6.46	15.51	6.35
	PS	33.45	7.76	4.31	3.45	9.78	5.83
	SAH	20.97	4.32	3.25	1.07	1.46	1.45
ERW10	Kaplan	51.14	9.88	1.66	8.22	17.31	8.39
	PS	54.15	9.70	6.65	3.05	7.50	6.41
	SAH	32.35	5.35	4.27	1.07	2.65	2.65
Office	Kaplan	58.80	12.76	7.47	5.29	11.63	6.03
	PS	60.04	12.10	10.64	1.46	3.39	2.73
	SAH	35.09	6.53	5.37	1.15	3.46	3.36
Theater	Kaplan	98.22	18.21	15.03	3.19	12.52	7.96
	PS	88.48	15.19	13.44	1.74	5.21	4.07
	SAH	64.86	10.40	9.13	1.28	3.79	3.68
Conference	Kaplan	68.10	14.25	9.48	4.78	9.91	5.63
	PS	68.91	13.61	12.31	1.29	2.82	2.38
	SAH	38.32	6.87	5.63	1.24	2.53	2.30
Soda Hall (inside)	Kaplan	61.96	8.70	5.45	3.25	9.58	6.20
	PS	60.06	8.24	6.81	1.43	3.73	2.98
	SAH	50.12	5.34	4.22	1.12	2.64	2.62
Soda Hall (outside)	Kaplan	99.92	17.10	14.29	2.81	8.04	5.52
	PS	73.16	11.56	10.38	1.17	2.89	2.67
	SAH	62.70	9.136	8.09	1.04	1.78	1.78
Cruiser	Kaplan	74.95	11.05	6.77	4.28	14.84	11.15
	PS	78.40	11.2	9.52	1.68	5.31	4.08
	SAH	52.34	7.019	5.74	1.28	2.73	2.57
PowerPlant (inside)	Kaplan	108.7	15.62	11.30	4.33	105.22	81.73
	PS	90.65	12.52	10.73	1.79	41.25	35.12
	SAH	72.79	9.18	7.93	1.25	5.82	5.69
PowerPlant (outside 2) ("overview")	Kaplan	189.1	32.45	28.52	3.93	40.06	28.26
	PS	132.7	22.02	19.82	2.20	15.75	12.13
	SAH	109.7	19.61	17.99	1.62	10.12	9.79

Table 5: Impact of the different BSP generation strategies on traversal parameters: This table shows (for different scenes and views) the average number of BSP traversal steps per ray, average number of leaves encountered during traversal (empty vs. non-empty leaves), and number of ray-triangle intersections with and without mailboxing, respectively[27]. Generation strategies measured include "Kaplan", "PlaneShifting", and Surface Area Heuristic see Table 4). For both Kaplan and PS, several parameter sets have been tested, the number given here corresponds to the parameter set that achieved best performance. Note that the exceptionally high number of triangles visited for the Kaplan BSP in the PowerPlant model results from the high memory consumption of the Kaplan BSP, which did not allow for "deeper" BSP trees in a 32-bit address space.

4 Current RTRT Performance

As described in the previous section, the RTRT software ray tracing kernel builds the combination of highly optimized traversal and intersection routines, tracing packets of rays for efficient SIMD support, and a special emphasis on caching and memory optimizations. Though the newest version of the RTRT core still uses the same ideas as discussed in its original publication [Wald01], the RTRT kernel since then has been significantly improved and completely re-implemented to achieve significantly higher performance [Wald03]. This increase in performance is due to a combination of several factors:

Faster CPUs: Obviously, CPUs have become significantly faster since 2001 (from around 800MHz Pentium-III's to 3GHz Pentium-IV's today). While many other applications cannot fully benefit from this increase in clock rate, the RTRT core has been designed to fully exploit the available CPU performance (e.g. by minimizing cache misses, pipeline stalls and branch mis-predictions), and as such benefits linearly from improved CPU performance. Though the performance increase of modern CPUs is obviously not an achievement of the RTRT core itself, it is due to its special design – especially its emphasis on SIMD support and caching optimizations – that have enabled the RTRT kernel to benefit linearly from any increase in CPU performance.

Better BSP Trees: The "Coherent Ray Tracing" paper cared mostly about the fast traversal of an existing BSP tree, and neglected the algorithms for building these BSPs. The new RTRT core uses an improved "surface area heuristic" (SAH) cost prediction function for generating optimized BSP tree (see Section 3), which result in up to *twice* the performance than with the BSP construction code as used in the original Coherent Ray Tracing system.

Better Compilers: Modern compilers offer increasingly powerful tools for writing better and faster code. For example, RTRT achieves roughly *twice* the performance when compiling its single-ray code (which is written in plain "C/C++") with Intel's ICC (Version 7.1) compiler as compared to compiling it with the 2001 version of the GNU gcc compiler as used in the original system[28]. Comparing to most up-to-date code written in ICC intrinsics with the performance of the original 2001 SSE code written in hand-coded assembler yields similar speedups.

Better Implementations: The RTRT core algorithms cover only a few hundred lines of code, and are continuously being optimized. Since its original publication in 2001 [Wald01], the core code has been re-implemented several times, having resulted in a significant increase in performance.

[28]The new gcc versions 3 and higher are supposed to offer similarly increased performance over pre-3.0 gcc's. Preliminary tests with gcc 3.3.1 have been positive, but a thorough evaluation has not yet been performed.

CPU / scene	#tris	absolute performance (fps@1024x1024, 1CPU)			
ray tracing		SSE	SSE	SSE	C
shading		none	SSE	C	C
ERW6 (static)	804	8.95	5.38	3.80	2.09
ERW6 (dynamic)	804	4.00	3.05	2.57	1.33
Office (static)	34,000	4.68	3.45	2.86	1.39
Office (dynamic)	34,000	2.61	2.17	1.87	0.88
ERW10	83,600	5.82	3.88	3.27	1.65
Theater	112,306	2.68	2.18	1.95	1.05
Conference (dynamic)	282,801	3.17	2.50	1.98	1.01
Conference (static)	282,801	4.40	3.26	2.61	1.44
Soda Hall (in)	2,247,870	3.68	2.85	2.46	1.19
Soda Hall (out)	2,247,870	4.47	3.28	3.19	1.78
Cruiser	3,637510	3.38	2.65	2.31	1.17
Power Plant (in)	12,748,510	1.43	1.27	1.19	0.53
Power Plant (out)	12,748,510	1.59	1.39	1.40	1.17

Table 6: RTRT core performance in million rays per second on a *single* 2.5GHz Pentium-IV notebook CPU at a resolution of 1024×1024 pixels, in different shading configurations: SSE/none corresponds to pure ray traversal and intersection performance without shading at all; SSE/SSE means SSE packet tracing with a hard-coded simple SSE shading model; SSE/C means SSE ray tracing with C-code shading (including SoA-to-AoS data re-packing overhead); and C/C means pure C-code single ray traversal and shading. Though ray tracing scales nicely with scene complexity, even simple shading can already cost more than a factor of two given current ray tracing performance! The above numbers directly correspond to the achievable frame rate on a single 2.5GHz Pentium-IV notebook CPU at full-screen resolution (1024×1024 pixels). The respective benchmarking scenes can be found in Figure 7.

Taken together, these methods allow the current core to significantly outperform the old system even when running the old code on an up-to-date CPU. Even when traversing single, incoherent rays (i.e. *without* using the SSE instruction set) the new kernel is slightly faster than the originally published SSE code tracing packets of rays.

Exploiting the full performance of the newest SIMD code then achieves an additional performance improvement of 2–3 when shooting coherent rays (see Table 6). It is important to note that the RTRT kernel does not use any approximations to achieve this speedup. It still performs at least the operations of a traditional ray tracer. Considering only the pure traversal and intersection cost – i.e. without shading and without support for dynamic scenes – the RTRT kernel achieves up to $\sim$ 9 million rays per second on simple scenes, and still 1.4–4.4 million rays per second on as complex scenes as the soda hall and power plant scenes (with 1.5 and 12.5 million triangles, respectively).

Casting only primary rays with relatively simple shading, this performance allows for computing several (1.3–5.4) full screen frames per second even on a single notebook with a typical 2.5GHz Pentium-IV CPU (see Table 6 and Figure 7). Using a state of the art dual-CPU PC, this level of ray tracing performance allows generate impressive frame-rates even on a single desktop machine.

5 Future Work

As can be seen by the results mentioned in Table 6, it is clear that the biggest individual bottleneck – and thus the biggest remaining problem to be solved – is the cost for shading. As the cost for shading has traditionally been cheap compared to the cost for tracing a ray, this problem so far has not received much attention. With the current increase in ray tracing performance however even simple shading incurs a severe performance impact. As such, the biggest potential for future performance gains lies in finding ways for faster shading. However, it is still unclear how this can be achieved.

Apart from faster shading, we expect that even higher ray tracing performance can be achieved by exploiting even more coherence by using larger packets. Larger packets should allow for optimizations in which not all individual rays in a packet have to be considered in each traversal step. For example, two out of the three traversal cases could be accelerated by only looking at the "corner rays" of a packet[29]. Similarly, the efficiency of the SSE code could probably be increased by larger packets, as any setup cost (such as fetching triangle data) could be amortized over more rays. Though larger packets obviously suffer from decreased coherence, this may be offset by the continuing trend towards higher image resolutions.

Furthermore, it has to be investigated how the ideas that have proven so successful in accelerating ray tracing for polygonal scenes could also be employed for other kind of ray tracing primitives, such as volumetric objects, isosurfaces, or parameteric patches.

Finally, it has to be investigated how much it is possible to further improve the quality of the BSP trees. While the average number of triangles hit by a ray is close to the optimum (see Table 5), it may still be possible to further reduce the number of traversal steps.

References

[AMD] *Advanced Micro Devices.* Software Optimization Guide for AMD Athlon(tm) 64 and AMD Opteron(tm)

[29]For primary rays, it is obvious to define the corner rays for a packet. For secondary rays, the "corner" rays could be defined by the corners of an imaginary shaft bounding the rays.

Processors. Available from http://www.amd.com/us-en/Processors/TechnicalResources/.

[Badouel92] *Didier Badouel.* An Efficient Ray Polygon Intersection. In David Kirk, editor, *Graphics Gems III*, pages 390–393. Academic Press, 1992. ISBN: 0124096735.

[Carey97] *Rikk Carey, Gavin Bell, and Chris Marrin.* ISO/IEC 14772-1:1997 Virtual Reality Modelling Language (VRML97), April 1997. http://www.vrml.org/Specifications/VRML97.

[Erickson97] *Jeff Erickson.* Pluecker Coordinates. *Ray Tracing News*, 1997. http://www.acm.org/tog/resources/RTNews/html/-rtnv10n3.html#art11.

[Glassner89] *Andrew Glassner. An Introduction to Ray Tracing.* Morgan Kaufmann, 1989. ISBN 0-12286-160-4.

[Goldsmith87] *Jeffrey Goldsmith and John Salmon.* Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[Haines91] *Eric Haines.* Efficiency Improvements for Hierarchy Traversal in Ray Tracing. In James Arvo, editor, *Graphics Gems II*, pages 267–272. Academic Press, 1991.

[Havran97] *Vlastimil Havran.* Cache Sensitive Representation for the BSP Tree. In *Compugraphics'97*, pages 369–376. GRASP – Graphics Science Promotions & Publications, December 1997.

[Havran99] *Vlastimil Havran.* Analysis of Cache Sensitive Representation for Binary Space Partitioning Trees. *Informatica*, 23(3):203–210, May 1999. ISSN: 0350-5596.

[Havran00] *Vlastimil Havran, Jan Prikryl, and Werner Purgathofer.* Statistical Comparison of Ray-Shooting Efficiency Schemes. Technical Report TR-186-2-00-14, Department of Computer Science, Czech Technical University; Vienna University of Technology, July 2000.

[Havran01] *Vlastimil Havran.* Heuristic Ray Shooting Algorithms. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[Hurley02] *James T. Hurley, Alexander Kapustin, Alexander Reshetov, and Alexei Soupikov.* Fast Ray Tracing for Modern General Purpose CPU. In *Proceedings of Graphicon*, 2002. Available from http://www.graphicon.ru/2002/papers.html.

[Intel] Intel Corp. *Intel Computer Based Tutorial.* http://developer.-intel.com/vtune/cbts/cbts.htm.

[Keller98] *Alexander Keller.* Quasi-Monte Carlo Methods for Realistic Image Synthesis. PhD thesis, University of Kaiserslautern, 1998.

[MacDonald89] *J. David MacDonald and Kellogg S. Booth.* Heuristics for Ray Tracing using Space Subdivision. In *Proceedings of Graphics Interface '89*, pages 152–63, Toronto, Ontario, June 1989. Canadian Information Processing Society.

[MacDonald90] *J. David MacDonald and Kellogg S. Booth.* Heuristics for Ray Tracing using Space Subdivision. *Visual Computer*, 6(6):153–65, 1990.

[Möller] *Tomas Möller.* Practical Analysis of Optimized Ray-Triangle Intersection. http://www.ce.chalmers.se/staff/-tomasm/raytri/.

[Möller97] *Tomas Möller and Ben Trumbore.* Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[Pharr97] *Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan.* Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Computer Graphics*, 31(Annual Conference Series):101–108, August 1997.

[Shirley03] *Peter Shirley and R. Keith Morley. Realistic Ray Tracing.* A K Peters, Second edition, 2003. ISBN 1-56881-198-5.

[Shoemake98] *Ken Shoemake.* Pluecker Coordinate Tutorial. *Ray Tracing News*, 1998. http://www.acm.org/tog/resources/RTNews/-html/rtnv11n1.html#art3.

[Subramanian90] *K. R. Subramanian.* A Search Structure based on K-d Trees for Efficient Ray Tracing. PhD thesis, The University of Texas at Austin, December 1990.

[Wald01] *Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner.* Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[Wald03] *Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek.* Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.

[Wald04] *Ingo Wald*. Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[Woop05] *Sven Woop, Joerg Schmittler, and Philipp Slusallek*. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *Proceedings of ACM SIGGRAPH*, (to appear), 2005.

Efficiency Issues for Ray Tracing

Brian Smits[*]
University of Utah

February 19, 1999

Abstract

Ray casting is the bottleneck of many rendering algorithms. Although much work has been done on making ray casting more efficient, most published work is high level. This paper discusses efficiency at a slightly lower level, presenting optimizations for bounding volume hierarchies that many people use but are rarely described in the literature. A set of guidelines for optimization are presented that avoid some of the common pitfalls. Finally, the effects of the optimizations are shown for a set of models.

1 Introduction

Many realistic rendering systems rely on ray casting algorithms for some part of their computation. Often, the ray casting takes most of the time in the system, and significant effort is usually spent on making it more efficient. Much work has been done and published on acceleration strategies and efficient algorithms for ray casting, the main ideas of which are summarized in Glassner [5]. In addition, many people have developed optimizations for making these algorithms even faster. Much of this work remains unpublished and part of oral history. This paper is an attempt to write down some of these techniques and some higher level guidelines to follow when trying to speed up ray casting algorithms. I learned most of the lessons in here the hard way, either by making the mistakes myself, or by tracking them down in other systems. Many of the observations in here were confirmed by others.

This paper will discuss some mid-level optimization issues for bounding volume hierarchies. The ray casting algorithm uses the hierarchy to determine if the ray intersects an object. An intersection involves computing the distance to the intersection and the intersection point as well as which object was hit. Sometimes it includes computing surface normal and texture coordinates. The information computed during an intersection is sometimes called the hit information. In ray tracing based renderers, rays from the eye are called primary rays. Reflected and transmitted rays are known as secondary rays. Together, these rays are called intersection rays. Rays from hits to lights to determine shadowing are called shadow rays.

[*]bes@cs.utah.edu

2 Principles of Optimization

Optimization can be a seductive activity leading to endless tweaks and changes of code. The most important part of optimization is knowing when not to do it. Two common cases are:

- Code or system is not run frequently.
- Code is a small fraction of overall time.

In other words, code should only be optimized if it will make a significant effect on the final system and the final system will be used frequently enough to justify the programmer's time and the chance of breaking something.

It helps to have a set of principles to follow in order to guide the process of optimization. The set I use is:

- Make it work before you make it fast.
- Profile everything you do.
- Complexity is bad.
- Preprocessing is good.
- Compute only what you need.

2.1 Make it Work Before You Make it Fast

Code should be made correct before it is made fast [2]. As stated repeatedly by Knuth [9] "Premature optimization is the root of all evil". Obviously, slow correct code is more useful than fast broken code. There is an additional reason for the rule, though. If you create a working, unoptimized version first, you can use that as a benchmark to check your optimizations against. This is very important. Putting the optimizations in early means you can never be completely sure if they are actually speeding up the code. You don't want to find out months or years later that your code could be sped up by removing all those clever optimizations.

2.2 Profile Everything You Do

It is important to find out what the bottleneck is before trying to remove it. This is best done by profiling the code before making changes [3]. The best profilers give time per line of code as well as per function. They also tell you how many times different routines are called. Typically what this will tell you is that most of the time is spent intersecting bounding boxes, something that seems to be universally true. It also can tell you how many bounding boxes and primitives are checked.

Like many algorithms, the speed will vary based on the input. Obviously large data sets tend to take more time than small ones, but the structure of the models you use for benchmarking is also important. Ideally you use a set of models that are characteristic of the types of models you expect to use.

Profiling is especially critical for low-level optimizations. Intuition is often very wrong about what changes will make the code faster and which ones the compiler was already doing for you. Compilers are good at rearranging nearby instructions. They are bad at knowing that the value you are continually reading through three levels of indirection is constant. Keeping things clean and local makes a big difference. This paper makes almost no attempt to deal with this level of optimization.

2.3 Complexity is Bad

Complexity in the intersection algorithm causes problems in many ways. The more complex your code becomes, the more likely it is to behave unexpectedly on new data sets. Additionally, complexity usually means branching, which is significantly slower than similar code with few branches. If you are checking the state of something in order to get out of doing work, it is important that the amount of work is significant and that you actually get out of doing the work often enough to justify the checks. This is the argument against the caches used in Section 4.4.

2.4 Preprocessing is Good

In many of the situations where ray casting is used, it is very common to cast hundreds of millions of rays. This usually takes a much longer time than it took to build the ray tracing data structures. A large percentage increase in the time it takes to build the data structures may provide a significant win even if the percentage decrease in the ray casting time of each ray is much smaller. Ideally you increase the complexity and sophistication of the hierarchy building stage in order to reduce the complexity and number of intersections computed during the ray traversal stage. This principle motivates Section 4.3.

2.5 Compute Only What You Need

There are many different algorithms for many of the components of ray casting. Often there are different algorithms because different information is needed out of them. Much of the following discussion will be based on the principle of determining the minimum amount of information needed and then computing or using that and nothing more. Often this results in a faster algorithm. Examples of this will be shown in Sections 4.1 and 4.2.

3 Overview of Bounding Volume Hierarchies

A bounding volume hierarchy is simply a tree of bounding volumes. The bounding volume at a given node encloses the bounding volumes of its children. The bounding volume of a leaf encloses a primitive. If a ray misses the bounding volume of a particular node, then the ray will miss all of its children, and the children can be skipped. The ray casting algorithm traverses this hierarchy, usually in depth first order, and determines if the ray intersects an object.

```
BoundingVolume **BuildHierarchy**(bvList, start, end, axis)
    if(end - start == 0)      // only a single bv in list so return it.
        return bvList[start]
    BoundingVolume parent
    foreach bv in bvList
        expand parent to enclose bv
    sort bvList along axis
    axis = next axis
    parent.AddChild(BuildHierarchy(bvList, start, (start + end) / 2, axis)
    parent.AddChild(BuildHierarchy(bvList, 1 + (start + end) / 2, end, axis)
    return parent
```

Figure 1: Building a bounding volume hierarchy recursively.

There are several ways of building bounding volume hierarchies [6, 10]. The simplest way to build them is to take a list of bounding volumes containing the primitives and sort along an axis[8]. Split the list in half, put a bounding box around each half, and then recurse, cycling through the axes as you recurse. This is expressed in pseudocode in Figure 1. This method can be modifi ed in many ways to produce better hierarchies. A better way to build the hierarchy is to try to minimize the cost functions described by Goldsmith and Salmon [6].

4 Optimizations for Bounding Volume Hierarchies

4.1 Bounding Box Intersections

Intersecting rays with bounding volumes usually accounts for most of the time spent casting rays. This makes bounding volume intersection tests an ideal candidate for optimization. The fi rst issue is what sort of bounding volumes to use. Most of the environments I work with are architectural and have many axis-aligned planar surfaces. This makes axis-aligned bounding boxes ideal. Spheres tend not to work very well for this type of environment.

There are many ways to represent and intersect an axis-aligned bounding box. I have seen bounding box code that computed the intersection point of the ray with the box. If there was an intersection point, the ray hits the box, and if not, the ray misses. There are optimizations that can be made to this approach, such as making sure you only check faces that are oriented towards the ray, and taking advantage of the fact that the planes are axis aligned [11]. Still, the approach is too slow. The fi rst hint of this is that the algorithm computes an intersection point. We don't care about that, we just want a yes or no answer. Kay [8] represented bounding volumes as the intersection of a set of slabs (parallel planes). A slab is stored as a direction, D_s, and an interval, I_s, representing the minimum and maximum value in that direction, effectively as two plane equations. The set of slab directions is fi xed in advance. In my experience, this approach is most effective when there are three, axis aligned, slab directions. This is just another way of storing a bounding box, we store minimum and maximum values

```
bool **RaySlabsIntersection**(ray, bbox)
  Interval inside = ray.Range()
  for i in (0,1,2)
    inside = Intersection(inside,(slab[i].Range()-ray.Origin[i])/ray.Direction[i])
    if(inside.IsEmpty())
      return false
  return true
```

Figure 2: Pseudocode for intersecting a ray with a box represented as axis aligned slabs.

along each axis.

Given this representation, we can intersect a bounding box fairly efficiently. We show this in pseudocode in Figure 2. This code isn't as simple as it looks due to the comparisons of the IsEmpty and Intersection functions and the need to reverse the min and max values of the interval when dividing by a negative number, but it is still much faster than computing the intersection point with the box.

One important thing to notice about this representation and this intersection code is that it gives the right answer when the ray direction is 0 for a particular component. In this case the ray is parallel to the planes of the slab. The divide by zero gives either $[-\infty, -\infty]$ or $[+\infty, +\infty]$ when the ray is outside the slab and $[-\infty, +\infty]$ when the ray is inside. This saves additional checks on the ray direction.

4.2 Intersection Rays versus Shadow Rays

It is important to know what kind of information you need from the ray casting algorithm in order to keep from doing more work than necessary. There are three commonly used ray casting queries: closest hit, any hit, and all hits. Closest hit is used to determine the first object in a given direction. This query is usually used for primary, reflected, and transmitted rays. Any hit is used for visibility tests between two points. This is done when checking to see if a point is lit directly by a light and for visibility estimation in radiosity algorithms. The object hit is not needed, only the existence of a hit. All hits is used for evaluating CSG models directly. The CSG operations are performed on the list of intervals returned from the all hits intersection routine.

For efficiency reasons it is important to keep these queries separate. This can be seen by looking at what happens when using the most general query, all hits, to implement the others. Any hit will simply check to see if the list of intersections is empty. Clearly we computed more than we needed in this case. Closest hit will sort the list and return the closest intersection. It may seem as if the same or more work is needed for this query, however this is usually not the case. With most ray tracing efficiency schemes, once an intersection is found, parts of the environment beyond the intersection point can be ignored. Finding intersections usually speeds up the rest of the traversal. Also, the list of hit data does not need to be maintained.

Shadow (any hit) rays are usually the most common type of rays cast, often accounting for more than 90 percent of all rays. Because of this, it is worth considering how to make them faster than other types of rays. Shadow rays need not compute any

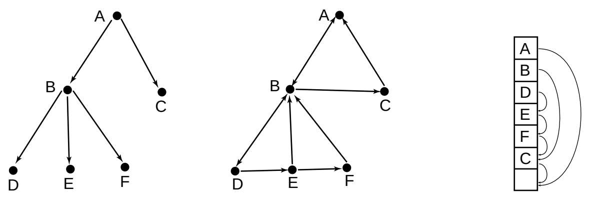


Figure 3: Three different representations for a tree. (a) Children pointers. (b) Left child, right sibling, parent pointers. (c) Array in depth-first order with skip pointers.

of the commonly needed intersection information, such as intersection point, surface normal, uv coordinates, or exact object hit. Additionally, the traversal of the efficiency structure can be terminated immediately once an intersection is guaranteed. A special shadow routine taking these factors into account can make a significant difference in efficiency.

The difference between shadow rays and intersection rays determined which acceleration scheme I use. I have tried both grids [4] and bounding volume hierarchies. In my experience (based on models I typically render) grids are a little faster on intersection rays (closest hit) and slower for shadow rays (any hit). Grids sort the environment spatially, which is good for finding the closest intersection. The bounding volume hierarchies built by trying to minimize Goldsmith and Salmon's cost function [6] tend to keep larger primitives near the root, which is good for shadow rays. It is still unknown as to which acceleration scheme is better, and it is almost certainly based on the model.

4.3 Traversal Code

Casting a ray against a bounding volume hierarchy requires traversing the hierarchy. If a ray hits a bounding volume, then the ray is checked against the children of the bounding volume. If the bounding volume is a leaf, then it has an object inside it, and the object is checked. This is done in depth-first order. Once bounding volume intersection tests are as fast as they can be, the next place for improvement is the traversal of the hierarchy. Traversal code for shadow rays will be used in the following discussion.

In 1991, Haines[7] published some techniques for better traversals. Several of these techniques used extra knowledge to mark bounding boxes as automatically hit and to change the order of traversal. In my experience these methods do not speed up the ray tracer and greatly increase the complexity of the code. This difference in experience may be due to changes in architecture over the last 8 years that make branches and memory accesses instead of floating point the bottleneck. It may also be due to faster bounding box tests. I have found that the best way to make the traversal fast is to make it as minimal as possible.

The simplest traversal code is to use recursion to traverse the tree in depth-first order. Figure 3(a) shows a hierarchy of bounding boxes. Depth first traversal means that bounding box A is tested, then box B, then the boxes with primitives D, E, and F. The idea is to find an intersection as soon as possible by traveling down into the tree. The

```
TreeShadowTraversal(ray, bvNode)
  while(true)     // termination occurs when bvNode→GetParent() is NULL
    if(bvNode→Intersect(ray))
      if(bvNode→HasPrimitive())
        if(bvNode→Primitive().Intersect(ray))
          return true
      else
        bvNode = bvNode→GetLeftChild()
        continue
    while(true)
      if(bvNode→GetRightSibling() != NULL)
        bvNode = bvNode→GetRightSibling()
        break
      bvNode = bvNode→GetParent()
      if(bvNode == NULL)
        return false
```

Figure 4: Traversal of bounding volume tree using left child, right sibling, parent structure.

biggest problem with this is the function call overhead. The compiler maintains much more state information than we need here. We can eliminate much of this overhead by changing our representation of the tree. A representation that works well is to store the left-most child, the right sibling, and the parent for each node, as in Figure 3. Using this representation we can get rid of the recursion by following the appropriate pointers. If the ray intersects the bounding box, we get to its children by following the left-most child link. If the ray misses, we get to the next node by following the right sibling link. If the right sibling is empty, we move up until either there is a right sibling, or we get back up to the root, as shown in pseudocode in Figure 4.

This tree traversal also does too much work. Notice that when the traversal is at a leaf or when the ray misses a bounding volume, we compute the next node. The next node is always the same, there is no reason to be computing it for each traversal. We can pre-compute the node we go to when we skip this subtree and store this skip node in each node. This step eliminates all computation of traversal related data from the traversal. There are still intersection computations, but no extra computation for determining where to go. This is expressed in pseudocode in Figure 5

The final optimization is the recognition that we only need to do depth-first traversals on the tree once it is built. This observation lets us store the tree in an array in depth-first order as in Figure 3. If the bounding volume is intersected, the next node to try is the next node in the array. If the bounding volume is missed, the next node can be found through the skip mechanism. We have effectively thrown out all the information we don't need out of the tree, although it is still possible to reconstruct it. The traversal code can be seen in Figure 6.

The array traversal approach works significantly better than the previous one, and has a couple subtle advantages. The first is better memory usage. In addition to the

```
SkipTreeShadowTraversal(ray, bvNode)
  while(bvNode != NULL)
    if(bvNode→Intersect(ray))
      if(bvNode→HasPrimitive())
        if(bvNode→Primitive().Intersect(ray))
          return true
        bvNode = bvNode→SkipNode()
      else
        bvNode = bvNode→GetLeftChild()
    else
      bvNode = bvNode→SkipNode()
  return false
```

Figure 5: Traversal of bounding volume tree using left child, and skip pointers.

```
ArrayShadowTraversal(ray, bvNode)
  stopNode = bvNode→GetSkipNode()
  while(bvNode < stopNode)
    if(bvNode→Intersect(ray))
      if(bvNode→HasPrimitive())
        if(bvNode→Primitive().Intersect(ray))
          return true
      bvNode++
    else
      bvNode = bvNode→GetSkipNode()
  return false
```

Figure 6: Traversal of bounding volume tree stored as an array in depth-first order.

bounding volume, this method requires only a pointer to a primitive and a pointer to the skip node. This is very minimal. Since the nodes are arranged in the order they will be accessed in, there is more memory coherency for large environments. The second advantage is that this method requires copying data from the original tree into an array. Since the original tree is going to be thrown out, it can be augmented with extra information. Depending upon how the tree is created, this extra information can more than double the cost of each node. Now there is no penalty for this information. Storing the extra information can reduce the time to build the tree and more importantly can result in better trees. The fastest bounding volume test is the one you don't have to do.

4.4 Caching Objects

One common optimization is the use of caches for the object most recently hit. This optimization and variations on it were discussed by Haines[7]. The idea is that the next ray cast will be similar to the current ray, so keep the intersected object around and check it first the next time. To the extent that this is true, caches can provide a benefit, however rays often differ wildly. Also, cache effectiveness decreases as the size of the primitives get smaller. The realism of many types of models is increased by replacing single surfaces with many surfaces. Now caches will remain valid for a shorter amount of time.

There are two different types of caches, those for intersection (closest hit) rays and those for shadow (any hit) rays. If caches are used for intersection rays, the ray will still need to be checked against the environment to see if another object is closer. Usually the ray will again be checked against whatever object is in the cache. Mailboxes [1] can eliminate this second check (by marking each tested object with a unique ray id and then checking the id before testing the primitive). Mailboxes, however, create problems when making a parallel version of the code. Depending on the environment and the average number of possible hits per ray, the cache may reduce the amount of the environment that must be checked by shortening the ray length. In my experience, the cost of maintaining the cache and the double intersection against an object in it more than outweighs the benefi t of having a cache. If your primitives are very expensive and your environments are dense, the benefi t of reducing the length of the ray early may outweigh the costs, but it is worth checking carefully.

Evaluating the benefi t of caches for shadow rays is more complicated. In cases where there is a single light, there tends to be a speedup as long as the cache remains full much of the time and the objects in it stay there for a long enough time. In cases where there are multiple lights we often lose shadow ray coherence because the lights are in different regions of the environment. Now each shadow ray is signifi cantly different from the previous one. A solution for this is to have a different cache for each light.

For both types of caches, we have ignored what happens for reflected and transmitted rays. These rays are spatially very different from primary rays and from each other. Each additional bounce makes the problem much worse. If rays are allowed to bounce d times, there are $2^{d+1} - 1$ different nodes in the ray tree. In order for caching to be useful, a separate cache needs to be associated with each node. For shadow rays, that means a separate cache for each light at each node This can increase the complexity of the code signifi cantly. Another option is to store a cache for each light on each object (or collection of objects) in the environment as discussed by Haines[7]. Note that caching only helps when there is an object in the cache. If most shadow rays won't hit anything (due to the model or the type of algorithm using the shadow tests) then the cache is less likely to be benefi cial. In my experience, shadow caching wasn't a signifi cant enough gain, so I opted for simplicity of code and removed it, although after generating the data for the result section I am considering putting it back in for certain situations. Others have found that caches are still benefi cial.

5 Results

Now we look at the cumulative effects for shadow rays of the three main optimizations described in the paper. First we speed up bounding box tests. Next we speed up the traversal using the different methods from Section 4.3. We then treat shadow rays differently from intersection rays and lastly we add a shadow cache. In all of the

	1	2	3	4	5	6	7	8
theater	64	36	30	21	22	11	10	6
lab	79	41	32	22	20	12	12	7
10,000 small	415	223	191	142	110	48	50	27
10,000 mid	392	185	154	103	81	77	79	65
10,000 big	381	179	152	104	82	79	77	69
100,000 small	995	620	550	449	351	62	63	33
100,000 mid	932	473	424	324	230	146	148	89
100,000 big	1024	508	442	332	240	210	212	156
300,000 mid	1093	597	536	421	312	120	121	64

Table 1: Results of the different experiments described in the text on different environments. Times rounded to the nearest second.

experiments 1,000,000 rays are generated by choosing random pairs of points from within a bounding box 20% larger than the bounding box of the environment. In the last experiment, 500,000 rays are generated, each generated ray is cast twice, resulting in 1,000,000 rays being cast overall. The first two test cases are real environments, the rest are composed of randomly oriented and positioned unit right triangles. The number gives the number of triangles. Small, mid, and big refer to the space the triangles fill. Small environments are 20 units cubed, mid are 100 units cubed, and big are 200 units cubed. The theater model has 46502 polygons. The science center model has 4045 polygons. The code was run on an SGI O2 with a 180 MHz R5000 using the SGI compiler with full optimization turned on[1]. No shading or other computation was done and time to build the hierarchies was not included.

The experiments reported in Table 1 are explained in more detail below:

1. Bounding box test computes intersection point, traversal uses recursion, and shadow rays are treated as intersection rays.
2. Bounding box test replaced by slab version from Section 4.1.
3. Recursive traversal replaced by iterative traversal using left child, right sibling, and parent pointers as in Section 4.3.
4. Skip pointer used to speed up traversal as in Section 4.3.
5. Tree traversal replaced by array traversal as in Section 4.3.
6. Intersection rays replaced by shadow rays as in Section 4.2.
7. Shadow caching used as in Section 4.4.
8. Shadow caching used, but each ray checked twice before generating a new ray. The same number of checks were performed.

[1]-Ofast=ip32_5k

The first thing to notice is that real models require much less work than random polygons. This is because the polygons are distributed very unevenly and vary greatly in size. The theater has a lot more open space and even more variation in polygon size than the lab, resulting in many inexpensive rays and a faster average time. In spite of this, the results show very similar trends for all models. In the first 5 experiments we haven't used any model-specific knowledge, we have just reduced the amount of work done. Special shadow rays and caching are more model specific. Shadow rays are more effective when there are many intersections along the ray and are almost the same when there is zero or one intersection. Shadow caching is based on ray coherence and the likelihood of having an intersection. In experiment 7 there is an unrealistically low amount of coherence (none). In experiment 8 we guaranteed that there would be significant coherence by casting each ray twice.

6 Conclusions

The optimization of ray casting code is a double-edged sword. With careful profiling it can result in significant speedups. It can also lead to code that is slower and more complicated. The optimizations presented here are probably fairly independent of the computer architecture. There are plenty of significant lower level optimizations that can be made which may be completely dependent upon the specific platform. If you plan on porting your code to other architectures, or even keeping your code for long enough that the architecture changes under you, these sorts of optimizations should be made with care.

Eventually you get to a point where further optimization makes no significant difference. At this point you have no choice but to go back and try to create better trees requiring fewer primitive and bounding box tests, or to look at entirely different acceleration strategies. Over time, the biggest wins come from better algorithms, not better code tuning.

The results presented here should be viewed as a case study. They describe some of what has worked for me on the types of models I use. They may not be appropriate for the types of models you use.

7 Acknowledgments

Thanks to Peter Shirley, Jim Arvo, and Eric Haines for many long discussions on ray tracing. Thanks to Peter and Eric for encouraging me to write up these experiences, and to both of them and Bill Martin for helpful comments on the paper. This work was partially funded by Honda and NSF grant ACI-97-20192.

References

[1] ARNALDI, B., PRIOL, T., AND BOUATOUCH, K. A new space subdivision method for ray tracing CSG modelled scenes. *The Visual Computer 3*, 2 (Aug. 1987), 98–108.

[2] BENTLEY, J. L. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[3] BENTLEY, J. L. *Programming Pearls (reprinted with corrections)*. Addison-Wesley, Reading, MA, USA, 1989.

[4] FUJIMOTO, A., TANAKA, T., AND IWATA, K. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications* (Apr. 1986), 16–26.

[5] GLASSNER, A., Ed. *An Introduction to Ray Tracing*. Academic Press, 1989.

[6] GOLDSMITH, J., AND SALMON, J. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20.

[7] HAINES, E. Effi ciency improvements for hierarchy traversal. In *Graphics Gems II*, J. Arvo, Ed. Academic Press, San Diego, 1991, pp. 267–273.

[8] KAY, T. L., AND KAJIYA, J. T. Ray tracing complex scenes. In *Computer Graphics (SIGGRAPH '86 Proceedings)* (Aug. 1986), D. C. Evans and R. J. Athay, Eds., vol. 20, pp. 269–278.

[9] KNUTH, D. E. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[10] RUBIN, S. M., AND WHITTED, T. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics 14*, 3 (July 1980), 110–116.

[11] WOO, A. Fast ray-box intersection. In *Graphics Gems*, A. S. Glassner, Ed. Academic Press, San Diego, 1990, pp. 395–396.

Fast, Minimum Storage Ray/Triangle Intersection

Tomas Möller
Prosolvia Clarus AB
Chalmers University of Technology
E-mail: `tompa@clarus.se`

Ben Trumbore
Program of Computer Graphics
Cornell University
E-mail: `wbt@graphics.cornell.edu`

Abstract

We present a clean algorithm for determining whether a ray intersects a triangle. The algorithm translates the origin of the ray and then changes the base of that vector which yields a vector $(t\ u\ v)^T$, where t is the distance to the plane in which the triangle lies and (u, v) represents the coordinates inside the triangle.

One advantage of this method is that the plane equation need not be computed on the fly nor be stored, which can amount to significant memory savings for triangle meshes. As we found our method to be comparable in speed to previous methods, we believe it is the fastest ray/triangle intersection routine for triangles which do not have precomputed plane equations.

Keywords: ray tracing, intersection, ray/triangle-intersection, base transformation.

1 Introduction

A ray $R(t)$ with origin O and normalized direction D is defined as

$$R(t) = O + tD \tag{1}$$

and a triangle is defined by three vertices V_0, V_1 and V_2. In the ray/triangle-intersection problem we want to determine if the ray intersects the triangle. Previous algorithms have solved this by first computing the intersection between the ray and the plane in which the triangle lies and then testing if the intersection point is inside the edges [**?**].

Our algorithm uses minimal storage (i.e only the vertices of the triangle need to be stored) and does not need any preprocessing. For triangle meshes, the memory savings are significant, ranging from about 25% to 50 %, depending on the amount of vertex sharing.

In our algorithm, a transformation is constructed and applied to the origin of the ray. The transformation yields a vector containing the distance, t, to the intersection and the coordinates, (u, v), of the intersection. In this way the ray/plane intersection of previous algorithms is avoided. It should be noted that this method has been known before, by for example [?] and [?].

2 Intersection Algorithm

A point, $T(u, v)$, on a triangle is given by

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2, \tag{2}$$

where (u, v) are the barycentric coordinates, which must fulfill $u \geq 0$, $v \geq 0$ and $u + v \leq 1$. Note that (u, v) can be used for texture mapping, normal interpolation, color interpolation etc. Computing the intersection between the ray, $R(t)$, and the triangle, $T(u, v)$, is equivalent to $R(t) = T(u, v)$, which yields:

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2 \tag{3}$$

Rearranging the terms gives:

$$\begin{bmatrix} -D, & V_1 - V_0, & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0 \tag{4}$$

This means the barycentric coordinates (u, v) and the distance, t, from the ray origin to the intersection point can be found by solving the linear system of equations above.

The above can be thought of geometrically as translating the triangle to the origin, and transforming it to a unit triangle in y & z with the ray direction aligned with x, as illustrated in figure 1 (where $M = [-D,\ V_1 - V_0,\ V_2 - V_0]$ is the matrix in equation 4).

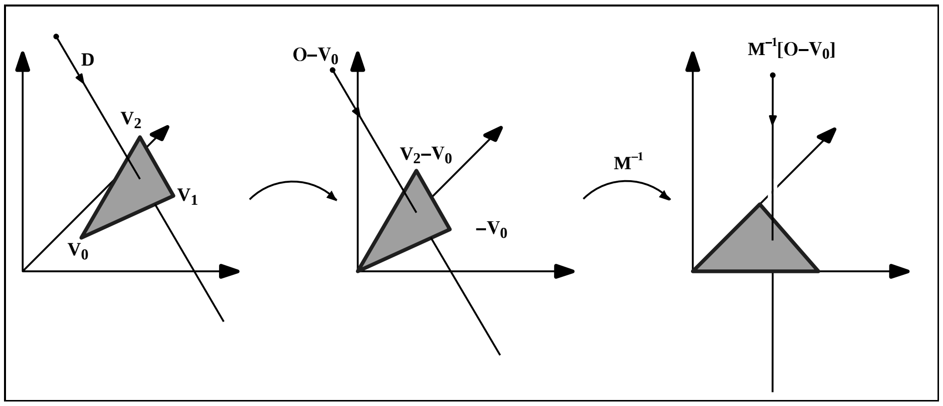


Figure 1: Translation and change of base of the ray origin.

Arenberg, in [?], describes a similar algorithm to the one above. He also constructs a 3×3 matrix but uses the normal of the triangle instead of the ray direction D. This method requires storing the normal for each triangle or computing them on the fly.

Denoting $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$ and $T = O - V_0$, the solution to equation (4) is obtained by using Cramer's rule:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D,\ E_1,\ E_2|} \begin{bmatrix} |\,T,\ E_1,\ E_2\,| \\ |-D,\ T,\ E_2| \\ |-D,\ E_1,\ T| \end{bmatrix} \tag{5}$$

¿From linear algebra, we know that $|A,\ B,\ C| = -(A \times C) \cdot B = -(C \times B) \cdot A$. Equation (5) could therefore be rewritten as

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}, \tag{6}$$

where $P = (D \times E_2)$ and $Q = T \times E_1$. In our implementation we reuse these factors to speed up the computations.

3 Implementation

The following C implementation (available online) has been tailored for optimum performance. There are two branches in the code; one which efficiently culls all back facing triangles (`#ifdef TEST_CULL`) and the other which performs the intersection test on two-sided triangles (`#else`). All computations are delayed until it is known that they are required. For example, the value for v is not computed until the value of u is found to be within the allowable range.

The one-sided intersection routine eliminates all triangles where the value of the determinant (`det`) is negative. This allows the routine's only division operation to be delayed until an intersection has been confirmed. For shadow test rays this division is not needed at all, since all we need is whether the triangle is intersected.

The two-sided intersection routine is forced to perform that division operation in order to evaluate the values of u and v. Alternatively, this function could be rewritten to conditionally compare u and v to 0 based on the sign of `det`.

Some aspects of this code deserve special attention. The calculation of edge vectors can be done as a pre-process, with `edge1` and `edge2` being stored in place of `vert1` and `vert2`. This speedup is only possible when the actual spatial locations of `vert1` and `vert2` are not needed for other calculations and when the vertex location data is not shared between triangles.

To ensure numerical stability, the test which eliminates parallel rays must compare the determinant to a small interval around zero. With a properly adjusted `EPSILON` value, this algorithm is extremely stable. If only front facing triangles are to be tested, the determinant can be compared to `EPSILON`, rather than 0 (a negative determinant indicates a back facing triangle).

The value of u is compared to an edge of the triangle ($u = 0$) and also to a line parallel to that edge, but passing through the opposite point of the triangle ($u = 1$). Though not actually testing an edge of the triangle, this second test efficiently rules out many intersection points without further calculation.

```
#define EPSILON 0.000001
#define CROSS(dest,v1,v2) \
          dest[0]=v1[1]*v2[2]-v1[2]*v2[1]; \
          dest[1]=v1[2]*v2[0]-v1[0]*v2[2]; \
          dest[2]=v1[0]*v2[1]-v1[1]*v2[0];
#define DOT(v1,v2) (v1[0]*v2[0]+v1[1]*v2[1]+v1[2]*v2[2])
#define SUB(dest,v1,v2)
          dest[0]=v1[0]-v2[0]; \
          dest[1]=v1[1]-v2[1]; \
          dest[2]=v1[2]-v2[2];


int
intersect_triangle(double orig[3], double dir[3],
                   double vert0[3], double vert1[3], double vert2[3],
                   double *t, double *u, double *v)
{
   double edge1[3], edge2[3], tvec[3], pvec[3], qvec[3];
   double det,inv_det;

   /* find vectors for two edges sharing vert0 */
   SUB(edge1, vert1, vert0);
   SUB(edge2, vert2, vert0);

   /* begin calculating determinant - also used to calculate U parameter */
   CROSS(pvec, dir, edge2);

   /* if determinant is near zero, ray lies in plane of triangle */
   det = DOT(edge1, pvec);

#ifdef TEST_CULL         /* define TEST_CULL if culling is desired */
   if (det < EPSILON)
      return 0;

   /* calculate distance from vert0 to ray origin */
   SUB(tvec, orig, vert0);

   /* calculate U parameter and test bounds */
   *u = DOT(tvec, pvec);
   if (*u < 0.0 || *u > det)
      return 0;

   /* prepare to test V parameter */
   CROSS(qvec, tvec, edge1);

    /* calculate V parameter and test bounds */
   *v = DOT(dir, qvec);
   if (*v < 0.0 || *u + *v > det)
      return 0;

   /* calculate t, scale parameters, ray intersects triangle */
   *t = DOT(edge2, qvec);
   inv_det = 1.0 / det;
   *t *= inv_det;
   *u *= inv_det;
   *v *= inv_det;
#else                   /* the non-culling branch */
   if (det > -EPSILON && det < EPSILON)
     return 0;
   inv_det = 1.0 / det;

   /* calculate distance from vert0 to ray origin */
   SUB(tvec, orig, vert0);

   /* calculate U parameter and test bounds */
   *u = DOT(tvec, pvec) * inv_det;
   if (*u < 0.0 || *u > 1.0)
     return 0;

   /* prepare to test V parameter */
   CROSS(qvec, tvec, edge1);

   /* calculate V parameter and test bounds */
   *v = DOT(dir, qvec) * inv_det;
   if (*v < 0.0 || *u + *v > 1.0)
     return 0;

   /* calculate t, ray intersects triangle */
   *t = DOT(edge2, qvec) * inv_det;
#endif
   return 1;
}
```

4 Results

In [?], a ray/triangle intersection routine that also computes the barycentric coordinates was presented. We compared that method to ours. The two non-culling methods were implemented in an efficient ray tracer. Figure **??** presents ray tracing runtimes from a Hewlett-Packard 9000/735 workstation for the three models shown in figures **??**-**??**. In this particular implementation, the performance of the two methods is roughly comparable (detailed statistics is available online).

Model	Objects	Polygons	Lights	Our method sec.	Badouel sec.
Car	497	83408	1	365	413
Mandala	1281	91743	2	242	244
Fallingwater	4072	182166	15	3143	3184

Figure 2: Contents and runtimes for data sets in figures **??**-**??**.

5 Conclusions

We present an algorithm for ray/triangle intersection which we show to be comparable in speed to previous methods while significantly reducing memory storage costs, by avoiding storing triangle plane equations.

6 Acknowledgements

Thanks to Peter Shirley, Eric Lafortune and the anonymous reviewer whose suggestions greatly improved this paper.

This work was supported by the NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization (ASC-8920219), and by the Hewlett-Packard Corporation and by Prosolvia Clarus AB.

References

[Arenberg88] Jeff Arenberg, *Re: Ray/Triangle Intersection with Barycentric Coordinates*, in Ray Tracing News, edited by Eric Haines, Vol. 1, No. 11, November 4, 1988, `http://www.acm.org/tog/resources/RTNews/`.

[Badouel90] Didier Badouel, *An Efficient Ray-Polygon Intersection*, in Graphics Gems, edited by Andrew S. Glassner, Academic Press Inc., 1990, pp. 390-393.

[Haines94] Eric Haines, *Point in Polygon Strategies*, in Graphics Gems IV, edited by Paul S. Heckbert, AP Professional, 1994, pp. 24-46.

[Patel96] Edward Patel, personal communication, 1996.

[Shirley96] Peter Shirley, personal communication, 1996.

Web information

Source code, statistical analysis and images are available online at
http://www.acm.org/jgt/papers/MollerTrumbore97/

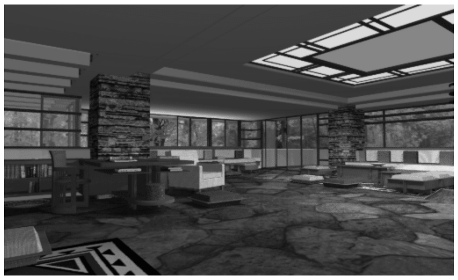


Figure 3: Falling Water

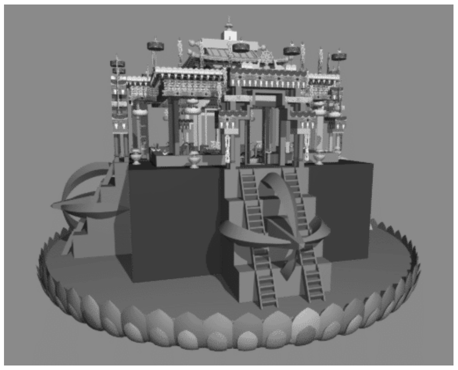


Figure 4: Mandala

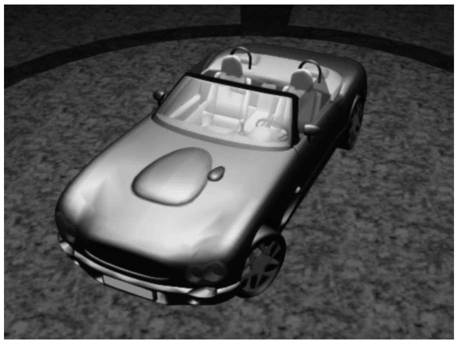


Figure 5: Car (model is courtesy of Nya Perspektiv Design AB).

Fast 3D Triangle-Box Overlap Testing

Tomas Akenine-Möller*
Department of Computer Engineering,
Chalmers University of Technology

March 2001, updated June 2001

Abstract

A fast routine for testing whether a triangle and a box are overlapping in three dimensions is presented. The test is derived using the separating axis theorem, whereafter the test is simplified and the code is optimized for speed. We show that this approach is 2.3 vs. 3.8 (PC vs. Sun) times faster than previous routines for this. It can be used for faster collision detection and faster voxelization in interactive ray tracers. The code is available online.

1 Introduction

Testing whether a triangle overlaps a box is an important routine to have in a graphics programmer's toolbox. For example, the test can be used to voxelize triangle meshes in ray tracers, and it can be used in collision detection algorithms that are based on boxes [3]. Gottschalk et al's collision detection framework only used OBB/OBB tests and triangle-triangle tests. However, it has been noted that both memory and speed can be gained [7] by not having an OBB around each triangle, and instead test a triangle against an OBB.

Previously, Voorhies has presented code for testing a triangle against a unit cube centered at the origin [8]. His test tries to eliminate work by doing some simple acceptance/rejection tests early on, and then testing each triangle edge for intersection with the cube faces. Finally, he checks whether the interior of the triangle is penetrated by the cube. Green and Hatch [4] improve on the efficiency of Voorhies' work and generalize it to handle arbitrary polygons as well. They also use fast acceptance/rejectance tests, but recast the testing of an edge against the cube into testing a point against a skewed rhombic dodecahedron, which is more robust. Finally, they test whether one diagonal of the cube intersect the polygon, which further improves the efficiency.

*Previously known as Tomas Möller.

2 Derivation and Optimization

Our test is derived from the *separating axis theorem* (SAT) [1, 3, 6]. The theorem states that two convex polyhedra, A and B, are disjoint if they can be separated along either an axis parallel to a normal of a face of either A or B, or along an axis formed from the cross product of an edge from A with and edge from B.

We focus on testing an axis-aligned bounding box (AABB), defined by a center $\mathbf{c}$, and a vector of half lengths, $\mathbf{h}$, against a triangle $\Delta\mathbf{u}_0\mathbf{u}_1\mathbf{u}_2$. To simplify the tests, we first move the triangle so that the box is centered around the origin, i.e., $\mathbf{v}_i = \mathbf{u}_i - \mathbf{c}$, $i \in \{0, 1, 2\}$. To test against an oriented box, we would first rotate the triangle vertices by the inverse box transform, then use the presented test. Based on SAT, we test the following 13 axes:

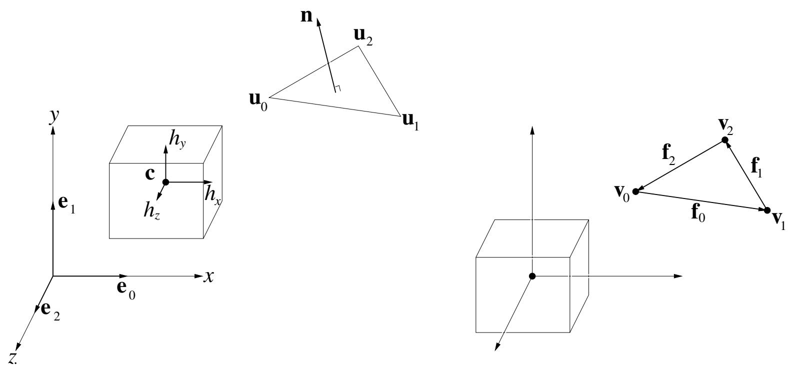


Figure 1: Notation used for the triangle-box overlap test. To the left the inital position of the box and the triangle is shown, while at the right, the box and the triangle has been translated so that the box center coincides with the origin.

1. [3 tests] $\mathbf{e}_0 = (1, 0, 0)$, $\mathbf{e}_1 = (0, 1, 0)$, $\mathbf{e}_2 = (0, 0, 1)$ (the normals of the AABB). Test the AABB against the minimal AABB around the triangle.
2. [1 test] $\mathbf{n}$, the normal of Δ. We use a fast plane/AABB overlap test [5, 6], which only tests the two diagonal vertices, whose direction is most closely aligned to the normal of the triangle.
3. [9 tests] $\mathbf{a}_{ij} = \mathbf{e}_i \times \mathbf{f}_j$, $i, j \in \{0, 1, 2\}$, where $\mathbf{f}_0 = \mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{f}_1 = \mathbf{v}_2 - \mathbf{v}_1$, and $\mathbf{f}_2 = \mathbf{v}_0 - \mathbf{v}_2$. These tests are very similar and we will only show the derivation of the case where $i = 0$ and $j = 0$ (see below).

If all tests pass, i.e., there is no separating axis, then the triangle overlaps the box. Also, as soon as a separating axis is found the the algorithm terminates and returns “no overlap”.

Next, we derive one of the nine tests, where $i = 0$ and $j = 0$, in bullet 3 above. This means that $\mathbf{a}_{00} = \mathbf{e}_0 \times \mathbf{f}_0 = (0, -f_{0z}, f_{0y})$. So, now we need to project the triangle vertices onto $\mathbf{a}_{00}$ (hereafter called $\mathbf{a}$):

$$\begin{aligned} p_0 &= \mathbf{a}\cdot\mathbf{v}_0 = (0, -f_{0z}, f_{0y})\cdot\mathbf{v}_0 = v_{0z}v_{1y} - v_{0y}v_{1z} \\ p_1 &= \mathbf{a}\cdot\mathbf{v}_1 = (0, -f_{0z}, f_{0y})\cdot\mathbf{v}_1 = v_{0z}v_{1y} - v_{0y}v_{1z} = p_0 \\ p_2 &= \mathbf{a}\cdot\mathbf{v}_2 = (0, -f_{0z}, f_{0y})\cdot\mathbf{v}_2 = (v_{1y} - v_{0y})v_{2z} - (v_{1z} - v_{0z})v_{2y} \end{aligned} \tag{1}$$

Normally, we would have had to find $\min(p_0, p_1, p_2)$ and $\max(p_0, p_1, p_2)$, but fortunately $p_0 = p_1$, which simplify the computations a lot. Now we only need to find $\min(p_0, p_2)$ and $\max(p_0, p_2)$, which is significantly faster because conditional statements are expensive on modern CPUs.

After the projection of the triangle onto $\mathbf{a}$, we need to project the box onto $\mathbf{a}$ as well. We compute a "radius", called r, of the box projected on $\mathbf{a}$ as

$$r = h_x|a_x| + h_y|a_y| + h_z|a_z| = h_y|a_y| + h_z|a_z| \tag{2}$$

where the last step comes from that $a_x = 0$ for this particular axis. Then this axis test becomes:

$$\texttt{if(}\ \min(p_0, p_2) > r\ \texttt{ or }\ \max(p_0, p_2) < -r)\ \texttt{return false;} \tag{3}$$

Now, if all these 13 tests pass, then the triangle overlaps the box.

3 Performance Evaluation

To evaluate performance, we used the same test as Voorhies [8], i.e., we randomly select the triangle vertices inside a $4 \times 4 \times 4$ cube centered around the origin and the AABB is the unit cube: from $(-0.5, -0.5, -0.5)$ to $(0.5, 0.5, 0.5)$. To get accurate timings we randomly selected $100,000$ triangles and tested these in a sequence 100 times. We verified that our code generated the same result as Green and Hatch [4], and compared runtimes (we did not test against Voorhies code since that was found to be incorrect [2]).

On a Sun Sparc Ultra 10 at 333 MHz, the presented code was 3.8 times faster on average in this test[1]. On a Linux PC with a 1333 MHz AMD Athlon, the speed up was found to be 2.3[2]. Also, the best order to perform the tests on the Sun was found to be: 3, 1, and finally 2 (the most expensive). On the PC, the order did not matter significantly.

Note that Green and Hatch's code handles the more general case of testing a general polygon against a cube, while we only test a triangle against a cube, and hence we can expect some degradation in performance due to this.

The only place, where there is a robustness issue, is when the normal of the triangle is computed; $\mathbf{n} = \mathbf{f}_0 \times \mathbf{f}_1$. If the triangle has an area close to zero, then the normal calculation is not robust, and our code does not solve that problem. However, in most applications thin long triangles are best avoided.

We have used the code for fast voxelization in a ray tracer, and it has been used in a 3D engine [7].

[1]Our code was compiled using `gcc`, and Green and Hatch's code was compiled using Sun's `cc`, because the runtimes were best for the different routines like that.

[2]Compiled with `gcc -O9 -fomit-frame-pointer -funroll-loops -march=athlon`.

4 Acknowledgement

Thanks to Pierre Terdiman for suggesting different ways to optimize the code, and for trying the code in his game engine. Thanks to Peter Rundberg for letting me use his PC for timings.

Code is available at: `http://www.acm.org/jgt/AkenineMoller01/`

References

[1] Eberly, David, *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, Morgan Kaufmann Publishers Inc., San Francisco, 2000. `http://www.magic-software.com/`

[2] *Graphics Gems III Errata Listing*, `http://www.graphicsgems.org/`

[3] Gottschalk, S., M.C. Lin, and D. Manocha, "OBBTree: A Hierarchical Structure for Rapid Interference Detection," *Computer Graphics (SIGGRAPH '96 Proceedings)*, pp. 171–180, August, 1996. `http://www.cs.unc.edu/~geom/OBB/OBBT.html`

[4] Green, D. and D. Hatch, "Fast Polygon-Cube Intersection Testing," in Alan Paeth, ed., *Graphics Gems V*, AP Professional, Boston, pp. 375–379, 1995. `http://www.graphicsgems.org/`

[5] Haines, Eric, and John Wallace, "Shaft Culling for Efficient Ray-Traced Radiosity," in P. Brunet and F.W. Jansen, eds., *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*, Springer-Verlag, New York, pp. 122–138, 1994. `http://www.acm.org/tog/editors/erich/`

[6] Möller, Tomas, and Eric Haines, *Real-Time Rendering*, AK Peters Ltd., Natick, MA, 1999. `http://www.realtimerendering.com/`

[7] Terdiman, Pierre, Personal communication, 2001.

[8] Voorhies, Douglas, "Triangle-Cube Intersection," in David Kirk, ed., *Graphics Gems III*, AP Professional, Boston, pp. 236–239, 1992. `http://www.graphicsgems.org/`

Vol. 10, No. 1: 55–60

An Efficient and Robust Ray-Box Intersection Algorithm

Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley
University of Utah

Abstract. The computational bottleneck in a ray tracer using bounding volume hierarchies is often the ray intersection routine with axis-aligned bounding boxes. We describe a version of this routine that uses IEEE numerical properties to ensure that those tests are both robust and efficient. Sample source code is available online.

1. Introduction

Naive implementations of ray-box intersection algorithms can have numerical problems for rays that have slopes near zero along any axis. Smits [Smits 98] pointed out that properties given in the IEEE floating point standard [IEEE 85] can be used to avoid explicit tests for these values, but did not provide the implementation details. The following is an implementation of Smits' algorithm. It expects a box with ordered corners `min` and `max`, a ray `r`, and a valid intersection interval of `(t0, t1)` to be given. We assume that the `Vector3` and `Ray` classes are implemented; their usages below should be obvious.

```
class Box {
  public:
    Box(const Vector3 &min, const Vector3 &max) {
      assert(min < max);
      bounds[0] = min;
      bounds[1] = max;
    }
    bool intersect(const Ray &, float t0, float t1) const;
    Vector3 bounds[2];
};

// Smits' method
bool Box::intersect(const Ray &r, float t0, float t1) const {
  float tmin, tmax, tymin, tymax, tzmin, tzmax;
  if (r.direction.x() >= 0) {
       tmin = (bounds[0].x() - r.origin.x()) / r.direction.x();
       tmax = (bounds[1].x() - r.origin.x()) / r.direction.x();
  }
  else {
       tmin = (bounds[1].x() - r.origin.x()) / r.direction.x();
       tmax = (bounds[0].x() - r.origin.x()) / r.direction.x();
  }
  if (r.direction.y() >= 0) {
       tymin = (bounds[0].y() - r.origin.y()) / r.direction.y();
       tymax = (bounds[1].y() - r.origin.y()) / r.direction.y();
  }
  else {
       tymin = (bounds[1].y() - r.origin.y()) / r.direction.y();
       tymax = (bounds[0].y() - r.origin.y()) / r.direction.y();
  }
  if ( (tmin > tymax) || (tymin > tmax) )
       return false;
  if (tymin > tmin)
        tmin = tymin;
  if (tymax < tmax)
        tmax = tymax;
  if (r.direction.z() >= 0) {
       tzmin = (bounds[0].z() - r.origin.z()) / r.direction.z();
       tzmax = (bounds[1].z() - r.origin.z()) / r.direction.z();
  }
  else {
       tzmin = (bounds[1].z() - r.origin.z()) / r.direction.z();
       tzmax = (bounds[0].z() - r.origin.z()) / r.direction.z();
  }
  if ( (tmin > tzmax) || (tzmin > tmax) )
       return false;
  if (tzmin > tmin)
        tmin = tzmin;
  if (tzmax < tmax)
        tmax = tzmax;
  return ( (tmin < t1) && (tmax > t0) );
}
```

Note that the reason we check the sign of each component direction is to ensure that the intervals produced are ordered (i.e., so that `tmin <= tmax` is true). This property is assumed throughout the code, and allows us to reason about whether the computed intervals overlap. Note also that since IEEE arithmetic guarantees that a positive number divided by zero is $+\infty$ and a negative number divided by zero is $-\infty$, the code works for vertical and horizontal lines (see [Shirley 02] for a detailed discussion).

2. Improved Code

The code from the previous section works correctly for almost all values, but there is a problem if `r.direction.x() == -0.0` In this case, the first if statement will be true (`-0 == 0` is true in IEEE floating point), and instead of the resulting interval being $(-\infty, +\infty)$, it will be the degenerate $(+\infty, -\infty)$. The same problem appears when either `r.direction.y()` or `r.direction.z()` are -0.0. When such a degenerate interval is obtained, the function will return false. The algorithm therefore fails to detect a valid intersection in this situation. While this scenario may seem unlikely, negative zeroes can arise in practice, and indeed have in our applications, which is how we discovered this problem. Note how easy it is to generate a negative zero:

```
float u = -2.0;
float v =  0.0;
float w =  u*v;  // w is now negative zero
```

Many implementations of ray-box intersection replace the two divides in each if clause with a single divide and two multiplies:

```
divx =  1 / r.direction.x();
tmin = (bounds[0].x() - r.origin.x()) * divx;
tmax = (bounds[1].x() - r.origin.x()) * divx;
```

This is done because the two multiplies are usually faster than the single divide they replace, but it also allows a way out of the negative zero problem. `divx` captures the sign of `r.direction.x()` even when it is zero: `1 / 0.0 =` $+\infty$ and `1 / -0.0 =` $-\infty$. The updated algorithm for the x component (y and z are analogous) is:

```
// Improved method for x component
divx =  1 / r.direction.x();
if (divx >= 0) {
     tmin = (bounds[0].x() - r.origin.x()) * divx;
     tmax = (bounds[1].x() - r.origin.x()) * divx;
}
```

```
else {
     tmin = (bounds[1].x() - r.origin.x()) * divx;
     tmax = (bounds[0].x() - r.origin.x()) * divx;
}
```

Note that it is important to test the sign of `divx` rather than `r.direction.x()` in order for `-0.0` to be properly detected. This does result in an efficiency penalty on some systems because the evaluation of the `if` statement must wait for the result of the divide. Nonetheless, to ensure the correctness of the ray-box test in all cases, this penalty must be accepted. The code with a test on `divx` was first presented by Smits [Smits 02]; although he did not explicitly state its advantage for handling zeroes, he was probably aware of it because the associated efficiency penalty makes it otherwise unattractive.

3. Optimizing for Multiple Box Tests

Rays are often tested against numerous boxes in a ray tracer, e.g., when traversing a bounding volume hierarchy. The above algorithm can be optimized by precomputing values that remain constant in each test. Rather than computing `divx = 1 / r.direction.x()` each time a ray is intersected with a box, the ray data structure can compute and store this and other pertinent values. Storing the inverse of each component of the ray direction as well as the boolean value associated with the tests (such as `divx >= 0`) provides significant speed improvements. The new code is fairly simple:

```
class Ray {
  public:
    Ray(Vector3 &o, Vector3 &d) {
      origin = o;
      direction = d;
      inv_direction = Vector3(1/d.x(), 1/d.y(), 1/d.z());
      sign[0] = (inv_direction.x() < 0);
      sign[1] = (inv_direction.y() < 0);
      sign[2] = (inv_direction.z() < 0);
    }
    Vector3 origin;
    Vector3 direction;
    Vector3 inv_direction;
    int sign[3];
};

// Optimized method
bool Box::intersect(const Ray &r, float t0, float t1) const {
  float tmin, tmax, tymin, tymax, tzmin, tzmax;
```

```
    tmin = (bounds[r.sign[0]].x() - r.origin.x())
       * r.inv_direction.x();
    tmax = (bounds[1-r.sign[0]].x() - r.origin.x())
       * r.inv_direction.x();
    tymin = (bounds[r.sign[1]].y() - r.origin.y())
       * r.inv_direction.y();
    tymax = (bounds[1-r.sign[1]].y() - r.origin.y())
       * r.inv_direction.y();
    if ( (tmin > tymax) || (tymin > tmax) )
         return false;
    if (tymin > tmin)
          tmin = tymin;
    if (tymax < tmax)
          tmax = tymax;
    tzmin = (bounds[r.sign[2]].z() - r.origin.z())
       * r.inv_direction.z();
    tzmax = (bounds[1-r.sign[2]].z() - r.origin.z())
       * r.inv_direction.z();
    if ( (tmin > tzmax) || (tzmin > tmax) )
         return false;
    if (tzmin > tmin)
          tmin = tzmin;
    if (tzmax < tmax)
          tmax = tzmax;
    return ( (tmin < t1) && (tmax > t0) );
  }
```

We ran tests to ensure that the multibox optimization did not incur a decrease in efficiency for the case in which a single box or shallow bounding volume hierarchy is intersected. Our results show that the optimized method is indeed faster for both cases. While the runtimes are dependent on processor type and scene content, we found these timings to be typical for most scene complexities and architectures.

Scene	Smits' method	Improved method	Optimized method
Single box - 1e8 rays	77.78s	71.39s	66.82s
1e6 triangles in BVH - 1e8 rays	1027.43s	961.23	739.21s

Table 1.

In both the single-box and BVH tests approximately half of the rays fired hit the test object while the other half were near misses. The tests were performed on a Pentium4 1800 MHz processor.

Acknowledgments. We would like to acknowledge Brandon Mansfield, Steve Parker, and Erik Reinhard for providing test code. John McCorquodale also provided some useful information about the speed of float-point multiplies and divides. Brian Smits' advocacy for BVH intersection methods and care with IEEE properties gave us the initial impetus for this work. The anonymous reviewer of the article provided very helpful comments, and pointed out that the divides could be performed once for a full hierarchy. This work was partially supported by NSF grant 03-06151.

Web Information:

Sample C++ source code for the optimized method described above is available online at http://www.acm.org/jgt/WilliamsEtAl05.

References

[IEEE 85] IEEE Standards Association. "IEEE Standard for Binary Floating-Point Arithmetic." IEEE Report (New York), ANSI/IEEE Std 754-1985, 1985.

[Shirley 02] Peter Shirley. *Fundamentals of Computer Graphics.* Wellesley, MA: A K Peters, Ltd., 2002.

[Smits 98] Brian Smits. "Efficiency Issues for Ray Tracing." *journal of graphics tools* 3:2 (1998), 1–14.

[Smits 02] Brian Smits. "Efficient Bounding Box Intersection." *Ray Tracing News* 15:1 (2002).

Amy Williams, University of Utah, Computer Science Department, 50 Central Campus Drive, Salt Lake City, UT 84112 (amy@mit.edu)

Steve Barrus, University of Utah, Computer Science Department, 50 Central Campus Drive, Salt Lake City, UT 84112 (email address)

R. Keith Morley, University of Utah, Computer Science Department, 50 Central Campus Drive, Salt Lake City, UT 84112 (email address)

Peter Shirley, University of Utah, Computer Science Department, 50 Central Campus Drive, Salt Lake City, UT 84112 (shirley@cs.utah.edu)

Received October 29, 2002; accepted November 6, 2002.

Notes on efficient ray tracing

Solomon Boulos
University of Utah

There are many ways to make your ray tracer faster. If you're writing an interactive ray tracer, you've got to turn to your bottlenecks in your code and make them scream. You're probably spending the majority of your time computing ray-scene intersections (in some applications, ray-scene intersection may not be the bottleneck, for example Perlin noise is commonly a performance bottleneck for applications that use it heavily). To speed up ray-scene intersections, you use acceleration structures, but how do you get that extra factor of two in performance? This document is some informal notes on experience we've had at Utah on this topic. I do not include citations here. For the sources of these techniques see the bibliography for the chapters from the second edition of *Fundamentals of Computer Graphics* included in these notes. Several papers discussing these techniques are also included in these notes.

I cover two different classes of acceleration structures and what you can do to make them even faster: bounding volume hierarchies (BVHs) and uniform grids (UGs). We do not have as much experience with BSP trees and interested readers should see the work from the University of Saarland group for BSP tree implementation techniques. I'll show code, and discuss the trade-offs involved between each choice. The code examples from the BVH section are slightly modified versions of code from *Realistic Ray Tracing, 2nd Edition*. That original code is available at http://www.cs.utah.edu/˜shirley/galileo/.

Bounding volume hierarchies

A BVH is conceptually simple. It's a tree of bounding volumes, where a bounding volume is usually an axis aligned bounding box that encloses all the surfaces you've got underneath it in the tree. An example of a simple BVH class in C++ looks like this:

```
class BVH : public Surface
{
public:
    // Constructors and such here
    BBox bbox;
    Surface* left;
    Surface* right;
};
```

As you can see, we have a bounding box for our node and pointers to our two children. To build a BVH, you choose some way to split up a list of primitives into two separate lists and put them into the left and right children as you see fit while making sure that your bounding box surrounds all the primitives. In C++ you get something like this:

```
BVH::BVH(Surface** surfaces, int num_surfaces, int axis)
{
   if (num_surfaces == 1) { *this = BVH(surfaces[0], surfaces[0], axis); return; }
   if (num_surfaces == 2) { *this = BVH(surfaces[0], surfaces[1], axis); return; }

   // surround all the objects in the list
   bbox = surround(surfaces, num_surfaces);
   Vector3 pivot = (bbox.max() + bbox.min()) / 2.0;

   // split up the primitives and tell me where the end of the left node is
   int mid_point = qsplit(surfaces, num_surfaces, pivot[axis], axis);

   // create a new bounding volume
   int next_axis = (axis + 1) % 3;
   left  = buildBranch(surfaces, mid_point, next_axis);
   right = buildBranch(&surfaces[mid_point], num_surfaces - mid_point, next_axis);
}
```

This constructor takes a list of Surface pointers and an axis, and produces a BVH. You include the axis parameter so you can switch which axis you split the primitives along. The *qsplit* function called here is similar to the way a standard qsort works, except that we only have to move objects to one side of a splitting plane (the pivot point). Again, the choice of construction algorithm is entirely up to you and the performance of your BVH depends strongly upon it, but it is an open question as to how you might build an optimal BVH (or at least something that performs really well for a variety of situations). *buildBranch* is essentially a copy-and-paste from this default constructor except you can return something other than a BVH pointer for those first cases (e.g. return surfaces[0] if there is only one object).

One of the nicest things about the BVH is how simple it is to intersect with a ray:

```
bool BVH::hit(Ray &r, HitRecord& rec, Context& context)
{
    if ( !(bbox.rayIntersect(r, r.tmin, r.tmax))) return false;

    bool isahit1 = left->hit(r, rec, context);
    bool isahit2 = right->hit(r, rec, context);
    return (isahit1 || isahit2);
}
```

From this code we can see that we first test a ray against our bounding volume. If we don't hit the bounding volume, we immediately return false. If we do hit the bounding volume we recurse. You may have just realized we're about to do a lot of bounding box intersection tests. Currently, the best method I know of asks for a little bit of extra storage in your Ray class but gives a substantial improvement in performance (Williams et al. 2005). There is also a recent JGT submission discussing a Ray-box test using Plücker coordinates, but we have not implemented this algorithm ourselves.

So those are the basics of BVH. How do we make it better? Assuming you think your construction is rock solid, but you just wish the traversal were faster, the first question is probably "why left before right, why not right before left?" This is a very good question. In fact, if you switch between left and right you'll even notice a difference for some scenes. What if we could choose the side based on something we know about the ray? Since we were already using the Williams bounding box test, which required us to store bitwise values that determined whether or not we are going in the positive or negative x,y and z directions, we use this to our advantage. The BVH node changes slightly:

```
class BVH : public Surface
{
public:
    // Constructors and such here
    BBox bbox;
    Surface* child[2];
    int split_axis;
};
```

and the hit function changes similarly:

```
bool BVH::hit(Ray &r, HitRecord& rec, Context& context)
{
    if ( !(bbox.rayIntersect(r, r.tmin, r.tmax))) return false;

    bool isahit1 = child[r.posneg[(split_axis*2)]]->hit(r, rec, context);
    bool isahit2 = child[r.posneg[(split_axis*2)+1]]->hit(r, rec, context);
    return (isahit1 || isahit2);
}
```

Here *r.posneg* stores a 0 if the ray is moving to the right in that axis and a 1 otherwise. In our experience this modification gives a non-trivial performance benefit over either static choice (left then right or right then left). Alternatively, if we switch the order of traversal, we perform worse than either of the static choices. It should be noted that this modification is essentially an algorithmic change in traversal. You're trying to find the earliest intersection in a scene, so this algorithm chooses the node that would be "in your way". If you're going to the right, it first checks the left node, and if you're going left it first checks the right node.

Other researchers have tried other things like reordering the nodes in depth first search order to obtain higher memory coherence (Smits 1998). There have been other discussions of how to choose a splitting axis, but the most commonly used scheme is that shown here: to start with some axis and cycle through the axes in order. More discussion about these issues can be found in the *Ray Tracing News* (http://www.acm.org/tog/resources/RTNews/html/rtn_index.html#spatial).

Uniform grids

The UG is also conceptually simple. Take your list of objects, build a big box around them then cut it up into smaller boxes. When a ray hits your grid, you iteratively traverse your grid using a 3D-DDA algorithm (Woo 87). Grid traversal has been covered in great detail, and the basic thing to remember to do is to avoid recomputing anything you don't need to during the inner most loop. Grid construction has also been discussed by many researchers and the best resource for any of this is the ray tracing news. Once you've got a basic grid implementation, the question is how to make it faster. First, if you're adding adding geometry to grid cells because their bounding boxes overlap, you're paying a high price without a good reason. Most likely, your large scene has at least some number of triangles in it. An excellent code for box-triangle overlap is on Tomas Akenine-Möller's web site. It is faster and more stable than previous methods and very simple to add to your code library. I strongly recommend that any object you are inserting into your grid is tested to make sure it actually overlaps your grid cell. This simple change gave a 15-17% boost in performance for a simple scene with the Stanford bunny. Other Box-object tests also exist, and a list of them can be found on the *Real-Time Rendering* website (http://www.realtimerendering.com/int).

Most grid implementations have some sort of way they store their grid data, for example a 3D array of lists of pointers. In a similar manner to the common Matrix-Matrix multiply optimization, you get very different results based on how you traverse this data due to the memory layout. If for example, you had a 3D array such as this:

```
Surface* data[nx][ny][nz];
```

you would pay very little cost in memory penalties for traversing in the z direction, but a very large cost for traversing in the x direction (and this only gets worse as your memory requirements increase). In ray tracing, and more so in path tracing, rays are bouncing in all sorts of directions. You could definitely layout your memory for a particular view if you wanted to, but doing this for each view is incredibly costly (and could almost certainly never be done interactively). Instead, it is better to arrange your data in a bricked fashion so that you never pay a huge cost in stride for any direction you travel. You won't necessarily do as well for the rays that would have been at ideal cost, but you won't do nearly as poorly for the rays that would have had the worst cost possible.

Again, a lot of improvement can also be gained at the algorithmic level. If we instead use a hierarchy of uniform grids (unfortunately there is no standard term for this in the literature) we can reduce the size of the object lists in each cell. Automatically generating a grid to perform well is essentially black magic, but without any explanation of how to do it, you can achieve up to 40% improvements in run time from simply building a new uniform grid whenever a cell is too densely populated. For example, if you build a grid by having a 3D array of lists of object pointers, you could do a pass over the grid after you've built it and check for lists that are say longer than 16 elements. In any such cell, you could take that list of objects and turn it into a new grid. This would be a particularly simple implementation, and seems to work pretty well in practice.

Another improvement involves maximizing cache coherence. For example, if you allocate a pointer for each object as you insert it into the grid, you will cause a large amount of fragmentation within your grid. If instead in a first pass you created a "grid" holding the number of objects that overlap a cell (instead of pointers to the objects that will eventually go there) and then allocated a big chunk of memory you can remove the penalty of fragmentation (you then loop over your grid again plugging in values for the pointers). This technique may also reduce your construction time despite the two passes due to the reduction in memory allocation calls (system calls always cost a fortune).

Grids vs BVHs

So the question now is which one to use? Or should you use a BSP? The short answer is that it depends. The long answer involves explaining what it depends on. The correct answer is that nobody really knows. But I'll give the long answer.

There are a few different issues that warrant some (mostly high-level) discussion, including very large scenes, object distribution and material properties. We'll talk about each of these issues in turn, and remember that for the most part this discussion assumes that each of your acceleration structures is implemented equally well (which may or may not be true in practice as people have very different mileage for each data structure).

Large scenes

Large scenes are those which are simply not possible in a 32-bit address space. A scene including the David model from Stanford (the model file alone is 1.1GB) would be a good example. In the 64-bit address space pointers are now 8 bytes long to allow you to address all that memory. The impact for you is that your data structures may now suddenly require twice as much storage.

Instead of using pointers we can store a list of objects that we wish to access and index into them using an appropriately sized integer value. For example, as long as you have less than 2^{32} objects in your scene, you can get away with a simple 4 byte unsigned integer. In the general case, you only need n-bit indices, where n is such that 2^n is greater than the number of objects you need to index. Unless memory was really tight, I'd recommend sticking with the simple integer.

This brings up a common technique whenever you have lots and lots of instances of a data structure: make it smaller. For example, a common representation for a KD-Tree node would contain two pointers to child nodes, an integer for the split axis and a floating point position of the splitting plane. This leads to at least a 24 byte structure on a 64-bit machine. Ingo Wald has demonstrated a more efficient representation requiring only 8 bytes of storage. This improves cache line reuse and greatly reduces memory requirements, and his representation does so without a loss in accuracy.

Object distribution

So let's say you have a big list of primitives (spheres, polygons, etc), what kind of an acceleration structure should you put them into? I find this question is best answered by looking at each data structure separately and then comparing them afterwards.

A BVH is ideal for sparse scenes. When you build a BVH, you have the ability to group the objects into two separate clusters that may be separated by large portions of space. Also, if the bounding boxes of your node's children (the left and right children's bounding boxes) don't overlap you can get an instant stopping criteria. For example, assume that you have a ray entering from the right and some objects clustered as shown in Figure 1. If you were to test the right box first, you'd find the first intersection and produce a shorter ray, which would no longer hit the left box. This exit early condition is not possible if the boxes overlap a lot because even the clipped ray will still hit the other box. So one of the biggest weaknesses of the BVH is when you have geometry with strong overlap.

What does this mean to you? For a dense mesh, such as the Stanford Buddha, your BVH may not perform as well as it would for a scene composed with the same number of non-overlapping primitives. This doesn't mean it won't perform pretty well, but there's definitely room for improvement. The take home message: BVHs are very natural for sparse scenes, since you can take advantage of early exits but maybe you should use something else for dense regions of your scene.

In contrast to the BVH, the Uniform Grid is meant for dense data. When you build a uniform grid, you usually dice up the overall bounding box into equally sized cells, which means that for a sparse scene, you have a lot of empty cells, which you'll still end up checking when you go along intersecting (although you still move pretty quickly through them). Wasting time moving through empty cells, and worse yet spending any amount of storage on empty cells is a problem for uniform grids. To avoid this problem, you can make cells bigger so that you jump over more empty space more quickly, but then you have some cells with lots of primitives inside of them. This problem is commonly referred to as the "teapot in the stadium problem", where you have a high resolution version of the Utah teapot in the center of a large low resolution stadium. This scene would have a very large bounding box, but to obtain a suitable grid resolution for the teapot you might have to use very small cells.

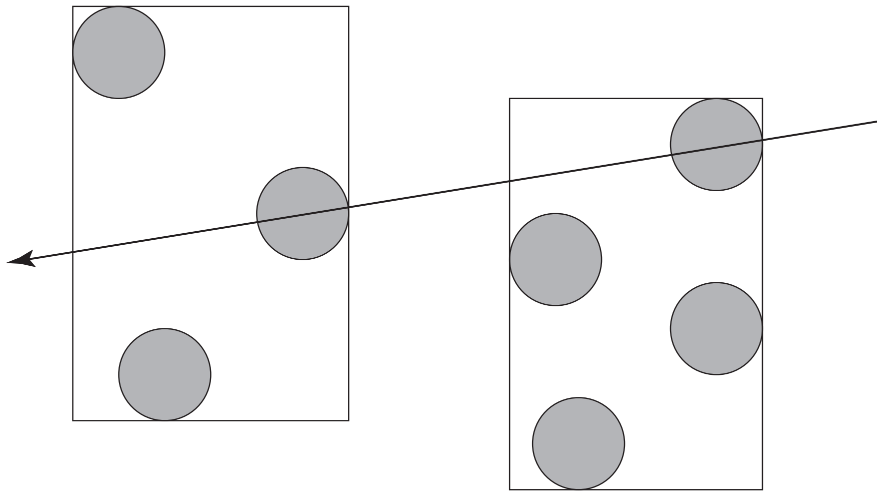


Figure 1: A ray coming from the right should test the right subtree first.

The common solution for this is not to put things into any acceleration structure blindly. Most likely, you have a high level understanding that your teapot is an object on its own and could make a uniform grid out of this object and then place the teapot-grid into a BVH in combination with the stadium (thus taking advantage of the sparse structure of the stadium). How do you do this automatically? There have been lots of papers, but this is largely an unanswered question. One of the largest barriers to answering the question is that there isn't a suitable set of test scenes to test the performance of acceleration structures. The SPD scenes have been useful as a a ray tracing benchmark, but are no longer representative of the types of scenes you would want to render in a modern rendering system.

The take home message: grids work very well for dense data such as meshes and volumes, but you pay a price for traversing and storing the empty cells. There has not been much work in adaptive resolution grid structures in ray tracing (although there were a handful between 1987 and 1997), but the basic idea involves automatically isolating dense regions of space and putting them into a structure and then placing the result into a coarser representation (or a different structure entirely, such as a BVH).

Ray casting versus ray tracing

This is not a commonly discussed problem with these different acceleration structures, but in practice is incredibly important. For example, some acceleration structure papers have only considered ray casting (sending primary rays from the observer towards the scene) which involves no secondary bounces. This usually means that all rays are starting well outside the acceleration structure and are very coherent (traveling in the same direction and likely to touch adjacent memory). A more interesting situation occurs when we consider rays that start on or inside the acceleration structure.

The two data structures we've discussed above perform quite differently for what I'll call a "starting cost." For example, for a uniform grid, you can determine in constant time the grid cell you are in when you start a ray inside the grid. For a BVH, you usually provide a ray to the top level node and traverse down the hierarchy, despite the fact that you might know you're inside the acceleration structure already. As you consider larger and larger scenes, the height of the hierarchy continues to grow and suddenly the $\log n$ traversal starting cost becomes larger and larger. This applies to all hierarchical data structures.

This basic problem leads to an optimization present in some interactive ray tracers: if you don't allow objects to be placed inside your dielectrics, you can avoid a scene intersection test for transmitted rays and only perform a test against the dielectric object. This is an interesting optimization because it offers a huge performance benefit for large scenes containing dielectrics (imagine a glass coffee table in a complex scene). An open question would be how to take advantage of this property automatically, without requiring the user to tell you that nothing is inside the space you're interested in testing. This problem comes up any time you have rays entering the model, but again as I mentioned the same issue is true when you're sending secondary rays from off of the model as well. It would be interesting to see more research on data structures that might be able to take advantage of these situations. For the most part the uniform grid already achieves this due to its negligible (constant time) startup cost, so the simplest solution might be to investigate how to create adaptive resolution hierarchical grids, so that you can avoid the empty cells.

Summary

We've discussed two of the most common data structures for accelerating ray-scene queries. Hopefully some of the basic optimizations such as memory layout (data bricking, compact data structures) and algorithmic optimizations (early exits, precomputed results) came across clearly as they can greatly improve the performance of your renderer.

There are a lot of other solutions out there, but at Utah we've found the advice given here to be fairly useful in practice. All of the techniques apply to parallel code as well, and we haven't spent any time considering optimizations that only work on a single processor (e.g. mailboxing). For the most part, the basic rules of optimization always hold: optimize the portions of the code that show up in profiling, always consider improving your algorithm and getting it right is more important than making it fast.

Interactive Ray Tracing

Steven Parker William Martin Peter-Pike J. Sloan Peter Shirley Brian Smits Charles Hansen

University of Utah, http://www.cs.utah.edu

Abstract

We examine a rendering system that interactively ray traces an image on a conventional multiprocessor. The implementation is "brute force" in that it explicitly traces rays through every screen pixel, yet pays careful attention to system resources for acceleration. The design of the system is described, along with issues related to material models, lighting and shadows, and frameless rendering. The system is demonstrated for several different types of input scenes.

CR Categories: 1.3.0 [Computer Graphics]: General; 1.3.6 [Computer Graphics]: Methodology and Techniques.

Keywords: Ray tracing, parallel systems, shading models

1 INTRODUCTION

Interactive rendering systems provide a powerful way to convey information, especially for complex environments. Until recently the only interactive rendering algorithms were hardware-accelerated polygonal renderers. This approach has limitations due to both the algorithms used and the tight coupling to the hardware. Software-only implementations are more easily modified and extended which enables experimentation with various rendering and interaction options.

This paper describes our explorations of an interactive ray tracing system designed for current multiprocessor machines. This system was initially developed to examine ray tracing's performance on a modem architecture. We were surprised at just how responsive the resulting system turned out to be. Although the system takes careful advantage of system resources, it is essentially a brute force implementation (Figure 1). We intentionally take the simple path wherever feasible at each step believing that neither limiting assumptions nor complex algorithms are needed for performance.

The ray tracing system is interactive in part because it runs on a high-end machine (SGI Origin 2000) with fast frame buffer, CPU set, and interconnect. The key advantages of ray tracing are:

- ray tracing scales well on tens to hundreds of processors;
- ray tracing's frame rendering time is sub-linear in the number of primitives for static scenes;
- ray tracing allows a wide range of primitives and user programmable shading effects.

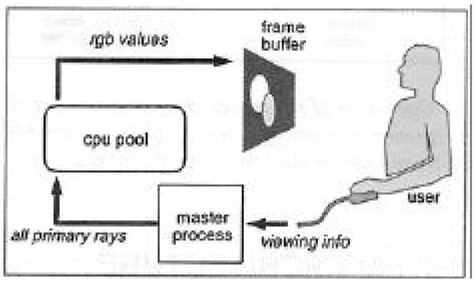


Figure 1: ***The ray tracing system discussed in this paper explicitly traces all rays on a pool of processors for a viewpoint interactively selected by the viewer.***

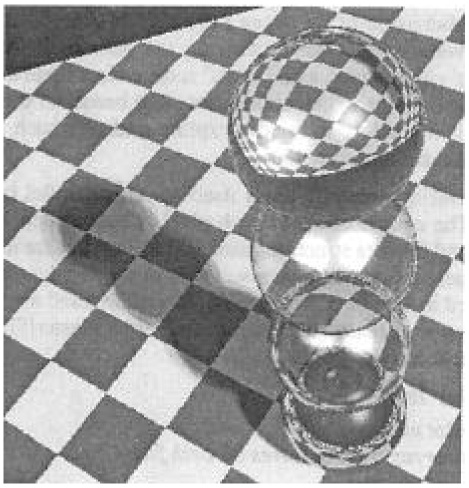


Figure 2: ***A portion of a 600 by 400 pixel image from our system running at approximately fifteen frames per second.***

The first item allows our implementation to be interactive, the second allows this interactivity to extend to relatively large (e.g. gigabyte) scenes, and the third allows the familiar ray traced look with shadows and specular reflection (Figure 2).

In the paper we stress the issues in ray tracing that change when we move from the static to the interactive case. These include achieving performance in synchronous or asynchronous (frameless) fashions (Section 2), and modifications to traditional Whitted-style lighting/shadowing model to improve appearance and performance (Section 3). We also discuss a few areas that might benefit from interactive ray tracing and show some of the environments we used in Section 4. We compare our work to the other work in parallel ray tracing in Section 5. We do not compare our work to the many object space methods available for simulating shadows and non-diffuse effects (e.g. Ofek and Rappoport [22]) which we believe comprise a different family of techniques. Our interactive implementation of ray tracing isosurfaces in trilinear volumes is described elsewhere [23].

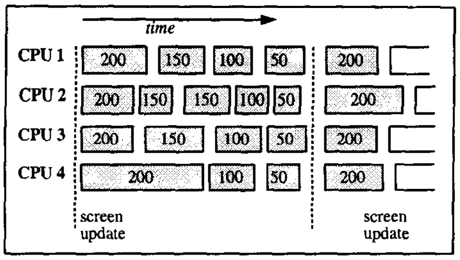


Figure 3: *Operation of ray tracer in synchronous mode. Numbers in boxes represent number of pixels in a block being processed. All pixels are traced before the screen swaps buffers.*

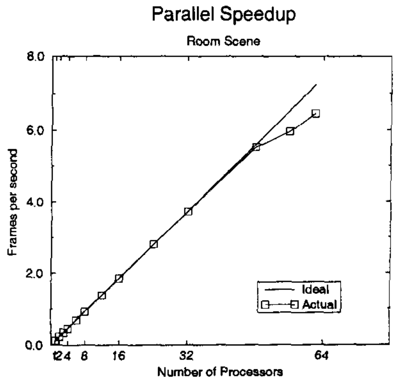


Figure 4: *Performance results for varying numbers of processors for a single view of the scene shown in Figure 16.*

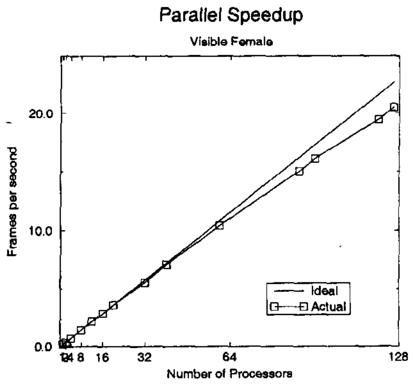


Figure 5: *Performance results for the visible female dataset, shown in Figure 9.*

2 SYSTEM ARCHITECTURE

It is well understood that ray tracing can be accelerated through two main techniques [26]: accelerating or eliminating ray/object intersection tests and parallelization. We employ both techniques in our system. We use a hybrid spatial subdivision which combines a grid based subdivision of the scene [10] with bounding volumes [17]. For a given scene, we can empirically test both methods to arrive at the 'best' combination where 'best' is dependent upon the scene geometry and the particular application. The beauty of the interactive system is the ability to rapidly explore tradeoffs such as different spatial subdivision techniques.

Ray tracing naturally lends itself towards parallel implementations. The computation for each pixel is independent of all other pixels, and the data structures used for casting rays are usually read-only. These properties have resulted in many parallel ray tracers, as discussed in Section 5. The simplest parallel shared memory implementation with reasonable performance uses Master/Slave demand driven scheduling as follows:

```
Master Task
initialize model
initialize ray tracing slaves on each free CPU
loop
  update viewing information
  lock queue
  place all primary rays in queue
  unlock queue
  when the queue is empty redraw screen and handle user input
end loop
```

The ray tracing slaves are simple programs that grab primary rays from the queue and compute pixel RGB values:

```
Slave Task
initialize memory
loop
  if queue is not empty then
    lock queue
    pop ray request
    unlock queue
    compute RGB for pixel
    write RGB into frame buffer pixel
  end if
end loop
```

This implementation would work, but it would have excessive synchronization overhead because each pixel is an independent task. The actual implementation uses a larger basic task size and runs in conventional or frameless mode as discussed in the next two sections.

2.1 Conventional Operation

To reduce synchronization overhead we can assign groups of rays to each processor. The larger these groups are, the less synchronization is required. However, as they become larger, more time is potentially lost due to poor load balancing because all processors must wait for the last job of the frame to finish before starting the next frame. We address this through a load balancing scheme that uses a static set of variable size jobs that are dispatched in a queue where jobs linearly decrease in size. This is shown in Figure 3.

Figure 3 has several exaggerations in scale to make it more obvious. First, the time between job runs for a processor is smaller than is shown in the form of gaps between boxes. Second, the actual jobs are multiples of the finest tile granularity which is a 128 pixel tile (32 by 4). We chose this size for two reasons: cache coherency for the pixels and data cache coherency for the scene. The first reason is dictated by the machine architecture which uses 128 byte cache lines (32 4-byte pixels). With a minimum task granularity of a cache line, false sharing between image tiles is eliminated. A further advantage of using a tile is data cache reuse for the scene geometry. Since primary rays exhibit good spatial coherence, our system takes advantage of this with the 32 by 4 pixel tile.

The implementation of the work queue assignment uses the hardware fetch and op counters on the Origin architecture. This allows efficient access to the central work queue resource. This approach to dividing the work between processors seems to scale very well. In Figure 4 we show the scalability for the room scene shown in Figure 16. We used up to 64 processor (all that are available lo-

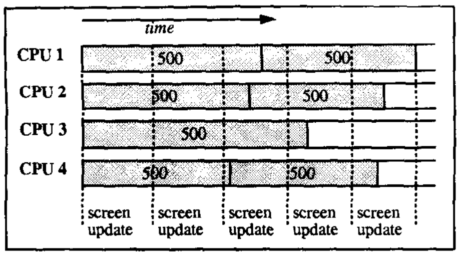


Figure 6: *Operation of ray tracer in asynchronous (frameless) mode. Screen is constantly updating and each processor is repeatedly tracing its set of pixels.*

cally) and found that up through about 48 we achieved almost ideal performance. Above 48 there is a slight drop off. We also show performance data for interactively ray tracing the iso-surfaces of the visible female dataset in Figure 5. For this data we had access to a 128 processor machine and found nearly ideal speed ups for up to 128 processors.

Since most scenes fit within the secondary cache of the processor (4 Mb), the memory bandwidth used is very small. The room scene, shown in Figure 4 uses an average of 9.4 Mb/s of main memory bandwidth per processor. Ironically, rendering a scene with a much larger memory footprint (rendering of isosurfaces from the visible female dataset [23]) uses only 2.1 to 8.4 Mb/s of main memory bandwidth. These statistics were gathered using the SGI perfex utility, benchmarked with 60 processors.

Since ray tracing is an inherently parallel algorithm, efficient scaling is limited by only two factors: Load balance and synchronization. The dynamic work assignment scheme described earlier is used to limit the effect of load imbalance. Synchronization for each frame can limit scaling due to the overhead of the barrier. The standard barrier provided in Irix requires an average of 5 milliseconds to synchronize 64 processors, which limits the scaling at high framerates. An efficient barrier was implemented using the "fetchop" atomic fetch-and-op facilities in the Origin. A barrier operation consumes 61 microseconds on average, which is an insignificant percentage of the frame time.

2.2 Frameless Rendering

For frameless rendering [3, 7, 36] the viewpoint and screen are updated synchronously, but the pixels are updated according to an asynchronous quasi-random pattern. Our implementation for this is summarized in Figure 6.

The implementation assigns a static pixel distribution to the rendering threads - every processor has a list of pixels that it will update, requiring minimal synchronization between threads. The rendering thread handles user input and draws the buffer to the screen at regular intervals. This is done asynchronously to the rendering threads. The rendering threads periodically update their camera - this is done at a specified rate expressed as a percentage of the pixels that thread owns. The display thread is modified so that it updates the screen at some user defined frame rate.

When creating a "static" pixel distribution (partitioning the screen between processors), there are two conflicting goals: 1) maintain coherent memory access; 2) have a more random distribution (incoherent memory) of pixels. The first is important for raw system efficiency, and the second is important to avoid visually distracting structure during updates.

In the current system we partition the image plane using a Hilbert curve (this maps the image to a 1D line), and then break this line into "chunks", these chunks are distributed to the processors in a round robin fashion (processors interleaved with chunk granularity along the 1D domain of the Hilbert curve). Each thread then randomly permutes its chunks so that the update doesn't always exactly track the Hilbert curve.

When updating the image, pixels can be blended into the frame buffer. This causes samples to have an exponential decay and creates a smoother image in space and time. We can use jittered sampling where there are four potential sample locations per pixel and two of them are updated when the pixel is updates, so the pixel is only fully updated after two passes. This implements the "frameless anti-aliasing" concept of Scher Zagier [35].

One nice property of a static pixel distribution is the ease of keeping extra information around (each thread just stores it - and no other threads will access this memory.) This can be used for computing a running average, sub pixel offsets for jittered sampling, a running variance computation or other information about the scene associated with that pixel (velocity, object ids, etc.).

3 IMPLEMENTATION DETAILS

Users of traditional ray tracers feel free to change the lighting and material parameters when the viewpoint is changed, and to add multiple lights to achieve a desired lighting effect. These are not practical in an interactive ray tracer where the lighting and material parameters are static as the viewpoint changes, and where even one light is expensive in terms of framerate. To help reduce the need for such traditional hacks, our implementation modifies the traditional key components of a ray tracer: lighting, material models, shadows, and ray-object intersection routines. In Section 3.1 we discuss how we handle and modify material models and lighting in a dynamic context. In Section 3.2 we discuss how we approximate soft shadows efficiently. In Section 3.3 we discuss how we compute ray-object intersections for spline surfaces.

3.1 Lighting and Materials

The traditional "Whitted-style" illumination model has many variations, but for one light the following formula is representative:

$$L = k_d \left(l_a + s l_e \hat{\mathbf{n}} \cdot \hat{\mathbf{l}} \right) + s l_e k_h \left(\hat{\mathbf{h}} \cdot \hat{\mathbf{l}} \right)^N + k_s L_s + k_t L_t, \quad (1)$$

where the vector quantities are shown in Figure 7, and L is the radiance (color) being computed, s is a shadow term that is either zero or one depending on whether the point luminaire is visible, k_d is the diffuse reflectance, l_a is the ambient illumination, l_e is the luminaire color, k_h is the Phong highlight reflectance, k_s is the specular reflectance, L_s is the radiance coming from the specular direction, k_t is the specular transmittance, and L_t is the radiance coming from the transmitted direction. Although this basic formula serves us well, we believe some alterations can improve performance and appearance. In particular, we are careful in allowing k_s and k_t to change with incident angle, we modify the ambient component l_a to be a very crude approximation to global illumination (Section 3.1.1), and we allow soft shadowing by making s vary continuously between zero and one (Section 3.2). Finally, we break the materials into several classes to compute only non-zero coefficients for efficiency.

One well-known problem with Equation 1 is that the specular terms do not change with incident angle. This is different from the behavior of materials in the real world [14]. In a conventional ray tracer the values of k_d, k_s and k_t can be hand-tuned to depend on viewpoint but in an interactive setting this does not work well. Instead, we first break down materials into a few distinct subjective categories suggested in [31]: *diffuse, dielectric, metal,* and *polished.* The modifications for these materials is described below:

Diffuse. For diffuse surfaces we use Equation 1 with $k_h = k_s = k_t = 0$. This is the same as a conventional ray tracer.

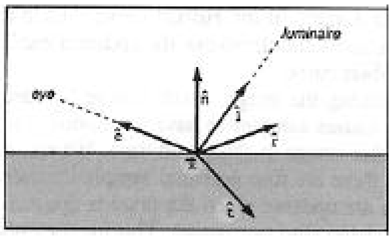


Figure *7:* ***The directional quantities associated with Equation 1.***

Metal. Metal has a reflectance that varies with incident angle [6]. We are currently ignoring this effect, and other effects of real metal, and using traditional Whitted-style lighting. We use Equation 1 with $k_d = k_t = 0$, and $k_h = k_s$.

Dielectric. Dielectrics, such as glass and water, have reflectances that depend on viewing angle. These reflectances are modeled by the Fresnel Equations, which for the unpolarized case can be approximated by a polynomial developed by Schlick [28]:

$$k_s(\hat{\mathbf{e}}, \hat{\mathbf{n}}) = R_0 + (1 - R_0)(1 - \hat{\mathbf{e}} \cdot \hat{\mathbf{n}})^5,$$

and k_t is determined by conservation of energy:

$$k_t(\hat{\mathbf{e}}, \hat{\mathbf{n}}) = 1 - k_s(\hat{\mathbf{e}}, \hat{\mathbf{n}}).$$

The internal attenuation of intensity I is the standard exponential decay with distance t according to extinction coefficient κ: I(t) = $I(0)exp(-\kappa t)$. To approximate the specular reflection of an area light source we add a Phong term to dielectrics as well.

Polished. We use the coupled model presented in [30]. This model allows the k_s to vary with incident angle, and allows the diffuse appearance to decrease with angle. As originally presented, it is a BRDF, but it is modified here to be appropriate for a clamped RGB lighting model with an ambient component:

$$\begin{aligned} L \quad = \quad & k_d l_a(1 - k_s(\hat{\mathbf{e}}, \hat{\mathbf{n}})) + & (2) \\ & s k_d l_e \left(1 - (1 - \hat{\mathbf{n}} \cdot \hat{\mathbf{e}})^5\right) \left(1 - (1 - \hat{\mathbf{n}} \cdot \hat{\mathbf{l}})^5\right) + \\ & s k_h (\hat{\mathbf{h}} \cdot \hat{\mathbf{l}})^N + \\ & k_s(\hat{\mathbf{e}}, \hat{\mathbf{n}}) L_s, \end{aligned}$$

where the first term assumes the ambient component arises from directionally uniform illumination.

3.1.1 Ambient Lighting

The ambient term l_a in Equation 1 is a crude approximation used in conventional ray tracers to avoid computing an indirect lighting term. It is not meant to be physically accurate, but instead to illuminate those areas that are not directly lit by the luminaires. Given this, its main failing is that the uniform intensity causes diffuse objects to appear flat when the surface faces away from the light source. One way to avoid this is to put a fill-light at the eye point, but we feel this is distracting for a moving viewpoint. An alternative is to allow the ambient coefficient to vary with position $\vec{\mathbf{p}}$ and orientation: $l_a(\vec{\mathbf{p}}, \hat{\mathbf{n}})$. This can be a simple heuristic, or based on radiosity solutions [13]. Our motivation for using a more sophisticated ambient term is to allow shading variation on surfaces that are not directly lit without the computational cost of adding additional lights. This can be accomplished by assuming the ambient term arises due to illumination from a background divided evenly between two intensities A and B (Figure 8). The angle θ will vary from zero to π radians. For θ = 0 the surface will only "see" *A so* the ambient term will be *A.* As θ increases the ambient term will

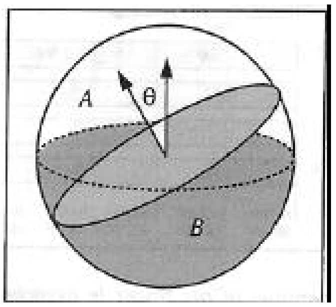


Figure *8:* ***A surface is illuminated by a hemisphere with colors A and B.***

gradually change to $(A + B)/2$ at $\theta = \pi/2$, and finally change to B as the surface fully faces the bottom hemisphere. Nusselt's analog [4] allows us to derive the full relationship:

$$l_a(\theta) = \begin{cases} \left(1 - \frac{\sin\theta}{2}\right) A + \left(\frac{\sin\theta}{2}\right) B & \text{if } \theta < \frac{\pi}{2}, \\ \left(\frac{\sin\theta}{2}\right) A + \left(1 - \frac{\sin\theta}{2}\right) B & \text{otherwise.} \end{cases}$$

Either the user can set A and B algorithmically or by hand. We set ours by hand, but some heuristics can aid in selection. If we envision the hemisphere-pair as approximating indirect lighting of an object in a room, then the "walls" opposite the light are well illuminated and bright. So the hemisphere can be roughly aligned to the light source, with the hemisphere in the direction of the light source darker than the one pointing away from the light source. As advocated by Gooch et al. [12], we can accent the shape using a cool-to-warm color shift by making sure the light source is yellow (warm) and the hemisphere facing away from the light is blue (cool). Our ambient approximation is shown in Figure 9 and is not measurably slower than a constant ambient component for non-trivial mmodels.

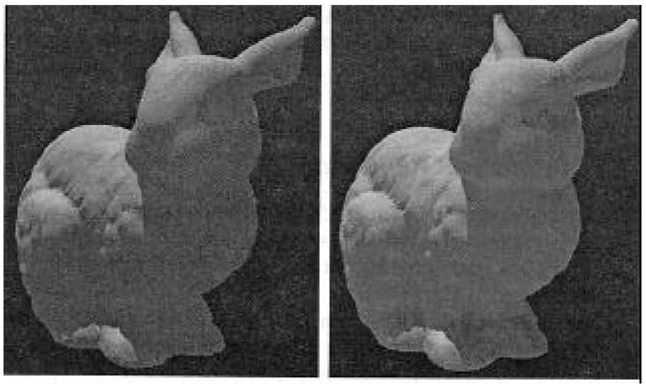


Figure *9:* ***Left: simple ambient approximation. Right: directionally varying ambient approximation.***

3.2 Shadows

One of the limitations of ray tracing is the hard edges computed for shadows. In addition to aesthetic reasons, there is evidence that soft edged shadows aid in accurate spatial perception [19]. Ray tracing methods that produce accurate soft shadows such as ray tracing with cones [l] or probabilistic ray tracing [5] stress accurate soft shadows, but dramatically increase computation time relative to hard shadow computation. In this section we examine how to compute soft-edged shadows approximately so that interactivity can be maintained.

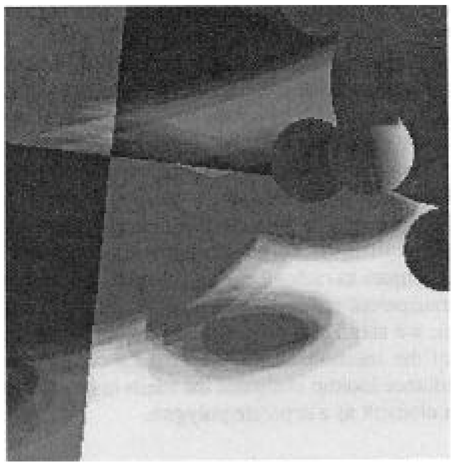


Figure 10: *A beam traced shadow with five samples. Note that there are discontinuities within the shadow.*

One option to improve performance is to do explicit multi-sampling of the light with a "beam" made up of a small number of rays [15]. Since the number of rays is small, there will be visual artifacts but interactive performance will be possible (Figure 10). To speed up this computation we can precompute the rays in the beam if we assume the luminaire is far away, and we can vectorize the intersection computation against each geometric primitive. This is similar to traversing efficiency structures using bundles rather than single rays [9]. Although this optimization gives us a factor of two performance over the unvectorized version, it is still too slow for many shadow rays.

An alternative to computing accurate soft shadows is to soften the edges of hard shadows. This is essentially the technique used in depth buffer algorithms [25] where the binary shadow raster can be filtered. However we want to simulate the change in penumbra width we see in real shadows. Such an effect requires more sophisticated filtering. This means shadow penumbra width should behave in a believable way, starting at zero at the occluder and increasing linearly with distance from the occluder.

It is hard for observers to tell the difference between shadows cast by differently shaped lights. For this reason we assume spherical lights. We do a rough calculation at each illuminated point of what fraction s of the light is visible, and attenuate the unshadowed illumination by s. Thus our goal is to estimate s in efficiently and to visually plausible results.

Rather than creating a correct shadow created by an area source, the algorithm creates a shadow of a ***soft-edged** object* from a point source (Figure 11). The penumbra is the shadow of the semi-opaque (outer) object that is not also shadowed by the opaque (inner) object. The transparency of the outer object increases from no transparency at the inner object to full transparency at the boundary of the outer object. For an isolated object, we can use inner and outer offsets of the real object to achieve believable results. We also need to make the intensity gradient in the penumbra natural. This can be achieved by computing the shadowing variable s beginning at $s = 0$ on the penumbra/umbra boundary (the surface of the inner object) and increasing non-linearly with distance to $s = 1$ on the outer boundary of the penumbra (the surface of the outer object).

The above approach will give an approximate soft shadow. The size of the penumbra is based on the size of the offsets used to create the inner and outer objects. In order to have the penumbra width change plausibly, the offsets need to change based on the distance along the shadow ray and the size of the light source, as illustrated in Figure 12. This requires modifying the intersection tests

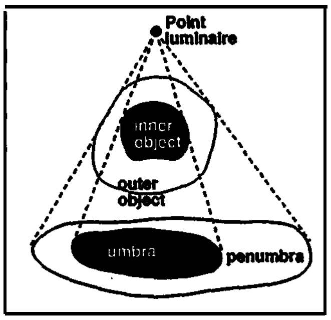


Figure 11: ***The inner object is opaque and the outer object's opacity falls off toward its outer boundary.***

for shadow rays. The details of this approach, including solutions to light leaking between two objects and the intersection tests and bounding box construction for polygons and spheres with varying offsets are discussed in more depth by Parker *et al. [24]. The* results are shown in Figure 13.

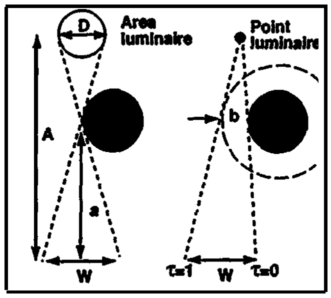


Figure 12: ***Choosing the size of the outer objectfor a given configuration.***

3.3 Spline Surfaces

In most traditional rendering systems NURBS are tessellated, often outside the graphics API in order to have more control over the accuracy. This can lead to an explosion in the amount of data that needs to be stored and then sent down the rendering pipeline. Ray tracing does not have this limitation.

Intersection tests with NURBS have been done in several ways (e.g., [16, 29, 33]. Our approach computes an estimate to the intersection point and then uses a brute force approach to compute

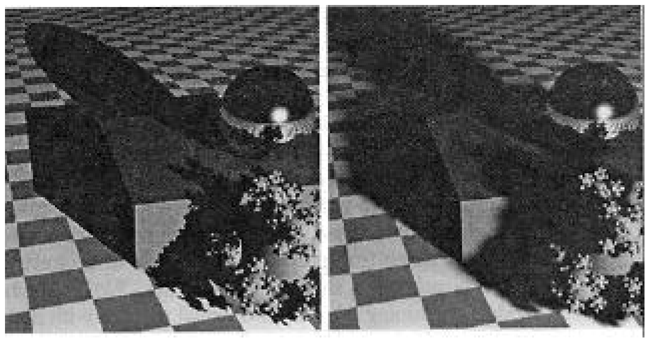


Figure 13: ***Left: one sample per pixel with hard shadows. Right: one sample per pixel with soft shadows. Note that the method captures the singularity near the box edge.***

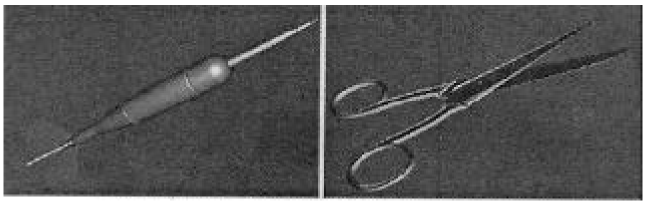


Figure 14: ***Two images rendered directly from the spline model.***

the actual intersection point. Surface parameter spaces are subdivided to a user-specified depth, and the quadtree that results is used to construct an inn-a-surface bounding volume hierarchy. Axis-aligned bounding volumes are used to preserve consistency with the overall infrastructure. Bounding volume hierarchies are built bottom up. It is important to note that this will result in tighter volumes than top down construction (since the subdivided control meshes converge to the surface).

Intersections with the leaf nodes of the bounding volume tree are computed using Broyden's method. This is a pseudo-Newton solver which approximates the value of the Jacobian. It converges more slowly than Newton, but requires fewer function evaluations. The initial guess is given by the average of the boundary parameter values of the patch in question. Patches are allowed to overlap by a small percentage of their parametric domains, thereby lessening the chance of cracks.

Usually fewer than three iterations of the root finder are required to converge to a suitably refined surface. The cost of storage is one copy of the original control mesh, and for each leaf node in the intra-surface bounding volume hierarchy, four doubles denoting the parametric interval it covers. In addition, we require each processor to reserve a scratch area so that the spline basis functions can be computed without needing to lock the data. The cost of this storage is $m+n$ where m and n are the maximum control mesh dimensions over all surfaces in the scene.

4 RESULTS

The final rendering system is interactive on relatively few (8) processors, and approaches real time for complex environments on 64 or more processors. It runs well on a variety of models from different application areas. Its flexibility allows several different display modes, all of which are applicable to 'the various models.

Ray tracing is ideal for showing dynamic effects such as specular highlights and shadows. Dynamic objects are more difficult to incorporate into a ray tracer than into a z-buffer algorithm as current acceleration schemes are not dynamic [11]. Our current workaround is to keep dynamic objects outside the acceleration scheme and check them individually for each ray. Obviously this only works for limited numbers of dynamic objects. In Figure 2 we show a static image from a set of bouncing balls using the soft shadow approximation.

Computer-aided design usually uses both curved surfaces and non-diffuse objects, such as a windshield made from glass. Ray tracing can render curved surfaces directly, making it ideal for spline models. The ability to calculate accurate reflections across the surface make is possible to evaluate the smoothness and curvature of the models for aesthetic purposes. A sample of a directly ray traced spline primitives is shown in Figure 14. We have run on several models containing 20-2000 individual patches with runtimes ranging from 1-20 fps at 512 by 512 pixels on 60 processors.

Ray tracing time is sub-linear with respect to model size. This allows us to interact with very large models. One area that creates large models is scientific visualization. In Figure 15 we show a visualization of a stress simulation. Each node in the simulation is represented by a sphere. There are 35 million spheres in this model. Unlike conventional rendering systems, the high depth complexity has very little effect on the rendering times. Another area that can create complex models is architectural design. The model in Figure 16 contains roughly 75,000 polygons and a spline teapot. An area we would like to explore is the use of interactive ray tracing for walk throughs of globally illuminated static environments, where the illumination information has been computed in advance by such techniques as radiosity or density estimation. Usually specular and transparent effects are missing from such walk throughs. In addition, we should be able to easily allow higher order reconstruction of the solution. Also, we could greatly reduce polygon count if radiance lookup evaluates the mesh instead of representing each mesh element as a separate polygon.

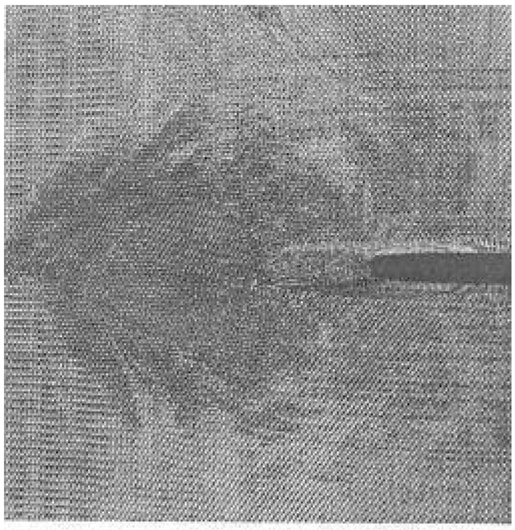


Figure 15: ***Simulation of crack propagation visualized using 35M spheres. This image at 512 by 512 pixels runs approximately 15 frames per second on 60 CPUs.***

5 RELATED WORK

Ray tracing has long been a focus for acceleration through parallel techniques. There are two general parallel methods which are used for such acceleration: demand scheduling and data parallel. Demand driven scheduling refers to distributing tasks upon demand. Data parallel methods partition the data and assign tasks to the processors which contain the required data. Hybrid methods combine these two paradigms generally by partitioning the tasks into those requiring a small amount of data and those requiring a large amount. Since shared memory processors with large amounts of memory have only recently been commercially available, most parallel ray tracing research has focused on distributed memory architectures. For shared memory parallel computers, demand driven scheduling methods tend to lead to the best performance [26]. Our implementation is based on demand driven scheduling where the task granularity is rendering an 32 by 4 pixel tile. In this section, we provide a comparison with several related parallel ray tracing implementations. A more thorough general review is provided by Reinhard and Jansen [26].

Muuss and researchers from ARL have experimented with parallel and distributed ray tracing for over a decade [20, 21]. In their recent work, they describe a parallel and distributed real-time ray

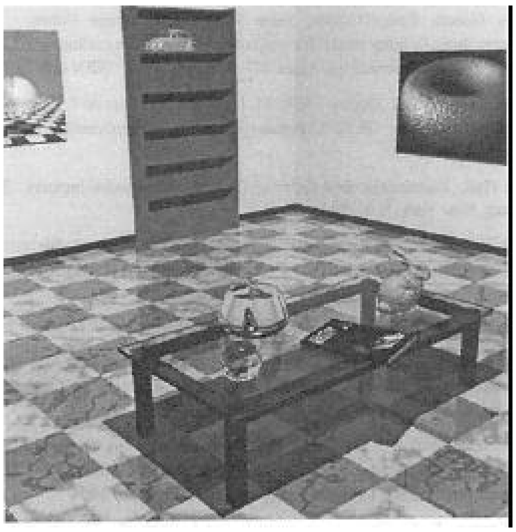


Figure 16: A ***model with splines, glass, image textures, and procedural solid textures. At 512 by 512 pixels this image is generated at approximately 4 frames per second on 60 CPUs.***

tracer running on a cluster of SGI Power Challenge machines [21]. One of the differences between BRL's effort and ours is the geometric primitives used. Their geometry is defined by through a CSG modeler, BRL-CAD. Additionally, we leverage the tight coupling of the graphics hardware on the SGI Origin while their system uses an image decomposition scheme combined with a network attached framebuffer. Muuss points out that synchronization, particularly at the frame level, is critical for real-time ray tracing [21]. Our research indicates that synchronization within a frame is also critical as noted by our dynamic load balancing scheme. Although not reported in the literature, ARL's current effort seems to have a comparable framerate as ours (Muuss, personal communication at SIGGRAPH98).

Keates and Hubbold use a distributed shared memory architecture, the KSRl, to implement a demand driven ray tracer which renders a simple scene is slightly over 1.8 seconds for 256 processors [18]. Their implementation is similar to ours in that they use the brute force technique of parallelizing over rays. However, their work differs in the granularity of work distribution, the method used for load balancing, and results based upon architecture. Their implementation split the screen into regions and divided the work among the CPUs. It is not clear how large the regions were but one is lead to believe the regions are larger than the 32 pixel regions used in our implementation. They report problems with load balancing and synchronization. They address these by a two level hierarchy for screen space subdivision similar to Ellsworth [8]. Our system uses a different strategy for load balancing of decreasing granularity of assigned work which empirically yields better results. This also assists in synchronization which is why this issue has not been a problem for us.

Singh et al. reported on using the Stanford DASH distributed shared memory machine for ray tracing [32]. Their implementation used an image decomposition scheme which subdivided the image among the available processors. Within a processor, the sub-image a further subdivided into 8 by 8 pixel tiles. As in our system, their implementation noted the advantage of data cache reuse for object intersection test. Their work differed from ours in the load balancing scheme. They used task stealing rather than demand driven scheduling. We find that the simpler approach of using a task queue with good dynamic load balancing provides excellent results without the complexity of performing task stealing. The fetch and op hardware in the Origin architecture allows the task queue to perform well even on a large number of processors.

Yoon et al. use an image partitioning scheme which statically load balances the tasks by interleaving pixels and distributing among nodes the scene data while replicating the spatial hierarchy on each node [34]. Their work attempts to prefetch data for each ray task. Their work differs from ours in two major respects: load balancing and machine architecture. Our implementation effectively exploits dynamic load balancing through the heuristic of decreasing task size while Yoon et al. employ static load balancing through pixel assignment. Since their work focuses on a distributed memory architecture, they need to explicitly address data distribution while our implementation exploits the CC-NUMA distributed shared memory.

Reinhard and Jansen use a hybrid scheduling method for parallel ray tracing [27]. Their implementation divides the ray tracing task into those tasks which require limited amounts of data and those that require more substantial amounts of data. Since their spatial subdivision hierarchy, but not the leaf nodes, is replicated on each processor, tasks using these are demand scheduled whereas tracing rays through the objects within the leaf nodes is performed in a data parallel fashion. Their method makes novel use of this combined scheduling scheme which provides better performance on distributed memory parallel computers. Since our method exploits the distributed shared memory architecture, we can achieve very good performance with only demand scheduling.

Bala et al. describe a bounded error method for ray tracing [2]. For each object surface, their method uses a 4D linetree to store a collection of interpolants representing the radiance from that surface. If available, these interpolants are reprojected when the user's viewpoint changes. If not, the system intersects the ray with the scene checking for a valid interpolant at the intersection point. If one is found, the radiance for that pixel is interpolated. Otherwise, using that linetree cell, an attempt is made to build an interpolant. If this is within an error predicate, it is used otherwise the linetree cell is subdivided and the system falls back to shading using a standard ray tracing technique. The acceleration is based upon the utilization of previously shaded samples bounded by an error predicate rather than fully tracing every ray. Our system is brute-force and traces every ray in parallel. Bala's method is oriented toward a more informed and less parallel strategy, and is currently not interactive. Moving objects would pose a problem for the linetree based system whereas they can be handled in our implementation. Using reprojection techniques might further accelerate our system.

6 CONCLUSION

Interactive ray tracing is a viable approach with high end parallel machines. As parallel architectures become more efficient and cheaper this approach could have much more widespread application. Ray tracing presents a new set of display options and tradeoffs for interactive display, such as soft shadows, frameless rendering, more sophisticated lighting, and different shading models. The software implementation allows us to easily explore these options and to evaluate their impact for an interactive display.

We believe the following possibilities are worth investigating:

- How should antialiasing be handled?
- How do we handle complex dynamic environments?
- How do we ensure predictable performance for simulation applications?
- What should the API be for an interactive ray tracer?
- How could an inexpensive architecture be built to do interactive ray tracing?

The first three items above highlight significant limitations of our current system: antialiasing is brute-force and thus too costly, and performance can be slow or unpredictable because there is a complex interaction between efficiency stucture build time, traversal time, and view-dependent performance. How much of these are due to the batch nature of traditional ray tracing methodology versus intrinsic limitations is not yet clear.

Additionally, we feel that an interactive ray tracer can help answer more general questions in interactive rendering, such as:

- How important are soft shadows and indirect illumination to scene comprehension and how accurate do they need to be?
- Are more physically accurate BRDF's more or less important in an interactive setting?
- Do accurate reflections give significant information about surface curvature/smoothness?

The ability to have more complete control over these features allows us to investigate their effects more completely.

Acknowledgments

Thanks to Chris Johnson for providing the open collaborative research environment that allowed this work to happen. Thanks to Elaine Cohen for many conversations on spline surface intersection and to Amy Gooch for help with model acquisition and conversion. Thanks to Dave Beazley and Peter Lomdahl at Los Alamos National Laboratory for the 35 million sphere data set. Special thanks to Jamie Painter and the Advanced Computing Laboratory at Los Alamos National Laboratory for access to a 128 processor machine for final benchmarks. The bunny model appeared courtesy of the Stanford University Graphics Laboratory. This work was supported by the SGI Visual Supercomputing Center, the Utah State Centers of Excellence, the Department of Energy and the National Science Foundation.

References

[1] J. Amanatides. Ray tracing with cones. *Computer Graphics*, pages 129–135, July 1984. ACM Siggraph '84 Conference Proceedings.

[2] Kavita Bala, Julie Dorsey, and Seth Teller. Bounded-error interactive ray tracing. Technical Report LCS TR-748, MIT Computer Graphics Group, March 1998.

[3] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J. Scher Zagier. Frameless rendering: Double buffering considered harmful. *Computer Graphics*, 28(3):175–176, July 1994. ACM Siggraph '94 Conference Proceedings.

[4] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, Boston, MA, 1993.

[5] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics*, 18(4):165–174, July 1984. ACM Siggraph '84 Conference Proceedings.

[6] Robert L. Cook and Kennneth E. Torrance. A reflectance model for computer graphics. *Computer Graphics*, 15(3):307–316, August 1981. ACM Siggraph '81 Conference Proceedings.

[7] Robert A. Cross. Interactive realism for visualization using ray tracing. In *Proceedings Visualization '95*, pages 19–25, 1995.

[8] David A. Ellsworth. A new algorithm for interactive graphics on multicomputers. *IEEE Computer Graphics and Applications*, 14(4), July 1994.

[9] Bernd Fröhlich. *Ray Tracing mit Strahlenbündeln (Ray Tracing with Bundles of Rays)*. PhD thesis, Technical University of Braunschweig, Germany, 1993.

[10] Akira Fujimoto, Takayu Tanaka, and Kansei Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics & Applications*, pages 16–26, April 1986.

[11] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

[12] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *SIGGRAPH 98 Conference Proceedings*, pages 447–452, July 1998. ISBN 0-89791-999-8.

[13] Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg. The irradiance volume. *IEEE Computer Graphics & Applications*, 18(2):32–43, March–April 1998.

[14] Roy Hall. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, N.Y., 1988.

[15] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 119–127, July 1984.

[16] James T. Kajiya. Ray tracing parametric patches. In *SIGGRAPH '82*, 1992.

[17] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, August 1986. ACM Siggraph '86 Conference Proceedings.

[18] M.J. Keates and R.J. Hubbold. Accelerated ray tracing on the KRS1 virtual shared-memory parallel computer. Technical Report UMCS-94-2-2, Computer Science Department, University of Manchester, February 1994.

[19] D. Kersten, D. C. Knill, Mamassian P, and I. Bülthoff. Illusory motion from shadows. *Nature*, 379:31, 1996.

[20] Michael J. Muuss. Rt and remrt - shared memory parllel and network distributed ray-tracing programs. In *USENIX: Proceedings of the Fourth Computer Graphics Workshop*, October 1987.

[21] Michael J. Muuss. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium*, June 1995.

[22] Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In *SIGGRAPH 98 Conference Proceedings*, pages 333–342, July 1998.

[23] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings Visualization '98*, 1998.

[24] Steven Parker, Peter Shirley, and Brian Smits. Single sample soft shadows. Technical Report UUCS-98-019, Computer Science Department, University of Utah, October 1998. http://www.cs.utah.edu/~bes/papers/coneShadow.

[25] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics*, 21(4):283–292, July 1987. ACM Siggraph '87 Conference Proceedings.

[26] E. Reinhard, A.G. Chalmers, and F.W. Jansen. Overview of parallel photo-realistic graphics. In *Eurographics '98*, 1998.

[27] Erik Reinhard and Frederik W. Jansen. Rendering large scenes using parallel ray tracing. *Parallel Computing*, 23(7), July 1997.

[28] Christophe Schlick. A customizable reflectance model for everyday rendering. In *Proceedings of the Fourth Eurographics Workshop on Rendering*, pages 73–84, June 1993.

[29] Thomas W. Sederberg and Scott R. Parry. Comparison of three curve intersection algorithms. *Computer-aided Design*, 18(1), January/February 1986.

[30] Peter Shirley, Helen Hu, Brian Smits, and Eric Lafortune. A practitioners' assessment of light reflection models. In *Pacific Graphics*, pages 40–49, October 1997.

[31] Peter Shirley, Kelvin Sung, and William Brown. A ray tracing framework for global illumination systems. In *Proceedings of Graphics Interface '91*, pages 117–128, June 1991.

[32] J.S. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7), July 1994.

[33] Wolfgang Stürzlinger. Ray-tracing triangular trimmed free-form surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 4(3), July-September 1998.

[34] H.J. Yoon, S. Eun, and J.W. Cho. Image parallel ray tracing using static load balancing and data prefetching. *Parallel Computing*, 23(7), July 1997.

[35] Ellen Scher Zagier. Frameless antialiasing. Technical Report TR95-026, UNC-CS, May 1995.

[36] Ellen Scher Zagier. Defining and refining frameless rendering. Technical Report TR97-008, UNC-CS, July 1997.

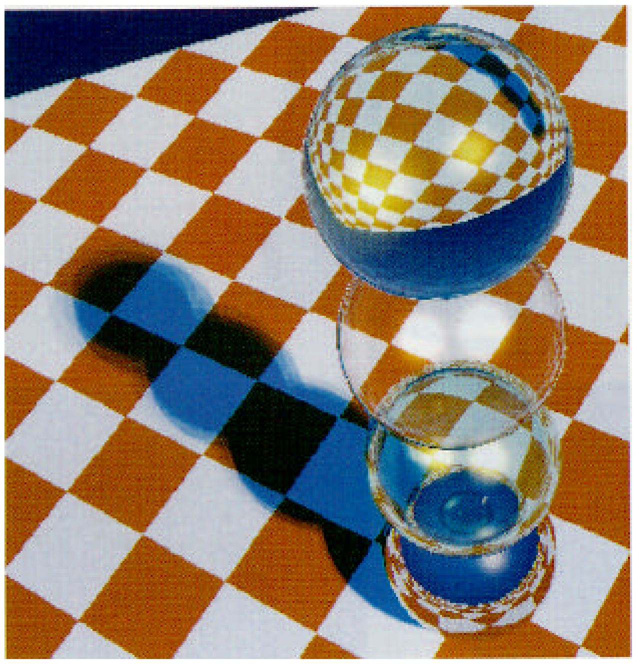


Figure A: *A portion of a 600 by 400 pixel image from our system running at approximately fifteen frames per second.*

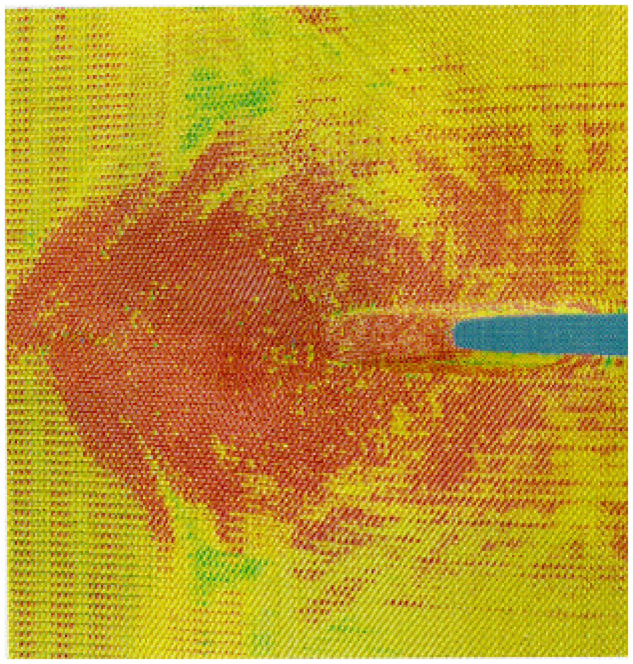


Figure C: *Simulation of crack propagation visualized using 35M spheres. This image at 512 by 512 pixels runs approximately 15 frames per second on 60 CPUs.*

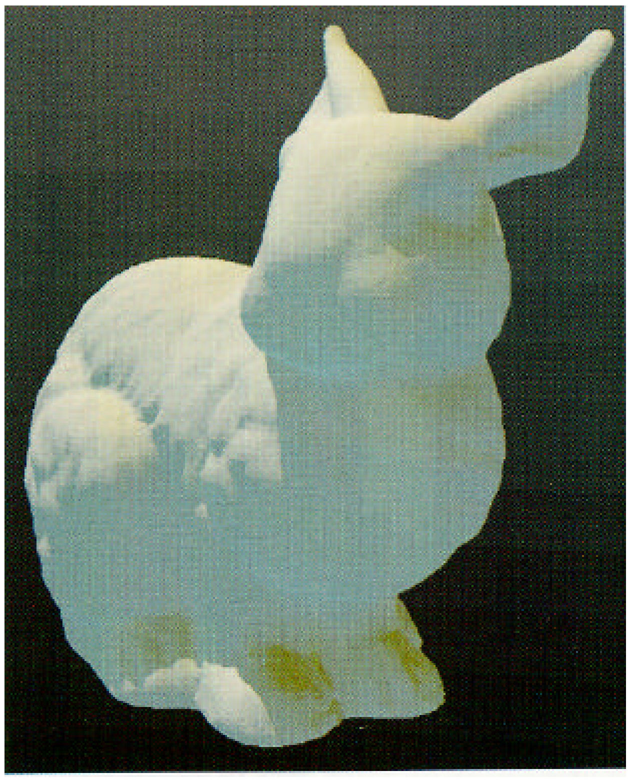


Figure B: *Directionally varying ambient approximation.*

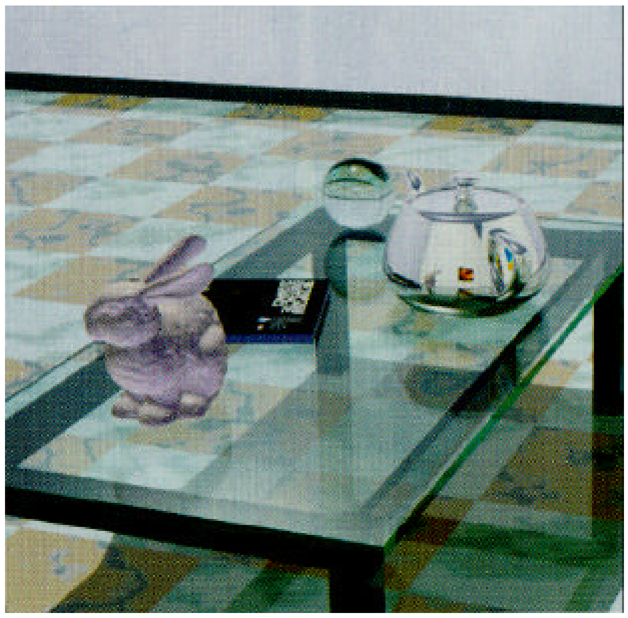


Figure D: *A model with splines, glass, image textures, and procedural solid textures. At 512 by 512 pixels this image is generated at approximately 4 frames per second on 60 CPUs.*

Siggraph 2005 Course on Interactive Ray Tracing
Handling Dynamic Scenes

Ingo Wald

This is an excerpt from
"Realtime Ray Tracing and Interactive Global Illumination",
PhD Thesis, Ingo Wald,
Computer Graphics Group, Saarland University.
Full Version available at `http://www.mpi-sb.mpg.de/∼wald/PhD`

"The time spent constructing the hierarchy tree should more than pay for itself in time saved rendering the image"

Timothy L. Kay, James T. Kajiya "Ray Tracing Complex Scenes" [Kay86] (in 1986!)

Even though ray tracing is a relatively old and well-understood technique, its use for interactive applications is still in its infancy. Several issues of interactive applications are all but fully solved. Especially the handling of dynamic scenes in an interactive context so far has received few attention by ray tracing researchers. Ray tracing research so far almost exclusively concentrated on accelerating the process of creating a *single* image, which could take from minutes to hours. Most of these approaches relied on doing extensive preprocessing by building up complex data structures to accelerate the process of tracing a ray.

Before realtime ray tracing, the time used for building an index structure such as kd-trees was insignificant compared to the long rendering times, as this preprocessing was then amortized over the remainder of a frame. Thus preprocessing times of several seconds to a few minutes could easily be tolerated in order to build a high-quality acceleration structure for an offline renderer. As long as the scene remains static, the same trick also worked for "interactive" ray tracing systems as described before – the acceleration structure was built once in an offline preprocessing step, and was then reused for all the remaining frames[1].

In dynamic scenes, however, this trick no longer works, as each new frame requires a new acceleration structure. Building this data structure for every

[1]Even though the scene itself has to remain static in this approach, it is still possible to arbitrarily change camera, material properties, shaders, and light source configurations in a scene.

frame then becomes a bottleneck, as this "preprocessing" alone would often exceed the total time available per frame in an interactive setting.

Even worse, this preprocessing phase cannot easily be parallelized: Though tracing the rays can be parallelized trivially once each client has access to scene and acceleration structure, the operations for building the acceleration structure have to be performed on each client, thereby incurring a non-parallelizable cost factor. As a result, any time spent on dynamic updates becomes extremely costly especially for parallel (distributed) interactive ray tracing systems[2]. This poses a major problem for ray tracing dynamic scenes, as virtually all of todays interactive ray tracing (e.g. [Wald01, Parker99, DeMarle03, Wald03]) systems have to rely on massive parallelization to achieve interactive frame rates.

Therefore, it is not surprising that all of those systems have in common that they mainly concentrate on the actual ray tracing phase and do not target dynamic scenes. Without methods for interactively modifying the scene, however, interactive ray tracing will forever be limited to simple walk-throughs of static environments, and can therefore hardly be termed truly interactive, as long as real *interaction* between the user and the environment is not impossible. In order to be truly interactive, ray tracing *must* be able to efficiently support dynamic environments. As such, efficient handling of dynamic scenes is probably one of the biggest challenges for realtime ray tracing.

1 Previous Work

Some methods have been proposed for the case where predefined animation paths are known in advance (e.g. [Glassner88, Gröller91]). These however are not applicable to our target setting of totally dynamic, unpredictable changes to the scene in which the motion of objects is not known in advance. Little research is available for such truly interactive settings. This research will be reviewed below.

First of all, excellent work on ray tracing in dynamic environments has recently been performed by Lext et al. with the BART project [Lext00], in which they provide an excellent analysis and classification of the problems arising in dynamic scenes. Based on this analysis, they proposed a representative set of test scenes (see Figure 1) that have been designed to stress the different aspects of ray tracing dynamic scenes. Thus, the BART benchmark provides an excellent tool for evaluating and analyzing a dynamic ray tracing engine. For future research on dynamic ray tracing, the BART benchmark suite might well play the same role that Eric Haines' "SPD Database" [Haines87] played for offline rendering.

In their analysis, Lext et al. have classified the behavior of dynamic scenes into two inherently different classes: Hierarchical motion, and unstructured mo-

[2] As an example, consider a system that spends only 5% of its time on dynamic scene updates. Parallelizing this system on 19 CPUs ($\frac{1}{5\%} - 1$) results in each node spending half its time on scene updates, and in a speedup of only 10 for 19 CPUs (i.e. a utlization of only $\frac{10}{19} \sim 50\%$)!

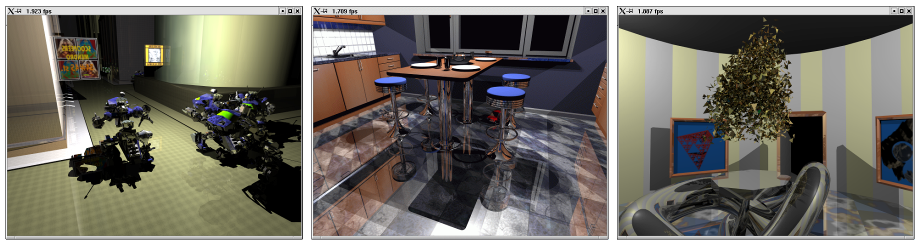


Figure 1: Some example screen-shots from the BART benchmark: (a) “robots”, where 10 robots (each consisting of 16 independently moving body parts) are walking through a city model; (b) “kitchen”, in which a small toy car is driving through a highly specular kitchen scene; and (c) “museum”, where a certain amount of reflective triangles is animated incoherently to form several different shapes. The number of triangles in the museum scene can be configured from a few hundred to some hundred thousand triangles.

tion. In *hierarchical motion*, the animation is described by having the primitives in a scene organized into several groups that are transformed hierarchically. While different groups may move independently of all other groups, all primitives in the same group are always subject to the same, usually affine, transformation[3]. The other class is *unstructured motion*, where each triangle moves without relation to all others. For example, the robots scene in Figure 1a is a good example of hierarchical motion, as there is no dynamic behavior except for hierarchical translation and rotation of the different robots’ body parts. In contrast to this, the museum scene (Figure 1c) features many incoherently moving triangles, and as such is a good example for unstructured motion. For a closer explanation of the different kinds of motion, see the BART paper [Lext00].

Though the BART paper provides an excellent analysis of dynamic ray tracing, it did not attempt to propose any practical algorithms or solutions to the problems. So far, few people have worked on this topic. In a first step, Parker et al. [Parker99] kept moving primitives out of the acceleration structure and checked them individually for every ray. This of course is only feasible for a small number of moving primitives.

Another approach would be to efficiently update the acceleration structure whenever objects move. Because objects can occupy a large number of cells in an acceleration structure this may require costly updates to large parts of the structure for each moving primitive (especially for large primitives, which tend to overlap many cells). To overcome this problem, Reinhard et al. [Reinhard00] proposed a dynamic acceleration structure based on a hierarchical grid. In order to quickly insert and delete objects independently of their size, larger objects are kept in coarser levels of the hierarchy. With this approach, objects always cover approximately a constant number of cells, thus updating the acceleration struc-

[3]Affine transformation are not limited to translation and rotation only, but also include e.g. shearing or scaling.

ture can be performed in constant time. However, their method resulted in a rather high overhead, and also required their data structure to be rebuilt once in a while to avoid degeneration. Furthermore, their method mainly concentrated on unstructured motion, and is not well suited for hierarchical animation.

Recently, Lext et al. [Lext01] proposed a way for quickly reconstructing an acceleration structure in a hierarchically animated scene by transforming the rays to the local object coordinate systems instead of transforming the objects and rebuilding their acceleration structures. Though their basic idea is similar to the way that our method handles hierarchical animation, to our knowledge their method so far has never been applied in an interactive context.

2 A Hierarchical Approach to Handling Dynamic Scenes

Essentially, our approach to handling dynamic scenes is motivated by the same observations as Lext et al. [Lext01] of how dynamic scenes typically behave: Usually large parts of a scene remain static over long periods of time. Other parts of the same scene undergo well-structured transformations such as rigid body motion or affine transformations. Yet other parts are changed in a totally unstructured way.

All these kinds of motion are fundamentally different. Even worse, many scenes actually contain a mix of all these different kinds of motion. It is unlikely that a single method can handle all these kinds of motion equally well. Because of this, we prefer an approach in which the different kinds of motion are handled with different, specialized algorithms that are then combined into a common architecture. To do this, geometric primitives are organized in separate *objects* according to their dynamic properties. Each of the three kinds of objects – static, hierarchically animated, and those with unstructured motion – is thus treated differently: Static objects will be handled as before, hierarchically animated objects are handled by transforming rays rather than the object, and objects with unstructured motion are handled by specially optimized algorithms for quickly rebuilding the affected parts of the data structure. Each object has its own acceleration structure and can be updated independently of the rest of the scene. These independent objects are then combined in a hierarchical way by organizing them in an additional top-level acceleration structure that accelerates ray traversal between the objects in a scene (see Figure 2).

2.1 Building the Hierarchy

To enable this scheme, all triangles that are subject to the same set of transformations (e.g. all the triangles forming the head of the animated robot in Figure 3) must be grouped by the application into the same object.

Note that we explicitly do *not* attempt to perform this grouping automatically. Instead, this grouping has to be performed by the application that drives the ray tracer. Though this somewhat shifts the problem to the application, the

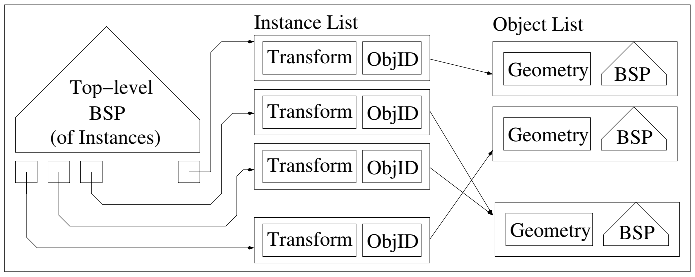


Figure 2: Two-level hierarchy as used in our approach: A top-level BSP contains references to instances, which contain a transformation and a reference to an object. Objects in turn consist of geometry and a local BSP tree. Multiple instances can refer to the same object with different transformations.

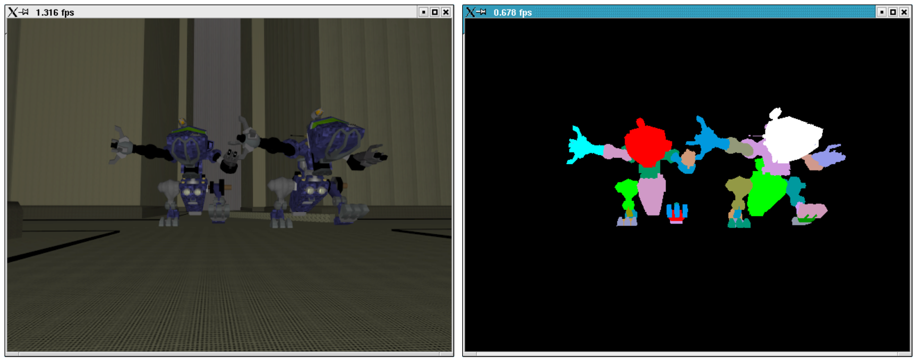


Figure 3: Grouping of triangles into objects for hierarchical animation. Triangles subject to the same hierarchical transformations are grouped into the same object. (a) Snapshot from the BART robots scene, (b) Same snapshot, with color-coded objects. Triangles with the same color belong to the same object.

application itself has the information about the motion of every part of the scene in its internal scene-graph, and can typically perform this classification without major effort. In fact, most applications already do this for OpenGL rendering, as the same grouping of objects is required for efficiently using OpenGL display lists (also see the discussion of the accompanying document on OpenRT). However, the actual grouping of objects into objects has a higher influence on rendering performance than for OpenGL display lists. As such, it is important to perform this grouping with extreme care in order to achieve good performance.

In the following, we will shortly describe how these different kinds of objects are treated, and how the top-level index structure is built and traversed.

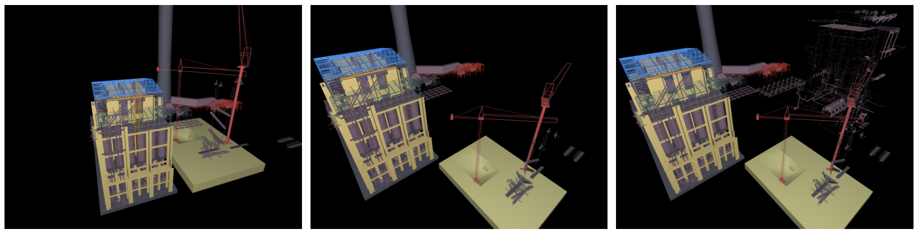


Figure 4: Snapshots of an interactive session in which complex parts of the 12.5 million triangle "UNC power plant" model are being moved interactively with our method: a) the original powerplant, b) moving the powerplant and the construction side apart, and c) moving part of the internal structure (the cool and warm water pipes, totalling a few million triangles!) out of the main structure).

3 Static and Hierarchical Motion

For *static objects*, ray traversal works as before by just traversing the ray with our usual, fast traversal algorithm.

For *hierarchically transformed objects*, we do not actually transform the geometry of the object itself, but rather store this transformation with the object, and inversely transform the rays that require intersection with this object[4].

For both static objects and for those with hierarchical motion, the local BSP tree must only be built once directly after object definition. Thus, the time for building these objects is not an issue, thereby allowing for the use of sophisticated and slow algorithms for building high-quality acceleration structures for these objects.

Obviously the transformation slightly increases the per-ray cost. However, this transformation has to be performed only once for each dynamic object visited by a ray, and is as such tolerable. This increased per-ray cost then totally removes the reconstruction cost for hierarchically animated objects, as all that is required for transforming the object is to update its transformation matrix. This is especially important as this kind of motion is usually the most common form in practical scenes. Furthermore, not having to rebuild any BSP trees make the update-cost for hierarchically transformed objects independent of object size. As such, this way of handling hierarchical animation can be used very efficiently even in extremely complex scenes. For example, Figure 4 shows how a complex part of the 12.5 million triangle "UNC power plant" is being moved in an interactive session.

3.1 Instantiation

Being able to handle objects that are subject to a transformation, the presented approach as a side effect also allows for "instantiation", i.e. using multiple in-

[4]Note that this way of handling is similar to the approach of Lext et al. [Lext01].

stances of the same object: Parts of a scene (e.g. one of the sunflowers in Figure 5) can re-used several times in the same scene by creating several *instances* of this object. An instance then consists only of two properties: a reference to the original model, and a transformation matrix that the instanced object is subject to. Thus, even highly complex scenes can be stored with a small memory footprint, which in turn allows for efficiently rendering even massively complex scenes at interactive rates. As an example, Figure 5 shows a slight modification of Oliver Deussen's "Sunflowers" scene, which consists of several large trees with millions of triangles each, plus 28,000 instances of 10 different sunflower models with roughly 36,000 triangles each. While only one kind of tree and 10 kinds of sunflowers have to actually be stored, in total roughly one billion triangles are potentially visible. By changing the transformation matrices of the instances, each object can be manipulated interactively while the scene renders at about 7 fps on 24 dual processor PCs at video resolution (see Table 4). Note that this performance can be achieved even tough a large number of rays needs to be cast in this scene: The leaves of both sunflowers and trees are modeled with transparency textures, which results in many rays for computing transparency and semi-transparent shadows.

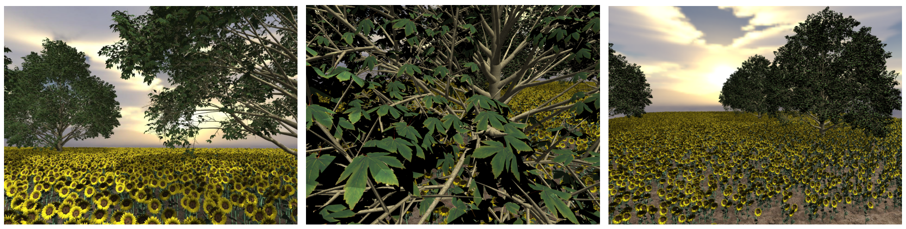


Figure 5: Instantiation: The "Sunflowers" scene consists of roughly 28,000 instances of 10 different kinds of sunflowers with 36,000 triangles each together with several multi-million-triangle trees. The whole scene consists of roughly one billion triangles. The center image shows a closeup of the highly detailed shadows cast by the sun onto the leaves. All leaves contain textures with transparency which increase the number of rays needed for rendering a frame. The whole scene renders at roughly 7 fps on 24 dual PCs at video resolution. All objects including the sun can be manipulated interactively.

4 Fast Handling of Unstructured Motion

While this simple trick of transforming rays instead of triangles elegantly avoids any reconstruction cost for hierarchical motion, it does not work for *unstructured motion*, as there the acceleration structure potentially has to be rebuilt for every frame. Even so, if triangles under unstructured motion are kept in a separate object, the BSP reconstruction cost can be localized to only those triangles that have actually been transformed. The local acceleration structures of such objects are discarded and rebuilt from the transformed triangles whenever necessary. Even though this process is costly, it is only required for objects with unstructured motion and does not affect any of the other objects. Obviously, only those objects have to be rebuilt whose primitives have actually be modified in the respective frame. Furthermore, it is possible to perform this reconstruction of dynamic objects lazily *on demand*, i.e. only once a ray actually demands intersection with that updated object. As such, the occlusion-culling feature of ray tracing also applies to the reconstruction of dynamic objects, as occluded objects do not have to be rebuilt.

The algorithms for creating highly optimized BSP trees may require several seconds even for moderately complex objects. Thus, they are not applicable to unstructured motion, for which the object BSP has to be rebuilt every frame (and thus in fractions of a second). For these cases we trade traversal performance for construction speed by using less expensive, simple heuristics for BSP plane placement, which allows for a high-performance implementation of the construction process.

4.1 Using less costly BSP Construction Parameters

Furthermore, we use less expensive quality parameters for the BSP plane placement heuristics. For example, a particularly important cost factor for BSP tree construction is the subdivision criterion of the BSP. As described in the accompanying document on the RTRT core, this criterion typically consist of a maximum tree depth and a target number of triangles per leaf cell. Subdivision continues on cells with more than the target number of triangles up to the maximum depth. Typical criteria specify 2 or 3 triangles per cell and usually result in fast traversal times – but also in deeper BSPs, which are more costly to create. Particularly costly are degenerate cases, in which subdivision can not reduce the number of triangles per cell, for example if too many primitives occupy the same point in space, e.g. at vertices with a valence higher than the maximum numbers of triangles.

In order to avoid such excessive subdivisions in degenerate regions, we modified the subdivision criterion (for unstructured object BSPs): The deeper the subdivision, the more triangles will be tolerated per cell. We currently increase the tolerance threshold by a constant factor for each level of subdivision. Thus, we generally obtain significantly lower BSP trees and larger leaf cells than for static objects. Though this of course slows down the traversal of rays hitting such objects, this slowdown is usually more than made up by the significantly shorter construction time. Furthermore often only few rays hit such objects with unstructured motion and are affected by this slowdown, so using a slower BSP tree for those rays is tolerable. With the described compromises on BSP construction, unstructured motion for moderate-sized objects can be supported by rebuilding the respective object BSP every frame.

5 Fast Top-Level BSP Construction

As mentioned before, all kinds of objects – static, hierarchically animated, and those with with unstructured motion – are hierarchically combined in an additional top-level acceleration structure. For this top-level structure, we also use a kd-tree, and as such can use exactly the same algorithms for traversing this top-level tree than for the object BSPs, except that visiting a voxel now requires to intersect objects rather than triangles. While traversing this top-level BSP thus requires only minor changes to the original implementation, this is not the case for the construction algorithm. A scene can easily contain hundreds or thousands of instances (see Figures 6 and 5), and a straight-forward approach would be too costly for interactive use. On the other hand, the top-level BSP is traversed by *every* ray, and thus few compromises on BSP quality can be made for the top-level BSP.

Fortunately, the task of building the top-level BSP is simpler than for object BSPs: Object BSPs require costly triangle-in-cell computations, careful placement of the splitting plane, and handling of degenerate cases. The top-level BSP however only contains instances represented by an axis-aligned bounding box (AABB) of its transformed object[5].

Considering only the AABBs, optimized placement of the splitting plane becomes much easier, and any degenerate cases can be avoided.

For splitting a cell, we follow several observations:

1. It is usually beneficial to subdivide a cell in the dimension of its maximum extent, as this usually yields the most well-formed cells [Havran01].

2. Placement of the BSP plane only makes sense at the boundary of objects contained within the current cell. This is due to the fact that the cost-function can be maximal only at such boundaries [Havran01].

3. It can been shown that the optimal position for the splitting plane lies between the cells geometric center and the object median [Havran01]

Following these observations, the BSP tree can be built such that it is both suited for fast traversal by optimized plane placement, and can still be built quickly and efficiently: For each subdivision step, we try to find a splitting plane in the dimension of maximum extent (observation 1). As potential splitting planes, only the AABB borders will be considered (observation 2). To find a good splitting plane, we first split the cell in the middle, and decide which side contains more objects, i.e. which one contains the object median. From this side, we choose the object boundary closest to the center of the cell. Thus, the splitting plane lies in-between cell center and object median, which is generally a good choice (observation 3).

[5]While the object itself already has an axis-aligned bounding box, this AABB is not necessarily axis-aligned any more when subject to a transformation. As such, we conservatively build the AABB of the instance by building a new instance AABB out of the transformed vertices of the objects AABB. While this somewhat overestimates the actual instance bounds, it is much less costly than computing the correct AABB by transforming all vertices.

As each subdivision step removes at least one potential splitting plane, termination of the subdivision can be guaranteed without further termination criteria. Degenerate cases for overlapping objects cannot happen, as only AABB boundaries are considered, and not the overlapping space itself. Choosing the splitting plane in the described way also yields relatively small and well-balanced BSP trees. Thus, we get a top-level BSP that can be traversed reasonable quickly, while still offering a fast and efficient construction algorithm.

```
BuildTree(instances,voxel)
for d = x,y,z in order of maximum extent
    P = $\{i.min_d, i.max_d | i \in instances\}$
    if ($\|P\|$ = 0) continue;
    c = center of voxel
    if (more instances on left side of c than on right)
        p = max($\{p \in P | p < c\}$)
    else
        p = min($\{p \in P | p >= c\}$)
    Split Cell (instances,cell) in d at p into
            (leftvox,leftinst),(rightvox,rightinst)
    l = BuildTree(leftinst,leftvox);
    r = BuildTree(rightinst,rightvox);
    return InnerNode(d,p,l,r);
end for
# no valid splitting plane found
return Leaf(instances)
```

5.1 High-Quality Top-level BSPs

Instead of this simplified BSP construction algorithm, it is also possible to use a surface area heuristic for the top-level BSP tree. The main problem with this approach is the question how to best estimate the cost for intersecting the object. As the respective object BSPs contain only triangles, the cost for each half voxel created by a split can be safely estimated to be mostly linear in the number of triangles on each side. For the top-level BSP however, each side contains objects of different size, for which the cost is hard to estimate.

However, the bigger problem with a surface area heuristic for the top-level BSP tree is its relatively high construction cost. While this may be neglectable for a few dozen objects, it currently becomes too expensive for a few hundred instances. As such, using an SAH would only make sense for a small number of instances as long as no fast implementations of an SAH tree builder are available. Though RTRT/OpenRT has an implementation of both SAH and the above mentioned algorithm, by default we use the simple and fast-to-build version as described above.

6 Fast Traversal

Once both the top-level BSP tree and all the object BSPs are built, each ray first starts traversing this top level structure. As soon as a voxel is found, the ray is intersected with the objects in the leaf by simply traversing the respective objects local acceleration structures. Once all BSPs are built, within both top-level BSP and within each object traversal is identical to traditional ray tracing in a static environment. Consequently, we use exactly the same algorithms and data structures for building and traversing that acceleration structure as for the static case [Wald04]. For the top-level BSP, the only difference is that each leaf cell of the top-level BSP tree contains a list of instance IDs instead of triangle IDs. Only minor changes have been required to implement this modified traversal code.

As with the original implementation, a ray is first clipped to the scene bounding box and is then traversed iteratively through the top-level BSP tree. As soon as it encounters a leaf cell, it sequentially intersects the ray with all instances in this cell: For each instance, the ray is first transformed to the local coordinate system of the object, then clipped to the correct AABB of the object, and finally traversed through its acceleration structure.

6.1 Mailboxing

Typically, the bounding volumes of different instances will overlap. In order to avoid having to intersect a ray with the same object multiple times, mailboxing [Amanatides87, Kirk91, Havran01] is very important to use for the top-level BSP tree. While the benefit of mailboxing for the triangle for an object BSP is rather small, the high cost of intersecting the same object several times clearly justifies the use of mailboxing for the top-level BSP.

6.2 SSE Traversal

As our traversal and intersection algorithms do not require normalized ray directions, transforming a ray is relatively cheap, as no costly re-normalization of the transformed rays is necessary. The ray-matrix multiplications themselves can very efficiently be done using SSE [Intel02]. Of course, our method also works with the fast SSE packet traversal code described in [Wald04].

7 Experiments and Results

The described framework is rather straightforward to implement and use as long as a shared-memory system (e.g. a dual-CPU PC) is available. However, the situation of dynamic ray tracing gets much more problematic for non-shared memory systems, as such systems often contain many non-scalable cost factors, such as communicating scene updates to the client, or having to rebuild parts of the hierarchy on every client.

As the described framework has been especially designed for performing well on loosely coupled (i.e. non-shared memory) clusters of workstations, it is of major importance to investigate the scalability of our method. To allow for representative results, we have chosen to use a wide range of experiments and test scenes. Therefore, we have chosen to use the BART benchmark scenes [Lext00], which represent a wide variety of stress factors for ray tracing of dynamic scenes. Additionally, we use several of the scenes that we encountered in practical applications [Wald02b], and a few custom-made scenes for stress-testing. Snapshots of these test scenes can be found in Figure 6.

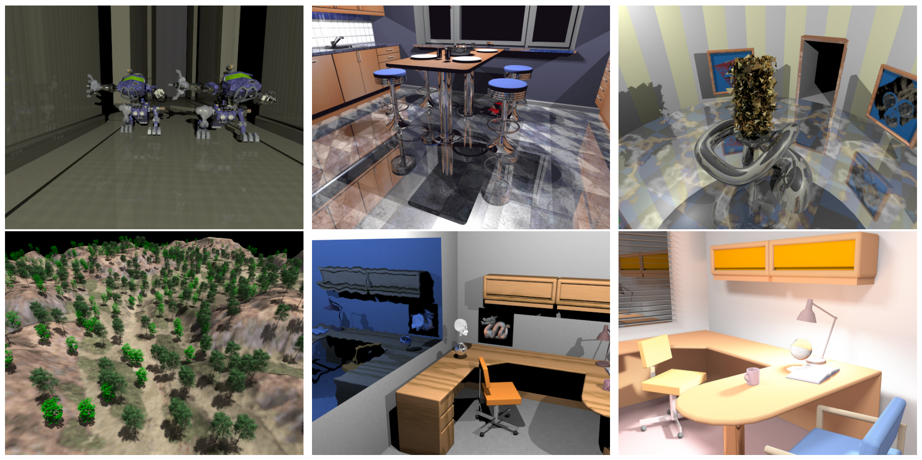


Figure 6: Several example frames from some of our dynamic scenes. a.) "BART robots" contains roughly 100,000 triangles in 161 moving objects, b.) "BART kitchen", c.) "BART museum" with unstructured motion of several thousand triangles. Note how the entire museum reflects in these triangles. d.) The "terrain" scene uses up to 661 instances of 2 trees, would contain several million triangles without instantiation, and also calculates detailed shadows. e.) The "office" scene in a typical ray tracing configuration, demonstrating that the method works fully automatically and completely transparently to the shader. f.) Office with interactive global illumination.

All of the following experiments have been performed on a cluster of dual AMD AthlonMP 1800+ machines with a FastEthernet (100Mbit) network connection. The network is fully switched with a single GigaBit uplink to a dual AthlonMP 1700+ server. The application is running on the server and is totally unaware of the distributed rendering happening inside the rendering engine. It manages the geometry in a scene graph, and transparently controls rendering via calls to the OpenRT API [Wald04]. While the application itself may internally use a scene-graph with multiple nested hierarchy levels, the OpenRT library internally "flattens" this multi-level scene graph to the two-level organization as described above (see Figure 2).

In the following experiments, all examples are rendered at video resolution

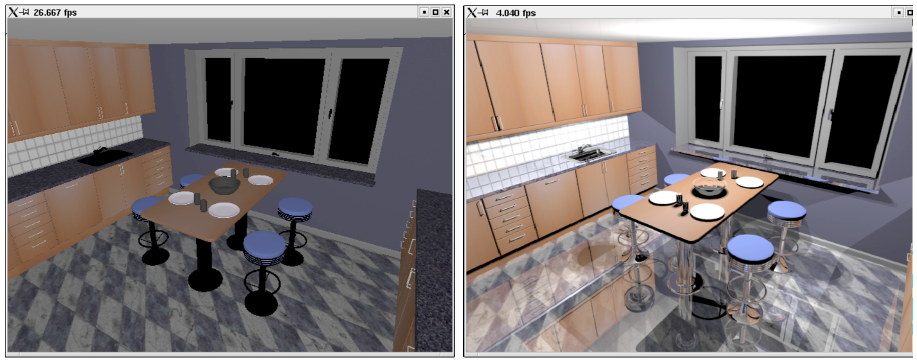


Figure 7: Two snapshots from the BART kitchen. a.) OpenGL-like ray casting at > 26 fps on 32 CPUs. b.) full-featured ray tracing with shadows and 3 levels of reflections, at > 7 fps on 32 CPUs.

of 640×480 pixels. Ray tracing is performed with costly programmable shaders featuring shadows, reflections and texturing.

7.1 BART Kitchen

The kitchen scene contains hierarchical animation of 110.000 triangles organized in 5 objects. It requires negligible network bandwidth and BSP construction overhead. Overlap of bounding boxes may results in a certain overhead, which is hard to measure exactly but is definitely not a major cost factor[6].

The main cost of this scene is due to the need for tracing many rays to evaluate shadows from 6 point lights. There is also a high degree of reflectivity on many objects. Due to fast camera motion and highly curved objects (see Figure 7), these rays are rather incoherent. However, these aspects are completely independent of the dynamic nature of the scene and are handled efficiently by our system.

We achieve interactive frame rates even for the large amount of rays to be shot. A reflection depth of 3 results in a total of 3.867.661 rays/frame. At a measured rate of 912.000 rays traced per second and CPU in this scene, this translates to a frame rate of 7.55 fps on 32 CPUs. As can be seen in Table 1, scalability is almost linear – using twice as many CPUs results in roughly twice the frame rate.

CPUs	2	4	8	16	32
OpenGL-like	3.2	6.4	12.8	25.6	> 26
Ray Tracing	0.47	0.94	1.88	3.77	7.55

Table 1: Scalability in the kitchen scene in frames/sec.

[6]Note that the robot, museum, kitchen, and terrain scenes are only available in a dynamic version, and can thus not be compared to a static version.

7.2 BART Robots

The robots scene features a game-like setting with 16 animated robots moving through a city. The scene consists of 161 objects: 16 robots with 10 animated body parts each, plus one object for the surrounding city. All dynamic motion is hierarchical with no unstructured motion at all. Therefore, the BSP trees for all objects have to be built only once, and only the top-level BSP have to be rebuilt for every frame.

Using the algorithms described above, rebuilding the top-level BSP is very efficient and takes less than one millisecond. Furthermore, updating the transformation matrices requires only a small network bandwidth of roughly 20 kb/frame for each client.

CPUs	2	4	8	16	32
OpenGL-like	2.8	5.55	10.8	21	> 26
Ray Tracing	0.54	1.07	2.15	4.3	8.6

Table 2: Scalability in the robots scene in frames/sec.

With such a small transmission and reconstruction overhead, we again achieve almost-linear scalability (see Table 2) and high rendering performance. Using 32 CPUs, we achieve a frame rate of 8 frames per second. Again, the high cost of this scene is due to the large number of reflection and shadow rays. Using a simple OpenGL-like shader (see Figure 8) results in frame rates of more than 26 frames per second.

7.3 BART Museum

The museum has been designed mainly for testing unstructured motion and is the only BART scene featuring non-hierarchical motion. In the center of the museum, several triangles are animated on predefined animation paths to form differently shaped objects. The number of triangles undergoing unstructured motion can be configured to 64, 256, 1k, 4k, 16k, or 64k. Even though the complete animation paths are specified in the BART scene graph, we do not make use of this information. User controlled movement of the triangles – i.e. without knowledge of future positions – would create the same results.

This scene also requires the computation of shadows from two point lights as well as large amounts of reflection rays. All of the moving triangles are reflective and incoherently sample the whole environment (see Figure 9). As the dynamic behavior of a scene is completely transparent to the shaders, integrating all these effects does not require any additional effort except for the cost for tracing the rays.

As expected, unstructured motion gets costly for many triangles. Building the BSP tree for the complex version of 64k triangles already requires more than one second (see Table 3). Note, however, that our current algorithms for building object BSPs still leave plenty of room for further optimizations.

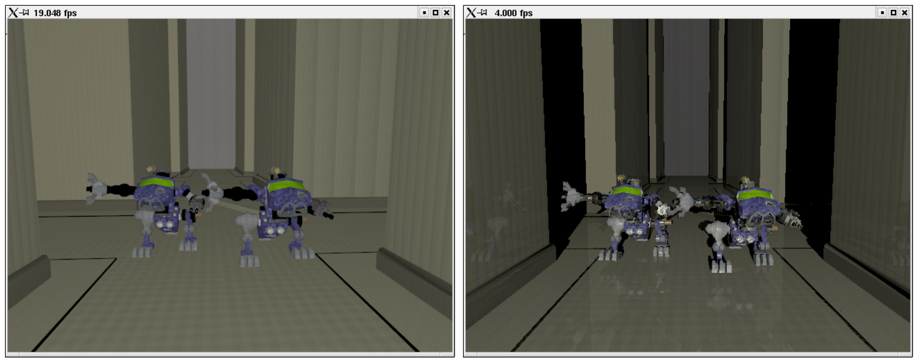


Figure 8: BART robots: 16 robots consisting of 161 objects rendered interactively. a.) Ray casting at > 26 fps on 32 CPUs. b.) Full ray tracing at > 8 fps at 32 CPUs.

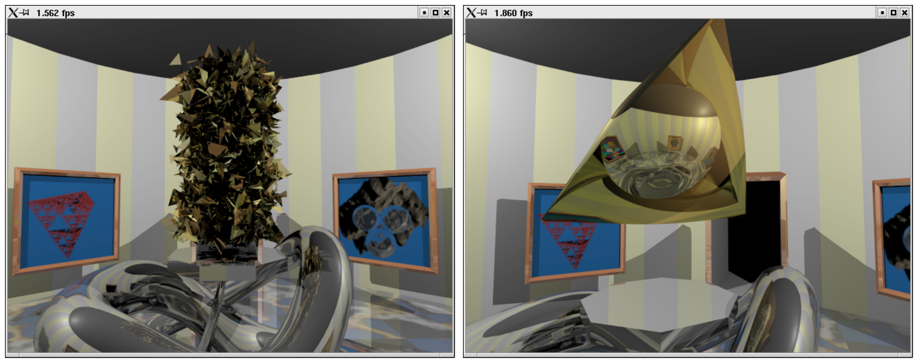


Figure 9: Unstructured motion in the BART museum: Up to 64,000 triangles are moving incoherently through the museum. Note how the triangles reflect the entire environment.

Furthermore, the reconstruction time is strongly affected by the distribution of triangles in space: In the beginning of the animation, all triangles are equally and almost-randomly distributed. This is the worst case for BSPs, which are best at handling uneven distributions, and construction is consequently costly. Furthermore, the randomly distributed triangles form many singularities when intersecting themselves, which is extremely bad for typical BSP trees. During the animation, the triangles organize themselves to form a single surface. At this stage, reconstruction time is much faster. Note that the numbers given in Table 3 are taken at the beginning of the animation, and are thus worst-case results.

num tris	64	256	1k	4k	16k	64k
reconst.time	1ms	2ms	8ms	34ms	0.1s	> 1s
bandwidth/client	6.4k	25.6k	102k	409k	1.6M	6.5M

Table 3: Unstructured motion in different configurations of the museum scene. Rows specify reconstruction time for the top-level BSP, and data sent to each client for updating the triangle positions.

7.3.1 Network Bottleneck

Apart from raw reconstruction cost, significant network bandwidth is required for sending all triangles to every client. Since we use reliable unicast (TCP/IP) for network transport, using 4096 triangles and 16 clients (32 CPUs), requires to transfer roughly 6.5 Mb (16 clients × 408*kb*, see Table 3) – for *every frame*. Though in theory this does not yet totally saturate the network, the network load is not equally distributed over time: Network bandwidth is especially high at the beginning of each frame, when all the scene updates have to be communicated to the clients. During this time, the network is already completely saturated when sending 16k triangles to the clients, implying that the performance of the server is already significantly affected during this time. Consequently, we can no longer scale linearly any more when dynamically updating more than a few thousand triangles (see Table 4).

7.3.2 "Geometry Shaders"

Note that this problem would be significantly reduced on a shared-memory platform, or even with the availability of a reliable hardware multicast. On a cluster configuration, the update problem could probably also be solved by using "geometry shaders" that can generate the triangles directly on the clients. Though this can only be used for a limited set of applications, it would already allow for many important and interesting applications. So far however this approach has not yet been sufficiently investigated.

Due to the discussed problems – high communication cost for the scene updates and high reconstruction cost for the dynamic object's BSP – the museum scene is the most problematic of all the scenes encountered so far. Even so, with all these effects – unstructured motion, shadows, and highly incoherent reflections in the animated triangles – the museum can still be rendered interactively: Using 8 clients, we achieve 4.8 fps for 1024 triangles, and still 4.2 fps for 4096 triangles in video resolution (see Table 4). Again, the frame rate is dominated by the cost for shadows and reflections. Using an OpenGL-like shader without these effects allows to render the scene at 19 frames per second on 8 clients.

7.4 Outdoor Terrain

The terrain scene has been specially designed to stress scalability with a large number of instances and triangles. It contains 661 instances of 2 different trees,

	with GL-like shading					with full ray tracing				
num CPUs	1	2	4	8	16	1	2	4	8	16
robots	2.8	5.55	10.8	21	26⋆	0.54	1.07	2.15	4.3	8.6
kitchen	3.2	6.4	12.8	25.6	26⋆	0.47	0.94	1.88	3.77	7.55
terrain	1.3	2.5	4.8	8	15	0.9	1.77	3.39	6.5	12
museum:										
w/ 1k	2.7	5.4	10.2	19.5	26⋆	0.6	1.2	2.4	4.8	9.3
w/ 4k	2.5	4.5	7.5	4.5	2.5	0.55	1.1	2.2	4.2	2.5
w/ 16k	1.6	2.4	1.7	1	0.5	0.45	0.9	1.65	0.98	0.53

Table 4: Scalability of our method in the different test scenes (BART robots, BART kitchen, Outdoor Terrain, and BART museum with 1k, 4k, and 16k dynamic triangles). "⋆" means that the servers network connection is completely saturated in our network configuration, and thus no higher performance can be achieved. The numbers in the left half of the table correspond to pure OpenGL like shading, the right half is for full ray tracing including shadows and reflections. As can clearly be seen, for scenes with hierarchical animation scalability is almost linear up to the maximum network bandwidth for the final pixel data. With an increasing amount of unstructured motion (museum 1k–16k), the required network bandwidth for sending the changed triangles to the clients soon becomes a bottleneck. In that case, adding more CPUs even reduces performance, as data has to be sent to even more clients. An overview of these scenes can be found in Figure 6

which correspond to more than 10 million triangles after instantiation. A point light source creates highly detailed shadows from the leaves (see Figure 6). All trees can be moved around interactively, both in groups or individually. The large number of instances results in construction times for the top-level BSP of up to 4 ms per frame. This cost — together with the transmission cost for updating all 661 instance matrices on all clients — limits the scalability for a large number of instances and clients (see Table 4).

8 Discussion

The above scenes have been chosen to stress different aspects of our dynamic ray tracing engine. Together with the terrain experiment, our test scenes contain a strong variation of parameters – from 5 to 661 instances, from a few thousand to several million triangles, from simple shading to lots of shadows and reflections, and from hierarchical animation to unstructured motion of thousands of triangles (for an overview, see Figure 6). Taken together, these experiments allow for a detailed analysis of our method.

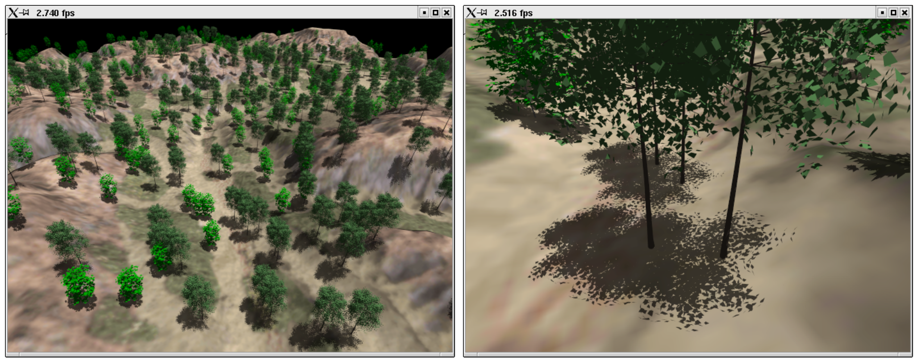


Figure 10: Terrain scene with up to 1000 instances of 2 kinds of complex trees (661 instances in the shown configuration, as some trees have been interactively moved off the terrain). Without instantiation, the scene would consist of roughly 10 million triangles. a.) Overview of the whole scene, b.) Detailed shadows from the sun, cast by the leaves onto both floor and other leaves.

8.1 Transformation Cost

For mainly hierarchical animation, the core idea of our method was to trade the cost for rebuilding the acceleration structure for the cost to transform the rays to the local coordinate system of each object. This implies that every ray intersecting an object has to be transformed via matrix-vector multiplications for both ray origin and direction (for every object encountered), potentially resulting in several matrix operations per ray. With a ray tracing performance of up to several million rays per second [Wald04], this can amount to many million matrix-vector multiplications per frame! For example, the *terrain* and *robots* scenes at 640×480 pixels require 1.6 and 1 million matrix operations, respectively (see Figure 5). Furthermore, more transformations are often required during shading, e.g. by transforming the shading normal or for calculating procedural noise in the local coordinate system.

	office	terrain	robots
objects	9	661	161
matrix ops	480k	1.6M	1M

Table 5: Number of matrix-vector multiplies for our benchmark scenes (resolution 640x480). A matrix operation can be performed in only 23 cycles even in plain C code, which is negligible compared to traversal cost.

However, the cost for these transformations in practice is quite tolerable: Even for a straight-forward C-code implementation, a matrix-vector operation costs only 23 cycles on an AMD AthlonMP CPU, which is rather small compared to the cost for tracing a ray (which is in the order of several hundred to a few thousand cycles). Note that the matrix-vector multiplies are ideally suited for fast SSE implementation. This reduces this cost even further, and makes the transformation overhead almost negligible.

8.2 Unstructured Motion

As could be expected, the museum scene has shown that unstructured motion remains costly for ray tracing. A moderate number of a few thousand independently moving triangles can easily be supported, but larger numbers still lead to high reconstruction times for the respective objects (see Table 3). As such, our method is still not suitable for scenes with strong unstructured motion.

To support such scenes, algorithms for faster reconstruction of dynamic objects have to be developed. Note that our method could also be combined with Reinhard's approach [Reinhard00] by using his method only for the unstructured objects. Even then, lots of unstructured motion would still create a performance problem due to the need to send all triangle updates to the clients. This is not a limitation of our specific method, but would be similar for any algorithm in a distributed environment.

8.3 Bounding Volume Overlap

One of the stress cases identified in [Lext00] was *Bounding Volume Overlap*. In fact, this does create some overhead, as in the overlap area of two objects, these two objects have to be intersected *sequentially* by each ray. As a result, a successful intersection found during traversal of the first object may later-on be invalidated by a closer one in the other object. In fact, this partially disables "early ray termination" and thus negatively affects the occlusion culling feature of ray tracing[7].

Though it is easy to construct scenarios where bounding volume overlap would lead to excessive overhead, it is rarely significant in practice. In fact, bounding volume overlap *does* happen in *all* our test cases, but has never shown to pose a major performance problem. In fact, overlapping objects are exactly what happens all the time in bounding volume hierarchies (BVHs) [Rubin80, Kay86, Haines91], which have also proven to work rather well in practice.

8.4 Teapot-in-a-Stadium Problem

The teapot-in-a-stadium problem is handled very well by out method: BSPs automatically adapt to varying object density in a scene [Havran01]. This is true for both object and top-level BSPs. In fact, our method can even increase performance for such cases: If the 'teapot' is contained in a separate object, the shape of the 'stadium' BSP is usually much better, as there is no need any more for several BSP subdivisions to tightly enclose the teapot.

[7]Note that for shadow rays, finding intersections in the wrong order is not a problem, as *any* valid intersection is sufficient to determine occlusion of a ray. Even so, Bounding Volume Overlap may still lead to the situation that the object containing an occluder will be traversed rather late during traversal of the toplevel structure.

8.5 Over-Estimation of Object Bounds

Building the top-level BSP requires an estimate of the bounding box of each instance in world coordinates. As transforming each individual vertex would be too costly, we conservatively estimate these bounds based on the transformed bounding box of the original object.

This somewhat over-estimates the correct bounds and thus results in some overhead: During top-level BSP traversal, a ray may be intersected with an object that it would not have intersected otherwise. However, this overhead is restricted to only transforming and clipping the ray: After transformation to the local coordinate system, such a ray is first clipped against the correct bounding box, and can thus be immediately discarded without further traversal.

8.6 Scalability with the Number of Instances

Apart from unstructured motion, the main cost of our method results from the need to recompute the top-level BSP tree. As such, a large number of instances is expensive, as can be seen in the terrain scene. Thus, the number of instances should be minimized in order to achieve optimal performance. Usually, it is better to use a small number of large objects instead of many small ones. For example, all static triangles in a scene should best be stored in a single object, instead of using multiple objects. This is completely different to approaches commonly used in OpenGL, in which many, small display lists are used. Thus, some amount of manual adjustment and optimization may be required when porting applications from OpenGL to OpenRT.

Even so, even the thousand complex instances can be rendered interactively, and top-level reconstruction has not yet proven a real limitation in any practical application. For moderate numbers of objects, top-level reconstruction is virtually negligible.

On the other hand, supporting instantiation (i.e. using exactly the same object multiple times in the same frame) is a valuable feature of our method, as this allows for rendering complex environments very efficiently: With instantiation, memory is required for storing only one copy of each object to be instantiated, plus the top-level BSP, allowing to render even many million triangles with a small memory footprint (see Figures 5 and 10). For triangle rasterization, all triangles would still need to be handled individually by the graphics hardware even when using display lists.

8.7 Scalability in a Distributed Environment

As can be seen by the experiments in Section 7, we achieve rather good scalability even for many clients except for scenes that require to update a lot of information on all clients, i.e. for a high degree of unstructured motion (where every moving triangle has to be transmitted), and for a large number of instances.

In the terrain scene, using 16 clients would require to send 676 Kb[8] per frame simply for updating the 661 transformation matrices on the clients. Though this data can be sent in a compressed form, load balancing and client/server communication further adds to the network bandwidth. Without broadcast/multicast functionality on the network, the server bandwidth increases linearly with the number of clients. For many clients and lots of updated data, this creates a bandwidth bottleneck on the server, and severely limits the scalability (see Table 4). In fact, performance could even *drop* when adding many more CPUs, as each new client increases the network load. In principle, the same is true for unstructured motion, where sending several thousand triangles to each client also creates a bandwidth bottleneck. On the other hand, both problems are not specific to our method, but will apply similarly to any method running on such a distributed hardware architecture.

8.8 Total Overhead

In order to estimate the total overhead of our method, we have compared several scenes in both a static and dynamic configuration. As there are no static equivalents for the BART benchmarks, we have taken several of our static test scenes, and have modified them in a way that they can be rendered in both a static configuration with all triangles in a single, static BSP tree, as well as in a dynamic configuration, where triangles are grouped into different objects that can then be moved dynamically.

For simple scenes, the total overhead is relatively high[9], compared to the small cost of rendering a static version of these scenes, and even reaches up to a factor of two in total rendering time. For more realistic scene sizes, however, the *relative* overhead is significantly less, and in practice is often in the range of 20 to 30 percent, sometimes even less. This is very fortunate, as the overhead is worst for simple scenes in which *absolute* performance is highest anyway, and is relatively small for more costly scenes in which a high overhead would hurt most. Furthermore, most practical applications use rather complex scenes, and thus have a small overhead. As such, we believe this overhead to be a reasonable price for the added flexibility gained through supporting dynamic scenes.

9 Conclusions

The presented method is a simple and practical approach to handling dynamic scenes in an interactive distributed ray tracing engine. It can handle a large variety of dynamic scenes, including all the BART benchmark scenes (see Figure 6). It imposes no limitations on the kind of rays to be shot, and as such

[8] 661 instances$\times$16 clients$\times(4 \times 4)$ floats

[9] Note that *total* overhead includes *all* the previously mentioned sources of overhead, e.g. including toplevel reconstruction, bounding volume overlap, traversal cost, multiple traversal setup cost, etc.

allows for all the usual ray-tracing features like shadows, reflections, and even global illumination [Wald04].

For unstructured motion, the proposed method still incurs a high reconstruction cost per frame, that makes it infeasible for a large number of incoherently moving triangles. For a moderate amount of unstructured motion however (in the order of a few thousand incoherently moving triangles), it is well applicable and results in frame rates of several frames per second at video resolution. For mostly hierarchical animation our method is highly efficient and achieves interactive performance even for highly complex models with hundreds of instances, and with millions of triangles per object [Wald02a]. This is especially furtunate since many of todays scene graph libraries (especially in VR/AR and other industrial applications) mostly use only this kind of animation.

The proposed technique forms a core part of the RTRT/OpenRT core, and has been used exclusively in all of the published applications of this system that all use this technique). Though it is certainly possible to construct cases in which the proposed method breaks down, so far is has been successfully able to handle all the dynamic scenes that have been encountered in practical applications of RTRT/OpenRT.

In conclusion, the proposed method of handling dynamic scenes is still limited but nonetheless already good enough for a large class of applications. Furthermore, the support for dynamic scenes in ray tracing is likely to improve significantly in the near future as more researchers start looking at this problem. Still, there remains a vast potential for future research in this area.

10 Future Work

At the moment, the main remaining scalability bottleneck lies in communicating all scene updates to all clients, making the total bandwidth linear in the number of clients. Thus, future work will investigate to use network broadcast/multicast to communicate the scene updates. As almost all of the updated data is the same for every client, this should effectively remove the server bottleneck. Furthermore, the above-mentioned concept of "geometry shaders" seems to be an interesting option for reducing the scene update bandwidth.

On the clients, the main bottleneck is the cost for reconstructing objects under unstructured motion. This could be improved by designing specialized algorithms for cases where motion is spatially limited in some form, such as for skinning, predefined animations, or for key-frame interpolation.

For the top-level BSP, it could be highly beneficial to investigate the use of cost functions. This especially includes finding ways of building such high-quality BSPs in a fast and efficient manner.

Apart from these technical issues, it is also important to investigate how existing applications can be mapped to our method, e.g. by evaluating how a scene graph library such as OpenInventor [Wernecke94], OpenSG [OpenSG01], OpenSceneGraph [OSG] or VRML [Carey97] can be efficiently implemented on top of our system. Preliminary results seem promising [Dietrich04].

Finally, it is an obvious next step to integrate this techniques into a hardware ray tracing architecture such as the SaarCOR architecture [Schmittler02]. As such an architecture avoids most of the distribution problems we have in our PC cluster, such a mapping should be highly successful. First results are encouraging.

References

[Amanatides87] *John Amanatides and Andrew Woo.* A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics '87*, pages 3–10. 1987.

[Carey97] *Rikk Carey, Gavin Bell, and Chris Marrin.* ISO/IEC 14772-1:1997 Virtual Reality Modelling Language (VRML97), April 1997. http://www.vrml.org/Specifications/VRML97.

[DeMarle03] *David E. DeMarle, Steve Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen.* Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, pages 87–94, 2003.

[Dietrich04] *Andreas Dietrich, Ingo Wald, Markus Wagner, and Philipp Slusallek.* VRML Scene Graphs on an Interactive Ray Tracing Engine. In *Proceedings of IEEE VR 2004*, pages 109–116, March 2004.

[Glassner88] *Andrew Glassner.* Spacetime Ray Tracing for Animation. *IEEE Computer Graphics and Applications*, 8(2):60–70, 1988.

[Gröller91] *Eduard Gröller and Werner Purgathofer.* Using temporal and spatial coherence for accelerating the calculation of animation sequences. In *Proceedings of Eurographics '91*, pages 103–113. Elsevier Science Publishers, 1991.

[Haines87] *Eric A. Haines.* A Proposal for Standard Graphics Environments. *IEEE Computer Graphics and Applications*, 7(11):3–5, November 1987. Available from http://www.acm.org/pubs/-tog/resources/SPD/overview.html.

[Haines91] *Eric Haines.* Efficiency Improvements for Hierarchy Traversal in Ray Tracing. In James Arvo, editor, *Graphics Gems II*, pages 267–272. Academic Press, 1991.

[Havran01] *Vlastimil Havran.* Heuristic Ray Shooting Algorithms. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[Intel02] *Intel Corp.* Intel Pentium III Streaming SIMD Extensions. http://developer.intel.com/vtune/cbts/simd.htm, 2002.

[Kay86] *Timothy L. Kay and James T. Kajiya.* Ray Tracing Complex Scenes. *Computer Graphics (Proceedings of SIGGRAPH 86)*, 20(4):269–278, June 1986. Held in Dallas, Texas.

[Kirk91] *David Kirk and James Arvo.* Improved Ray Tagging For Voxel-Based Ray Tracing. In James Arvo, editor, *Graphics Gems II*, pages 264–266. Academic Press, 1991.

[Lext00] *Jonas Lext, Ulf Assarsson, and Tomas Möller.* BART: A Benchmark for Animated Ray Tracing. Technical report, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, May 2000. Available at http://www.ce.chalmers.se/BART/.

[Lext01] *Jonas Lext and Tomas Akenine-Möller.* Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 – Short Presentations*, pages 311–318, 2001.

[OpenSG01] *OpenSG-Forum.* http://www.opensg.org, 2001.

[OSG] OpenSceneGraph. http://www.openscenegraph.org.

[Parker99] *Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan.* Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics*, pages 119–126, 1999.

[Reinhard00] *Erik Reinhard, Brian Smits, and Chuck Hansen.* Dynamic Acceleration Structures for Interactive Ray Tracing. In *Proceedings of the Eurographics Workshop on Rendering*, pages 299–306, Brno, Czech Republic, June 2000.

[Rubin80] *Steve M. Rubin and Turner Whitted.* A Three-Dimensional Representation for Fast Rendering of Complex Scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[Schmittler02] *Jörg Schmittler, Ingo Wald, and Philipp Slusallek.* SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 27–36, 2002.

[Wald01] *Ingo Wald, Philipp Slusallek, and Carsten Benthin.* Interactive Distributed Ray Tracing of Highly Complex Models. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques*, Proceedings of the 12th Eurographics Workshop on Rendering Techniques, London, UK, June 25-27, 2001, pages 274–285. Springer, 2001.

[Wald02a] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Saarland University, 2002. Available at http://graphics.cs.uni-sb.de/Publications.

[Wald02b] *Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek.* Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques*, pages 15–24, 2002. (Proceedings of the 13th Eurographics Workshop on Rendering).

[Wald03] *Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek.* Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.

[Wald04] *Ingo Wald.* Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[Wernecke94] *Josie Wernecke. The Inventor Mentor.* Addison-Wesley, 1994. ISBN 0-20162-495-8.

Interactive Ray Tracing for Volume Visualization

Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, Peter Shirley

Abstract— **We present a brute-force ray tracing system for interactive volume visualization. The system runs on a conventional (distributed) shared-memory multiprocessor machine. For each pixel we trace a ray through a volume to compute the color for that pixel. Although this method has high intrinsic computational cost, its simplicity and scalability make it ideal for large datasets on current high-end parallel systems. To gain efficiency several optimizations are used including a volume bricking scheme and a shallow data hierarchy. These optimizations are used in three separate visualization algorithms: isosurfacing of rectilinear data, isosurfacing of unstructured data, and maximum-intensity projection on rectilinear data. The system runs interactively (i.e., several frames per second) on an SGI Reality Monster. The graphics capabilities of the Reality Monster are used only for display of the final color image.**

Keywords— **Ray tracing, visualization, isosurface, maximum-intensity projection.**

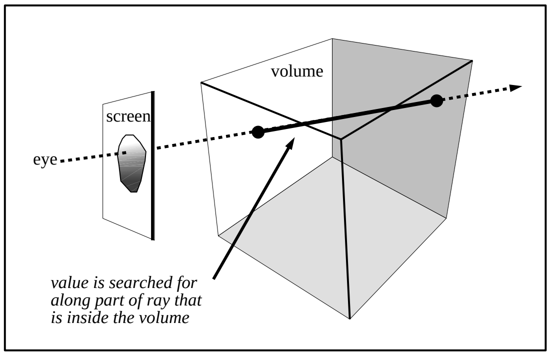

volume
screen
eye
value is searched for along part of ray that is inside the volume

Fig. 1. A ray traverses a volume looking for a specific or maximum value. No explicit surface or volume is computed.

I. Introduction

Many applications generate scalar fields $\rho(x, y, z)$ which can be visualized by a variety of methods. These fields are often defined by a set of point samples and an interpolation rule. The point samples are typically in either a rectilinear grid, a curvilinear grid, or an unstructured grid (simplical complex). The two main visualization techniques used on such fields are to display *isosurfaces* where $\rho(x, y, z) = \rho_{\text{iso}}$, and *direct volume rendering*, where there is some type of opacity/emission integration along the line of sight. The key difference between these techniques is that isosurfacing displays actual surfaces, while direct volume rendering displays some function of all the values seen along a ray throughout the pixel. Ideally, the display parameters for each technique are interactively controlled by the user. In this paper we present interactive volume visualization schemes that use ray tracing as their basic computation method.

The basic ray-volume traversal method used in this paper is shown in Fig. 1. This framework allows us to implement volume visualization methods that find exactly one value along a ray. Two such methods described in this paper are isosurfacing and maximum-intensity projection. Maximum-intensity projection is a direct volume rendering technique where the opacity is a function of the maximum intensity seen along a ray. The isosurfacing of rectilinear grids has appeared previously [1], while the isosurfacing of unstructured grids and the maximum-intensity projection are described for the first time in this paper. More general forms of direct volume rendering are not discussed in this paper.

The methods are implemented in a parallel ray tracing system that runs on an SGI Reality Monster, which is a conventional (distributed) shared-memory multiprocessor machine. The only graphics hardware that is used is the high-speed framebuffer. This overall system is described in a previous paper [2]. Conventional wisdom holds that ray tracing is too slow to be competitive with hardware z-buffers. However, when rendering a sufficiently large dataset, ray tracing should be competitive because its low time complexity ultimately overcomes its large time constant [3]. This crossover will happen sooner on a multiple CPU computer because of ray tracing's high degree of intrinsic parallelism. The same arguments apply to the volume traversal problem.

In Section II we review previous work, describe several volume visualization techniques, and give an overview of the parallel ray tracing code that provides the backbone of our system. Section III describes the data organizational optimizations that allow us to achieve interactivity. In Section IV we describe our memory optimizations for various types of volume visualization. In Section V we show our methods applied to several datasets. We discuss the implications of our results in Section VI, and point to some future directions in Section VII. Some material that is not research-oriented but is helpful for implementors is presented in the appendices.

II. Background

Ray tracing has been used for volume visualization in many works (e.g., [4], [5], [6]). Typically, the ray tracing of a pixel is a kernel operation that could take place within any conventional ray tracing system. In this section we review how ray tracers are used in visualization, and how they are implemented efficiently at a systems level.

A. Efficient Ray Tracing

It is well understood that ray tracing is accelerated through two main techniques [7]: accelerating or eliminating ray/voxel intersection tests and parallelization. Acceleration is usually accomplished by a combination of spatial subdivision and early ray termination [4], [8], [9].

Ray tracing for volume visualization naturally lends itself towards parallel implementations [10], [11]. The computation for each pixel is independent of all other pixels, and the data structures used for casting rays are usually read-only. These properties have resulted in many parallel implementations. A variety of techniques have been used to make such systems parallel, and

Computer Science Department, University of Utah, Salt Lake City, UT 84112. E-mail: [sparker | map | ylivnat | ppsloan | hansen | shirley] @cs.utah.edu.

many successful systems have been built (e.g., [10], [12], [13], [14]). These techniques are surveyed by Whitman [15].

B. Methods of Volume Visualization

There are several ways that scalar volumes can be made into images. The most popular simple volume visualization techniques that are not based on cutting planes are *isosurfacing*, *maximum-intensity projection* and *direct volume rendering*.

In isosurfacing, a surface is displayed that is the locus of points where the scalar field equals a certain value. There are several methods for computing images of such surfaces including constructive approaches such as marching cubes [16], [17] and ray tracing [18], [19], [20].

In maximum-intensity projection (MIP) each value in the scalar field is associated with an intensity and the maximum intensity seen through a pixel is projected onto that pixel [21]. This is a "winner-takes-all" algorithm, and thus looks more like a search algorithm than a traditional volume color/opacity accumulation algorithm.

More traditional direct volume rendering algorithms accumulate color and opacity along a line of sight [8], [4], [5], [6], [22]. This requires more intrinsic computation than MIP, and we will not deal with it in this paper.

C. Traversals of Volume Data

Traversal algorithms for volume data are usually customized to the details of the volume data characteristics. The three most common types [23] of volume data used in applications are shown in Figure 2.

To traverse a line through rectilinear data some type of incremental traversal is used (e.g., [24], [25]). Because there are many cells, a hierarchy can be used that skips "uninteresting" parameter intervals, which increases performance [26], [27], [28], [29].

For curvilinear volumes, the ray can be intersected against a polygonal approximation to the boundary, and then a more complex cell-to-cell traversal can be used [30].

For unstructured volumes a similar technique can be used [31], [32]. Once the ray is intersected with a volume, it can be tracked from cell-to-cell using the connectivity information present in the mesh.

Another possibility for both curvilinear and unstructured grids is to resample to a rectilinear grid [33], although resampling artifacts and data explosion are both issues.

III. TRAVERSAL OPTIMIZATIONS

Our system organizes the data into a shallow rectilinear hierarchy for ray tracing. For unstructured or curvilinear grids, a rectilinear hierarchy is imposed over the data domain. Within a given level of the hierarchy we use the incremental method described by Amanatides and Woo [24].

A. Memory Bricking

The first optimization is to improve data locality by organizing the volume into "bricks" that are analogous to the use of image tiles in image-processing software and other volume rendering programs [21], [34] (Figure 3). Our use of lookup tables is particularly similar to that of Sakas et al. [21].

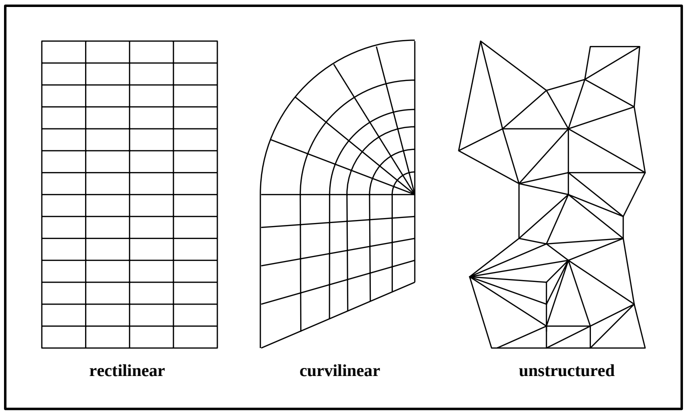


Fig. 2. The three most common types of point-sampled volume data.

Effectively utilizing the cache hierarchy is a crucial task in designing algorithms for modern architectures. Bricking or 3D tiling has been a popular method for increasing locality for ray cast volume rendering. The dataset is reordered into $n \times n \times n$ cells which then fill the entire volume. On a machine with 128 byte cache lines, and using 16 bit data values, n is exactly 4. However, using float (32 bit) datasets, n is closer to 3.

Effective translation lookaside buffer (TLB) utilization is also becoming a crucial factor in algorithm performance. The same technique can be used to improve TLB hit rates by creating $m \times m \times m$ bricks of $n \times n \times n$ cells. For example, a $40 \times 20 \times 19$ volume could be decomposed into $4 \times 2 \times 2$ macrobricks of $2 \times 2 \times 2$ bricks of $5 \times 5 \times 5$ cells. This corresponds to $m = 2$ and $n = 5$. Because 19 cannot be factored by $mn = 10$, one level of padding is needed. We use $m = 5$ for 16 bit datasets, and $m = 6$ for 32 bit datasets.

The resulting offset q into the data array can be computed for any x, y, z triple with the expression:

$$\begin{aligned} q \;=\; & ((x \div n) \div m) n^3 m^3 ((N_z \div n) \div m)((N_y \div n) \div m) + \\ & ((y \div n) \div m) n^3 m^3 ((N_z \div n) \div m) + \\ & ((z \div n) \div m) n^3 m^3 + \\ & ((x \div n) \bmod m) n^3 m^2 + \\ & ((y \div n) \bmod m) n^3 m + \\ & ((z \div n) \bmod m) n^3 + \\ & (x \bmod n \times n) n^2 + \\ & (y \bmod n) \times n + \\ & (z \bmod n) \end{aligned}$$

where N_x, N_y and N_z are the respective sizes of the dataset.

This expression contains many integer multiplication, divide and modulus operations. On modern processors, these operations are extremely costly (32+ cycles for the MIPS R10000). Where n and m are powers of two, these operations can be converted to bitshifts and bitwise logical operations. However, the ideal size is rarely a power of two thus, a method that addresses arbitrary sizes is needed. Some of the multiplications can be converted to shift/add operations, but the divide and modulus operations are more problematic. The indices could be computed incrementally, but this would require tracking 9 counters, with numerous comparisons and poor branch prediction performance.

Note that this expression can be written as:

$$q = F_x(x) + F_y(y) + F_z(z)$$

where

$$\begin{aligned}
F_x(x) &= ((x \div n) \div m)n^3m^3((N_z \div n) \div m)((N_y \div n) \div m) + \\
&\quad ((x \div n) \bmod m)n^3m^2 + \\
&\quad (x \bmod n \times n)n^2 \\
F_y(y) &= ((y \div n) \div m)n^3m^3((N_z \div n) \div m) + \\
&\quad ((y \div n) \bmod m)n^3m + \\
&\quad (y \bmod n) \times n \\
F_z(z) &= ((z \div n) \div m)n^3m^3 + \\
&\quad ((z \div n) \bmod m)n^3 + \\
&\quad (z \bmod n)
\end{aligned}$$

We tabulate F_x, F_y, and F_z and use x, y, and z respectively to find three offsets in the array. These three values are summed to compute the index into the data array. These tables will consist of N_x, N_y, and N_z elements respectively. The total sizes of the tables will fit in the primary data cache of the processor even for very large data set sizes. Using this technique, we note that one could produce mappings which are much more complex than the two level bricking described here, although it is not at all obvious which of these mappings would achieve the highest cache utilization.

For many algorithms, each iteration through the loop examines the eight corners of a cell. In order to find these eight values, we need to only lookup $F_x(x)$, $F_x(x+1)$, $F_y(y)$, $F_y(y+1)$, $F_z(z)$, and $F_z(z+1)$. This consists of six index table lookups for each eight data value lookups.

B. Multilevel Grid

The other basic optimization we use is a multi-level spatial hierarchy to accelerate the traversal of empty cells as is shown in Figure 4. Cells are grouped divided into equal portions, and then a "macrocell" is created which contains the minimum and maximum data value for its children cells. This is a common variant of standard ray-grid techniques [35] and is especially similar to previous multi-level grids [36], [37]. The use of minimum/maximum caching has been shown to be useful [28], [29], [38]. The ray-isosurface traversal algorithm examines the min and max at each macrocell before deciding whether to recursively examine a deeper level or to proceed to the next cell. The typical complexity of this search will be $O(\sqrt[3]{n})$ for a three level hierarchy [36]. While the worst case complexity is still $O(n)$, it is difficult to imagine an isosurface occurring in practice approaching this worst case. Using a deeper hierarchy can theoretically reduce the average case complexity slightly, but also dramatically increases the storage cost of intermediate levels. We have experimented with modifying the number of levels in the hierarchy and empirically determined that a tri-level hierarchy (one top-level cell, two intermediate macrocell levels, and the data cells) is highly efficient. This optimum may be data dependent and is modifiable at program startup. Using a tri-level hierarchy, the storage overhead is negligible ($< 0.5\%$ of the data size). The cell sizes used in the hierarchy are independent of the brick sizes used for cache locality in the first optimization.

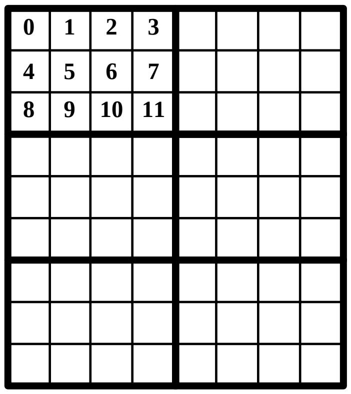


Fig. 3. Cells can be organized into "tiles" or "bricks" in memory to improve locality. The numbers in the first brick represent layout in memory. Neither the number of atomic voxels nor the number of bricks need be a power of two.

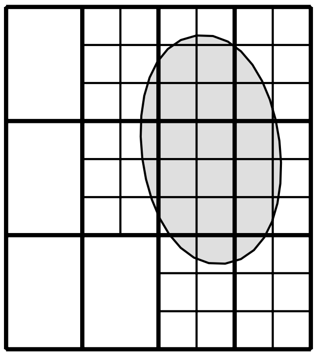


Fig. 4. With a two-level hierarchy, rays can skip empty space by traversing larger cells. A three-level hierarchy is used for most of the examples in this paper.

Macrocells can be indexed with the same approach as used for memory bricking of the data values. However, in this case there will be three table lookups for each macrocell. This, combined with the significantly smaller memory footprint of the macrocells made the effect of bricking the macrocells negligible.

IV. Algorithms

This section describes three types of volume visualization that use ray tracing:

- isosurfacing on rectilinear grids
- isosurfacing on unstructured meshes
- maximum-intensity projection on rectilinear grids

The first two require an operation of the form: find a specific scaler value along a ray. The third asks: what is the maximum value along a ray. All of these are searches that can benefit from the hierarchical data representations described in the previous section.

A. Rectilinear Isosurfacing

Our algorithm has three phases: traversing a ray through cells which do not contain an isosurface, analytically computing the isosurface when intersecting a voxel containing the isosurface, shading the resulting intersection point. This process is repeated for each pixel on the screen. A benefit is that adding incremental features to the rendering has only incremental cost. For example, if one is visualizing multiple isosurfaces with some of them

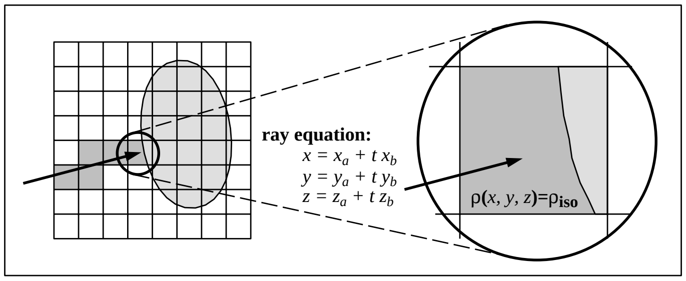


Fig. 5. The ray traverses each cell (left), and when a cell is encountered that has an isosurface in it (right), an analytic ray-isosurface intersection computation is performed.

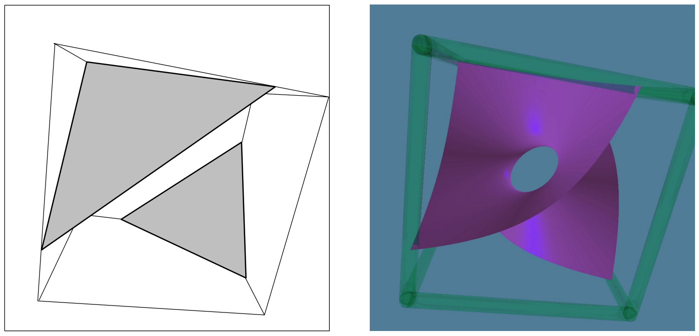


Fig. 6. Left: The isosurface from the marching cubes algorithm. Right: The isosurface resulting the true cubic behavior inside the cell.

rendered transparently, the correct compositing order is guaranteed since we traverse the volume in a front-to-back order along the rays. Additional shading techniques, such as shadows and specular reflection, can easily be incorporated for enhanced visual cues. Another benefit is the ability to exploit texture maps which are much larger than physical texture memory which is currently available up to 64 MBytes. However, newer architectures that use main memory for textures eliminate this issue.

For a regular volume, there is a one-to-one correspondence with the cells forming bricks and the voxels. This leads to a large branching factor for the shallow hierarchy which we have empirically found to yield the best results.

If we assume a regular volume with even grid point spacing arranged in a rectilinear array, then ray-isosurface intersection is straightforward. Analogous simple schemes exist for intersection of tetrahedral cells as described below.

To find an intersection (Figure 5), the ray $\vec{\mathbf{a}} + t\vec{\mathbf{b}}$ traverses cells in the volume checking each cell to see if its data range bounds an isovalue. If it does, an analytic computation is performed to solve for the ray parameter t at the intersection with the isosurface:

$$\rho(x_a + tx_b, y_a + ty_b, z_a + tz_b) - \rho_{\text{iso}} = 0.$$

When approximating ρ with a trilinear interpolation between discrete grid points, this equation will expand to a cubic polynomial in t. This cubic can then be solved in closed form to find the intersections of the ray with the isosurface in that cell. We use the closed form solution for convenience since its stability and efficiency have not proven to be major issues for the data we have used in our tests. Only the roots of the polynomial which are contained in the cell are examined. There may be multiple roots, corresponding to multiple intersection points. In this case, the smallest t (closest to the eye) is used. There may also be no roots of the polynomial, in which case the ray misses the isosurface in the cell. The details of this intersection computation are given in Appendix A. Note that using trilinear interpolation directly will produce more complex isosurfaces than is possible with a marching cubes algorithm. An example of this is shown in Figure 6 which illustrates case 4 from Lorensen and Cline's paper [17]. Techniques such as the Asymptotic Decider [39] could disambiguate such cases but they would still miss the correct topology due to the isosurface interpolation scheme.

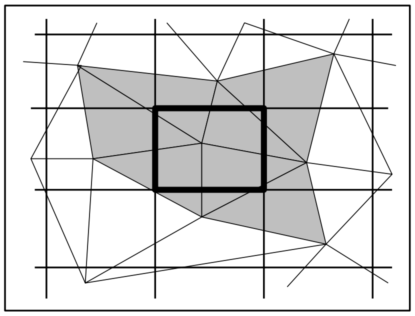


Fig. 7. For a given leaf cell in the rectilinear grid, indices to the shaded elements of the unstructured mesh are stored.

B. *Unstructured Isosurfacing*

For unstructured meshes, the same memory hierarchy is used as is used in the rectilinear case. However, we can control the resolution of the cell size at the finest level. We chose a resolution which uses approximately the same number of leaf nodes as there are tetrahedral elements. At the leaf nodes a list of references to overlapping tetrahedra is stored (Figure 7). For efficiency, we store these lists as integer indices into an array of all tetrahedra.

Rays traverse the cell hierarchy in a manner identical to the rectilinear case. However, when a cell is detected that might contain an isosurface for the current isovalue, each of the tetrahedra in that cell are tested for intersection. No connectivity information is used for the tetrahedra; instead they are treated as independent items, just as in a traditional surface-based ray tracer.

The isosurface for a tetrahedron is computed implicitly using barycentric coordinates. The intersection of the parameterized ray and the isoplane is computed directly, using the implicit equations for the plane and the parametric equation for the ray. The intersection point is checked to see if it is still within the bounds of the tetrahedron by making sure the barycentric coordinates are all positive. Details of this intersection code are described in Appendix B.

C. *Maximum-Intensity Projection*

The maximum-intensity projection (MIP) algorithm seeks the largest data value that intersects a particular ray. It utilizes the same shallow spatial hierarchy described above for isosurface extraction. In addition, a priority queue is used to track the cells or macrocells with the maximal values. For each ray, the priority queue is first initialized with single top level macrocell. The maximum data value for the dataset is used as the priority value for this entry in the priority queue. The algorithm repeatedly pulls the largest entry from the priority queue and breaks it into smaller (lower level) macrocells. Each of these cells are inserted into the priority queue with the precomputed maximum data value for that region of space. When the lowest-level cells are pulled from the priority queue, the algorithm traverses the segment of the ray which intersects the macrocell. Bilinear interpolation is used at the intersection of the ray with cell faces since these are the extremal values of the ray-cell intersection in a linear interpolation scheme. For each data cell face which intersects the ray, a bilinear interpolation of the data values is computed, and the maximum of these values in stored again in the priority queue. Finally, when one of these data maximums appears at the head of the priority queue, the algorithm has found the maximum data value for the entire ray.

To reduce the average length of the priority queue, the algorithm performs a single trilinear interpolation of the data at one point to establish a lower-bound for the maximum value of the ray. Macrocells and datacells which do not exceed this lower-bound are not entered into the priority queue. To obtain this value, we perform the trilinear interpolation using the t corresponding to the maximum value from whatever previous ray a particular processor has computed. Typically, this will be a value within the same block of pixels and exploits image-space coherence. If not, it still provides a bound on the maximum along the ray. If this t value is unavailable (due to program startup, or a ray missing the data volume), we choose the midpoint of the ray segment which intersects the data volume. This is a simple heuristic which improves the performance for many datasets.

Similar to the isosurface extraction algorithm, the MIP algorithm uses the 3D bricking memory layout for efficient cache utilization when traversing the data values. Since each processor will be using a different priority queue as it processes each ray, an efficient implementation of a priority queue which does not perform dynamic memory allocation is essential for performance of the algorithm.

V. Results

We applied ray tracing isosurface extraction to interactively visualize the Visible Woman dataset. The Visible Woman dataset is available through the National Library of Medicine as part of its Visible Human Project [40]. We used the computed tomography (CT) data which was acquired in 1mm slices with varying in-slice resolution. This rectilinear data is composed of 1734 slices of 512x512 images at 16 bits. The complete dataset is 910 MBytes. Rather than down-sample the data with a loss of resolution, we utilize the full resolution data in our experiments. As previously described, our algorithm has three phases: traversing a ray through cells which do not contain an isosurface, analytically computing the isosurface when intersecting a voxel containing the isosurface, and shading the resulting intersection point.

Figure 8 shows a ray tracing for two isosurface values. Figure 9 illustrates how shadows can improves the accuracy of our

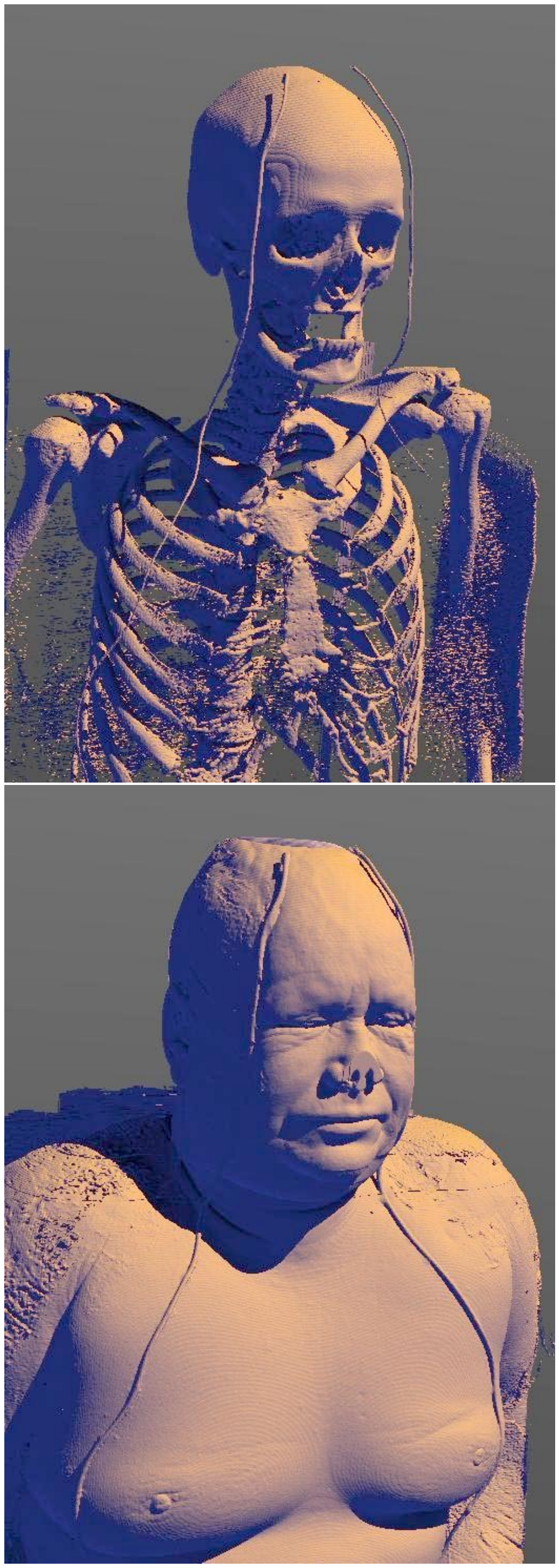


Fig. 8. Ray tracings of the bone and skin isosurfaces of the Visible Woman.

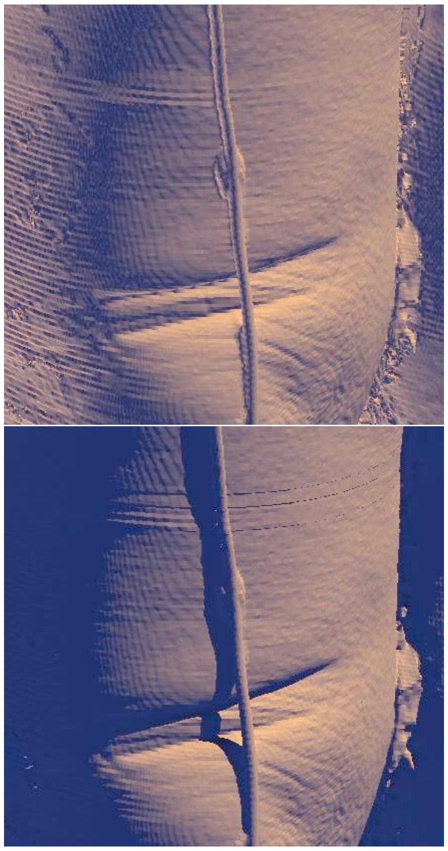


Fig. 9. A ray tracing with and without shadows.

TABLE I

DATA FROM RAY TRACING THE VISIBLE WOMAN. THE FRAMES-PER-SECOND (FPS) GIVES THE OBSERVED RANGE FOR THE INTERACTIVELY GENERATED VIEWPOINTS ON 64 CPUS.

Isosurface	Traversal	Intersec.	Shading	FPS
Skin ($\rho = 600.5$)	55%	22%	23%	7-15
Bone ($\rho = 1224.5$)	66%	21%	13%	6-15

TABLE II

SCALABILITY RESULTS FOR RAY TRACING THE BONE ISOSURFACE IN THE VISIBLE HUMAN. A 512x512 IMAGE WAS GENERATED USING A SINGLE VIEW OF THE BONE ISOSURFACE.

	View 1		View 2	
# cpus	FPS	speedup	FPS	speedup
1	0.18	1.0	0.39	1.0
2	0.36	2.0	0.79	2.0
4	0.72	4.0	1.58	4.1
8	1.44	8.0	3.16	8.1
12	2.17	12.1	4.73	12.1
16	2.89	16.1	6.31	16.2
24	4.33	24.1	9.47	24.3
32	5.55	30.8	11.34	29.1
48	8.50	47.2	16.96	43.5
64	10.40	57.8	22.14	56.8
96	16.10	89.4	33.34	85.5
128	20.49	113.8	39.98	102.5

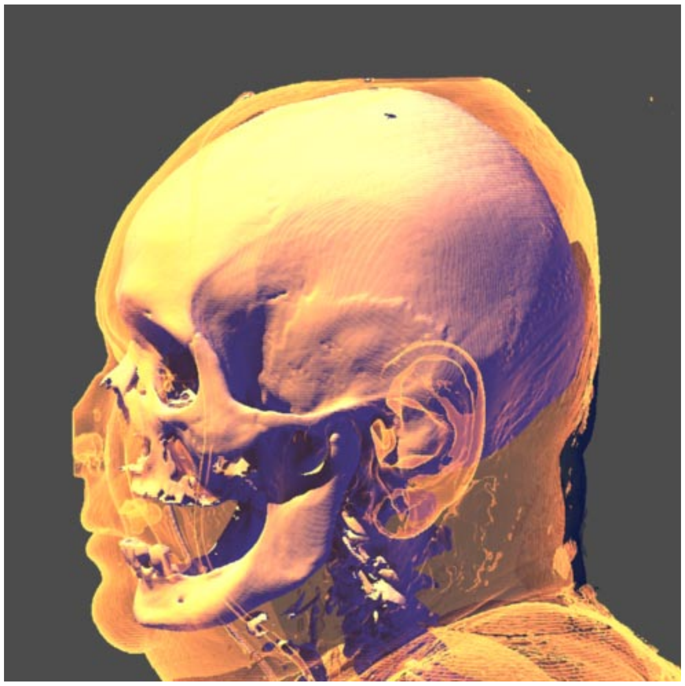


Fig. 10. Ray tracings of the skin and bone isosurfaces with transparency.

geometric perception. Figure 10 shows a transparent skin isosurface over a bone isosurface. Table I shows the percentages of time spent in each of these phases, as obtained through the cycle hardware counter in Silicon Graphics' Speedshop[1]. As can be seen, we achieve about 10 frames per second (FPS) interactive rates while rendering the full, nearly 1 GByte, dataset.

Table II shows the scalability of the algorithm from 1 to 128 processors. View 2 uses a zoomed out viewpoint with approximately 75% pixel coverage whereas view 1 has nearly 100% pixel coverage. We chose to examine both cases since view 2 achieves higher frame rates. The higher frame rates cause less parallel efficiency due to synchronization and load balancing. Of course, maximum interaction is obtained with 128 processors, but reasonable interaction can be achieved with fewer processors. If a smaller number of processors were available, one could reduce the image size in order to restore the interactive rates. Efficiencies are 91% and 80% for view 1 and 2 respectively on 128 processors. The reduced efficiency with larger numbers of processors (> 64) can be explained by load imbalances and the time required to synchronize processors at the required frame rate. The efficiencies would be higher for a larger image.

Table III shows the improvements which were obtained

[1]Speedshop is the vendor provided performance analysis environment for the SGI IRIX operating system.

TABLE III

TIMES IN SECONDS FOR OPTIMIZATIONS FOR RAY TRACING THE VISIBLE HUMAN. A 512X512 IMAGE WAS GENERATED ON 16 PROCESSORS USING A SINGLE VIEW OF AN ISOSURFACE.

View	Initial	Bricking	Hierarchy+Bricking
skin: front	1.41	1.27	0.53
bone: front	2.35	2.07	0.52
bone: close	3.61	3.52	0.76
bone: from feet	26.1	5.8	0.62

TABLE IV

FRAMERATES VARYING SHADOW AND TEXTURE FOR THE VISIBLE MALE DATASET ON 64 CPUS (FPS).

no shadows, no texture	15.9
shadows, no texture	8.7
no shadows, texture	12.6
shadows, texture	7.5

through the data bricking and spatial hierarchy optimizations.

Using a ray tracing architecture, it is simple to map each isosurface with an arbitrary texture map. The Visible Man dataset includes both CT data and photographic data. Using a texture mapping technique during the rendering phase allows us to add realism to the resultant isosurface. The photographic cross section data which was acquired in 0.33mm slices, and can be registered with the CT data. This combined data cab be used as a texture mapped model to add realism to the resulting isosurface. The size of the photographic dataset is approximately 13 GBytes which clearly is too large to fit into texture memory. When using texture mapping hardware it is up to the user to implement intelligent texture memory management. This makes achieving effective texture performance non-trivial. In our implementation, we down-sampled this texture by a factor of 0.6 in two of the dimensions so that it occupied only 5.1 GBytes. The framerates for this volume with and without shadows and texture are shown in Table IV. A sample image is shown in Figure 11. We can achieve interactive rates when applying the full resolution photographic cross sections to the full resolution CT data. We know of no other work which achieves these rates.

Figure 12 shows an isosurface from an unstructured mesh made up of 1.08 million elements which contains adaptively refined tetrahedral elements. The heart and lungs shown are polygonal meshes that serve as landmarks. The rendering times for this data, rendered without the polygonal landmarks at 512 by 512 pixel resolution, is shown in Table V. As would be expected, the FPS is lower than the structured data but the method scales well. We make the number of lowest-level cells proportional to the number of tetrahedral elements, and the bottleneck is the intersection with individual tetrahedral elements. This dataset composed of adaptively refined tetrahedral with volume differences of two orders of magnitude.

Figure 13 shows a maximum-intensity projection of the Visible Female dataset. This dataset runs in approximately 0.5 to 2 FPS on 16 processors. Using the “use last t” optimization saves

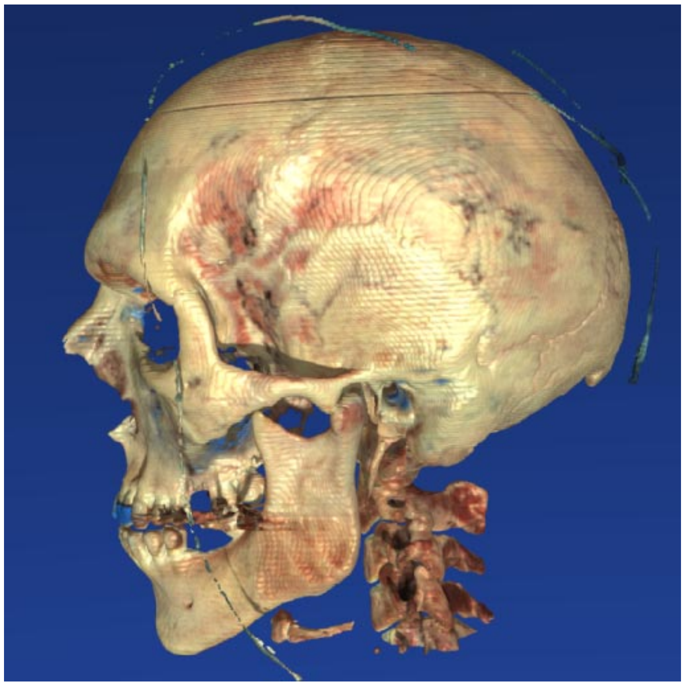


Fig. 11. A 3D texture applied to an isosurface from the Visible Man dataset.

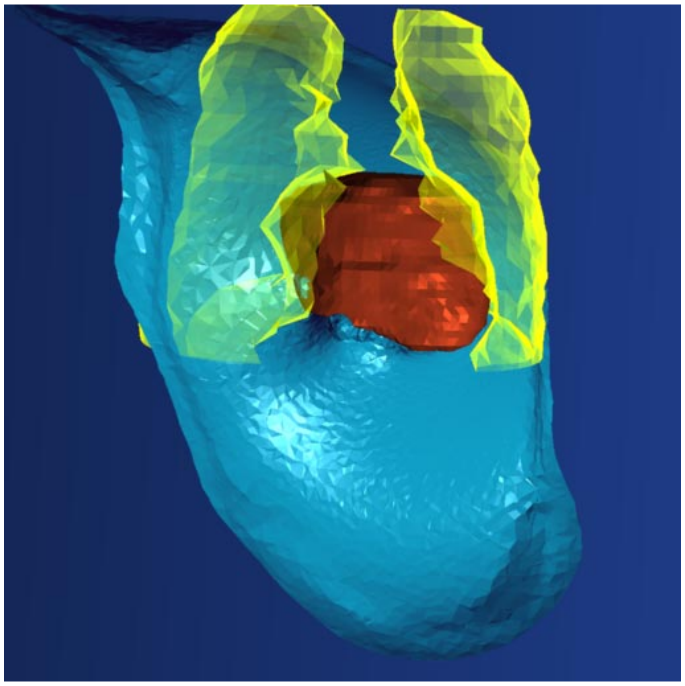


Fig. 12. Ray tracing of a 1.08 million element unstructured mesh from bioelectric field simulation. The heart and lungs are represented as landmark polygonal meshes and are not part of the isosurface.

TABLE V

DATA FROM RAY TRACING UNSTRUCTURED GRIDS AT 512X512 PIXELS ON 1 TO 126 PROCESSORS. THE ADAPTIVELY REFINED DATASET IS FROM A BIOELECTRIC FIELD PROBLEM.

# cpus	FPS	speedup
1	0.108	1
2	0.21	1.97
3	0.32	2.95
4	0.42	3.91
6	0.63	5.86
8	0.84	7.78
12	1.25	11.56
16	1.64	15.20
24	2.44	22.58
32	3.21	29.68
48	4.76	44.07
64	6.46	59.81
96	9.05	83.80
124	11.13	103.06

approximately 15% of runtime. Generating such a frame rate using conventional graphics hardware would require approximately a 1.8 GPixel/second pixel fill rate and 900 Mbytes of texture memory.

VI. DISCUSSION

We contrast applying our algorithm to explicitly extracting polygonal isosurfaces from the Visible Woman data set. For the skin isosurface we generated 18,068,534 polygons. For the bone isosurface we generated 12,922,628 polygons. These numbers are consistent with those reported by Lorensen given that he was using a cropped version of the volume [41]. With this number of polygons, it would be challenging to achieve interactive rendering rates on conventional high-end graphics hardware. Our method can render a ray-traced isosurface of this data at roughly ten frames per second using a 512 by 512 image on 64 processors. Table VI shows the extraction time for the bone isosurface using both NOISE [42] and marching cubes [17]. Note that because we are using static load balancing, these numbers would improve with a dynamic load balancing scheme. However, this would still not allow interactive modification of the isovalue while displaying the isosurface. Although using a downsampled or simplified detail volume would allow interaction at the cost of some detail. Simplified, precomputed isosurfaces could also yield interaction, but storage and precomputation time would be significant. Triangle stripping could improve display rates by up to a factor of three because isosurface meshes are usually transform bound. Note that we gain efficiency for both the extraction and rendering components by not explicitly extracting the geometry. Our algorithm is therefore not well-suited for applications that will use the geometry for non-graphics purposes.

The interactivity of our system allows exploration of both the data by interactively changing the isovalue or viewpoint. For example, one could view the entire skeleton and interactively zoom in and modify the isovalue to examine the detail in the

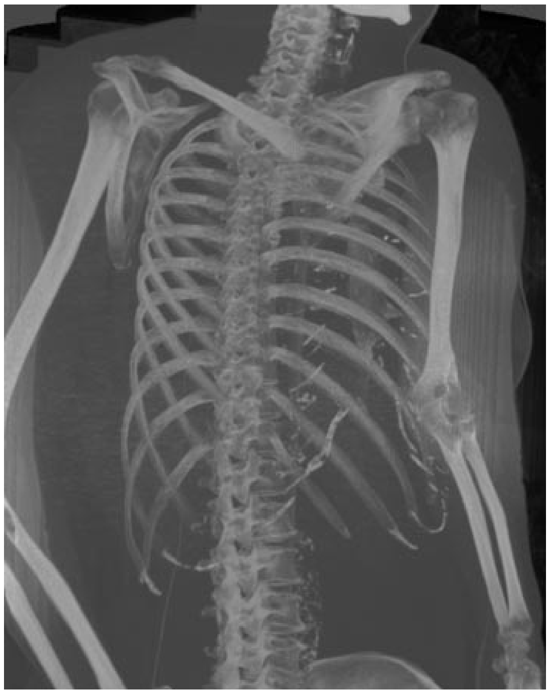
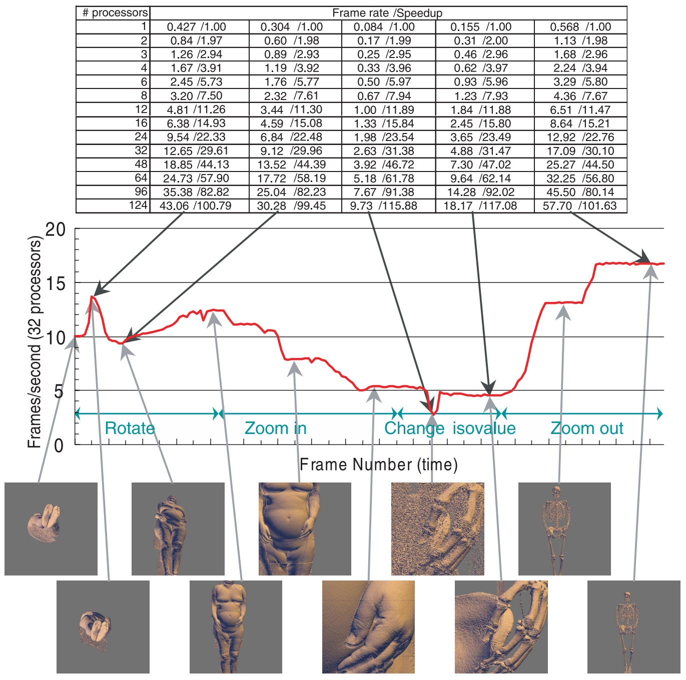


Fig. 13. A maximum-intensity projection of the Visible Female dataset.



# processors	Frame rate /Speedup				
1	0.427 /1.00	0.304 /1.00	0.084 /1.00	0.155 /1.00	0.568 /1.00
2	0.84 /1.97	0.60 /1.98	0.17 /1.99	0.31 /2.00	1.13 /1.98
3	1.26 /2.94	0.89 /2.93	0.25 /2.95	0.46 /2.96	1.68 /2.96
4	1.67 /3.91	1.19 /3.92	0.33 /3.96	0.62 /3.97	2.24 /3.94
6	2.45 /5.73	1.76 /5.77	0.50 /5.97	0.93 /5.96	3.29 /5.80
8	3.20 /7.50	2.32 /7.61	0.67 /7.94	1.23 /7.93	4.36 /7.67
12	4.81 /11.26	3.44 /11.30	1.00 /11.89	1.84 /11.88	6.51 /11.47
16	6.38 /14.93	4.59 /15.08	1.33 /15.84	2.45 /15.80	8.64 /15.21
24	9.54 /22.33	6.84 /22.48	1.98 /23.54	3.65 /23.49	12.92 /22.76
32	12.65 /29.61	9.12 /29.96	2.63 /31.38	4.88 /31.47	17.09 /30.10
48	18.85 /44.13	13.52 /44.39	3.92 /46.72	7.30 /47.02	25.27 /44.50
64	24.73 /57.90	17.72 /58.19	5.18 /61.78	9.64 /62.14	32.25 /56.80
96	35.38 /82.82	25.04 /82.23	7.67 /91.38	14.28 /92.02	45.50 /80.14
124	43.06 /100.79	30.28 /99.45	9.73 /115.88	18.17 /117.08	57.70 /101.63

Fig. 14. Variation in framerate as the viewpoint and isovalue changes.

TABLE VI
EXPLICIT BONE ISOSURFACE EXTRACTION TIMES IN SECONDS.

# cpus	NOISE build	NOISE extract	Marching cubes
1	4838	110	627
2	2109	81	324
4	1006	56	171
8	885	31	93
16	437	24	49
32	118	14	26
64	59	12	24

toes all at about ten FPS. The variation in framerate is shown in Fig. 14.

Brady et al. [43] describe a system which allows, on a Pentium workstation with accelerated graphics, interactive navigation through the Visible Human data set. Their technique is two-fold: 1) combine frustum culling with intelligent paging from disk of the volume data, and 2) utilize a two-phase perspective volume rendering method which exploits coherence in adjacent frames. Their work differs from ours in that they are using incremental direct volume rendering while we are exploiting isosurface or MIP rendering. This is evidenced by their incremental rendering times of about 2 seconds per frame for a 480x480 image. A full (non-incremental) rendering is nearly 20 seconds using their technique. For a single CPU, our isosurface rendering time is several seconds per frame (see Table II) depending on viewpoint. While it is difficult to directly compare these techniques due to their differing application focus, our method allows for the entire data set to reside within the view frustum without severe performance penalties since we are exploiting parallelism.

The architecture of the parallel machine plays an important role in the success of this technique. Since any processor can randomly access the entire dataset, the dataset must be available to each processor. Nonetheless, there is fairly high locality in the dataset for any particular processor. As a result, a shared memory or distributed shared memory machine, such as the SGI Origin 2000, is ideally suited for this application. The load balancing mechanism also requires a fine-grained low-latency communication mechanism for synchronizing work assignments and returning completed image tiles. With an attached Infinite Reality graphics engine, we can display images at high frame rates without network bottlenecks. We feel that implementing a similar technique on a distributed memory machine would be extraordinarily challenging, and would probably not achieve the same rates without duplicating the dataset on each processor.

VII. FUTURE WORK AND CONCLUSIONS

Since all computation is performed in software, there are many avenues which deserve exploration. Ray tracers have a relatively clean software architecture, in which techniques can be added without interfering with existing techniques, without re-unrolling large loops, and without complicated state management as are characteristic of a typical polygon renderer.

We believe the following possibilities are worth investigating:

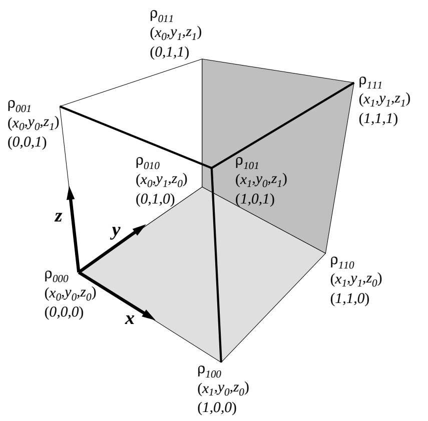

ρ_{011} (x_0,y_1,z_1) $(0,1,1)$
ρ_{111} (x_1,y_1,z_1) $(1,1,1)$
ρ_{001} (x_0,y_0,z_1) $(0,0,1)$
ρ_{010} (x_0,y_1,z_0) $(0,1,0)$
ρ_{101} (x_1,y_0,z_1) $(1,0,1)$
z
y
ρ_{110} (x_1,y_1,z_0) $(1,1,0)$
ρ_{000} (x_0,y_0,z_0) $(0,0,0)$
x
ρ_{100} (x_1,y_0,z_0) $(1,0,0)$

Fig. 15. The geometry for a cell. The bottom coordinates are the (u, v, w) values for the intermediate point.

- Exploration of other hierarchical methods in addition to the multilevel hierarchy described above.
- Combination with other scalar and vector visualization tools, such as cutting planes, surface maps, streamlines, etc.
- Using higher-order interpolants. Although numerical root finding would be necessary, the images might look better [19]. Since the intersection routine is not the bottleneck the degradation in performance might be reasonable.

We have shown that ray tracing can be a practical alternative to explicit isosurface extraction for very large datasets. As data sets get larger, and as general purpose processing hardware becomes more powerful, we expect this to become a very attractive method for visualizing large scale scalar data both in terms of speed and rendering accuracy.

VIII. ACKNOWLEDGMENTS

Thanks to Matthew Bane and Michelle Miller for comments on the paper. Thanks to Chris Johnson for providing the open collaborative research environment that allowed this work to happen. Special thanks to Steve Modica and Robert Cummins at SGI for crucial bug fixes in support code. This work was supported by the SGI Visual Supercomputing Center, the Utah State Centers of Excellence, the Department of Energy and the National Science Foundation. Special thanks to Jamie Painter and the Advanced Computing Laboratory at Los Alamos National Laboratory for access to a 128 processor machine for final benchmarks. Ruth Klepfer provide assistance in obtaining the various unstructured data sets.

APPENDIX

I. RAY-ISOSURFACE INTERSECTION FOR TRILINEAR BOXES

This appendix expands on some details of the intersection of a ray and a trilinear surface. It is not new research, but is helpful for implementors.

A rectilinear volume is composed of a three dimensional array of point samples that are aligned to the Cartesian axes and are equally spaced in a given dimension. A single cell from such a volume is shown in Figure 15. Other cells can be generated by exchanging indices (i, j, k) for the zeros and ones in the figure.

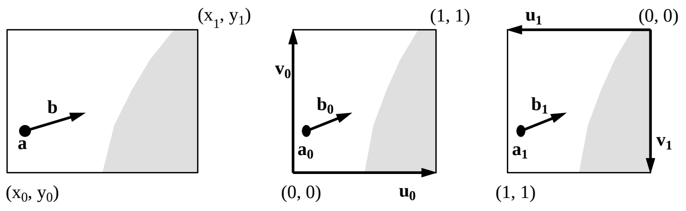


Fig. 16. Various coordinate systems used for interpolation and intersection.

The density at a point within the cell is found using *trilinear* interpolation:

$$\begin{aligned}\rho(u,v,w) \quad = \quad & (1-u)(1-v)(1-w)\rho_{000} + \\ & (1-u)(1-v)(w)\rho_{001} + \\ & (1-u)(v)(1-w)\rho_{010} + \\ & (u)(1-v)(1-w)\rho_{100} + \\ & (u)(1-v)(w)\rho_{101} + \\ & (1-u)(v)(w)\rho_{011} + \\ & (u)(v)(1-w)\rho_{110} + \\ & (u)(v)(w)\rho_{111}\end{aligned} \tag{1}$$

where

$$\begin{aligned} u &= \frac{x-x_0}{x_1-x_0} \\ v &= \frac{y-y_0}{y_1-y_0} \\ w &= \frac{z-z_0}{z_1-z_0}\end{aligned} \tag{2}$$

Note that

$$\begin{aligned} 1-u &= \frac{x_1-x}{x_1-x_0} \\ 1-v &= \frac{y_1-y}{y_1-y_0} \\ 1-w &= \frac{z_1-z}{z_1-z_0}\end{aligned} \tag{3}$$

If we redefine $u_0 = 1-u$ and $u_1 = u$, and similar definitions for v_0, v_1, w_0, w_1, then we get:

$$\rho = \sum_{i,j,k=0,1} u_i v_j w_k \rho_{ijk}$$

For a given point (x, y, z) in the cell, the surface normal is given by the gradient with respect to (x, y, z):

$$\vec{N} = \vec{\nabla}\rho = \left(\frac{\partial\rho}{\partial x}, \frac{\partial\rho}{\partial y}, \frac{\partial\rho}{\partial z}\right)$$

So the normal vector of $(N_x, N_Y, N_z) = \vec{\nabla}\rho$ is

$$N_x = \sum_{i,j,k=0,1} \frac{(-1)^{i+1} v_j w_k}{x_1 - x_0}\rho_{ijk}$$

$$N_y = \sum_{i,j,k=0,1} \frac{(-1)^{j+1} u_i w_k}{y_1 - y_0}\rho_{ijk}$$

$$N_z = \sum_{i,j,k=0,1} \frac{(-1)^{k+1} u_i v_j}{z_1 - z_0}\rho_{ijk}$$

Lin and Ching [18] described a method for intersecting a ray with a trilinear cell. We derive a similar result that is more tailored to our implementation.

See figure 16. Given a ray $\vec{p} = \vec{a} + t\vec{b}$, the intersection with the isosurface occurs where $\rho(\vec{p}) = \rho_{\text{iso}}$. We can convert this ray into coordinates defined by (u_0, v_0, w_0): $\vec{p_0} = \vec{a}_0 + t\vec{b_0}$ and a third ray defined by $\vec{p_1} = \vec{a}_1 + t\vec{b}_1$. These rays $\vec{p_0} = \vec{a}_0 + t\vec{b_0}$ and $\vec{p_1} = \vec{a}_1 + t\vec{b}_1$ are now used for the intersection computation. These two rays are in the two coordinate systems (Figure 16):

$$\vec{a}_0 = (u_0^a, v_0^a, w_0^a) = \left(\frac{x_1 - x_a}{x_1 - x_0}, \frac{y_1 - y_a}{y_1 - y_0}, \frac{z_1 - z_a}{z_1 - z_0}\right),$$

and

$$\vec{b}_0 = (u_0^b, v_0^b, w_0^b) = \left(\frac{x_b}{x_1 - x_0}, \frac{y_b}{y_1 - y_0}, \frac{z_b}{z_1 - z_0}\right).$$

These equations are different because $\vec{a}_0$ is a location and $\vec{b}_0$ is a direction. The equations are similar for $\vec{a}_1$ and $\vec{b}_1$:

$$\vec{a}_1 = (u_1^a, v_1^a, w_1^a) = \left(\frac{x_a - x_0}{x_1 - x_0}, \frac{y_a - y_0}{y_1 - y_0}, \frac{z_a - z_0}{z_1 - z_0}\right),$$

and

$$\vec{b}_1 = (u_1^b, v_1^b, w_1^b) = \left(\frac{-x_b}{x_1 - x_0}, \frac{-y_b}{y_1 - y_0}, \frac{-z_b}{z_1 - z_0}\right).$$

Note that t is the same for all three rays. This point can be found by traversing the cells and doing a brute-force algebraic solution for t. The intersection with the isosurface $\rho(\vec{p}) = \rho_{\text{iso}}$ occurs where:

$$\rho_{\text{iso}} = \sum_{i,j,k=0,1} \left(u_i^a + tu_i^b\right)\left(v_i^a + tv_i^b\right)\left(w_i^a + tw_i^b\right)\rho_{ijk}$$

This can be simplified to a cubic polynomial in t:

$$At^3 + Bt^2 + Ct + D = 0$$

where

$$A = \sum_{i,j,k=0,1} u_i^b v_i^b w_i^b \rho_{ijk}$$

$$B = \sum_{i,j,k=0,1} \left(u_i^a v_i^b w_i^b + u_i^b v_i^a w_i^b + u_i^b v_i^b w_i^a\right)\rho_{ijk}$$

$$C = \sum_{i,j,k=0,1} \left(u_i^b v_i^a w_i^a + u_i^a v_i^b w_i^a + u_i^a v_i^a w_i^b\right)\rho_{ijk}$$

$$D = -\rho_{\text{iso}} + \sum_{i,j,k=0,1} u_i^a v_i^a w_i^a \rho_{ijk}$$

The solution to a cubic polynomial is discussed the article by Schwarze [44]. We used his code (available on the web in several *Graphics Gems* archive sites) with two modifications: special cases for quadratic or linear solutions (his code assumes A is non-zero), and the EQN_EPS parameter was set to 1.e-30 which provided for maximum stability for large coefficients.

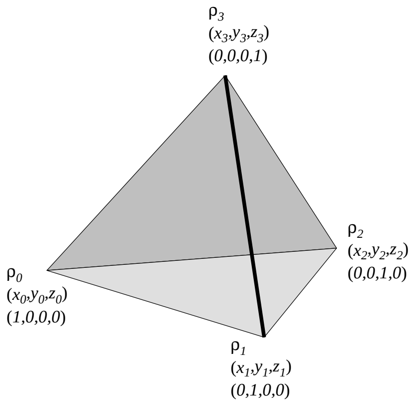


Fig. 17. The geometry for a barycentric tetrahedron. The bottom barycentric coordinates are the $(\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ values for the vertex.

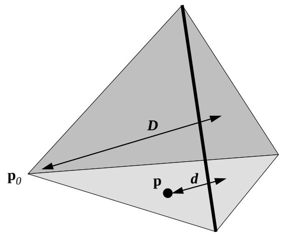


Fig. 18. The barycentric coordinate α_0 is the scaled distance d/D. The distances are d and D are signed distances to the plane containing the triangular face opposite $\mathbf{p}_0$.

II. Ray-Isosurface Intersection for Barycentric Tetrahedra

This appendix is geared toward implementors and discusses the details of intersecting a ray with a barycentric tetrahedral isosurface.

An unstructured mesh is composed of three dimensional point samples arranged into a simplex of tetrahedra. A single cell from such a volume is shown in Figure 17, where the four vertices are $\mathbf{p}_i = (x_i, y_i, z_i)$.

The density at a point within the cell is found using *barycentric* interpolation:

$$\rho(\alpha_0, \alpha_1, \alpha_2, \alpha_3) = \alpha_0\rho_0 + \alpha_1\rho_1 + \alpha_2\rho_2 + \alpha_3\rho_3,$$

where

$$\alpha_0 + \alpha_1 + \alpha_2 + \alpha_3 = 1.$$

Similar equations apply to points in terms of the vertices. For points inside the tetrahedron, all barycentric coordinates are positive.

One way to compute barycentric coordinates is to measure the distance from the plane that defines each face (Figure 18). This is accomplished by choosing a plane equation $f_0(\mathbf{p}) = 0$ such that $f_0(\mathbf{p}_0) = 1$. Such equations for all four plane-faces of the tetrahedron allow us to compute barycentric coordinates of a point $\mathbf{p}$ directly: $\alpha_i(\mathbf{p}) = f_i(\mathbf{p})$.

If we take the ray $\mathbf{p}(t) = \mathbf{a} + t\vec{\mathbf{b}}$, then we get an equation for the density along the ray:

$$\rho(t) = \sum_{i=0}^{3} f_i(\mathbf{a} + t\vec{\mathbf{b}})\rho_i.$$

If we solve for $\rho(t) = \rho_{\text{iso}}$, then we get a linear equation in t, so solution is straightforward. If the resulting barycentric coordinates of $\mathbf{p(t)}$ are all positive, the point is in the tetrahedron, and it is accepted. Finding the normal is just a matter of taking the gradient:

$$\vec{\nabla}\rho(\mathbf{p}) = \sum_{i=0}^{3} \rho_i \vec{\nabla} f_i(\mathbf{p}).$$

Because f_i is just a plane equation of the form $\vec{\mathbf{n}}_i \cdot (\mathbf{p} - \mathbf{q}_i)$ where $\mathbf{q}_i$ is a constant point, the normal vector $\vec{\mathbf{N}}$ is simply

$$\vec{\mathbf{N}} = \sum_{i=0}^{3} \rho_i \vec{\mathbf{n}}_i.$$

This is a constant for the cell, but we do not precompute it since it would require extra memory accesses.

References

[1] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan, “Interactive ray tracing for isosurface rendering,” in *Proceedings of Visualization ’98*, October 1998.
[2] Steven Parker, William Martin, Peter-Pike Sloan, Peter Shirley, Brian Smits, and Charles Hansen, “Interactive ray tracing,” in *Symposium on Interactive 3D Graphics*, April 1999.
[3] James T. Kajiya, “An overview and comparison of rendering methods,” *A Consumer’s and Developer’s Guide to Image Synthesis*, pp. 259–263, 1988, ACM Siggraph ’88 Course 12 Notes.
[4] Mark Levoy, “Display of surfaces from volume data,” *IEEE Computer Graphics & Applications*, vol. 8, no. 3, pp. 29–37, 1988.
[5] Paolo Sabella, “A rendering algorithm for visualizing 3d scalar fields,” *Computer Graphics*, vol. 22, no. 4, pp. 51–58, July 1988, ACM Siggraph ’88 Conference Proceedings.
[6] Craig Upson and Micheal Keeler, “V-buffer: Visible volume rendering,” *Computer Graphics*, vol. 22, no. 4, pp. 59–64, July 1988, ACM Siggraph ’88 Conference Proceedings.
[7] E. Reinhard, A.G. Chalmers, and F.W. Jansen, “Overview of parallel photo-realistic graphics,” in *Eurographics ’98*, 1998.
[8] Arie Kaufman, *Volume Visualization*, IEEE CS Press, 1991.
[9] Lisa Sobierajski and Arie Kaufman, “Volumetric Ray Tracing,” *1994 Workshop on Volume Visualization*, pp. 11–18, Oct. 1994.
[10] K.L. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh, “Parallel Volume Rendering using Binary-Swap Compositing,” *IEEE Comput. Graphics and Appl.*, vol. 14, no. 4, pp. 59–68, July 1993.
[11] Michael J. Muuss, “Rt and remrt - shared memory parllel and network distributed ray-tracing programs,” in *USENIX: Proceedings of the Fourth Computer Graphics Workshop*, October 1987.
[12] Guy Vézina, Peter A. Fletcher, and Philip K. Robertson, “Volume Rendering on the MasPar MP-1,” in *1992 Workshop on volume Visualization*, 1992, pp. 3–8, Boston, October 19-20.
[13] P. Schröder and Gordon Stoll, “Data Parallel Volume Rendering as Line Drawing,” in *1992 Workshop on volume Visualization*, 1992, pp. 25–31, Boston, October 19-20.
[14] Michael J. Muuss, “Towards real-time ray-tracing of combinatorial solid geometric models,” in *Proceedings of BRL-CAD Symposium*, June 1995.
[15] Scott Whitman, “A Survey of Parallel Algorithms for Graphics and Visualization,” in *High Performance Computing for Computer Graphics and Visualization*, 1995, pp. 3–22, Swansea, July 3–4.
[16] B. Wyvill G. Wyvill, C. McPheeters, “Data structures for soft objects,” *The Visual Computer*, vol. 2, pp. 227–234, 1986.
[17] William E. Lorensen and Harvey E. Cline, “Marching cubes: A high resolution 3d surface construction algorithm,” *Computer Graphics*, vol. 21, no. 4, pp. 163–169, July 1987, ACM Siggraph ’87 Conference Proceedings.
[18] Chyi-Cheng Lin and Yu-Tai Ching, “An efficient volume-rendering algorithm with an analytic approach,” *The Visual Computer*, vol. 12, no. 10, pp. 515–526, 1996.
[19] Stephen Marschner and Richard Lobb, “An evaluation of reconstruction filters for volume rendering,” in *Proceedings of Visualization ’94*, October 1994, pp. 100–107.
[20] Milos Sramek, “Fast surface rendering from raster data by voxel traversal using chessboard distance,” in *Proceedings of Visualization ’94*, October 1994, pp. 188–195.

[21] Georgios Sakas, Marcus Grimm, and Alexandros Savopoulos, "Optimized maximum intensity projection (MIP)," in *Eurographics Rendering Workshop 1995*. Eurographics, June 1995.

[22] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan, "Volume rendering," *Computer Graphics*, vol. 22, no. 4, pp. 65–74, July 1988, ACM Siggraph '88 Conference Proceedings.

[23] Don Speray and Steve Kennon, "Volume probes: Interactive data exploration on arbitrary grids," in *Computer Graphics (San Diego Workshop on Volume Visualization)*, 1990, pp. 5–12.

[24] John Amanatides and Andrew Woo, "A fast voxel traversal algorithm for ray tracing," in *Eurographics '87*, 1987.

[25] Akira Fujimoto, Takayu Tanaka, and Kansei Iwata, "Arts: Accelerated ray-tracing system," *IEEE Computer Graphics & Applications*, pp. 16–26, April 1986.

[26] John Danskin and Pat Hanrahan, "Fast algorithms for volume ray tracing," *1992 Workshop on Volume Visualization*, pp. 91–98, 1992.

[27] Marc Levoy, "Efficient ray tracing of volume data," *ACM Transactions on Graphics*, vol. 9, no. 3, pp. 245–261, July 1990.

[28] J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation," in *Computer Graphics (San Diego Workshop on Volume Visualization)*, Nov. 1990, pp. 57–62.

[29] J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation," *ACM Transactions on Graphics*, vol. 11, no. 3, pp. 201–227, July 1992.

[30] Jane Wilhelms and Judy Challinger, "Direct volume rendering of curvilinear volumes," in *Computer Graphics (San Diego Workshop on Volume Visualization)*, Nov. 1990, pp. 41–47.

[31] M. Garrity, "Ray Tracing Irregular Volume Data," in *1990 Workshop on Volume Visualization*, 1990, pp. 35–40, San Diego.

[32] Cláudio Silva, Joseph S. B. Mitchell, and Arie E. Kaufman, "Fast rendering of irregular grids," in *1996 Volume Visualization Symposium*. IEEE, Oct. 1996, pp. 15–22, ISBN 0-89791-741-3.

[33] C. E. Prakash and S. Manohar, "Volume rendering of unstructured grids–a voxelization approach," *Computers & Graphics*, vol. 19, no. 5, pp. 711–726, Sept. 1995, ISSN 0097-8493.

[34] Michael B. Cox and David Ellsworth, "Application-controlled demand paging for Out-of-Core visualization," in *Proceedings of Visualization '97*, October 1997, pp. 235–244.

[35] James Arvo and David Kirk, "A survey of ray tracing acceleration techniques," in *An Introduction to Ray Tracing*, Andrew S. Glassner, Ed. Academic Press, San Diego, CA, 1989.

[36] David Jevans and Brian Wyvill, "Adaptive voxel subdivision for ray tracing," in *Proceedings of Graphics Interface '89*, June 1989, pp. 164–172.

[37] Kryzsztof S. Klimansezewski and Thomas W. Sederberg, "Faster ray tracing using adaptive grids," *IEEE Computer Graphics & Applications*, vol. 17, no. 1, pp. 42–51, Jan.-Feb. 1997, ISSN 0272-1716.

[38] Al Globus, "Octree optimization," Tech. Rep. RNR-90-011, NASA Ames Research Center, July 1990.

[39] Greg Nielson and Bernd Hamann, "The asymptotic decider: Resolving the ambiguity in marching cubes," in *Proceedings of Visualization '91*, October 1991, pp. 83–91.

[40] National Library of Medicine (U.S.) Board of Regents, "Electronic imaging: Report of the board of regents. u.s. department of health and human services, public health service, national institutes of health," NIH Publication 90-2197, 1990.

[41] Bill Lorensen, "Marching through the visible woman," http://www.crd.ge.com/cgi-bin/vw.pl, 1997.

[42] Y Livnat, H. Shen, and C. R. Johnson, "A near optimal isosurface extraction algorithm using the span space," *IEEE Trans. Vis. Comp. Graphics*, vol. 2, no. 1, pp. 73–84, 1996.

[43] M.L. Brady, K.K. Jung, H.T. Nguyen, and T.PQ. Nguyen, "Interactive Volume Navigation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 3, pp. 243–256, July 1998.

[44] Jochen Schwarze, "Cubic and quartic roots," in *Graphics Gems*, Andrew Glassner, Ed., pp. 404–407. Academic Press, San Diego, 1990.

Steven Parker is a research scientist in the Department of Computer Science at the University of Utah. His research focuses on problem solving environments, which tie together scientific computing, scientific visualization, and computer graphics. He is the principal architect of the SCIRun Software System, which formed the core of his Ph.D. dissertation. He was a recipient of the Computational Science Graduate Fellowship from the Department of Energy. He received a B.S. in Electrical Engineering from the University of Oklahoma in 1992.

Michael Parker Michael Parker is a Ph.D. student in Computer Science at the University of Utah. He is interested in Computer Architecture and VLSI Design. He has recently concluded his work on a project to reduce communication latency and overhead in clusters of workstations. He is currently involved in the architecture of an adaptable memory controller. His dissertation deals with reducing I/O and communication overhead and latency. He received a B.S. in Electrical Engineering from the University of Oklahoma in 1995.

Yarden Livnat is a Research Associate at the Department of Computer Science at the University of Utah. working with the Scientific Computing and Imaging Research Group. Yarden received a B.Sc. in computer science in 1982 from Ben Gurion University Israel and an M.Sc. cum laude in computer science from the Hebrew University, Israel in 1991. He will receive his Ph.D from the University of Utah in 1999. His research interests include computational geometry, scientific computation and visualization and computer generated holograms.

Peter-Pike Sloan has recently joined the Graphics Research group at Microsoft as a Research SDE. He previously was a student at the University of Utah and worked in the Scientific Computing and Imaging group for Chris Johnson. He has also previously worked on a 3D Painting product at Parametric Technology in Salt Lake City. His interests span the spectrum of computer graphics, and most recently has been working/dabbling in the areas of interactive techniques, image-based rendering, surface parameterizations, and non-photorealistic rendering.

Charles Hansen is an Associate Professor of Computer Science at the University of Utah. From 1989 to 1997, he was a Research Associate Professor of Computer Science at Utah. From 1989 to 1997, he was a Technical Staff Member in the Advanced Computing Laboratory (ACL) located at Los Alamos National Laboratory where he formed and directed the visualization efforts in the ACL. His research interests include large-scale scientific visualization, massively parallel processing, parallel computer graphics algorithms, 3D shape representation, and computer vision. He received a B.S. in Computer Science from Memphis State University in 1981 and a Ph.D. in Computer Science from the University of Utah in 1987. He was a Bourse de Chateaubriand PostDoc Fellow at INRIA, in 1987 and 1988.

Peter Shirley is an Assistant Professor of Computer Science at the University of Utah. From 1994 to 1996 he was a Visiting Assistant Professor at the Cornell Program of Computer Graphics. From 1990 to 1994 he was an Assistant Professor of Computer Science at Indiana University. His research interests include visualization, realistic rendering, and application of visual perception research in computer graphics. He received a B.A. in Physics from Reed College in 1984 and a Ph.D. in Computer Science from the University of Illinois at Urbana/Champaign in 1991.

Eurographics Symposium on Rendering (2004)
H. W. Jensen, A. Keller (Editors)

An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models

Ingo Wald[†‡], Andreas Dietrich[‡], and Phlipp Slusallek[‡]

[†]MPI Informatik, Saarbrücken, Germany,
wald@mpi-sb.mpg.de

[‡]Saarland University, Saarbrücken, Germany
{dietrich,slusallek}@cs.uni-sb.de

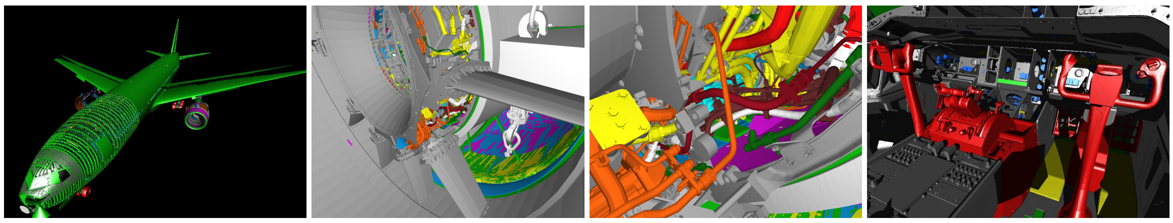


Figure 1: *The Boeing 777 model containing 350 million triangles. a.) Overview over the entire model, including shadows. b.) Zoom into the engine, showing intricately interweaved, complex geometry. c.) The same as b.), but zooming in even closer. All of the individual parts of the entire plane are modeled at this level of complexity. d.) The cockpit, including shadows. Using our out-of-core visualization scheme, all of these frames can be rendered interactively at 3 7 frames per second on a single desktop PC.*

Abstract

With the tremendous advances in both hardware capabilities and rendering algorithms, rendering performance is steadily increasing. Even consumer graphics hardware can render many million triangles per second. However, scene complexity seems to be rising even faster than rendering performance, with no end to even more complex models in sight.
In this paper, we are targeting the interactive visualization of the "Boeing 777" model, a highly complex model of 350 million *individual triangles, which – due to its sheer size and complex internal structure – simply cannot be handled satisfactorily by today's techniques. To render this model, we use a combination of real-time ray tracing, a low-level out of core caching and demand loading strategy, and a hierarchical, hybrid volumetric/lightfield-like approximation scheme for representing not-yet-loaded geometry. With this approach, we are able to render the full 777 model at several frames per second even on a single commodity desktop PC.*

Keywords: Real-time rendering, out-of-core rendering, complex models, distributed computing, ray tracing

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Ray tracing I.6.3 [Simulation and Modeling]: Applications I.3.2 [Computer Graphics]: Distributed/network graphics

1. Introduction

For many years now, the performance of commodity CPUs has increased at a rate of a factor of two roughly every 18 months. At least in the last few years, the performance of graphics hardware has grown even faster, having led to commodity graphics hardware that can render up to several million triangles per second. In addition to this "free" increase in rendering performance, we also see a steady improvement in rendering algorithms. With all this taken together, the model complexity that is affordable at interactive rates is constantly and rapidly increasing.

Unfortunately, the complexity of practical models seems to be rising even faster: First of all, users of modeling systems (and game designers as well) tend to immediately spend every grain of increased performance into even more detail, i.e. into more triangles.

Additionally, virtual prototyping is becoming increasingly important and hardwired into the design process. Traditionally, virtual reality has been but loosely coupled to the actual design process, and has merely visualized semi-manually prepared (i.e. simplified) versions of the CAD models. With VR getting increasingly involved into the production pro-

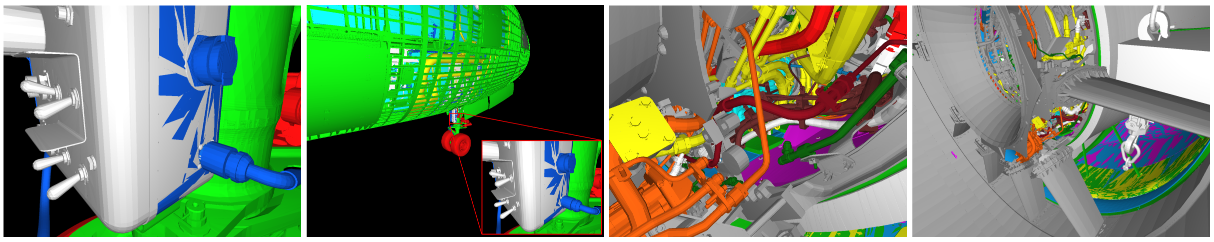


Figure 2: *Some example closeups of the 777, to show the high geometric complexity, small degree of occlusion, and complex topological structure of the model, which make it complicated for most simplification/approximation-based approaches. a.) Zoom onto a small object of roughly one cubic foot in size, showing each individual nut and bolt modeled with hundreds of badly-shaped triangles. Multiple surfaces with different materials overlap themselves, as can be seen e.g. on the mixed white/blue-patched structure. Due to the randomly jittered vertex positions (introduced to prevent data theft), such structures self-intersect with each other randomly. b.) The same view from a few meters away. The left image corresponds to the red rectangle in the middle. c.) & d.) The same for a view into the engine. Note how much detail is visible in that view, and how the many pipes and cables are intricately interweaved. The low degree of occlusion is also demonstrated in Figure 3.*

cess, there is a growing need to render models "directly out of the database", i.e. *without* any model "preparation" and simplification. Such CAD datasets, however, can be quite complex.

Furthermore, the increased use of "collaborative engineering" for large-scale industrial projects leads to models consisting of hundreds and thousands of individual parts (potentially created by different suppliers), each of which modeled at whatever complexity and accuracy has been affordable for that individual part. In practice, this often means that each individual nut and bolt of a model (also see Figures 1– 3) is represented in full geometric detail.

Taken together, these developments lead to a growth in model complexity that seems to be at least as fast as the growth in hardware resources. An end to these developments currently is not foreseeable.

In this paper, we are targeting the interactive visualization of the "Boeing 777" model, a model consisting of roughly 350 *million* individual triangles, i.e. without using instantiation to generate this triangle count. Just the raw input data of that model ships – in compressed form – on a total of eleven CDs. After unpacking and storing each triangle as a triple of three floats without any additional acceleration data, the model is 12 GByte in size, and requires several minutes just for reading it from disk. For this kind of model complexity, generating frame rates of several frames per second is quite challenging for contemporary massive model rendering approaches.

1.1. Outline

In the remainder of this paper, we will first discuss relevant related work regarding rendering complex models in Section 2, and will particularly discuss their problems in handling a model of the size, topological structure, and complexity of the 777. Based on this discussion, we will then develop and describe our new approach to such models: After giving an overview of our system in Section 3, we will then describe the caching and demand loading subsystem in Section 4, and our hierarchical approximation scheme for not-yet-loaded geometry in Section 5. Section 6 then summarizes some results of using our framework for rendering the full 777 model on a single dual-1.8 GHz AMD Opteron desktop PC with 6 GB RAM. Finally, Section 7 concludes and ends with an outlook on future work.

2. Previous Work

Due to the practical and industrial importance of rendering complex datasets, there exists a vast suite of different approaches to this problem. However, many of these techniques perform well only for specific kinds of models, but prove problematic for others.

Brute-Force Rendering. Obviously, a model of the size of the 777 cannot be handled by a pure brute-force approach. In theory, the most up-to-date graphics hardware (e.g. an NVIDIA Quadro FX 4000) features a theoretical peak performance of 133 million shaded and lit triangles per second, and could thus raster the full model in only a few seconds. Unfortunately, the practical performance usually is much lower, in particular for models that do not fit into graphics card memory. Thus, typical approaches to rendering complex datasets rely on reducing the number of triangles to be sent to the graphics card.

Culling Techniques. Typical approaches like view-frustum culling are quite limited for a model of as high a depth complexity as the 777. Depth complexity can only be handled by taking occlusion into account. At least for 2D or $2\frac{1}{2}$D scenes (e.g. urban walkthroughs), occlusion can be conservatively precomputed quite well [WWS00]. In three dimensions, in particular with as few occlusion as in the 777 (see Figures 2 and 3), visibility preprocessing is quite problematic [ACW*99].

Instead of precomputing visibility, the alternative is to use a hierarchical visibility culling mechanism, e.g. the hierarchical z-buffer [GKM93], possibly implemented via OpenGL occlusion queries [BMH99]. However, neither of these approaches has been designed for handling gigabyte-sized models that do not even fit into main memory. Recently, Correa et al. [CKS03] have proposed a visibility-based out-of-core rendering framework that can also cope with models larger than memory. However, even if visibility-based approaches would achieve perfect culling, for many views the low degree of occlusion in the 777 still results in millions of potentially visible triangles.

To handle this case, the randomized z-buffer algorithm [WFP*01] randomly selects one triangle out of the many triangles that project onto a pixel. This, however, works only for scenes in which it does not actually matter which of the triangles is chosen, e.g. for picking one of the thousands of leaves of a tree. For the 777 the exact ordering and mutual occlusion of even very close-by triangles is quite important. For example, in order to avoid the small yellow pipes "shining through" the green hull to which they are attached. Finally, like all the previously mentioned techniques, the randomized z-buffer is not designed for handling models that do not even fit into memory.

Model Simplification. As pure visibility culling even theoretically is not enough, many approaches try to "reduce" the model by some form of mesh simplification, e.g. via edge contraction, vertex removal, or remeshing (see e.g. [CMS98]), often requiring some form of "well-behaving" geometry. Typically, these methods perform best for highly tessellated surfaces that are otherwise relatively smooth, flat, and topologically simple. In the 777 the triangles actually form many detailed, loosely connected though interweaving parts of complex topological structure, such as mazes of tubes, pipes, and cables (see Figures 2 and 3). Such kinds of geometry are very hard to simplify effectively in a robust manner.

Moreover, each part of the 777 comes in a "soup" of unconnected triangles, without any connectivity information, often forming self-intersecting and overlapping surfaces (see Figure 2a) with different material properties. Even worse, the vertex positions have been slightly jittered to prevent public spreading of the sensitive original CAD data. Thus, overlapping surfaces are not perfectly aligned, but rather randomly intersect each other multiple times. For such kinds of input data, most geometrically based algorithms are likely to fail.

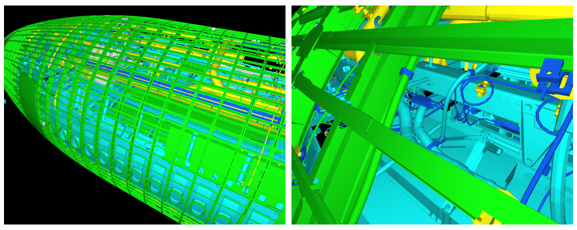


Figure 3: *In comparison to most other "massive" models, the 777 has a* much *lower degree of occlusion. a.) Zoom onto the front part of the model, where the rays penetrate deeply into the model. b.) Closeup of the geometry that can be seen through the ribs of the plane.*

As each individual technique usually has a weak point, the UNC's MMR/Gigawalk system [ACW*99, BSGM02, GLY*03] is based on a combination of different techniques, combining mesh simplification, visibility preprocessing, impostors [SDB97], textured depth meshes, and hierarchical occlusion maps [ZMHH97]. However, as just discussed *each* of these individual parts is problematic in the 777. This raises the question whether a combination of these techniques can still succeed in each technique masking the shortcomings of the other.

Image-based and Point-based Approaches. In addition to these "traditional" methods, researchers have also looked into image-based and point-based approaches. For example, the Holodeck [WS99], Render Cache [WDP99], and Edge-and-Point-Image [BWG03] progressively sample the model asynchronously to displaying it, and interactively reconstruct the image from these sparse samples. In principle, both approaches might be applicable to the 777. However, the rays traced by these systems are likely to cause significant paging, resulting in prohibitively long times for generating enough image samples. This is likely to result in severe subsampling, and in strong visual artifacts.

As yet another alternative, researchers have proposed to represent models using point samples (see e.g. [CH02, PZvBG00]). Though this decouples geometric complexity from display complexity, the sparse number of samples often limits the detail that is present in the reconstructed image. To avoid this problem, QSplat [RL00] employs a hierarchical scheme in which the entire mesh is represented by at least one sample per triangle. However, its hierarchical approximation scheme assumes that nearby triangles can, if seen from a distance, be well approximated by a disk-shaped "splat" with filtered color and normal information. Like mesh simplification, this works only for relatively smooth and topologically simple surfaces, and is likely to fail for the geometrical structure of the 777 as described above.

Interactive Ray Tracing. Finally, complex models can also be visualized using interactive ray tracing. Due to its logarithmic dependence on scene complexity, ray tracing can easily handle even highly complex scenes of several million triangles at full detail. For example, the OpenRT realtime ray tracing system [Wal04] has been shown to interactively render the one *billion* triangle "Sunflowers" scene even including shadows, semi-transparent leaves, and moving geometry. However, this has only been possible through instantiation, i.e. by reusing the same kind of sunflower several thousand times, therefore being able to keep the entire model in main memory. For the 777 model, we simply cannot store the entire dataset – which occupies 30–40 GByte including acceleration structures – in main memory. Once the operating system starts to generate the inevitable page faults the ray tracer would run idle while waiting for data, and could not maintain interactivity.

In order to solve that problem, Pharr et al. [PKGH97] have proposed a caching and reordering scheme that reorders the rays in a way that minimizes disk I/O. Though this allows for efficiently ray tracing models that are much larger than main memory, the approach is not easily applicable to interactive rendering. A simplified version of this scheme has also been used by Wald et al. [WSB01]. They have proposed to "suspend" rays that would cause a page fault and load the required data asynchronously over the network while tracing other rays in the meantime. The stalled rays then get "resumed" once the data is available. Though that approach worked well for the target model (the 12.5 million triangle UNC Power Plant), it fails in interactively rendering a model as complex as the 777: The proposed suspend/resume approach can hide the loading latency only within the duration of one frame. In the 777, however, even a small camera change often triggers thousands of disk read requests that simply cannot be fulfilled within a single frame. Though prefetching (in the sense of e.g. [CKS03]) would help, it can hide loading latencies only to a limited degree.

Furthermore, their demand loading scheme was based on splitting the model into "voxels" of several thousand triangles, which were then loaded and discarded as required. This caching granularity is far too large for our purposes, as each individual ray may cause loading another of these voxels. Additionally, this method is prone to memory fragmentation, and carries a certain overhead for managing the data (also see the discussion in [DGP04]).

3. An Out-of-Core Framework for Interactively Rendering Massively Complex Models

As shown by the discussion in the previous section, contemporary techniques to handle massive models cannot easily cope with a model of the size, structure, and complexity of the 777. Thus, a new approach had to be taken.

Since ray tracing can in principle handle such massive amounts of geometry, in a first experiment we ported the OpenRT ray tracer to a shared-memory architecture, and experimented with rendering the 777 on 16 UltraSPARC III CPUs in a SUN Sun Fire 11K with 180 GB RAM. This allowed for storing the model – including pre-built BSP data – into the RAM disk, making it possible to load the entire scene within a few seconds, and to interactively inspect it at several frames per second, even including shadows.

With these successful experiments, we started designing an architecture that could deliver similar performance even on a commodity PC. In order to be able to at least *address* the entire model, we decided to build on AMD's 64-bit Opteron CPUs [AMD03], which have recently become available in commodity desktop systems. Compared to e.g. the Intel Itanium CPU, the Opteron also supports the IA32 SSE Instruction set [Int02], and thus can exploit also those traversal and intersection routines of OpenRT that have been specifically optimized towards SSE [WSBW01, Wal04]. This support for using SSE instructions – together with a nominally higher clock rate – allow the Opteron to easily outpace the UltraSPARC III. Instead of having to use many CPUs in a Sun Fire, we can achieve similar performance on a single dual-CPU Opteron PC.

Unfortunately, having a 64-bit address space allows for *addressing* the entire model, but cannot help the fact that we still are not able to keep it entirely in memory. We therefore decided to follow the approach of Wald et al. [WSB01], and use a combination of manual memory management and demand loading in order to detect and avoid page faults due to access to out-of-core memory. As discussed in the previous section, however, their approach had several shortcomings with respect to a 777-class model, mainly with respect to the design and implementation of the memory management scheme. Most importantly, their system has mainly been designed for *hiding* the scene access latency by suspending and resuming rays, which we have argued cannot work successfully for the 777.

As a consequence, our framework builds on two pillars: First, on a new memory management scheme that has been redesigned from scratch. It avoids the fragmentation, caching granularity, and I/O problems of the original approach, and is thus much better suited for a 777-class model. Second, our approach does not even try to hide scene access latency, but instead kills off potentially page-faulting rays, which are then being replaced by shading information from so-called "proxies". This is achieved by efficiently determining in advance accesses to parts of the BSP that may potentially lead to a page fault. Proxies are a pre-computed coarse yet appropriate approximate representation for the respective subtree. This proxy mechanism is similar to a hierarchical level-of-detail representation intermixed with the spatial index structure, and will be described in more detail in Section 5.

4. Memory Management

As just motivated, a memory management scheme based on manually managing individual sub-parts of several thousand triangles is inappropriate for the 777 due to memory fragmentation, much too coarse cache granularity, and thus bad memory efficiency and high I/O cost. In contrast to this, the Linux/UNIX memory mapping facilities (`mmap()` [BC02]) provide a convenient way of addressing and demand loading memory on a per-page basis. In particular, it realizes a unified cache, i.e. it does not matter which data is contained in which physical page, and it never pages in any data (e.g. shading information) that might not be required.

Leaving the memory management (MM) to the operating system greatly simplifies the design, and improves the efficiency of the implementation: For example, manual memory management requires to take special care in order to avoid race conditions where one thread accesses data that is just being freed by another thread. If not avoided by costly synchronization via mutexes, such race conditions usually lead to program crashes. If a similar race condition happens in our OS-based MM scheme, the worst that can happen is a page fault, as the pointer to the not available memory region is still considered valid. Additionally, working on "real" pointers minimizes the address lookup overhead, as this is done automatically by the processor's hardware MMU, and do not cost precious CPU time (also see [DPH*03]).

Finally, using this scheme is quite simple: All one has to do to implement this scheme is to precompute all static data structures (e.g. BSP index structures etc.), store them on disk in binary form, and map them into the address space via `mmap()`. This preprocessing is done in an out-of-core approach similar to [WSB01].

4.1. Detecting and Avoiding Page Faults

Though an OS-based MM system has many advantages over manual caching, it also has a major drawback in that we lose control over what data is loaded into or discarded from memory at what time. Although data is automatically paged in on demand upon accessing it, the resulting page fault stalls the rendering thread until the data is available.

To retain control over the caching process we implemented a hybrid memory management system, which uses the operating system to perform demand paging, but which detects and avoids potential page faults before they occur, and which manually steers page loading and eviction.

In order to avoid page faults, we have to detect whether or not memory referenced by a pointer is actually in core memory. Though Linux for this purpose offers the `mincore()` function, performing an OS call on *each* memory access obviously is not affordable. When taking a closer look at the Linux memory mapping implementation, however, there are several important observations to be made: First, after having once loaded a page, it will stay in memory at least for a limited amount of time. Second, pages in memory will *not* be paged out as long as there is some unused memory available. Thus, as long as we know that there is some memory left, we can mark once-accessed pages, and can be (almost) sure that the respective page will still be in memory later on.

Obviously, this only works as long as we do not try to page in more data than fits into physical memory. Fortunately, this can be easily guaranteed: By using the Linux `madvise()` call, we can *force* the kernel to free pages of our choice, thereby guaranteeing that some free memory is available at any time, and that no pages become unavailable without us knowing it. Of course, this assumes that no other processes start using up our memory.

4.2. The Tile Table

In order to mark pages as either available or missing, we have to store at least one bit per page. Keeping an entry for each potential 4 KB page in a 64-bit address space would require 2^{52} entries and is not affordable. Instead, one could use a hierarchical scheme as used by the processor's MMU, which however would be quite costly to access. We therefore group several pages into one "tile" and keep our tiles organized in a hash table of tile addresses. If the hash table is large enough to minimize hash collisions, hashing is quite efficient, and can be implemented with a few bit operations on the address pointer. Furthermore, a hash table is quite memory efficient: For hashing 128 GB RAM of 4 KB sized tiles (one page per tile) we only need 32M entries. Using a larger cache granularity of 16 KB or 64 KB, this reduces even more to 8M and 2M entries, respectively. If the size of the tile table is a power of two, all addressing and hashing operation can be performed efficiently by simple binary `ands` and `shifts`.

Each tile table entry contains a 64-bit pointer with the virtual base address of the tile for detecting hashing collisions. The lower 12–16 bits of this entry are always zero, and can thus be used for other purposes, i.e. for marking whether the page is available (bit 0), and whether it has recently been referenced (bit 1). Thus, in order to check if a page is in memory, we simply have to find its entry in the tile table (one `shift` operation), validate there is no hash collision (one `and`), check bit 0 for availability (one more `and`), and, if required, set bit 1 (one `or`) to mark an access.

4.3. Tile Fetching

In case we found a tile that is not marked as available, we cancel the respective ray and schedule the tile's address for asynchronous loading by putting it into a request queue.

Once a tile is scheduled to be fetched, it will eventually be loaded by an asynchronous *fetcher thread*. In an infinite loop, this thread in each iteration takes one request from the request queue, reads in the page via `madvise()`, and then marks the tile as available. Though reading the page obviously stalls the fetcher thread, the ray tracing threads are not affected at all, and remain busy. Note that we run several (4–8) fetcher threads in parallel, thereby allowing the OS to schedule multiple parallel disk requests as it deems appropriate.

Fetch Prioritization. Missing data leads to cancellation of rays, so missing data that cancels many rays should be

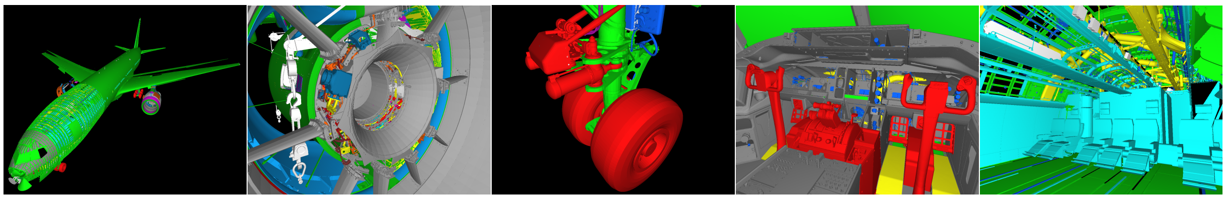


Figure 4: *Reference views for our experiments. From left to right: Overview over the whole model, a view into the engine, zoom onto the front wheels, the cockpit, and one of the main cabins. Using a single dual-CPU AMD Opteron 1.8 GHz PC, these respective views can be rendered at 4.1, 2.9, 7.1, 3.1, and 3.2 frames per second at video resolution (*640×480*).*

fetched faster than data affecting only a single ray. Counting the actual accesses to a tile, however, is too costly, as it would require to coordinate the different write accesses to the shared counter. Instead, we observe that the number of affected rays is proportional to the size of the BSP voxel they are about to enter, and inversely proportional to its distance to the camera. We use this value for prioritizing fetch requests. To avoid searching for the most important requests, we map the priority to 8 discrete values, and keep one request queue for each of them. The fetchers then always take the first entry out of the queue with highest priority. This mapping is performed linearly, relative to the minimum and maximum priorities of the previous frame.

4.4. Tile Eviction

As mentioned before, the tile fetcher can only fetch new tiles if some unused memory is available. Otherwise, the OS pages out tiles without us even noticing it (i.e. they are still being marked available). We therefore use the `madvise()` function to discard mapped pages from main memory. This obviously should be done only for pages that are likely not needed any longer. As a full "least recently used" strategy would be too expensive, we follow the same strategy as the Linux kernel swapper, and use a "second chance" strategy. The tile evictor slowly but continuously cycles through the tile table and resets the tile's "referenced" bit to zero (the page is still marked as present!). If the tile is still needed, this bit will soon be re-set by a rendering thread. If, however, the evictor visits a tile a second time with the R-bit still zero, it evicts the tile and marks it as missing. Similar to the Linux kernel swapper, tile eviction only starts once memory gets scarce, currently at a memory utilization of $\sim$80%.

4.5. Minimizing MM Overhead

While the just described memory management is an integral part of our system, we have to keep its performance impact to an absolute minimum. In particular, we have to minimize the number of semaphore synchronization operations, which otherwise tend to block the rendering threads.

Apart from the time consumed by the asynchronous fetcher and evictor threads, the main ray tracing threads have to constantly check each memory access for availability of the data. To minimize this overhead, we first check each pointer dereference for whether it crosses a tile boundary (with respect to the previous access). This can be done quite efficiently by simple bit operations, already reduces most of the tile table lookups, and does not require any costly semaphore operations.

Even in the case that we have to access the tile table, we can often get away without having to perform locking operations: If the tile is marked as available, or is marked as already being fetched, we can immediately return. Though this can result in a race condition – e.g. the evictor might evict the tile at exactly this moment – this event is extremely improbable. Even *if* it occurs, in the worst case it can lead to either a single, improbable page fault, or to scheduling a tile twice for being loaded. Both cases are well tolerable even in the rare event that they occur.

As such, there are only two cases where a ray tracing thread has to use a mutex. Once it adds a previously unvisited tile to the tile table, and every time it has to add a tile to the request queue. Both cases happen but relatively rarely. We also have to lock a mutex every time the tile fetcher or tile evictor want to modify the tile table or request queues. These threads, however, are not performance critical.

5. Geometry Proxies

Using our MM scheme, we can efficiently detect and avoid *any* page fault of the ray tracing threads, and thus maintain interactivity and high performance at all times. Unfortunately cache misses are detected but shortly before the data is actually required. Thus, the ray that caused this page fault obviously cannot be traversed any further.

As already discussed in Section 2, only suspending that ray until the data has been fetched will not work for a model of the 777's complexity, as we simply cannot load thousands of tiles within a single frame. Hence, we have no other way but to accept the fact that there eventually *will* be pixels in a frame for which we cannot completely trace the necessary ray(s). Therefore, we have to decide on what color to assign to such pixels. Obviously, coloring these pixels in a fixed color (like red in Figure 5) results in large parts of the image being unrecognizable.

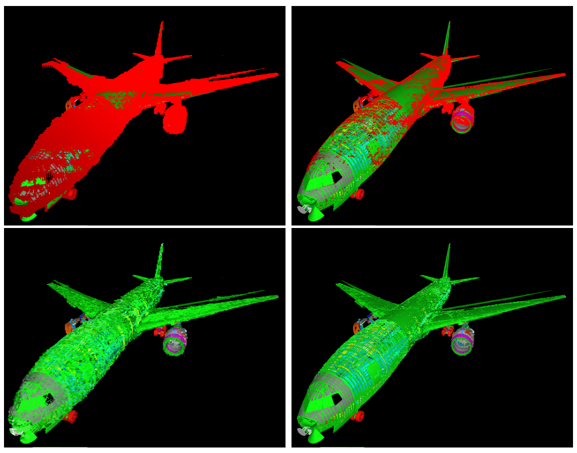


Figure 5: *Approximation quality during startup time. Left: Immediately after startup. Right: after loading for a few seconds. Even then only a fraction of the model has been loaded. Top row: Without proxy information, by just marking canceled rays in red. Bottom row: Using our geometry proxies. While the proxy quality after startup is quite coarse, it suffices to navigate the model. As can be seen, without the proxies almost no pixel contains sensible information. Eventually all data will be loaded, with no artifacts left at all. Also note that the positive influence of the proxies can hardly be shown in a still image, and becomes fully apparent only while interactively navigating the model.*

Alternatively one could fill in such a pixels color from the nearest valid sample, interpolate its color from several surrounding pixels, or even do sophisticated sparse sample reconstruction as done in e.g. the Render Cache [WDP99]. This approach however is quite problematic too: First, it requires costly (and badly parallelizable) post-filtering of the rendered image, which is too costly for full-screen resolutions. More importantly however, even a slight change of camera position can result in large fractions of the image becoming invalid (see Figure 5): Though most of the required nodes in the upper BSP levels will be in memory, many of the subpixel-sized leaf voxels will not yet be available, and will result in killing off many pixels, even after the rays could be traced "almost" up to he final hitpoint.

5.1. Proxies for Missing Data

For a cache miss however there are several important observation to be made: First, a cache miss can only be caused by a ray that wishes to traverse a specific *subtree* of the BSP that is not yet in memory. Such a subtree – no matter how many nodes or triangles it contains – is always a volume enclosed in an axis-aligned box. Furthermore, walkthrough applications tend to not change the view drastically, and similar views will touch similar data, particularly in the upper levels of the BSP tree. As such, going from one view to the next most of the upper-level BSP nodes will already be in memory, and only small subtrees close to the leaves are likely to be missing. These subtrees fortunately are quite small, and, when projected, often smaller than a pixel. For such small voxels it often does not matter which triangle exactly is hit by the ray, as long as there is some kind of "proxy" that mimics the subtrees appearance. As a result, we have chosen to compute such a proxy for each potentially missing subtree.

Note that this scheme is inherently hierarchical, as each proxy represents a subtree that in turn contains other subtrees and proxies. Moreover, this hierarchical approximation is tightly coupled to the BSP tree, and thus adapts well to the geometry.

Number of Proxies. Before discussing how exactly we are going to represent our proxies, we first have to evaluate how many of them we actually need (in order to estimate the amount of memory we can spend on them), and how to efficiently find the proxies. As we want to use our proxies for hiding the visual impact of a cache miss, we obviously need a proxy for each potentially occurring cache miss. As already discussed above, cache misses can only happen when following pointers from a parent node to its children that are located in a different tile. Instead of building a proxy for each child, we only build a proxy for the parent node.

More importantly, we change our BSP memory organization such that the number of pointers across tiles is minimized: Instead of storing BSP nodes in depth-first order [Wal04], we now use a scheme where we always fill cache-tile sized memory regions in breadth-first order [Hav99], thereby combining nodes forming small subtrees in the same tile. Apart from having fewer tile-crossing pointers, this has the positive and visually notable side effect that the proxy distribution is more symmetric: In depth-first order, the parents tile is usually filled up with nodes of the left child's subtree, almost always yielding a potentially faulting pointer for the right son. This insymmetry results is visually displeasing images while not all data is loaded.

granularity	16KB		64KB	
BSP	number	memory	number	memory
deep	15.6M	1.2GB	**4.3M**	**344MB**
shallow	**833K**	**66MB**	383K	30MB

Table 1: *Number of proxies with respect to cache granularity, and for two different BSP tree parameters. The deeper BSP generates somewhat faster performance (see Table 3), but requires more memory and many more proxies. For our experiments, we typically use the shallow BSP with 16 KB tiles, resulting in less than 70 MB of proxy memory. However, using only 80 bytes per proxy (see below), even the deep BSPs are affordable when using 64 KB tiles, using 344 MB out of 6 GB RAM. Note that the deep BSPs are a worst-case configuration.*

Obviously, the exact parameters with which the BSP was built influences the number of proxies. Deeper BSPs tend to achieve higher performance (see Table 3), but unfortunately also have more potentially page-faulting subtrees (see Table 1). Note that we can also influence the number of proxies by adjusting the caching granularity, as we can also perform our caching on e.g. 16 KB or 64 KB tiles. A larger cache granularity results in less tiles, in less pointers crossing tile borders, and thus in less proxies (see Table 1). Due to their lower memory consumption, by default, we use the shallow BSPs with a cache granularity of 16 KB, resulting in roughly 1.1 million proxies for the 777 model.

5.2. Hybrid Volumetric/Lightfield-like Proxies

As proxies, we have chosen a lightfield-like approach: As just argued, each proxy represents a volumetric subpart of the model, that will be viewed only from the outside, but from different directions. Thus, we only need to generate some meaningful shading information for each potentially incoming ray. This representation of discretized rays in fact is similar to a lightfield [LH96], except that we do not store readily shaded illumination samples in our proxy, but rather pre-filtered shading information. In particular, we store the averaged material information (currently only a single diffuse 5+6+5 bit RGB value) and the averaged normal (discretized into 16 bits). As mentioned above such a proxy will usually be subpixel-sized, we ignore the spatial distribution of the incoming ray on the proxy's surface, and rather only consider its direction. To this end, we triangulate the sphere of potentially incoming directions around the proxy, and precompute average normal and material value for each vertex of this discretized sphere of directions.

In case a canceled ray must use such a proxy, we then simply find the three nearest discretized directions with respect to the rays direction (i.e. the spherical triangle that contains this direction), compute the ray direction's barycentric coordinates with respect to its neighboring directions, and then interpolate the shading information from the data stored at these neighboring directions.

As discretized directions, we currently use the triangulation given by once subdividing the Octahedron given by the +X,-X,+Y,-Y,+Z, and -Z axes, which results in 18 discretized directions: 6 directions along the major axes, and 12 directions halfway in-between two adjoining axes (i.e. (1,1,0),(1,0,1),...). This discretization has been chosen very carefully, as it allows for finding the three nearest directions quite efficiently: The direction's three signs specify the octant of the spheres which has only 4 triangles. The coordinate with the maximum value then fixes the main axis, and leaves but two potential triangles, the one adjoining the axis, and the center triangle of the octant. By computing the four dot products between the ray and these triangles' four vertices, the nearest three vertices – and their barycentric coordinates – can be easily and efficiently determined.

The main problem with this approach is that averaging the normal tends to result in a normal that points more into the direction of the viewer than each individual normal. For example, looking symmetrically onto the edge of a box shows two sides facing the viewer in a 45-degree angle, but averaging the normals results in the averaged normal pointing *towards* the viewer. This effect leads to over-estimation of the cosine between normal and viewing direction and thus in overly bright proxies.

By only considering directional information, a proxy will for each individual direction look like a simple, colored box. This obviously leads to artifacts if a proxy covers many pixels. As these proxies are fetched with higher priority such large blocks however appear rarely, and disappear quickly. For proxies of small projected size our representation is sufficient and very compact. Alternatively, one could use a method in which this purely directional scheme is only used for small proxies, and proxies higher up in the BSP also get some positional information. So far however this scheme was not deemed necessary, and thus has not been implemented.

5.3. Discretization, Generation, and Reconstruction

Though we have just argued that the actual hitpoint is not important as long as we have a solid approximation, it is important to note that occlusion *has* to be taken into account. Most proxies contain a significant number of triangles, potentially with different materials and orientation. It often happens that a proxy contains e.g. lots of yellow cables being hidden behind a green metal part. In that case, just randomly picking a triangle is not appropriate, as it would lead to the proxy getting yellowish.

We therefore compute the directional information by sampling the proxy with ray tracing. Rays are traced from the outside onto the object, and only triangles actually visible from that respective direction will contribute to the proxy's appearance in that direction. Each proxy is sampled by a certain number ($\sim$10K) of random rays, whose hit information is stored within the proxy. To increase uniformity of the rays, we use Halton sequences [Nie92] for generating the rays.

5.4. Memory Overhead and Reconstruction Quality

For obvious reasons, we can spend only a small amount of memory for our proxies: We want to use the proxies to hide page faults, and thus currently need *all* proxies in physical memory. Otherwise, we would only replace paging for BSP nodes by paging for proxies. On the other hand, we still need a significant portion of main memory for our geometry cache, and cannot "waste" it on proxies.

As mentioned above, we use a discretization of 18 directions for each proxy. At two 16-bit values per direction, each proxy consumes 72 bytes, plus a float for specifying its surface area (for prioritized loading), plus a 32-bit unique ID specifying to which BSP subtree it belongs. In total, this requires a mere 80 bytes per proxy, or 66–344 megabytes for our two example configurations (shallow 16 KB, deep 64 KB).

Addressing of Proxies. In case of a cache miss, we have to efficiently find the corresponding proxy. As we can't add any pointer to the respective BSP node – at least not without changing the entire BSP structure of the ray tracer – we simply use the parents address as a unique identifier for the proxy, and use this address to index into an STL-"map" to find the proxy. Thus, we can implement our MM scheme without interfering *at all* with OpenRT's internal data structures, and can use the same data structures and algorithms as without our memory management unit (MMU).

Note that we only build proxies only for BSP subtrees, and not for faulting triangles or shading data. For such page faults, we simply use the last proxy encountered during traversal, which represents the subtree that this faulting triangle is located in.

6. Results

Once all the individual parts of our system are now together, we can start evaluating its performance. As target hardware platform, we have chosen a dual-processor 1.8 GHz AMD Opteron 246 system with 6 GB RAM, running SuSE 9.0 Linux with kernel 2.4.25. Though the machine is equipped with an NVIDIA graphics card, this card is only used for displaying the final image, and otherwise remains unused. For storing the model, the system uses a standard Western Digital WD2500JB IDE disk with a nominal throughput of roughly 50–55 MB/s. All of these parts are commodity PC parts, and the whole system costs is less than $5000.

6.1. Preprocessing

As mentioned before, all preprocessing – i.e. model splitting, BSP construction, and proxy computation – is performed out of core. For this preprocessing, by default we stop subdividing at 2–4 million triangles per voxels. At this size the individual blocks conveniently fit into memory for BSP construction. BSPs that are built in core can be built with advanced BSP generation schemes using cost prediction functions [Hav01, Wal04], which results in higher rendering performance than for the typical "split-in-the-middle" strategy adopted while splitting the model.

Depending on the actual BSP parameters, we need around 30–40 GB on disk for the preprocessed model. Preprocessing – including unpacking, conversion, splitting, BSP generation, and proxy computation – takes in the order of one day on a single PC, depending on the actual parameter values. Most of this time however is spent in BSP generation and proxy computation, which can be trivially parallelized by simply having N machines working on every Nth voxel each. This allows for performing the entire preprocessing in less than a night. For example, in the course of writing this paper we have performed this preprocessing several times a day in order to experiment with different parameters.

6.2. Proxy Memory Overhead and Cache Configuration

From these experiments, we have picked two different configurations that represent a range of typical values. For one setting, we have chosen high-quality BSP trees of up to 60 levels of depth for each voxel generated during out-of-core preprocessing. This obviously results in many BSP nodes, roughly 40 GB on disk, and many proxies (see Table 1). In the other experiments, we have used rather shallow BSP trees, which use only 30 GB of disk space, and much less proxies.

As mentioned before, we use a cache granularity of 64 KB for the deep BSPs, and 16 KB for the shallow BSPs, resulting in 66 MB and 344 MB for the proxies, respectively. With 6 GB of physical RAM, we can configure our cache size at 4–5 GB, with plenty of RAM left for the OS and for OpenRT runtime data. At this cache size, large parts of the model fit into the cache. In particular, each individual view fits into cache, and the proxies only have to bridge the loading latencies when changing views.

6.3. Demand Loading Time and Approximation Quality

After a complete restart of the entire system, our framework starts by parsing the list of voxel files, creates the yet-empty containers for the voxels, and builds a "top-level" BSP for these voxels. All this takes at most a few seconds. It then reads in all the proxies, which takes several seconds to a few minutes, depending on the actual proxies' data size. Once all proxies are read, the ray tracer immediately starts shooting rays, and uses the proxies while the data is being fetched asynchronously.

Depending on how much data is required for loading the working set of a frame, it can take in the order of up to several minutes until all data is loaded. Some views require up to more than a gigabyte of data, which simply cannot be loaded from disk within a few seconds. The memory footprint and loading time for our reference views (see Figure 4) are given in Table 2. As a worst-case example, we have taken the "overview" viewpoint, in which the entire airplane is seen from a viewpoint with minimal occlusion and in which the rays travel deeply into all parts of the model . This view requires more than 2 *giga*bytes of data, and can take minutes to page in.

While the approximation is rather coarse immediately after startup (see Figure 5), the structure of the model is already well recognizable after having loaded only a few percent of the data. Though this quality is far from perfect, it is totally sufficient for navigating through the model while it is being loaded and refined simultaneously. Additionally, when zooming onto a specific part the data is usually fetched quite fast, and shows the part in full detail after only a few frames.

Once a significant portion of the model has been loaded, most of the geometry needed for rendering is already present in the cache. In particular, most of the upper levels of the BSP are already in the cache, and potential cache misses will typically affect only a few pixels. In that case, the proxies can do a good job at masking these isolated missing pixels. As our proxies were never designed to provide a high-quality hierarchical approximation, they fulfill their planned task of providing solid feedback for interactive navigation.

BSP/View	overview	engine	wheels	cockpit	cabin
deep	2,300	145	40	122	254
shallow	2,150	215	36	105	236

Table 2: *Memory footprint (in MB) for our reference views. As expected, some views require up to hundreds of megabytes. Particularly the intentionally chosen worst-case "overview" requires more than 2 GB, which can take minutes to load completely. Note however that we do not have to wait until all data is loaded, but can already navigate the proxy-approximation from the very first second.*

6.4. Performance Overhead

Obviously, our demand loading scheme will not come for free. Through aggressively minimizing the tile table lookups and mutex locks (see Section 4.5), we have reduced the overhead of our MM scheme to the bare minimum. Even so, a certain overhead remains. In particular, testing if a pointer crosses a tile boundary – though it costs only a few bit tests – has to be performed for *each* memory access even in the ray tracers inner traversal loop, and thus affects performance. Tile table access is less common, but unfortunately more costly, and thus affects performance, too.

To determine the total overhead of our system, we have measured the frame rate for our reference views (see Figure 4), once using the "standard" OpenRT ray tracer without our MM scheme, and once with the MMU turned on. This experiment can be performed only for static views, as even small camera movements lead to long paging stalls if the MMU is turned off. To enable a fair comparison, for the MMU version we have measured the frame rate after all tiles have been paged in. This in fact is the worst-case scenario as rays have to be traversed quite deeply, and have to perform many checks and locks. Once some pixels get killed off – i.e. during startup or when accessing previously invisible parts of the scene – frame rate is rather higher than lower.

As can be seen from Table 3, the total overhead for our example views consistently is in the range of 25% for the shallow BSPs, and 20% for the deep BSPs, respectively. Note that this includes the *entire* overhead, including pointer checking, tile lookups, threading, tile fetching, mutexes, etc. As our MM scheme enables us to navigate fluently without *any* paging stalls, we believe this overhead to be quite tolerable.

BSP/View	overview	engine	wheels	cockpit	cabin
shallow BSPs					
w/o MMU	2.7	2.4	5.3	2.0	2.1
w/ MMU	1.9	1.8	4.0	1.5	1.6
overhead	26%	25%	24%	25%	23%
deep BSPs					
w/o MMU	4.9	3.6	9.0	4.0	4.0
w/ MMU	4.1	2.9	7.1	3.1	3.2
overhead	16%	19%	21%	22%	20%

Table 3: *Total overhead of our memory management scheme for different views (see Figure 4), and for BSPs built with different cost parameters, measured in frames per second. As can be seen, total overhead is in the range of 25% for the shallow BSPs, and only 20% for the higher-performing deeper BSPs.*

6.5. Overall System Performance

With this small performance overhead, the ray tracer is quite efficient at rendering the model. As can be seen from Table 4, using a single dual-Opteron PC we achieve interactive frame rates of 3–7 fps at video resolution of 640 × 480 pixels. Even at full-screen resolution of 1280 × 1024, we still maintain frame rates of 1–2 fps. Such high resolutions are particularly important for getting a feeling of the relative orientation of the highly detailed geometry. While the just mentioned frame rates do not include antialiasing, supersampling can still be added progressively as soon as the camera stops moving.

Also note that this performance data again corresponds to all data being present in the geometry cache. Upon cache misses and use of proxies, frame rates are even *higher*, as the rays perform less traversal steps. Thus, we can maintain these interactive frame rates at all times while navigating freely in and around the model.

Resolution	overview	engine	wheels	cockpit	cabin
shallow BSPs					
640x480	1.9	1.8	4.0	1.5	1.6
1280x1024	0.7	0.6	1.3	0.5	0.5
deep BSPs					
640x480	4.1	2.9	7.1	3.1	3.2
1280x1024	1.3	0.9	2.3	1.0	1.0

Table 4: *System performance in frames per second on a single dual-1.8 GHz Opteron, using our two configurations.*

6.6. Shading and Shadows

In the course of the paper, we have only concentrated on the memory management scheme and proxy mechanism, and so far have neglected secondary rays and shading at all. Of course, using a ray tracer allows for also computing shadows and reflections. Without sensible material data (which is not included in the 777 model), and in particular with the randomly jittered vertex positions (and therefore randomly jittered normals) however computing reflections simply does not make much sense.

Shadow effects can be added quite easily. While the performance and caching impact of shadows so far have shown to be surprisingly low, an exact discussion of these effects is beyond the scope of this paper. Nonetheless, adding shadows in practice is relatively simple. For example, Figure 6 shows some images that have been computed with shadows from a flashlight-like light source that is attached relative to the viewer.

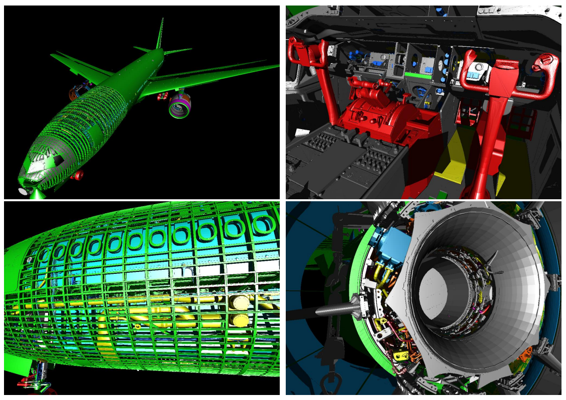


Figure 6: *Using a ray tracer, adding shadows to the model is (almost) trivial. As expected, shadows significantly improve the impression of shape and depth (compare to Figure 4). This is particularly the case during interaction.*

As expected, shadows add an important visual cue to the image, and significantly improve the impression of shape and depth, which can best be seen by comparing Figures 4 and 6. This improved sense of depth is particularly apparent once the shadows move with the light during interaction.

7. Conclusions

In this paper, we have presented a framework that allows for interactively ray tracing gigabyte-sized models consisting of hundreds of millions of individual triangles on a single desktop PC. This is achieved using a combination of real-time ray tracing, an out-of-core demand-loading and memory management scheme for handling the massive amounts of geometry, and a hybrid volumetric/lightfield-like approximation scheme for representing not-yet-loaded geometry.

By detecting and canceling potentially page-faulting rays, we can avoid system paging, and maintain high frame rates of several frames a second, even while flying into or around our example airplane model.

The visual impact of killing off rays is reduced by approximating the missing geometry using a lightfield-like approach. For not too drastic camera changes, the proxies can well hide the visual artifacts otherwise caused by the canceled rays. For large camera movements however, or when entering a previously occluded part of the model, the proxy structure becomes visible in form of blocky artifacts in the image. These artifacts then are similar to other approaches like Holodeck, Render Cache, point-based methods, or even MPEG/JPEG-compression. Using the surface-based loading prioritization however these artifacts disappear quite quickly. Furthermore, the quality is still sufficient for interactively navigating the model.

Using our approach, we achieve frame rates of 3–7 frames per second at 640×480 pixels, or still 1–2 frames per second at full-screen resolution of 1280×1024 pixels, even on a single dual-CPU desktop PC.

7.1. Future Work

As next steps, we will investigate ways to further improve the visual appearance of our proxies, potentially by including positional information at least for large voxels.

More importantly, we are planning to make this technology available to industrial end-users, which means that we have to target real-time frame rates at full-screen resolutions. Eventually, this will require using more than only two CPUs. Fortunately, quad-CPU systems are already available, and eight-way systems have been announced. Additionally, it seems interesting to parallelize and distribute the current system over a cluster of cheap dual-CPU PCs. Preliminary result already look promising.

Once the computational power is available, we also plan on evaluating how high-quality shadows, reflections, and in particular interactive lighting simulation can be achieved in models of this complexity.

Acknowledgements

We would like to express our thanks to Boeing Corp. for graciously providing this model, and to our system administration team for getting the prototype Opteron to run so quickly. Source 3D data provided by and used with permission of the Boeing Company.

References

[ACW 99] ALIAGA D. G., COHEN J., WILSON A., BAKER E., ZHANG H., ERIKSON C., HOFF III K. E., HUDSON T., STÜRZLINGER W., BASTOS R., WHITTON M. C., BROOKS JR. F. P., MANOCHA D.: MMR: An Interactive Massive Model Rendering System using Geometric and Image-Based Acceleration. In *ACM Symposium on Interactive 3D Graphics* (1999), pp. 199 206.

[AMD03] AMD: AMD Opteron Processor Model 8 Data Sheet. http://www.amd.com/us-en/Processors, 2003.

[BC02] BOVET D. P., CESATI M.: *Understanding the Linux Kernel (2nd Edition)*. O'Reilly & Associates, 2002. ISBN 0-59600-213-0.

[BMH99] BARTZ D., MEISSNER M., HÜTTNER T.: OpenGL assisted Occlusion Culling for Large Polygonal Models. *Computer and Graphics 23*, 3 (1999), 667 679.

[BSGM02] BAXTER III W. V., SUD A., GOVINDARAJU N. K., MANOCHA D.: Gigawalk: Interactive Walkthrough of Complex Environments. In *Rendering Techniques 2002 (Proceedings of the 13th Eurographics Workshop on Rendering)* (2002), pp. 203 214.

[BWG03] BALA K., WALTER B., GREENBERG D.: Combining Edges and Points for Interactive High-Quality Rendering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)* (2003), 631 640.

[CH02] COCONU L., HEGE H.-C.: Hardware-Accelerated Point-Based Rendering of Complex Scenes. In *Proceedings of the 13th Eurographics Workshop on Rendering* (2002), Eurographics Association, pp. 43 52.

[CKS03] CORREA W., KOSLOWSKI J. T., SILVA C.: Visibility-Based Prefetching for Interactive Out-Of-Core Rendering. In *Proceedings of Parallel Graphics and Visualization (PGV)* (2003), pp. 1 8.

[CMS98] CIGNIONI P., MONTANI C., SCOPIGNIO R.: A Comparison of Mesh Simpli cation Algorithms. *Computers and Graphics 22*, 1 (1998), 37 54.

[DGP04] DEMARLE D. E., GRIBBLE C., PARKER S.: Memory-Savvy Distributed Interactive Ray Tracing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2004). To appear.

[DPH 03] DEMARLE D. E., PARKER S., HARTNER M., GRIBBLE C., HANSEN C.: Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)* (2003), pp. 87 94.

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-Buffer Visibility. In *Computer Graphics (Proceedings of ACM SIGGRAPH)* (August 1993), pp. 231 238.

[GLY 03] GOVINDARAJU N. K., LLOYD B., YOON S.-E., SUD A., MANOCHA D.: Interactive Shadow Generation in Complex Environments. *ACM Transaction on Graphics (Proceedings of ACM SIGGRAPH) 22*, 3 (2003), 501 510.

[Hav99] HAVRAN V.: Analysis of Cache Sensitive Representation for Binary Space Partitioning Trees. *Informatica 23*, 3 (May 1999), 203 210. ISSN: 0350-5596.

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[Int02] INTEL CORP.: Intel Pentium III Streaming SIMD Extensions. http://developer.intel.com, 2002.

[LH96] LEVOY M., HANRAHAN P.: Light eld rendering. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (ACM SIGGRAPH)* (1996), pp. 31 42.

[Nie92] NIEDERREITER H.: *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, 1992.

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Computer Graphics 31*, Annual Conference Series (Aug. 1997), 101 108.

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: Surface elements as rendering primitives. In *Proc. of ACM SIGGRAPH* (2000), pp. 335 342.

[RL00] RUSINKIEWICZ S., LEVOY M.: QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proc. of ACM SIGGRAPH* (2000), pp. 343 352.

[SDB97] SILLION F., DRETTAKIS G., BEDELET B.: Ef cient Imposter manipulation for Real-Time Visualization of Urban Scenery. *Computer Graphics Forum, 16*, 3 (1997), 207 218. (Proceeding of Eurographics).

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[WDP99] WALTER B., DRETTAKIS G., PARKER S.: Interactive Rendering using the Render Cache. In *Rendering Techniques 1999 (Proceedings of Eurographics Workshop on Rendering)* (1999).

[WFP 01] WAND M., FISCHER M., PETER I., AUF DER HEIDE F. M., STRASSER W.: The Randomized z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In *Proc of ACM SIGGRAPH* (2001), pp. 361 370.

[WS99] WARD G., SIMMONS M.: The Holodeck Ray Cache: An Interactive Rendering System for Global Illumination in Nondiffuse Environments. *ACM Transactions on Graphics 18*, 4 (Oct. 1999), 361 398.

[WSB01] WALD I., SLUSALLEK P., BENTHIN C.: Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques* (2001), pp. 274 285. (Proceedings of Eurographics Workshop on Rendering).

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153 164. (Proceedings of Eurographics).

[WWS00] WONKA P., WIMMER M., SCHMALSTIEG D.: Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. In *Rendering Techniques* (2000), pp. 71 82. (Proceedings of Eurographics Workshop on Rendering).

[ZMHH97] ZHANG H., MANOCHA D., HUDSON T., HOFF III K. E.: Visibility Culling using Hierarchical Occlusion Maps. *Computer Graphics 31*, Annual Conference Series (1997), 77 88.

University of Texas at Austin Dept. of Computer Sciences Technical Report #TR-05-18, May 2, 2005

Real-Time Rendering Systems in 2010

William R. Mark [*] Donald Fussell [†]

Department of Computer Sciences
The University of Texas at Austin

Abstract

We present a case for future real-time rendering systems that support non-physically-correct global illumination techniques by using ray tracing visibility algorithms, by integrating scene management with rendering, and by executing on general-purpose single-chip parallel hardware (CMP's). We explain why this system design is desireable and why it is feasible. We also discuss some of the research questions that must be addressed before such a system can become practical.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture— [I.3.2]: Computer Graphics—Graphics Systems

1 Introduction

For many years, real-time graphics systems have used the traditional Z-buffer pipeline model, which is limited to local illumination computations. With appropriate modifications, this pipeline can support some restrictive global illumination techniques, but doing so is awkward and often inefficient. A different strategy is possible – VLSI technology has now progressed to the point where we are on the verge of having sufficient raw computational capability to use more general global illumination techniques. But there is no consensus yet about how future graphics systems supporting global illumination should be organized.

If we look a few years into the future, several major questions become evident: What rendering algorithms are most appropriate for this new era? What architectures should we build to support these algorithms? And what overall system organization should tie together the application, rendering algorithms, and hardware? We believe that these questions have not yet been answered satisfactorily.

The purpose of this paper is to argue that these questions are closely coupled and that addressing them will require simultaneous investigation of software algorithms and hardware architectures. We also propose a set of algorithmic and architectural approaches that we believe present one promising avenue of investigation. Our hope is that this paper will stimulate discussion in the research community and help to inspire the combined software and hardware research that we believe is critical to forward progress.

The application-level goal that drives our investigation is support for real-time global illumination for dynamic scenes. We place greater emphasis on non-physically-correct global illumination techniques than on fully physically-based techniques, since non-physically-correct techniques represent an intermediate step between today's local illumination models and eventual use of 100% physically-based techniques.

Most global illumination techniques require a more general visibility-computation capability than that provided by today's Z buffer. We present an algorithmic approach organized around ray tracing visibility algorithms that efficiently supports dynamic scenes by integrating scene management with rendering. But this tighter integration requires that the graphics hardware directly support model management as well as rendering.

At the hardware level, we advocate a very flexible architecture: a multi-core, multi-threaded, MIMD architecture with coherent access to a single address space. This architecture efficiently supports application-specific scene management code as well as the creation and traversal of dynamic, irregular data structures.

1.1 Background

The Z-buffer 3D graphics pipeline has been widely used for more than 20 years. As VLSI technology has advanced, this system organization has progressed down the cost curve from multimillion dollar flight simulators, through high-end graphics workstations made by companies such as SGI (e.g. [Akeley 1993]), down to single-chip GPUs made by companies such as NVIDIA and ATI.

For most of this history, Z-buffer graphics hardware was configurable but not programmable. However, over the past four years, we have seen the introduction of user-accessible programmability at both the vertex [Lindholm et al. 2001] and fragment [NVIDIA Corp. 2003] stages of the pipeline. The vertex programmability merely exposed a programmable engine that had already existed in various forms for many years, but the fragment programmability exposed fundamentally new hardware functionality. Its introduction was driven by the realization that beyond a certain point, the best way to use additional VLSI transistors to improve image quality is to increase the quality of each pixel rather than increasing the number of pixels or increasing the geometric detail.

Fragment programmability enabled commodity real-time systems [Mark et al. 2003] to support programmable shading capabilities inspired by those of Renderman [Hanrahan and Lawson 1990]. However, this programmability has proven to be sufficiently flexible that researchers have begun to think of graphics processors as general-purpose stream processors [Kapasi et al. 2002], capable of supporting a variety of non-shading computations [Purcell et al. 2002; Thompson et al. 2002; Bolz et al. 2003]. But at the current time, most of these other uses of the GPU are not yet fully practical. The reason is that the current GPU programming model has limitations that limit performance on general-purpose computations to much less than peak performance. We expect that this situation will change with time, but not as rapidly as many researchers are expecting.

Thus, the primary economic force driving GPU design is still real-time rendering, which leads us to the following question: What rendering requirements should drive the future evolution of graphics hardware? Another way of asking this question is, what additional capabilities could best be put to good use by applications? Of course, it only makes sense to consider capabilities that have the potential to be cost effective in the time frame of interest.

2 Application needs

We believe that there is still unmet application demand for higher-fidelity real-time imagery. For example, most observers would

[*] e-mail: billmark@cs.utexas.edu
[†] e-mail: fussell@cs.utexas.edu

agree that the images produced by batch-rendering systems are noticeably superior to those produced by interactive graphics systems, and that they would like to see these higher-quality images produced by real-time systems.

Some of the demand for improved image quality in real-time graphics can be met by adding support for object-space shading like that used in batch rendering systems such as REYES [Cook et al. 1987]. In particular, REYES provides better temporal and spatial anti-alising than the screen-space shading used in current real-time graphics systems. However, much of the current difference between batch rendering and real-time rendering results from the poor modeling of global illumination effects in real-time rendering systems as compared to batch rendering systems. We are already seeing demand for realistic global illumination with the current focus on the special case of real-time hard shadow generation. REYES and similar systems do not support global illumination computations in any general sense.

Some observers argue that REYES and similar algorithms can be used to fake a wide variety global illumination effects, as demonstrated by their use for over 10 years for movie rendering. However, interactive graphics applications are fundamentally different from batch movie rendering because the viewpoints and scene configurations are not known a-priori by the programmers and artists. Most of the techniques used to fake global illumination with REYES-like systems rely on viewpoint-dependent hand tuning and thus are not appropriate for use in real-time graphics.

2.1 Use ray tracing visibility

Almost all algorithms that model global illumination effects without the use of extensive hand-tuning rely on global visibility computations. Examples include radiosity, ray tracing [Whitted 1980], photon mapping [Wann Jensen 2001], approximation of far-field illumination using spherical basis functions, etc. Thus, we believe that robust support for global illumination requires support for global visibility computations, and specifically for ray tracing visibility.

Recent work shows that raw computational capability has now advanced to the point where it is reasonable to consider using ray tracing visibility in real-time graphics systems. Over the past several years, several groups have built near-real-time ray tracing systems with steadily improving price/performance ratios. Most of these systems run on standard CPUs (e.g. [Parker et al. 1999a; Parker et al. 1999b; Hurley et al. 2002; Wald et al. 2003b], but one runs on a specialized ray tracing architecture implemented with an FPGA [Schmittler et al. 2004], and another uses the fragment processors of mainstream GPUs [Purcell et al. 2002]. The system with the best price/performance ratio [Hurley 2005] runs on a desktop PC with frame rates over 30 frames/sec for eye+shadow rays on complex scenes. Its raw performance has been quoted at over 100M Ray segments/sec. A recent review article [Wald et al. 2003c] provides an excellent overview of recent developments in this area.

2.2 Use non-physically-correct global illumination

Experience has proven [Gritz and Hahn 1996; Kato 2002] that ray tracing algorithms and variants such as photon mapping provide the most robust and general solution to the global illumination problem. However, we do not expect 2010-era real-time game applications to rely primarily on *physically correct* global illumination. Instead, we expect that these applications will use the point-to-point visibility queries enabled by a ray tracing visibility framework to implement various non-physically correct approximations to global illumination. For example, we expect techniques such as ambient occlusion [Moyer 2004], instant radiosity [Keller 1997], and variations of precomputed radiance transfer [Sloan et al. 2002] to be used. For most of its history, computer graphics has relied heavily on phenomenological or quasi-physical approximations to illumination computation, and we do not expect that situation to change immediately. In fact, we expect that new phenomenological approximation techniques will be developed that leverage the capabilities of a ray tracing visibility engine.

2.3 Dynamic scenes are the challenge

Most interactive applications, particularly those in the economically important gaming market, use dynamic scenes. These scenes include geometrically complex objects that move, and, more significantly, deform. Unfortunately, there has been very little effort devoted to raytracing for dynamic scenes, and in particular for deformable objects.

Deformable objects such as the skinned characters [Lander 1998] used in QuakeIII and Doom present a significant challenge. The deformable nature of these characters is not well supported by any existing method for ray tracing. In particular, the simple approach of pre-building an acceleration data structure for the object and repositioning that object within the scene [Lext and Akenine-Moller 2001; Wald et al. 2003a] does not work for objects in which many polygons deform every frame. We believe that any practical real-time raytracing system must support moving and deformable objects with reasonable performance.

3 Integrate scene management with rendering

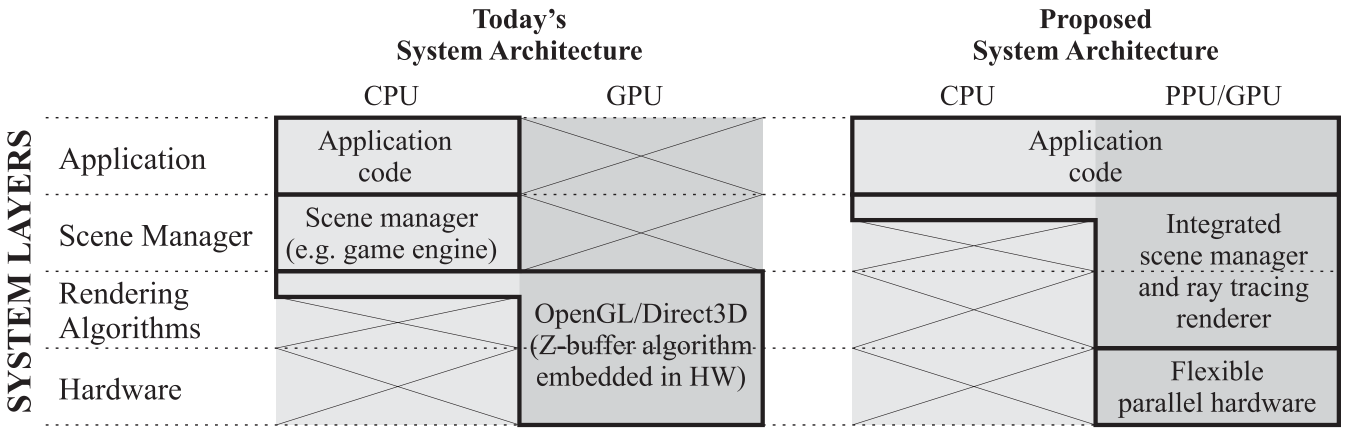
To ray trace dynamic scenes in real time we must reassess the crucial role of acceleration structures in making the ray tracing process efficient. The highest performance ray tracers use a space partitioning acceleration structure such as an octree or BSP tree, but the scene data is not originally stored in this form. Instead, the space partitioning data structure is constructed from data stored in a scene graph represented as a hierarchy of (potentially overlapping) bounding volumes.

A simple approach is to begin the computation of each frame by rebuilding an acceleration data structure of the type used in batch ray tracing. The problem with this approach is that the cost of rebuilding the acceleration structure may exceed the ray tracing cost itself. This problem is particularly serious if the scene has very high depth complexity, forcing the system to perform work for objects that are not hit by any rays. Even if we only rebuild those portions of the acceleration structure containing moving objects [Reinhard et al. 2000] the system may be performing much unnecessary work. For dynamic scenes it becomes apparent that minimizing the rebuild cost may be as important as minimizing the traversal cost, since the minimization of the total cost is the overriding criterion.

The most promising approach is to use lazy evaluation techniques to build the acceleration structure (building on and extending work by Ar *et al.* [Ar et al. 2002]). When a ray enters a previously untouched portion of the space partitioning data structure, the system puts the ray traversal on hold; then constructs that portion of the space partitioning data structure from the scene graph; and finally lets ray traversal resume through the newly created geometry.

However, this approach requires a close interaction between the acceleration data structure and the scene graph used to model the world at the application level. We believe that this recognition is the key to designing an effective system organization for real time dynamic ray tracing.

Consider a system in which scene management is tightly integrated with rendering (Figure 1). Such a system does not necessarily eliminate the need to store geometry using two different organizations – hierarchical scene graph and space partitioning – but such a system can tightly control which data is converted into the space partitioning form and when it is stored in this form. In particular,



Today's System Architecture
Proposed System Architecture
CPU
GPU
CPU
PPU/GPU
SYSTEM LAYERS
Application
Scene Manager
Rendering Algorithms
Hardware
Application code
Scene manager (e.g. game engine)
OpenGL/Direct3D (Z-buffer algorithm embedded in HW)
Application code
Integrated scene manager and ray tracing renderer
Flexible parallel hardware

Figure 1: We propose that scene management be tightly integrated with rendering and that both be executed on the flexible parallel hardware. We refer to this flexible parallel hardware as a PPU (parallel processing unit).

the system can insure that only visible or nearly-visible surfaces are stored in space partitioning form.

Requiring the rendering system to integrate scene management with rendering is a major change from today's systems, so it is reasonable to ask why it is possible to separate scene management from rendering in a Z-buffer system but not in a ray tracer. In a simple Z-buffer system, visibility computations are performed in object order, so that each polygon in the scene is touched once and only once each frame by the visibility algorithm. Thus, for the purpose of the visibility computation, there is no need to store more than one polygon in local memory at a time. Of course the geometry must be stored somewhere in the system, but this can be done by the application or scene graph in any manner that is desired, with the geometry streamed across the immediate mode interface to the Z-buffer system. Commonly, the geometry is stored in a hierarchical data structure for the application to animate and otherwise modify.

Typically, ray tracing algorithms are "ray order" algorithms, in which the basic visibility algorithm can touch one polygon, then touch a second polygon, and eventually return to the first polygon. This type of algorithm requires direct access to the geometric database describing the scene. However, the geometric database need not be stored in the same format as the scene graph that is manipulated by the application. By transferring data lazily between the two data structures, we can minimize the cost of maintaining two different data structures.

3.1 Additional improvements

If ray traversal is managed so that most rays touching a particular portion of space are processed simultaneously [Pharr et al. 1997], then the system has the option of treating geometry represented in space-partitioning form as disposable. That is, when a particular volume of space is visited by a batch of rays, first the system creates an acceleration structure in on-chip memory for the geometry residing in that volume of space, then performs ray/triangle intersection tests, then discards the acceleration structure. The acceleration structure for that volume of space can be recreated later from the scene graph if it happens to be needed again.

Several other optimizations become convenient in this framework. If the system stores scene graph data using higher-order representations such as subdivision surfaces, these representations may be tesselated into triangles as the system creates the spatial acceleration structure. The data explosion that occurs during this step can be confined to on-chip memory, just as it is for a Z-buffer pipeline that includes a tesselation processor. Pharr and Hanrahan describe a variant of approach for displacement surfaces [Pharr and Hanrahan 1996].

An additional advantage of tight integration of scene management with rendering is that the system can automatically adapt the LOD of geometry to local ray density, even instantiating the same geometry at two different levels of detail, as is often needed when different types of rays (eye and reflected, for example) intersect the same geometry. A recent paper from Pixar [Christensen et al. 2003] has clearly demonstrated the value of using ray differentials to manage geometric level of detail in a raytracer.

4 Is a unified system organization practical?

We recognize that proposing to tightly couple scene management with rendering flies in the face of conventional wisdom about graphics system design. Current systems, following the lead of Iris GL and OpenGL [Segal and Akeley 2002], are characterized by the separation of scene management from rendering, mediated by a carefully-designed immediate mode rendering interface (Figure 1).

Why do we have this interface? Because experience has shown that it is not possible to build an efficient, fully general-purpose scene manager. Attempts to standardize systems of this type, such as CORE [Graphics standards planning committee 1979] and PHIGS [(american national standards institute) 1988], failed largely because of their attempt to integrate support for modeling and rendering using an API framework.

So why do we think we can do better? Because experience has also shown that it *is* possible to build reusable scene managers specialized for particular application domains. The most prominent examples are Performer [Rohlf and Helman 1994] which is specialized for visual simulation and id software's widely licensed game engines, which are specialized for first-person-shooter games. However, these systems do not use a standard API framework – either the engine is either highly configurable through internal hooks (Performer) or it can be directly modified in source code form (id's game engines).

We conclude that it is probably not possible to build a fully generic scene engine behind an API, but that it *is* possible to build specialized engines that implement performance critical tasks and can be adapted for particular applications. Thus, if one is willing to allow a scene manager to be implemented in "user" code (i.e. not embedded in unprogrammable hardware, or behind a one-size-fits-all interface), then it is perfectly possible to build a high-performance scene manager. If this scene manager can run on the same hardware that supports the rendering, then we believe that the scene manager can include rendering code, and thus provide the integrated renderer / scene manager that we propose. This approach is analogous to the programmable shaders in today's hardware, but carried much farther.

5 Parallel architecture supporting late binding

To efficiently support the ray tracing system we have described, the hardware architecture and corresponding parallel programming model must be very flexible and allow most control and data binding decisions to be deferred until run time. For example, a highly-specialized architecture, a SIMD architecture, or a streaming architecture would not be appropriate for this workload.

Several factors drive this need for generality:

- **Application-dependent scene management:** The architecture cannot be designed for particular scene management code.
- **Irregular data structures:** The scene graph and acceleration data structures are irregular, requiring pointer-chasing or its equivalent.
- **Dynamic data structures:** The irregular data structures must be built and modified with high performance, as well as being traversed with high performance.
- **Data dependent control flow:** Adaptive tesselation, ray tracing, and other tasks use highly data-dependent control flow.
- **Data locality:** Many of the data structures exhibit temporal locality in their access patterns, but the exact form of the locality is not known at compile time due to the irregular nature of the data structures.

We take as a starting point that our target architecture provides explicit parallelism, which provides better power efficiency than a single mainstream high-ILP CPU [Sasanka et al. 2004].

5.1 MIMD control flow

We advocate a MIMD programming model because it supports data parallel execution of computation kernels that use data-dependent conditionals and looping. Support for MIMD control flow is critical for efficiently creating and traversing adaptive spatial data structures such as k-d trees, as well as for executing short data-dependent loops such as those used in vertex skinning and anisotropic texture filtering [Sankaralingam et al. 2003a]. MIMD computation also supports general task level parallelism, i.e. it allows multiple distinct "kernels" to run concurrently. A primary example of the need for this is to allow closely coupled scene graph management and rendering tasks to run concurrently, particularly when these tasks are not individually sufficiently parallelizable to be able to occupy the entire machine.

Current graphics hardware (e.g. NVIDIA 6800 with shader model 3.0) supports a more restrictive SPMD (single-program, multiple data) programming model in which MIMD-style control flow is supported, but all fragment or vertex processors must be running the same program. However, the hardware implementation of the control flow is closer to a SIMD implementation, so that code with divergent branching behavior is inefficient [Nvidia Corp. 2004].

Even if future architectures use a MIMD organization as we advocate, that does not preclude support for simpler programming models as well. Most other parallel programming models (e.g. various variants of "stream programming") can be described as restricted subsets of the one we have outlined and thus can be supported by the same hardware. For tasks that can tolerate these limitations, the restricted programming models are often easier to use and typically encourage the programmer to express the task in a form that will perform well. For example, the stream programming model forbids the code within one kernel from directly communicating with the code within another kernel, thereby eliminating the potential for many types of concurrency and performance bugs.

Recent industry designs seem to endorse our view that MIMD architectures are a better choice than SIMD architectures for general-purpose single-chip parallel computation. Sun's Niagara [Krewell 2003] and IBM/STI's CELL [Pham et al. 2005] are both fully MIMD. The most advanced graphics processors (e.g. GeForce 6800) currently have a MIMD programing model (actually SPMD) implemented as a MIMD execution model in the vertex processor and a SIMD execution model in the fragment processor. We expect future architectures to gradually move towards a MIMD implementation, although maintaining current fragment ordering semantics in a MIMD machine presents some challenges. Several interesting research architectures that use a highly-parallel MIMD organization are IBM's Cyclops [Caşcaval et al. 2002] (not yet built), Stanford's Smart Memories [Mai et al. 2000] (not yet built), MIT's RAW [Taylor et al. 2004] (already built), and the MIT M-Machine [Keckler et al. 1998] (already built) which demonstrated some promising approaches for supporting fine-grained MIMD parallelism.

Note that although all of the architectures mentioned above are MIMD in their overall organization, many of them support 4-wide SIMD instructions within each core. These short-vector SIMD instructions are an efficient mechanism for exploiting what is really just a particularly common form of instruction-level parallelism found in graphics and scientific code. Even in machines that are designed to exploit MIMD thread-level parallelism, it is still worthwhile to support such low-cost forms of instruction level parallelism, since exploiting such parallelism improves performance without requiring an increase in on-chip storage such as would be required by support for additional threads.

5.2 Hardware caches and global address space

For processors built using modern VLSI technology it is desirable to include a multi-level memory hierarchy on chip, since for workloads with temporal memory-access locality this strategy provides a favorable combination of low power consumption, low average memory-access latency, and high load/store bandwidth [Kapasi et al. 2002].

There are a variety of mechanisms by which a programming model can provide access to high-speed on-chip memory. The two most popular mechanisms are a hardware-managed cache and a software-managed scratchpad memory. The difference between these two approaches is fundamental. For a cache, the decision as to which elements of data should be stored on chip is automatically made by the hardware at run-time, with the decision typically made at a fine granularity (e.g. blocks of 32 bytes). With a software-managed scratchpad memory, the decision as to which data should be stored on chip is made either at compile time or made explicitly by software at runtime, usually at a coarser granularity.

In applications with highly regular memory access patterns, such as classical DSP applications, a software-managed memory is the right choice. Software-managed memories carry less hardware overhead, allow static scheduling of the entire machine (particularly important for SIMD architectures), and provide the user and compiler with better performance guarantees than a hardware-managed cache.

In contrast, applications that manipulate adaptive data structures such as k-d trees, BSP trees, or short variable-length lists cannot easily manage memory at compile time. The application writer and compiler may know that there will be significant spatial and temporal locality of the memory accesses, but they do not know exactly what form this locality will take for any particular data set. For these applications, the binding of particular data elements to the on-chip memory is best performed at a fine spatial granularity. This approach is exactly that used by conventional hardware-managed caches. Since we believe that the construction, modification, and use of adaptive spatial data structures will be a performance-critical part of future real-time 3D graphics applications, we believe that future hardware architectures should support hardware-managed caches or at a minimum must include hardware primitives from which equivalent behavior can be efficiently implemented in software.

One important advantage of an architecture with traditional hardware-managed caches – especially if cache-coherency is supported – is that the architecture can provide the illusion of a single large memory, in which the storage hierarchy is simply an automatic hardware-supported performance optimization. In practice, parallel software must be heavily tuned to achieve good performance from such an architecture, but this performance tuning can be done incrementally. In contrast, software-managed memories are usually exposed to the programmer and/or compiler as a series of architecturally visible capacity "cliffs", which must be painfully overcome even in the earliest software prototypes.

The recently announced CELL architecture [Pham et al. 2005], is an interesting hybrid between the traditional cache strategy and scratchpad strategy. CELL's parallel cores (called SPE's) have a local scratchpad memory, but the DMA transfers between this scratchpad and main memory are coherent within a single global address space. The difficulty of managing a scratchpad memory is mitigated by the fact that the scratchpad is unusually large (256 KB per core). For a programmer, is is qualitatively easier to manage this L2-sized scratchpad than it is to manage a more traditional L1-sized scratchpad. Nevertheless, we believe that it will prove to be challenging to efficiently implement some irregular-datastructure algorithms on CELL. Even the strategy of using software to mimic traditional cache behavior is unlikely to perform well on CELL, due to the long branch mis-predict penalty and lack of hardware-supported multithreading. However, we believe that adding minimalist multithreading capability to the CELL SPE architecture would substantially improve this situation at relatively low cost, and we hope that this capability will be considered for future versions of CELL.

5.3 Hardware support for multithreading

A major problem encountered by most modern architectures is that the latency for moving data between the processing chip and off-chip DRAM memory can be 100 or more cycles. To maintain high ALU utilization, the machine must perform other work while such requests are outstanding. With a hardware-managed cache, the problem is particularly severe, because the compiler and hardware do not know in advance whether a particular 'load' or 'store' will miss the cache(s). Thus, *every* access to the unified address space potentially incurs a 100 cycle delay, whereas in a machine with a scratchpad, only the explicit accesses to off-chip memory can incur this delay.

Fortunately, highly parallel computations such as those in 3D graphics normally have other work (i.e. other threads) that can be processed during an off-chip memory access. There are two strategies for switching to other thread(s), which we will now describe.

The first strategy is to assume that every memory access misses the cache. This approach is followed by classical texture caching systems [Igehy et al. 1998], by the specialized SaarCOR raytracing architecture [Schmittler et al. 2004], and by cacheless multithreaded architectures like Tera [Alverson et al. 1990]. The ALU switches to other thread(s) (e.g. another fragment or vertex) for the required number of cycles, regardless of whether or not the memory request actually missed the cache. Unfortunately, this strategy requires that the number of active threads per ALU be approximately equal to the off-chip memory latency. The memory needed to store the working set for these threads can easily dominate the die area of the parallel processor, particularly when one considers the data-cache or scratchpad-memory footprint of each thread as well as its registers.

The second strategy is to switch to another thread only if the data access actually misses the cache. This approach is the one used by modern multithreaded machines such as Niagara [Krewell 2003], Cyclops [Caşcaval et al. 2002], and MAJC [Kowalczyk et al. 2001]. The advantage of this second approach is that fewer threads are required, particularly if cache misses are infrequent. Thus we consider this strategy to be the better one, at least if the machine is already a MIMD machine. However, it is worth noting that this strategy may not perform well if the memory accesses by different threads are highly correlated, leading to situations where all threads stall at the same time waiting for the same cache line. For example, this situation can occur for texture map lookups in a fragment shader. In some cases careful use of 'prefetching' can mitigate this problem, but it is not yet clear if this strategy would be effective for texture mapping.

There is an unfortunate tension between the goal of maximizing overlap of the working sets of different threads (which in turn reduces the per-thread SRAM requirements) and minimizing the temporal correlation between cache misses of different threads (which in turn allows a reduction in the ratio of threads-per-ALU). We expect that managing this tradeoff will be a major focus of future performance-optimization efforts for both hardware and software in single-chip parallel systems. One advantage of SIMD control flow that is often under-appreciated is that SIMD execution provides implicit but very tight inter-thread synchronization that facilitates reasoning about and management of this tradeoff. Managing this tradeoff in MIMD systems can require that fine-grained inter-thread synchronization be used for this purpose as well as for the traditional purpose of managing the more obvious control and data dependencies in the parallel computation.

5.4 Parallelism Summary

The various design decisions for a parallel machine are closely coupled to each other. For example, the decision to use a hardware-managed L1 cache in each core is at odds with a decision to use SIMD control. Broadly speaking, there appear to be two reasonable points in the design space, which can be referred to as "static" and "dynamic". Static machines such as Imagine bind and schedule most fine-grain resources at compile time – ALUs, on-chip memory, off-chip memory accesses, etc. The static strategy can use compile-time information about the program, but cannot not use much if any information about data-dependent behavior. In contrast, dynamic machines such as Niagara [Krewell 2003], Cyclops [Caşcaval et al. 2002] and the Intel IXP network processor [Adiletta et al. 2002] bind and schedule most resources at run time with hardware assistance. The dynamic-binding strategy uses both program information and runtime information derived from the data being processed.

For tasks in which the runtime information can significantly improve the quality of resource binding and scheduling, we believe that the dynamic approach will provide superior performance and will also be easier to program. However, for problems that can be effectively scheduled at compile time, there is no benefit to the dynamic approach, and the hardware support needed for it reduces the performance/price ratio of the hardware. Thus, the decision as to what type of machine to build should rest largely on anticipated application characteristics. We have argued that future real-time 3D graphics algorithms will use adaptive data structures, and thus that future architectures targeted to support these algorithms should use dynamic binding and scheduling. The close coupling we find here between the choice of software algorithms and the choice of hardware architectures is one of the reasons that we are advocating that algorithmic and hardware questions be investigated in tandem.

As with most such design-space tradeoffs, hybrid strategies exist. For example CELL has MIMD control flow, seemingly placing it in the dynamic category, but its high branch-mispredict penalty coupled with lack of multithreading somewhat penalize highly dynamic algorithms, as does CELL's choice of scratchpad memory rather than cache for local storage. A useful perspective on the general static-vs-dynamic tradeoff can be found in the architectural taxonomy found at the end of [Taylor et al. 2004].

5.5 CPU and parallel processor on the same die

Experience teaches us that very few problems are perfectly parallelizable. Historically, Cray's vector machines outperformed their competitors because they had superior performance for scalars and short vectors [Hennessy et al. 2003]. 3-D graphics is no exception to this general rule – modern graphics hardware has serialization points, although these potential bottlenecks are normally not user programmable.

For this reason, we believe that future graphics algorithms will split their work between an array of parallel processors optimized for high, power-efficient throughput on parallel code and at least one CPU-like processor optimized for maximum performance on a single thread. We believe that these two core types will be implemented with different sets of transistors, rather than by reconfiguring a single underlying substrate [Sankaralingam et al. 2003b; Taylor et al. 2004; Mai et al. 2000]. The reason is that a well-designed throughput-optimized processor differs from a single-thread-optimized processor in almost every respect, including the physical design of the individual transistors. The flexibility gained from a single reconfigurable substrate is likely to be more than offset by the cost of the necessary compromises.

To facilitate the low-latency, high-bandwith transfer of work between the throughput-optimized and single-thread-optimized processing cores, they must be integrated on a single die. Network processors [Adiletta et al. 2002] and CELL use this organization already, and we believe that in the long term these technical benefits as well as market trends towards cost reduction make such integration inevitable for graphics processors.

5.6 More than one kind of throughput-optimized core?

One important but open question is whether future chip-multiprocessors should have just one kind of throughput-optimized core, or two or more varieties of such cores. For example, it would be reasonable to build an architecture which has one set of cores that can only write to memory via stream outputs (like today's GPU fragment processors), and a second set of cores supporting cache-coherent memory writes and reads. The first set of cores would have higher peak performance, but would be restricted to a narrower class of computations than the second set of cores.

Other kinds of cores may also be useful. For example, current graphics chips include a simple configurable hardware unit (the raster-operation unit) located next to each of several memory controllers. We have shown that adding additional capabilities to this unit enables it to efficiently assist the task of building linked lists [Johnson et al. 2005]. Others have shown that such "near-memory processing" can be useful for traversing linked lists [Hughes and Adve 2005].

Finally, if a single-chip parallel architecture is expected to be heavily used for one particular task such as 3D rendering, it may be advantageous to include highly-specialized cores optimized for particular tasks such as texture filtering (included in today's GPU's) or ray/triangle intersection testing.

Most of these decisions must be made based on detailed cost/benefit analysis of both the workload and the hardware implementation, but there is one broad issue that will impact all such decisions. It is possible that future power budgets will prohibit architectures from using all of their transistors at once. This constraint would favor heterogeneous specialization of the architecture's processing units, a point that was first brought to our attention by Mark Horowitz.

6 Conclusion

We have argued that the next frontier in improved real-time image quality is to simulate global illumination effects for dynamic scenes. We claim that ray tracing will be the visibility algorithm of choice, but that it will initially be used to support non-physically-correct global illumination techniques.

We believe that a ray tracing system that efficiently supports dynamic scenes will need to integrate scene management with rendering. Since scene management code is somewhat application specific, this tight integration implies that the parallel architecture used to accelerate rendering must also be capable of executing application-specific scene management code. In turn, this requires that the parallel architecture support a general-purpose parallel programming model, with inter-thread communication, synchronization, and perhaps cache-coherent memory operations. The programming model supported by today's GPUs lacks most of these capabilities, and in particular it does not provide adequate support for creating and modifying adaptive data structures.

We believe the most promising hardware architecture to support this programming model is a MIMD multithreaded machine with cache-coherent shared memory. However, this conjecture remains unproven, and many questions remain about the details of such an architecture as well as its price/performance ratio.

To date we have not built either the software or the hardware necessary to confirm our hypothesis. What we have presented is a set of informed opinions backed by reasonable arguments and some initial results from architecture and algorithm simulations [Johnson et al. 2005]. Our purpose in presenting these opinions is two-fold. First, we think the ideas are sufficiently interesting that they will stimulate useful discussion within the research community. Second, we hope to persuade the research community that the particular approach we have outlined is sufficiently promising to be worthy of detailed investigation.

7 Acknowledgements

Gordon Stoll has contributed substantially to our thinking about real-time ray tracing systems and in particular is responsible for the insight that ray casting visiblity is likely to be used initially to support non-physically-correct techniques such as ambient occlusion rather than to support physically-correct techniques such as photon mapping.

Kurt Akeley is responsible for the insight that scene-graph rendering interfaces can be successful if they are tailored to sufficiently narrow application domains.

Our research group and collaborators – Greg Johnson, Paul Navratil, Calvin Lin, Chris Burns, Juhyun Lee, Karthikeyan Sankaralingam, Doug Burger, Steve Keckler, the Stanford Smart Memories group, Alex Joly, Ikrima Elhassan, David Whiteford, Chris Lundberg, Chendi Zhang and Peter Djeu – have been building systems and gathering results that have influenced our thinking about both rendering algorithms and parallel architectures.

We thank Microsoft, Intel, NVIDIA, and the University of Texas for supporting this research.

References

ADILETTA, M., ROSENBLUTH, M., BERNSTEIN, D., WOLRICH, G., AND WILKINSON, H. 2002. The next generation of Intel IXP network processors. *Intel technology journal 6*, 3 (Aug.).

AKELEY, K. 1993. RealityEngine graphics. In *SIGGRAPH 93*, 109–116.

ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. 1990. The tera computer system. *SIGARCH Comput. Archit. News 18*, 3, 1–6.

(AMERICAN NATIONAL STANDARDS INSTITUTE), A. 1988. programmer's hierarchical interactive graphics system (PHIGS) functional description. Tech. rep., ANSI.

AR, S., MONTAG, G., AND TAL, A. 2002. Deferred, self-organizing bsp trees. In *Eurographics 2002*.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRODER, P. 2003. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *SIGGRAPH 2003*.

CAŞCAVAL, C., NOS, J. G. C., CEZE, L., DENNEAU, M., GUPTA, M., LIEBER, D., MOREIRA, J. E., STRAUSS, K., AND JR, H. S. W. 2002. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, IEEE Computer Society, 311–322.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Eurographics 2003*.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The REYES image rendering architecture. *SIGGRAPH 87 21*, 4 (July), 95–102.

GRAPHICS STANDARDS PLANNING COMMITTEE. 1979. Status report of the graphics standards planning committee. *Computer graphics 13*, 3 (Aug.).

GRITZ, L., AND HAHN, J. K. 1996. BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools 1*, 3, 29–47.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *SIGGRAPH 90*, 289–298.

HENNESSY, J. L., PATTERSON, D. A., AND GOLDBERG, D. 2003. *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann.

HUGHES, C. J., AND ADVE, S. V. 2005. Memory-side prefetching for linked data structures for processor-in-memory systems. *Journal of Parallel and Distributed Computing 65*, 4 (Apr.), 448–463.

HURLEY, KAPUSTIN, RESHETOV, AND SOUPIKOV. 2002. Fast ray tracing for modern general purpose CPU. In *Graphicon 2002*.

HURLEY, J., 2005, Mar. Personal Communication.

IGEHY, H., ELDRIDGE, M., AND PROUDFOOT, K. 1998. Prefetching in a texture cache architecture. In *Proc. of 1998 Eurographics/SIGGRAPH workshop on graphics hardware*.

JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular z-buffer. *ACM Transactions on Graphics (to appear)*.

KAPASI, U. J., DALLY, W. J., RIXNER, S., OWENS, J. D., AND KHAILANY, B. 2002. The Imagine stream processor. In *Proc. of IEEE Conf. on Computer Design*, 295–302.

KATO, T. 2002. The "Kilauea" massively parallel ray tracer. In *Practical Parallel Rendering*, A K Peters, A. Chalmers, T. Davis, and E. Reinhard, Eds.

KECKLER, S. W., DALLY, W. J., MASKIT, D., , CARTER, N. P., CHANG, A., AND LEE, W. S. 1998. Exploiting fine-grain thread level parallelism on the MIT multi-alu processor. In *ISCA 1998*, 306–317.

KELLER, A. 1997. Instant radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 49–56.

KOWALCZYK, A., ADLER, V., AMIR, C., CHIU, F., CHNG, C. P., LANGE, W. J. D., GE, Y., GHOSH, S., HOANG, T. C., HUANG, B., KANT, S., KAO, Y. S., KHIEU, C., KUMAR, S., LEE, L., LIEBERMENSCH, A., LIU, X., MALUR, N. G., MARTIN, A. A., NGO, H., OH, S.-H., ORGINOS, I., SHIH, L., SUR, B., TREMBLAY, M., TZENG, A., VO, D., ZAMBARE, S., AND ZONG, J. 2001. The first majc microprocessor: A dual cpu system-on-a-chip. *IEEE Journal of Solid-State Circuits 36*, 11 (Nov.), 1609–1616.

KREWELL, K. 2003. Sun weaves multithreaded future. Available online at http://www.sun.com/processors/throughput/MDR_Reprint.pdf.

LANDER, J. 1998. Skin them bones: game programming for the web generation. *Game Developer Magazine* (May), 11–16.

LEXT, J., AND AKENINE-MOLLER, T. 2001. Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001*.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *SIGGRAPH 2001*.

MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W. J., AND HOROWITZ, M. 2000. Smart memories: A modular reconfigurable architecture. In *ISCA 2000*.

MARK, W. R., GLANVILLE, S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. In *SIGGRAPH 2003*.

MOYER, B., 2004. Ambient occlusion: It's better than a kick to the head. WWW page visited December 2004, http://www-viz.tamu.edu/students/bmoyer/617/ambocc/.

NVIDIA CORP. 2003. NV_fragment_program. In *NVIDIA OpenGL Extension Specifications*. Jan.

NVIDIA CORP. 2004. *NVIDIA GPU programming guide, v2.2.1*, Nov.

PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *Symposium on interactive 3D graphics*.

PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P.-P., HANSEN, C., AND SHIRLEY, P. 1999. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3, 238–250.

PHAM, D., S.ASANO, BOLLIGER, M., DAY, M., HOFSTEE, H., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI2, Y., RILEY1, M., SHIPPY1, D., STASIAK1, D., M.WANG, J.WARNOCK, S.WEITZEL, D.WENDEL, T.YAMAZAKI, AND K.YAZAWA. 2005. The design and implementation of a first-generation cell processor. In *Proceedings of 2005 IEEE Intl. Solid-State Circuits Conf.*

PHARR, M., AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. In *1996 Eurographics workshop on rendering*.

PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. 1997. Rendering complex scenes with memory-coherent raytracing. In *SIGGRAPH 1997*.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *SIGGRAPH 2002*, 703–712.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the 11th Eurographics Workshop on Rendering*, Eurographics Association, 299–306.

ROHLF, J., AND HELMAN, J. 1994. IRIS performer: A high performance multiprocessing toolkit for real–time 3D graphics. In *SIGGRAPH 94*, 381–394.

SANKARALINGAM, K., KECKLER, S. W., MARK, W. R., AND BURGER, D. 2003. Universal mechanisms for data-parallel architectures. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*.

SANKARALINGAM, K., NAGARAJAN, R., LIU, H., KIM, C., HUH, J., BURGER, D., KECKLER, S. W., AND MOORE, C. R. 2003. Exploiting ilp,tlp, and dlp with the polymorphous trips architecture. In *Proc. of the 30th Annual Intl. Symp. on Computer Architecture (ISCA)*.

SASANKA, R., ADVE, S. V., CHEN, Y.-K., AND DEBES, E. 2004. The energy efficiency of cmp vs. smt for multimedia workloads. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, ACM Press, New York, NY, USA, 196–206.

SCHMITTLER, J., WOOP, S., WAGNER, D., PAUL, W. J., AND SLUSALLEK, P. 2004. Realtime ray tracing of dynamic scenes on an fpga chip. In *Graphics Hardware 2004*.

SEGAL, M., AND AKELEY, K. 2002. *The OpenGL Graphics System: A Specification (Version 1.4)*. OpenGL Architecture Review Board. Editor: Jon Leech.

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 527–536.

TAYLOR, M. B., LEE, W., MILLER, J., WENTZLAFF, D., BRATT, I., GREENWALD, B., HOFFMANN, H., JOHNSON, P., KIM, J., PSOTA, J., SARAF, A., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2004. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA 2004*.

THOMPSON, C. J., HAHN, S., AND OSKIN, M. 2002. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Intl. symposium on computer architecture*.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed interactive ray tracing of dynamic scenes. In *Proc. IEEE symp. on parallel and large-data visualization and graphics*.

WALD, I., PURCELL, T. J., SCHMITTLER, J., BENTHIN, C., AND SLUSALLEK, P. 2003. Realtime ray tracing and its use for global illumination. In *Eurographics 2003*.

WALD, I., PURCELL, T. J., SCHMITTLER, J., BENTHIN, C., AND SLUSALLEK, P. 2003. Realtime ray tracing and its use for interactive global illumination. In *State of the Art Reports, EUROGRAPHICS 2003*.

WANN JENSEN, H. 2001. *Realistic image synthesis using photon mapping*. AK Peters.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June), 343–349.

Siggraph 2005 Course on Interactive Ray Tracing
The OpenRT API

Ingo Wald

This is an excerpt from
“Realtime Ray Tracing and Interactive Global Illumination”,
PhD Thesis, Ingo Wald,
Computer Graphics Group, Saarland University.
Full Version available at `http://www.mpi-sb.mpg.de/∼wald/PhD`

In the preceding course sections, all the basic constituents of a complete realtime ray tracing engine have been described: A highly efficient ray tracing kernel for modern CPUs, its efficient parallelization, and a simple yet efficient framework for handling dynamic scenes.

Once these “building blocks” have successfully been merged, essentially all the *technical* requirements for realtime ray tracing are fulfilled. However, a key issue for reaching the scenario of realtime ray tracing on everybody’s desktop is widespread application support, which requires a standardized and commonly accepted API. Roughly speaking, having a powerful new technology is one thing, having a good means of making the power of this technology available to the “average end user” is a totally different story. For hardware rasterization, this role of a powerful and widely accepted API has been taken by OpenGL [Neider93][1], which today is used by almost any graphics application, and which is well-known to virtually any graphics developer. Ideally, one could simply adopt OpenGL for ray tracing, in which case any existing OpenGL application could transparently render its images using ray tracing. Thus, one could (in theory) write an OpenGL “wrapper library” in the spirit of WireGL/Chromium [Humphreys02], that would perform the state tracking, would extract a ray-tracing suitable scene description from that, and would then call the ray tracer. The extended capabilities of ray tracing – namely shaders and global effects – could then be made available to the graphics programmer via the use of the well-known OpenGL extension mechanism (i.e. by offering a “GL_ARB_RAYTRACE” extension).

Unfortunately, this is but a mere theoretical option. as OpenGL and similar graphics APIs are too closely related to the rasterization pipeline. These APIs clearly reflect the stream of graphics commands that is fed to the rasterization pipeline, and as such also closely reflect the capabilities *and* limitations of the rasterization approach. In contrast to OpenGL, RenderMan [Pixar89, Apodaca90, Hanrahan90, Upstill90, Apodaka00] offers a more suitable high-level API that also supports ray tracing. However, these APIs offer almost no

[1] And, more recently, also by DirectX and Direct3D [DirectX].

support for interactive applications, and are thus not well suited for driving interactive applications.

Another option would be the use of an existing high-level scene graph library such as Performer, OpenInventor, OpenSG, OpenSceneGraph, or others [Rohlf94, Wernecke94, OpenSG01, OSG] for driving the ray tracer. This would already enable many new applications and would already reach a large number of potential users. However, the level of abstraction of such high-level scene graphs is too high for a generic ray tracing API, often is too application specific, and is unnecessarily restrictive. Being a low-level API, OpenGL allowed for all the "non-intended" uses (e.g. multipass rendering) while still allowing for layering higher-level APIs on top of it. In order not to unneccessarily restrict the potential uses of the API, it seems appropriate to follow this approach and design the API to be as low-level and flexible as possible. This allows for performing all the tasks that it is mainly thought for today, while still being flexible enough to adapt to potentially changing demands in the future.

As discussed above, none of the commonly available graphics APIs could be easily adopted for our ray tracing engine without unnecessary restrictions of its functionality. As such, we have decided to design a new API explicitly for realtime ray tracing. Ideally, such an API for realtime ray tracing should *not* only be an API specific to a certain implementation (i.e. the RTRT kernel), but should be both general and powerful enough to support the upcoming trend towards more widespread use of realtime ray tracing in general. Thus, we have designed our API with the following guidelines in mind:

- The API should be as low-level as possible in order to be able to layer higher-level scene graph APIs on top of it.

- It should be syntactically and semantically as similar to an existing, widely used graphics API as possible, in order to facilitate porting of existing applications and for leveraging programmers' experiences. Being the most commonly adopted graphics API, we have chosen OpenGL as a "parent API" to our new API, and have thus called it "OpenRT".

- It should be as powerful and flexible as RenderMan for writing shaders in order not to restrict the kinds of shading functionality that can be realized through this API.

The OpenRT API [Wald02, Wald03, Dietrich03] has been designed explicitly with these key observations in mind. While OpenRT so far has been implemented only on top of the previously mentioned RTRT kernel, it is not specific to this system. For example, the entire cluster infrastructure of the RTRT system has been completely abstracted from, and is not reflected in the API. Already today, two different implementations of this API are available, a distributed cluster version and a "stand-alone" shared-memory version (although both actually build on the same RTRT core). In the near future, it is planned to also use this OpenRT API for driving the SaarCOR architecture [Woop05].

1 General Design of OpenRT

As briefly mentioned above, one problem in designing OpenRT was choosing the right "parent" API to inherit ideas from: While it is generally a good idea to stay close to popular APIs – allowing to draw from a wide range of experienced people – there is the open question what API exactly to inherit from. On one side, OpenGL is the favorite API for writing interactive applications – it is very powerful, many people are experienced in OpenGL, and there is a huge amount of documentation and practical applications using OpenGL. On the other hand, OpenGL does not really fit a ray tracing engine: For example, it is mostly an immediate-mode API, and does not have any support for specifying shaders or for handling secondary effects (reflections, refraction) in a sensible manner.

In contrast to OpenGL, there are many APIs (like RenderMan[2], POV-Ray, etc.) that allow for taking advantage of all the benefits of ray tracing, but which are usually not applicable to interactive settings.

On the other hand, writing shaders and writing applications are usually two fundamentally different (though inter-playing) parts that can be realized with different APIs. As such, it is possible to take the best of both worlds, by using a RenderMan like API for writing shaders, and an OpenGL like API for writing the application. With this in mind, OpenRT has not been designed as *one* single graphics API, but actually consists of two mostly independent parts: One part is concerned with application programming, i.e. specifying geometric primitives, handling transformations and user input, handling textures, loading and communicating with shaders (but not writing them!), etc. This part of the API has been designed to be as close to OpenGL as possible. The second part of OpenRT describes how shaders are written – essentially describing a *shading language* – which has inherited much functionality from the RenderMan language, though it is not yet as flexible as "full" RenderMan.

1.1 Shader API – Application API Communication

All that is required to use this concept of having two separate parts of the same API is a minimal interface between these two subsystems. In OpenRT, this interface is realized via shader parameters (see below): Shaders are written independently from the application, and are stored in shared library files. Each shader "exports" a description of its shader parameters which control its shading calculations (e.g. the surface material properties to be implemented by this shader) but otherwise performs all its computations independently from the application. The application API then offers calls for loading these shaders, for binding them to geometry, for acquiring "handles" to their parameters, and for writing data to their parameters. For a closer description of this process, see below.

[2]RenderMan was originally not designed to be a "ray tracing" API, but mainly to drive the REYES [Cook87] architecture. However, its flexibility and powerful shading language allow for also using it for ray tracing and global illumination, see e.g. [Gritz96, Slusallek95].

Having a clear abstraction layer between application interface and shading language it is also possible to exchange any of these two parts without affecting the other. For example, it would be possible to use different shading languages like e.g. Cg [Mark03, Fernando03], OpenGL 2.0 Shading language [Kessenich02], or RenderMan [Upstill90, Apodaca90, Apodaka00] for writing the shaders, while still using the same application interface.

Instead of adopting another API as a parent API, it would also have been possible to create a completely new, independent API from scratch. Such approaches however tend to reinvent the wheel, and often have problems getting widely accepted (and used) by the users.

2 Application Programming Interface

As just described the application programming part of OpenRT has been designed to be as close to OpenGL as possible. As a rule of thumb, OpenRT offers the same calls as OpenGL wherever possible (albeit using "rt" as a prefix instead of "gl"), and only uses different calls where a concept of ray tracing has no meaningful match in OpenGL (or vice versa). In particular any calls for specifying geometry (i.e. vertices or primitives), transformations, and textures have identical syntax and semantics as OpenGL. This simplifies porting of applications where large parts of the OpenGL code can be reused without changes.

2.1 Semantical Differences

Note however that OpenRT is *not* compatible with OpenGL. In fact, the general rule often has been to use the same syntax where possible but not supporting all semantical details that do not easily fit a ray tracer. In practice that means that there are several concepts in which OpenRT can be used just as a "typical" user would use OpenGL, even though the actual semantics slightly differ. For example, the viewpoint in OpenGL is usually specified via calls to `gluLookAt` and `gluPerspective`. While OpenRT offers exactly the same functions with the same parameters (consequently called `rtLookAt` and `rtPerspective`) that also specify the camera position, OpenRT does *not* exactly follow the OpenGL semantics of having these functions change a "perspective transform" which in OpenGL could also be used for other applications, e.g. projective textures. Supporting these semantical details in OpenRT does not make much sense, as a ray tracer uses the much more general and flexible concept of a camera shader instead of a perspective transformation.

Though there are actually several of such low-level semantical differences, most are actually not very important for practical applications, as they usually appear only for concepts in which the ray tracer offers a more general concept (such as a freely programmable camera shader) anyway. In fact, many users of OpenRT take quite a while to discover the first of these differences at all. While these semantical differences obviously make porting more complicated, the two main goals of making OpenRT similar to OpenGL are not successfully realized with this approach: First, to the average user, OpenRT appears quite similar to OpenGL, and thus is easy to learn, understand, and accept as a new API. Second, none of the flexibility, features and functionality of ray tracing have to be sacrificed in order to be comply to OpenGL features that simply don't match.

2.2 Geometry, Objects and Instances

The main issue with using OpenGL for ray tracing is the fact that no information is available about the changes between successive frames. In OpenGL, even unchanged display lists can be rendered differently in successive frames due to global state changes in-between the frames[3]. This however does not map to a ray tracing engine, which needs information on which parts of a scene did or did not change since the last frame in order to achieve interactive performance (see accompanying document on "Handling Dynamic Scenes"). Therefore, instead of display lists OpenRT offers *objects* (see Figure 1). Objects encapsulate geometry together with references to shaders and their attributes. In contrast to display lists, objects may not have any side effects and as such changing the definition of one object can never affect the shape or appearance of any other object. On the other hand, global side effects are still possible (and usually beneficial) for the appearance of an object: As the primitives only store *references* to shaders, changing a shader at any later time will immediately change the appearance of all the primitives that this shader is bound to[4].

Objects are defined using an `rtNewObject(id)/rtEndObject()` pair. Each object is assigned a unique ID that is used to instantiate it later by a call to `rtInstantiateObject(ID)`. Note how this is (intentionally) very similar to OpenGL's way of handling display lists (i.e. `glNewList(id),glEndList()` and `glCallList(id)`).

Essentially, an instance consists of a reference to an object, together with a transformation matrix that this object is subject to (see Figure 1). Therefore, re-instantiating an object with a different transformation will change the position of the object in the scene (also see the accompanying document on "Handling Dynamic Scenes").

In order to support unstructured motion, each object can be redefined at any time by calling `rtNewObject` with the same object ID. Note that here too, global side effects can take place once an object is changed: Redefining an object automatically changes all the instances that have instantiated the redefined

[3]Actually, this "feature" of changing the effects of a display list by global state changes often even happens in the same frame.

[4]In OpenRT, the *shape* of the object (i.e. its triangles and vertices) is captured in the kd-tree, and will not be affected by any global state changes lateron. The *appearance* of the object if described by its references to the respective shaders (and, of course, by the global light sources shaders), and thus can change lateron by changing these respective shaders (see Figure 1. Even though this allows for side effects, it is conceptually slightly different from side effects through global state changes in OpenGL.

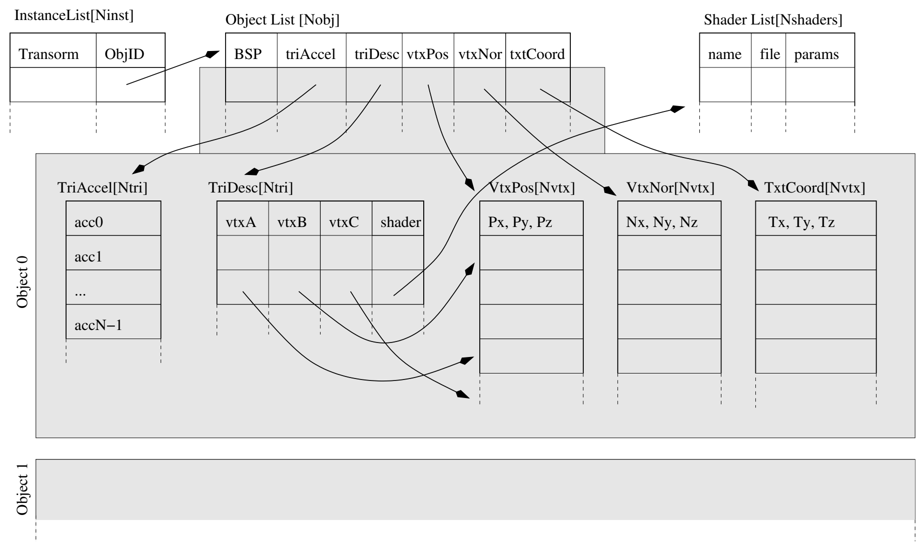


Figure 1: In the RTRT/OpenRT system, all geometry is encapsulated in *objects*. Each object (the grey block) contains the vertices, triangle description records, as well as their local acceleration structure. Each triangle contains references to its three vertices, as well as to its globally defined shader. In fact, each of these objects corresponds exactly to the RTRT Kernel data structures as described earlier in this course. In order to take effect, objects are instantiated, where each instance consists of an object ID and a transformation that this object is subject to (which corresponds to our method for handling dynamic scenes, as described in the accompanying document on “Handling Dynamic Scenes”). The entire scene then consists of the list of objects, the list of shaders, and the list of instances. Objects, shaders, and instances reference themselves by ID only, thereby allowing for dynamic and fully automatic side effects when changing any of these records.

object. Note that this API functionality perfectly matches the requirements of the previously proposed method to handle dynamic scenes.

Here again, the detailed semantics of OpenGL display lists and OpenRT objects/instances are slightly different. For example, certain “special features” (such as the above-mentioned global state changes) are not supported by OpenRT objects. However, the way that the “average user” uses a display list (i.e. for encapsulating a certain part of a scene graph for faster rendering) corresponds exactly to what an OpenRT object is being used for. As such, most users will hardly see the difference at all.

2.3 Shading, Shaders and Lighting

In order not to be limited by the fixed reflectance model of standard OpenGL, OpenRT does not offer or emulate the OpenGL shading model at all, but rather supports *programmable shaders* similar to RenderMan [Pixar89]. Shaders provide a flexible "plug-in" mechanism that allows for modifying almost any functionality in a ray tracer, e.g. the appearance of objects, the behavior of light sources, the way that primary rays are generated, how radiance values are mapped to pixel values, or what the environment looks like. In its current version, OpenRT supports all these kinds of programmability by offering support for "surface", "light", "camera", "pixel", and "environment" shaders, respectively.

In terms of the API, shaders are named objects that receive parameters and can then be referenced lateron by name or ID, e.g. in order to attach ("bind") a surface shader to geometry. The syntax and functionality are essentially the same as the functionality to specify texture objects in OpenGL: A set of shader IDs is allocated by `rtGenShaders()`, and a shader with a certain ID is then loaded by `rtNewShader(ID)`. lateron, a previously defined shader can be activated at any time by `rtBindShader(ID)`, e.g. in order to assign to some geometric primitives. Binding shaders to geometry works similarly to how materials properties are "assigned" in OpenGL: The application just binds a certain shader, at which stage all primitives issues after this call get this respective shader assigned to them.

Once the primitive is issues, the ID of the shader bound to the respective triangle is stored with the respective triangle. As changing individual triangles is only possible by redefining the respective object containing that triangle, this shader-primitive binding can not be changed any more without redefining the object and re-issueing the primitives with a differently bound shader. Note however that the triangles actually store only *references* to their respective shader (in fact, the *ID* of the shader). As such, changing the shader associated to these triangles itself (i.e. loading a new shader with the same ID as the original one) thus allows for changing the appearance of the respective triangle or object without having to touch any triangle or object at all.

2.3.1 Parameter Binding

For communicating with the applications, shaders export "parameters", each parameter having a symbolic name (e.g. "diffuse"). The application can then register a handle to a specific shader parameter (`rtParameterHandle()`), and can write to that parameter with a generic `rtParameter()` call. Note that the syntax and semantics for defining and accessing shader parameters is almost exactly the same as proposed in the Stanford shader API [Proudfoot01, Mark01]. A shader can specify its parameters to reside in different "scopes", i.e. a shader can be specified to be stored per vertex, per triangle, per object, or per scene. For example, a Phong shader would most likely want to have its material parameters stored per shader, whereas a radiosity viewer might want to store certain radiosity values in the vertices[5]. These different ways to specify parameters allow for optimizing shaders and minimize storage requirements.

Using a parameter binding by name allows for a very flexible way of having an application communicate with many different kinds of shaders. For example, if a VRML viewer [Dietrich04]) follows the convention to always assign the diffuse component of its VRML material to the “diffuse” parameter of a shader, all that different shaders have to do to get access to the applications material model is to implement and export the respective shaders. In this example, the same diffuse parameter value can be used for both a simple flat shader as well as for a shader implementing interactive global illumination. Neither application nor shader have to know anything else about each other except that they follow this convention[6]. Overhead due to binding by name is not an issue: Once the “handle” to the parameter has been acquired by the application, the assignment itself does not have to consider any symbolic names any more.

2.3.2 Lighting

The same argument given for materials is actually true for lighting: The OpenGL lighting model simply is too limited for a ray tracer to be useful. As such, all lighting calculations are implemented via programmable light source shaders (see below).

For convenience and compatibility, the OpenRT library comes equipped with default implementations for all the typical OpenGL (or VRML) light source types like point lights, spot lights, directional lights, or ambient lights. Even so, loading these shaders is different from specifying a light source in OpenGL (via `glLight()`), and requires special handling when porting applications.

2.4 A Simple Example

Obviously, this thesis can not give a complete description of the full OpenRT API with all its details. However, for readers being familiar with both OpenGL and with the concepts of a ray tracer, the following simple example should give a good overview of how OpenRT is used in practical applications [7].

```
// EightCubes.c:
// Simple OpenRT example showing
// eight rotating color cubes
#include <rtut/rtut.h> // include GLUT-replacement
```

[5]Obviously, it could do this also by storing them in the triangles vertex colors

[6]If the application tries to assign a value to a parameter that a shader never actually exported, this “invalid” assignment will be detected and ignored. This can be very useful for many applications: For example, a typical VRML application [Dietrich04] might simply assign the typical VRML material properties to each shader (writing to parameters named “diffuseColor”, “specularColor”, etc.). If the shader writer wants to have access to the VRML materials “diffuseColor”, it simply has to export a parameter with that name.

[7]The example given below uses a slightly outdated version of the OpenRT API (pre-1.0). In the most up-to-date version (currently 1.0R2), the example would look slightly different.

```
#include <openrt/rt.h> // include OpenRT header files

RTint createColorCubeObject()
{
  // Create an object for our
  // vertex-colored cube

  // Step1: Define the *class* of a vertex color shader
  int cid = rtGenShaderClasses(1);
  //allocate one slot for a shader class
  rtNewShaderClass(cid,''VertexColor'',''libVertexColor.so'');
  // load shader class ``VertexColor'' from a shared library

  // Step2: Create one instance of that shader class
  int sid = rtGenShaders(1);
  // allocate one slot for a shader instance
  rtNewShader(sid); // creates an instance of the
                    // currently bound shader class
  ...

  // Step3: Define the object
  RTint objId = rtGenObjects(1);
  rtNewObject(objId, RT_COMPILE);
    // Step3a: Bind the shader
    rtBindShader(sid);
    // Step3b: Specify transforms
    rtMatrixMode(RT_MODELVIEW);
    rtPushMatrix();
    rtLoadIdentity();
    // scale the cube to [-1,1]^3
    rtTranslatef(-1, -1, -1);
    rtScalef(2, 2, 2);
    // first cube side
    // Step3c: Issue geometry
    rtBegin(RT_POLYGON);
      rtColor3f(0, 0, 0);
      rtVertex3f(0, 0, 0);
      rtColor3f(0, 1, 0);
      rtVertex3f(0, 1, 0);
      rtColor3f(1, 1, 0);
      rtVertex3f(1, 1, 0);
      rtColor3f(1, 0, 0);
      rtVertex3f(1, 0, 0);
    rtEnd();
    // other cube sides
    ...
    rtPopMatrix();
  rtEndObject(); // finish building the object
  return objId;  // return object's ID to the caller
}

int main(int argc, char *argv[]) {
  // Init, open window, etc.
  // virtually exactly the same as any GLUT program
  rtutInit(&argc, argv);
  rtutInitWindowSize(640, 480);
  rtutCreateWindow("Simple OpenRT Example");

  // set Camera
  rtPerspective(65, 1, 1, 100000);
  rtLookAt(2,4,3, 0,0,0, 0,0,1);

  // generate object *once*
  objId = createColorCubeObject();
  for (int rot = 0; ; rot++) {
    // instantiate object eight times,
    // re-instantitate object for every frame
    // with different transformation
    rtDeleteAllInstances();
    for (int i=0; i<8; i++) {
      int dx = (i&1)?-1:1;
      int dy = (i&2)?-1:1;
      int dz = (i&4)?-1:1;

      // position individual objects
      rtLoadIdentity();
      rtTranslatef(dx,dy,dz);
      rtRotatef(4*rot*dx,dz,dy,dx);
      rtScalef(.5,.5,.5);
      rtInstantiateObject(objId);
    }
    // start rendering and display the image
    // frame buffer automatically handled by RTUT
    rutSwapBuffers();
  }
  return 0;
}
```

After opening a window, the “main” function first generates a vertex-colored RGB cube with a shader that just displays the interpolated vertex color. The cube is generated by first loading the “VertexColor” shader class from its shared library file, creating a single instance of it, and defining an object containing the geometry for the sides of the triangle. After the object has been completed, the "for"-loop creates eight rotating instances of this cube by re-instantiating each of the eight instances with a different transformation in subsequent frames. In fact, this simple example already features most of the important features of OpenRT: Specifying objects and instantiating them, issuing geometry, loading shaders, animating the objects, specifying the camera, and opening and using a window[8].

Being similar to OpenGL, this example should be easy to understand – and extend – by any slightly experienced OpenGL programmer. Of course, this is but a very simple example, and real programs will be considerably more complex. For example, a real program also has to load textures, specify light shaders, assign shader parameters, aso. Still, using advanced ray tracing effects in OpenRT is significantly simpler than generating the same effect in an OpenGL program: For example, rendering a scene once with global illumination effects and once without only requires to load a different shader – e.g. changing the shader name in "rtNewShaderClass" from "VertexColor" to "InstantGlobalIllumination" [Wald04, Benthin03] – without having to touch any other code in the program.

2.5 Semantical Differences to OpenGL

As already mentioned before, there are several issues on which OpenRT semantically differs from OpenGL.

2.5.1 Retained Mode and Late Binding

For example, OpenRT differs from the semantics of OpenGL when binding references. OpenGL stores parameters on its state stack and binds references immediately when geometry is specified. This is natural for immediate-mode rendering, but does not easily fit a ray tracer. OpenRT instead extends the notion of identifiable objects embedding state, similar to OpenGL texture objects. However, this binding is performed only during rendering once the frame is fully defined. This approach significantly simplifies the reuse of unchanged geometric objects across frames, thus getting rid of the need to redefine such unchanged objects every frame. On the other hand this means that any changes to an objects or shader defined in a previous frame might also affect the appearance of geometry defined earlier. For example, changing a shader parameter will automatically change the appearance of all triangles that this shader is bound to, even if those triangles have been specified in an earlier frame. Similarly, redefining a geometric object will automatically and instantly change the shape of all instances of that object, even if those have been defined in a previous frame. Though this sounds obvious, it can lead to somewhat unexpected results for people being used to OpenGL. For example, the code sequence

[8]Though the example uses RTUT (a GLUT) replacement, it is not required to use this interface. It is also possible to directly get access to the ray tracers frame buffer, and to display this e.g. via OpenGL

```
rtGenNewShaderClass(‘‘Diffuse’’,’’libDiffuse.so’’);
RTint diffuse = rtParameterHandle(‘‘diffuse’’);
rtParameter3f(diffuse, 1.f,0.f,0.f);
<triangle A>
rtParameter3f(diffuse, 0.f,1.f,0.f);
<triangle B>
rtSwapBuffers(); // render frame
```

will actually result in two triangle that are *both* green[9], which is not what an OpenGL-experienced programmer would expect.

Thus, these semantics are natural for a ray tracer but require careful attention during porting of existing OpenGL applications. More research is still required to better resolve the contradicting requirements of rasterization and ray tracing in this area.

2.5.2 Unsupported GL Functionality

Finally, some OpenGL functions are meaningless in a 3D ray tracing context and consequently are not supported in OpenRT. For instance, point and line drawing operations are not (currently) supported, and effects like "stipple bits" and "fill modes", as well as 2D frame buffer operations make little sense for a ray tracing engine either. Similarly, fragment operations, fragment tests, and blending modes are no longer useful and can be better implemented using surface and pixel shaders if necessary. Traditionally ray tracing writes only a single "fragment" to each pixel in the frame buffer after a complete ray tree has been evaluated. Thus the usual ordering semantics of OpenGL and its blending operations that are based on the submission order of primitives are no longer meaningful, either.

However the lack of this functionality so far has not been a problem for any of the applications already written on top of OpenRT: While these unsupported operations are very important for triangle rasterization, their main use is for multi-pass rendering. With the powerful shader concept offered by OpenRT, multipass-rendering is not neccessary any more, so this functionality so far has not been missed yet.

2.5.3 Frame Buffer Handling

Instead of writing the pixels to a hardware frame-buffer OpenRT renders into an application-supplied memory region as a frame buffer. This, however, is only due to the current hardware setup which uses a software implementation. For more dedicated ray tracing hardware, this is likely to change. For example, an OpenRT application on top of the SaarCOR architecture [Woop05] would most likely have the option to use a hardware frame buffer with direct VGA output

[9]Both triangles share *the same* shader. Until the two triangles are actually rendered during `rtSwapBuffers`, that shaders diffuse parameter has been set to green. Whether or not that parameter has had a different value when specifying triangle A does not make a difference.

instead of always transferring the rendered pixel values back to the application for display.

The above described "late binding"[10] also results in up to one frame of additional latency compared to OpenGL. The rasterization hardware can already start rendering as soon as the first geometric primitive is received by the renderer, and renders each primitive directly once it is specified (except for some buffering in the driver). Once all primitives have been sent to the graphics card, the resulting image as such is already finished. In contrast to this, the ray tracer has to wait for the full scene to be completely finished before it can actually start tracing any rays.

3 OpenRTS Shader Programming Interface

As motivated in the introduction of this chapter, the shader API in OpenRT (called "OpenRTS") has intentionally been designed to be mostly independent of the core API for writing applications. In order to allow for all the typical ray tracing effects that users are already used to, this API is as similar to RenderMan as possible.

3.1 Shader Structure Overview

The base class of all shaders in OpenRT is the "OpenRT Plug-in", i.e. an entity that can be loaded dynamically from a file, and which offers functionality for registering itself and exporting its parameters[11]. Once a parameter has been exported, the application can lateron bind a 'handle' to this parameter, and can assign values to it (see the above OpenRT example). Apart from registration and parameter export, all `RTPlugins` are equipped with an `Init` and NewFrame method that can be overwritten by its subclasses.

All other shader types – i.e. surface, light, camera and pixel shaders, and the rendering object (see below) – are derived from this base class, and as such can all be parameterized by the application.

3.1.1 Surface Shaders

The most common shader types in OpenRT obviously are surface shaders. Surface shaders have a virtual "`Shade`" function that is expected to return the color of the ray it got passed. For the shading operations, the surface shader has access to an extensive API for accessing scene data (e.g. vertex positions, normals, or texture coordinates) and for querying data concerning the ray and hit point (such as the shading normal, the ray origin and direction, the transformation that the hit object is subject to, etc). To differentiate these shader

[10]Sometimes also called "frame semantics" to stress its difference from "immediate mode semantics"

[11]For convenience, we only speak about C++ classes for specifying shaders. Though OpenRT in principle also allows for writing pure C-code shaders, C++ classes are actually more natural for implementing a shader concept and as such are usually preferred.

API functions from those of the core OpenRT API, all these functions (except class methods) start with the prefix “rts”.

3.1.2 Accessing Light Sources

In order to access light sources, a surface shader can query a list of light shaders over which it can iterate. The surface shader can then call back to each light shader (via `rtsIlluminate(...)`) to ask it for an “illumination sample”, or “light sample”. A “light sample” consists of all 3 values required for doing the lighting calculations in the surface shader: The direction towards the light, its distance (possibly infinite), and the intensity with which it influences the hit position.

Once a light shader has returned its light sample, this sample forms a complete shadow ray description with origin, direction, and maximum distance. This shadow ray description can then (but does not have to) be used by the surface shader to compute shadows by calling `rtsOccluded(...)` with this light sample, which in turn uses the ray tracing core to cast a shadow ray. If semi-transparent occluders are used, the surface shader can also use `rtsTransparentShadows()` instead of `rtsOccluded`, which will iterate over all the potential occluders along the shadow ray to compute the attenuated contribution of the light source.

3.1.3 Casting Secondary Rays

Except for casting shadow rays via `rtsOccluded()` (or via `rtsTransparent-Shadows()` for computing transparent shadows), further secondary rays can also be shot via `rtsTrace`. This `rtsTrace` shoots an arbitrarily specified ray, determines the hit point, calls the respective shader at that hit point, and returns the color computed by that shader. In case the ray did not hit any objects, `rtsTrace` automatically calls the *environment shader* for computing the color of that ray.

While `rtsTrace` already allows for all kinds of rays to be generated and shot, OpenRT offers several “convenience functions” for the most often used kinds of secondary rays, like e.g. `rtsReflectionRay()`, `rtsRefractionRay()`, `rtsTransparencyRay()`, etc.

3.1.4 Light Shaders

Similarly to the “Shade” function of the surface shaders, light shaders have a virtual `Illuminate` method that can be overridden to write new kinds of light shaders. As described above, OpenRT already comes equipped with the most common light source shaders like point, spot, and directional lights. For global illumination purposes, OpenRT also contains a few area light source shaders. As the surface shader expects illuminate to return a *single* light sample, these area light shaders take a list of pseudo-random numbers that they got passed from the surface shader to create a light sample. If a surface sample needs multiple samples from the same light source, it has to call `rtsIlluminate` several times with different random numbers.

3.1.5 Camera Shader

Camera shaders work in a similar way as surface and light shaders: Each camera shader has a single virtual function for initializing and returning a primary ray through the pixel, which will then be cast into the scene via `rtsTrace()`.

3.1.6 Environment Shader

Environment shaders are automatically called for all rays traced via `rtsTrace` that did not hit an object. In fact, an environment shader is a shader like any other (i.e. with a `Shade()` function), except that it does not make any sense to query any hit point information within the shading code.

3.1.7 The Rendering Object Concept

Whereas all the surface, light, camera, and environment shaders are typical shader types in any programmable shader concept, OpenRT additionally offers the concept of a "rendering object". A rendering object is responsible for actually computing pixel values, and as such enables the user to completely change the way that the ray tracer works. Typically, a rendering object will call a camera shader to generate a primary ray through each pixel, will call `rtsTrace`, and will let the respective surface shaders do the rest.

For special applications however, the rendering object can skip this flexible though costly shader concept, and can perform the rendering in a more "hard-coded" way, e.g. by directly using the fast RTRT packet tracing code with a hard-coded shading model. Similarly, many global illumination algorithms do not easily fit the above surface shader concept[12], but can be quite efficiently implemented as a rendering object. As such, rendering objects greatly extend the range of applications that can be realized with OpenRT. However, rendering objects are an advanced concept of OpenRT, and should be used with extreme care.

3.2 A Simple Shader Example

Obviously, the above explanation is but a very brief sketch of the OpenRT shader concept. The complete description of the shader API is beyond the scope of this thesis. More information on OpenRT and OpenRT shading can also be found in the respective OpenRT manuals and tutorials (see e.g. [Wald]).

As for the previously described application part of the OpenRT API, how the OpenRT Shader API is actually used in practice can best be described

[12]For example, the above shader concept expects a shader to compute the color of a ray, whereas many global illumination algorithms require evaluation of a BRDF with given incoming and outgoing directions (such as bidirectional path tracing), or sampling of a BRDF (e.g. for photon shooting or generation of light- and eye-paths).

with a simple example. As such, the following example implements some simple (though typical) OpenRT shaders, one light shader and one surface shader. The surface shader implements a simple diffuse shader, parameterized by a diffuse color and an ambient term). The light shader implements a typical point light source consisting of position and intensity, and with a hard-coded quadratic intensity falloff.

3.2.1 Simple Diffuse Shader

```
class SimpleDiffuse : public RTShader {
  RTVec3f diffuse;
  RTVec3f ambient;

  RTvoid Register() {
    // register parameters
    rtDeclareParameter("diffuse", PER_SHADER,
                       offsetof(diffuse),sizeof(diffuse));
    rtDeclareParameter("ambient", PER_SHADER,
                       offsetof(ambient),sizeof(ambient));
  }

  RTvoid Shade(RTState *state)
  {
    RTVec3f color = ambient;      // init with ambient color
    RTVec3f P;                    // surface hit position
    RTVec3f N;                    // normal
    rtsGetHitPosition(state,P);
    rtsFindShadingNormal(state,N);// interpolate normal, make
                             // sure it faces toward the viewer
    RTState shadow = *state;      // init shadow ray state
    RTenum *light; RTint lights;
    lights = rtsGlobalLights(&light);
    for (int i=0;i<lights;i++)
    { // iterate over all light sources
      rtsIlluminate(light[i],P,&shadow,NULL);
      if (rtsOccluded(&shadow))
         continue; // test for shadows

      Vec3f L; // light direction
      Vec3f I; // light intensity
      rtsGetRayDirection(&shadow,L);
      rtsGetRayColor(&shadow,I);
      RTfloat cosine = N * L;     // dot product
      I *= diffuse;               // component-wise mult.
      color += (cosine * I);
    }
    rtsReturnColor(state,color);
  }
};
rtsDeclareShader(SimpleDiffuse, SimpleDiffuse);
```

3.2.2 Simple PointLight Shader

```
class SimplePointLight : public RTLight {
  RTVec3f position;
  RTVec3f intensity;

  RTvoid Register() {
    rtDeclareParameter("position",
         offsetof(position),sizeof(position));
    rtDeclareParameter("intensity",
         offsetof(intensity),sizeof(intensity));
  }

  RTvoid Illuminate(RTState *state) {
    RTVec3f P;                   // surface hit point
    rtsGetRayOrigin(state,P);

    RTVec3f L = position - P;     // direction towards light
    RTfloat distance = Lenght(L);
    Normalize(L);

    RTVecf3 I = intensity * 1./(distance * distance);
    // quadratic distance attenuation

    rtsSetRayDirection(state,L);
    rtsSetRayMaxDistance(state,length - Epsilon);
    rtsReturnColor(state,I);
  }
};
rtsDeclareShader(SimplePointLight, SimplePointLight);
```

4 Taking it all together

Having now described all the different parts of the API, it is important to briefly summarize how these different parts actually play together. To do this, we will briefly go – step by step – through the process of rendering a frame:

1. First, the application specifies the scene itself, i.e. it loads and parameterizes shaders, specifies objects and instances, issues geometry, sets the frame buffer, etc. All the time, the OpenRT implementation makes sure

that all these calls get executed on all rendering clients, be it the local CPU, remote cluster clients, or a hardware architecture.

2. Once the scene is specified, the application calls `rtSwapBuffers` to tell the ray tracer that any scene updates are finished and that it should render a frame.

3. Upon `rtSwapBuffers` the OpenRT library calls the user-programmable rendering object to actually perform the rendering computations. In a single-CPU or shared-memory version, the rendering object will simply render a complete frame. In the distributed cluster version, the ray tracer will automatically perform the load distribution, load balancing, and communication between the clients and the server. As such, it will automatically request each client's respective rendering object to render one or more tiles.

4. The rendering object iterates over all the pixels in its frame (respectively tile), and calls the user-programmable camera shader to generate a primary ray through that tile.

5. Once a valid primary ray has been generated, the rendering object tells OpenRT to trace this ray and compute its color. To do this, OpenRT uses the RTRT kernel to trace the ray and find a hit point.

6. If no valid hit could be found, OpenRT automatically calls the (user-programmable) environment shader to shade the ray. If a hit was found, OpenRT determines the respective surface shader and calls its `Shade` method.

7. The (user-programmable) surface shader uses the shader API to call back to the library while performing its shading computations, e.g. by asking OpenRT for the list of active lights, or for the shading normal of the hit point. This also includes asking OpenRT for a light sample from a given light shader. OpenRT will then look up that light shader, and call its respective`Illuminate` function.

8. The light shader generates this light sample (probably with some additional calls into the shader API), and returns this – via OpenRT – to the surface shader.

9. Having processed all light samples, the surface shader may tell OpenRT to shoot some additional secondary rays, for which stages 5–9 are recursively repeated[13].

10. Once the entire shading tree has been processed, the rendering object has the color of the hit point as determined by the surface shader. It may now do some final operations on this ray (in the spirit of a "pixel

[13]Obviously, the secondary rays can be shot at any time, not only at the end of the shader routine.

shader"), e.g. for performing tone mapping. Once this is done, it writes the pixel to the frame buffer. Again, in the distributed version all these pixels that have been computed on different machines get automatically communicated back to the server (where they can be again manipulated by a user-programmable routine).

11. Once all pixels have been computed, the OpenRT library returns the frame buffer to the application, and returns from the `rtSwapBuffers` call.

12. The application can now display the frame buffer, and can start over by starting to specify the next frame.

Though this is in fact exactly the same ray tracing pipeline that any decent ray tracer uses as well, two things are important to note: first, the modularity and programmability of this framework, and second, the hardware abstraction model used in OpenRT.

4.1 Modularity and Programmability

First of all, taking a closer look at the above topics makes clear that OpenRT is a highly flexible API in which almost all parts are user programmable and can be arbitrarily replaced. Surface, light, environment and camera shaders, the rendering mode, and to a certain degree even the parallelization can be changed by the user.

The OpenRT library in fact provides only the basic infrastructure – such as abstracting from the distributed architecture, automatic handling of all parallelization and communication, scene management etc – and nicely glues the different user-programmable parts together. Last but not least, the OpenRT library also drives the ray tracing kernel and makes it available to all the respective subsystems.

Of course, for all these user-programmable parts (such as generating the tiles in the rendering object, generating primary rays, or assembling the pixels to the final image) there are optimized default routines. Most users will never make contact with any of these advanced issues, and will concentrate on writing surface and/or light source shaders.

4.2 Hardware Abstraction Model

The second important issue to mention is how this design carefully abstracts from the underlying hardware. For example, the shader-application communication works *entirely* over the shader parameter concept, and never assumes any direct communication between shaders and application. As such, the shaders can either be located on the same machine as the application, or could run on another, remote machine that does not even know about the application. It would just as well possible that the shaders themselves are not software C++ classes at all, but might reside directly on a ray tracing hardware architecture such as SaarCOR.

Similarly, the shader API (i.e. the API used by the shader programmer) is strictly kept apart from the main OpenRT application API. As such, the same application program could be used even if the shader API changes. For example, the SaarCOR architecture [Woop05] obviously will not use the same C/C++ shader API that is currently used on the CPU[14].

5 Conclusions and Future Work

In summary, OpenRT is a simple yet highly flexible API for realtime ray tracing. It is simple to use and flexible enough to support all typical ray tracing effects though a RenderMan like shading API and a highly modular user-programmable plug-in concept.

While the application API is not actually semantically 100% compatible to OpenGL, the syntax and semantics for typical programs are still very similar. Thus, novice OpenRT users with (some) previous OpenGL experience so far found OpenRT easy to learn and use. In fact, many concepts (e.g. shaders) appeared easier and more natural to these users. Because of this, OpenRT so far has shown to be well accepted by current users. However, highly experienced OpenGL users (which tend to know – and use – all the subtle details of OpenGL) sometimes found it hard to understand that certain concepts are different (e.g. that a `rtLookAt` call does not have any side effects on the matrix stack that could then be exploited for projective textures). Though some open questions remain, OpenRT has already been used for several practical projects, and so far has been very successful for those projects. In particular, it is already being using in real-world industrial project, and so far has been very successful.

For really widespread use, however, still some more work has to be invested: First, it would be desirable if more different implementations of the OpenRT API would be available, e.g. on the SaarCOR architecture, on a GPU-based implementation (e.g. [Purcell02]), or on an open source ray tracer.

Furthermore, it has to be evaluated whether – and how – the remaining differences between OpenGL and OpenRT could be bridged. Eventually, it would be a highly interesting option to somehow combine OpenGL and OpenRT, e.g. by making OpenRT to be an OpenGL extension. Due to fundamentally different semantics as discussed above, it yet unclear if this is possible at all, let alone in which way.

An even more important issue to work on is an efficient shading API that supports coherent packets of rays. As described in earlier on, the full performance of the RTRT core can only be unleashed if SIMD packet traversal with efficient SIMD shading can be used. In its current form, however, the OpenRT shader API actually supports only the shading and tracing of single rays. For those having the actual RTRT sources, it is still possible to use both packet

[14]Though it is still imaginable to use the same "shading language" for both the software and the hardware implementation, e.g. by using different shading language compilers (with the same syntax) for the different platforms.

traversal code and OpenRT API at the same (e.g. by performing the packet traversal code inside a rendering object), but a clean external API does not exist. As it is not yet even clear how such packet shading could be efficiently performed at all, it seemed premature to already discuss its API issues.

Finally, probably the biggest challenge for the success of OpenRT is to create new, powerful interactive applications. This also implies making it available to a much wider range of users to actually build these applications. Though all our experiences with OpenRT so far have been highly encouraging, only once many different kinds of users will actually use if for solving their everyday practical rendering problems will it be possible to objectively evaluate the real potential – and the limitations – of this API.

As most applications in fact operate on a much higher level of abstraction – usually working on scene graphs rather than directly on the API level – making OpenRT available to a wider range of users also implies to investigate how scene graphs can be efficiently mapped to the new API. Preliminary work has already investigated how a VRML-based scene graph (the XRML engine [Bekaert01]) can be mapped to OpenRT [Wagner02, Dietrich04]. However, an even deeper investigation of this problem has yet to be performed.

References

[Apodaca90] *A. Apodaca and M. Mantle.* RenderMan: Pursuing the Future of Graphics. *IEEE Computer Graphics & Applications*, 10(4):44–49, July 1990.

[Apodaka00] *Anthony Apodaka and Larry Gritz. Advanced RenderMan: Creating CGI for Motion Pictures.* Morgan Kaufmann, 2000. ISBN: 1558606181.

[Bekaert01] *Philippe Bekaert.* Extensible Scene Graph Manager, August 2001. http://www.cs.kuleuven.ac.be/∼graphics/XRML/.

[Benthin03] *Carsten Benthin, Ingo Wald, and Philipp Slusallek.* A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum*, 22(3):621–630, 2003. (Proceedings of Eurographics).

[Cook87] *Robert L. Cook, Loren Carpenter, and Edwin Catmull.* The REYES Image Rendering Architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 1987)*, pages 95–102, July 1987.

[Dietrich03] *Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek.* The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, pages 23–31, Darmstadt, Germany, 2003. Eurographics Association.

[Dietrich04] *Andreas Dietrich, Ingo Wald, Markus Wagner, and Philipp Slusallek.* VRML Scene Graphs on an Interactive Ray Tracing Engine. In *Proceedings of IEEE VR 2004*, pages 109–116, March 2004.

[DirectX] Microsoft DirectX 8.0. http://www.microsoft.com/windows/-directx/.

[Fernando03] *Randima Fernando and Mark J. Kilgard. The Cg Tutorial – The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley, 2003.

[Gritz96] *Larry Gritz and James K. Hahn.* BMRT: A Global Illumination Implementation of the RenderMan Standard. *Journal of Graphics Tools*, 1(3):29–47, 1996.

[Hanrahan90] *Pat Hanrahan and Jim Lawson.* A language for shading and lighting calculations. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 24(4):289–298, August 1990. ISBN: 0-201-50933-4.

[Humphreys02] *Greg Humphreys, Mike Houston, Ren Ng, Sean Ahern, Randall Frank, Peter Kirchner, and James T. Klosowski.* Chromium: A Stream Processing Framework for Interactive Graphics on Clusters of Workstations. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, 21(3):693–702, July 2002.

[Kessenich02] *John Kessenich, Dave Baldwin, and Randi Rost.* The OpenGL Shading Language, Version 1.051, February 2002. Available from http://www.3dlabs.com/support/developer/ogl2/-downloads/ShaderSpecV1.051.pdf.

[Mark01] *William Mark.* Shading System Immediate-Mode API, v2.1. In *SIGGRAPH 2001 Course 24 Notes – Real-Time Shading*, August 2001.

[Mark03] *William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard.* Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 22(3):896–907, 2003.

[Neider93] *Jackie Neider, Tom Davis, and Mason Woo. OpenGL Programming Guide.* Addison-Wesley, 1993. ISBN 020163-2748.

[OpenSG01] *OpenSG-Forum.* http://www.opensg.org, 2001.

[OSG] OpenSceneGraph. http://www.openscenegraph.org.

[Pixar89] *Pixar. The RenderMan Interface.* San Rafael, September 1989.

[Proudfoot01] *Kekoa Proudfoot, William Mark, Svetoslav Tzvetkov, and Pat Hanrahan.* A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of ACM SIGGRAPH*, pages 159–170, August 2001.

[Purcell02] *Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan.* Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002. (Proceedings of SIGGRAPH 2002).

[Rohlf94] *John Rohlf and James Helman.* IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Computer Graphics*, 28(Annual Conference Series):381–394, July 1994.

[Slusallek95] *Philipp Slusallek, Thomas Pflaum, and Hans-Peter Seidel.* Using Procedural RenderMan Shaders for Global Illumination. In *Computer Graphics Forum (Proc. of Eurographics '95*, pages 311–324, 1995.

[Upstill90] *Steve Upstill. The RenderMan Companion.* Addison-Wesley, 1990.

[Wagner02] *Markus Wagner.* Development of a Ray-Tracing-Based VRML Browser and Editor. Master's thesis, Computer Graphics Group, Saarland University, Saarbrücken, Germany, 2002.

[Wald] *Ingo Wald and Tim Dahmen. OpenRT User Manual.* Computer Graphics Group, Saarland University. http://www.openrt.de.

[Wald02] *Ingo Wald, Carsten Benthin, and Philipp Slusallek.* OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics. Technical report, Saarland University, 2002. Available at http://graphics.cs.uni-sb.de/Publications.

[Wald03] *Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek.* Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports*, 2003.

[Wald04] *Ingo Wald.* Realtime Ray Tracing and Interactive Global Illumination. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[Wernecke94] *Josie Wernecke. The Inventor Mentor.* Addison-Wesley, 1994. ISBN 0-20162-495-8.

[Woop05] *Sven Woop, Joerg Schmittler, and Philipp Slusallek.* RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *Proceedings of ACM SIGGRAPH*, (to appear), 2005.

Realtime Ray Tracing for Current and Future Games

Jörg Schmittler, Daniel Pohl, Tim Dahmen, Christian Vogelgesang, and Philipp Slusallek
{schmittler,sidapohl,morfiel,chrvog,slusallek}@graphics.cs.uni-sb.de

Abstract: Recently, realtime ray tracing has been developed to the point where it is becoming a possible alternative to the current rasterization approach for interactive 3D graphics. With the availability of a first prototype graphics board purely based on ray tracing, we have all the ingredients for a new generation of 3D graphics technology that could have significant consequences for computer gaming. However, hardly any research has been looking at how games could benefit from ray tracing.

In this paper we describe our experience with two games: The adaption of a well known ego-shooter to a ray tracing engine and the development of a new game especially designed to exploit the features of ray tracing. We discuss how existing features of games can be implemented in a ray tracing context and what new effects and improvements are enabled by using ray tracing. Both projects show how ray tracing allows for highly realistic images while it greatly simplifies content creation.

1 Introduction

Ray tracing is a well-known method to achieve high quality and physically-correct images, but only recently its performance was improved to the point that it can now also be used for interactive 3D graphics for highly complex and dynamic scenes including global illumination [Wa04, WDS04, WBS03, BWS03, GWS04].

While the above systems still rely on distributed computing to achieve realtime performance, a first prototype of a purely ray tracing based graphics chip [SWWS04] shows that efficient hardware implementations are indeed possible and provide many advantages over rasterization

This encourages the research on possible effects of ray tracing technology for computer games. In this paper we describe our experiences with two games which use ray tracing for rendering and the physics engine.

2 Computer Games Based on Ray Tracing

Ray tracing and rasterization technology are basically two different algorithms to solve the same problem: the visibility calculation. While rasterization uses a set of potential visible triangles which are rendered sequentially into the Z-buffer, ray tracing starts at the virtual camera and for every pixel shoots rays into the scene. Since rays are terminated as soon as they hit an object, the visibility calculation is highly efficient and fully output sensitive. As shading is performed after visibility calculation, you only pay for what you see.

While ray tracing has access to the entire scene database and only reads what it needs to, current rasterization technology operates on a stream of independent triangles sent by the application. Therefore it cannot efficiently and accurately render *global effects* such as shadows, reflections, and indirect illumination on demand, i.e. after finding out that these effects are actually visible. Every effect has to be split into several render passes by the application and relies on tricks and approximations (e.g. shadow and reflection maps) which are inaccurate and break down in many situations (e.g. multiple reflections).

In contrast ray tracing trivially supports global effects by shooting on demand additional rays for shadows, reflections, and refractions. This output sensitivity allows for efficiently rendering even highly complex scenes. For every pixel this recursive approach automatically combines all visible shading effect correctly without involving the application or the need for separate rendering passes. Even memory management of the graphics card's memory is handled automatically by the ray tracer [SLS03].

Basic shading computations are the same as for rasterization. Thus, the same shaders (e.g. for texture filtering and calculation of light intensities) and image filters (e.g. for anti-aliasing) can be used. However, ray tracing allows to adaptively shoot new rays as required. While both techniques can eventually achieve similar results, this requires complex and costly operation by both the graphics hardware and the application. In contrast ray tracing handles most effects automatically and internally. This greatly simplifies content creation for games, which is increasingly becoming a limiting factor for the gaming industry.

2.1 Traditional Ego Shooter

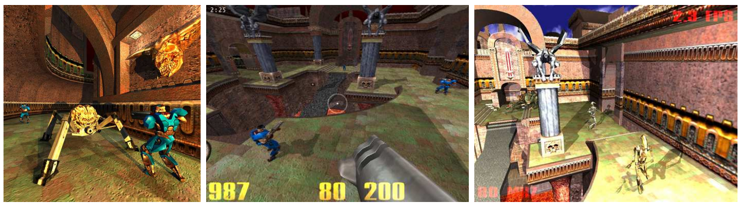
We started our research with adapting the existing, well-known ego-shooter *Quake 3: Arena* by Id-Software to use ray tracing for rendering. We concentrated our efforts on adapting shading effects and general game management because most otherwise difficult rendering effects (e.g. shadows and reflections) were automatically handled by the ray tracer.

The game engine was written from scratch and supports player and bot movement including shooting and jumping, collision detection, and many special effects like jump-pads and teleporters. The main development was done by a single student in less than six months.

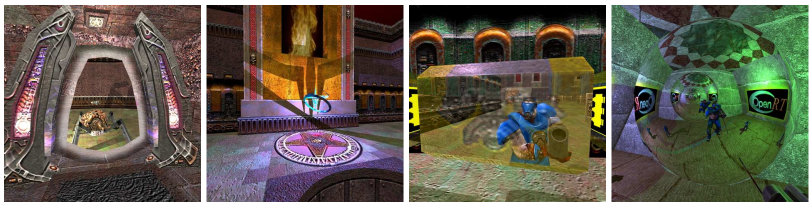
The game engine interfaces with the ray tracer through the OpenRT-API [DWBS03], which is very close to the OpenGL. OpenRT manages all ray tracing events fully transparent to the application, making it unaware of the underlying ray tracing implementation, which may run on a single computer, a cluster of PCs, or a dedicated ray tracing hardware.

Figure 1 shows several example images from the game with many shading effects. The engine supports *all* of the standard effects of traditional computer games like dynamic placement of (blood) splats, texture animation and blending, volumetric fog, and pre-computed light maps (if desired).

Figure 1: Screen shots of the ray traced version of *Quake 3: Arena*.



Screen shots of live game play. While most images are taken from the PC-cluster-based version, the right-most image was rendered on the hardware prototype (1024x768, 32bit). All images were rendered at fully interactive rates of 5-20 fps.



Some of the effects supported by ray tracing: a portal providing a view into distant places, light effects in the power-up, and correct reflections in the ammo-box and on some spheres.

More ray tracing specific effects like dynamic lighting including shadows and physically-correct reflections and refractions are trivially supported by simply specifying the corresponding material attributes. This also holds for camera portals and surveillance cameras, which are automatically rendered correctly by default even if they recursively see each other.

Since ray tracing is output sensitive there is no need for any level-of-detail mechanism to reduce scene complexity. This allows for highly crowded scenes with many players, monsters, and complex trees in a forest. Furthermore as ray tracing efficiently supports multiple instantiations, even crowded scenes have negligible memory requirements and scene complexity has only a minimal impact on performance.

In summary, we were able to support all the traditional effects of Quake 3 while most effects were significantly simpler to implement. Looking at newer engines such as *Unreal 3*, we still see no effects that could not be supported easily by ray tracing.

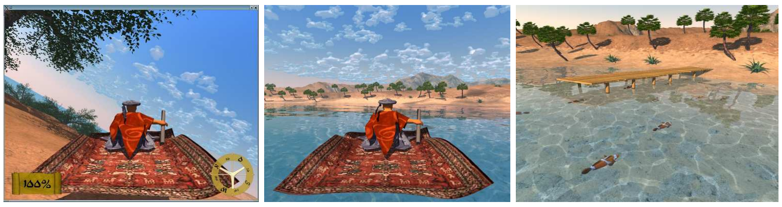
2.2 Novel Game Design for Ray Tracing

Ray tracing offers new ways to design a game which led us to the development of *Oasen* game. Oasen operates in a fully open space on a huge world consisting of several islands and includes day time simulation with changing sky and light situations (see Figure 2). The player takes the role of a salesman on a flying carpet visiting different places, buying and selling goods while fighting off other players or bots.

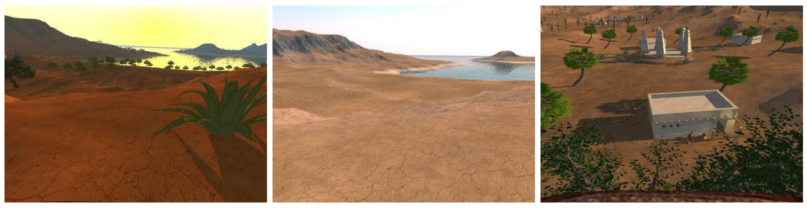
While ray tracing is basically a method for visibility calculation, we were also able to use it for the physics engine, acoustics, and collision detection. Similar to a radar system it uses rays to determine the distance to nearby objects. By reusing the existing fast ray tracer on the original geometry we avoided having to build special algorithms and data structures for those tasks.

No level-of-detail mechanisms, clipping-planes, or fog have been used to reduce scene complexity, since ray tracing efficiently handles huge amounts of geometry and objects automatically. This avoids any artifacts such as popping and results in smooth flights. Huge numbers of light sources are efficiently handled by exploiting the restricted range of illumination of each light and organizing them in a spatial index structure. For each pixel we can then efficiently locate light sources that contribute to its illumination, allowing physically-correct illumination at very low costs even in the presents of hundreds of visible light sources.

Figure 2: Screen shots of the ray tracing based game *Oasen*.



Typical life screen shots showing correct shadows, nicely rendered water including caustic-effects, and volumetric clouds.



The two left-most images show the inherent scene management capability of ray-tracing: vegetation and buildings add 40-times triangles over the basic landscape geometry while the performance drops by less than 10% – without the use of any level-of-detail or clipping techniques.

3 Conclusion

In this paper we briefly summarized the experience we gained from implementing two games using a ray tracing based game engine. We have been able to easily port all of the exiting rendering effects to ray tracing, where their implementation has been much simpler and more efficient. Furthermore ray tracing adds many novel aspects that help designing more realistic and compelling game contents.

Any shading effect can be approximated with rasterization, but every combination of shaders requires special support and complex programming for both the application and shaders. In contrast, ray tracing automatically handles all shader interaction allowing for plug-and-play use of arbitrary shading effects. As a result, content creation is greatly simplified and game designers can again concentrate on the content and game experience instead of working around the many limitations of current technology. Furthermore ray tracing offers new ways to design the physics engine including collision detection and acoustics.

The main limitation of ray tracing has been its still limited support for dynamic scenes [WBS03], but ongoing research will soon remove this constraint. Even though LOD mechanisms were not needed for supporting complex scenes, new approaches are required to efficiently handle resulting geometric aliasing.

The highly realistic and physically-correct images together with a greatly simplified rendering engine make ray tracing an interesting technology for future computer games. Today, the system requirements for ray tracing based games seem to be very high as a cluster of PCs forming a virtual CPU with 30 GHz is required to render the images present here at interactive rates (5-20 fps for 640x480 pixels). But future hardware will allow to have even higher performance on a single PC board similar to today's graphics cards [SWWS04]. This encourages further research on ray tracing and computer games. More details and videos are available at `http://graphics.cs.uni-sb.de/RTGames/`

Literatur

[BWS03] Benthin, C., Wald, I., und Slusallek, P.: A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum*. 22(3):621–630. 2003. (Proceedings of Eurographics).

[DWBS03] Dietrich, A., Wald, I., Benthin, C., und Slusallek, P.: The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In: *Proceedings of the 2003 OpenSG Symposium*. S. 23–31. Darmstadt, Germany. 2003. Eurographics Association.

[GWS04] Günther, J., Wald, I., und Slusallek, P.: Realtime caustics using distributed photon mapping. In: *To appear in Proceedings of Eurographics Symposium on Rendering*. 2004.

[SLS03] Schmittler, J., Leidinger, A., und Slusallek, P.: A Virtual Memory Architecture for Real-Time Ray Tracing Hardware. *Computer and Graphics, Volume 27, Graphics Hardware*. S. 693–699. 2003.

[SWWS04] Schmittler, J., Woop, S., Wagner, D., und Slusallek, P.: Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In: *To appear in Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*. 2004.

[Wa04] Wald, I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis. Computer Graphics Group, Saarland University. 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[WBS03] Wald, I., Benthin, C., und Slusallek, P.: Distributed Interactive Ray Tracing of Dynamic Scenes. In: *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*. 2003.

[WDS04] Wald, I., Dietrich, A., und Slusallek, P.: An interactive out-of-core rendering framework forvisualizing massively complex models. In: *To appear in Proceedings of Eurographics Symposium on Rendering*. 2004.

Technical Report
NWU-CS-05-07
April 26, 2005

Adaptive Frameless Rendering

Abhinav Dayal[1], Cliff Woolley[2], Benjamin Watson[1] and David Luebke[2]
[1]Northwestern University, [2]University of Virginia

Abstract

We propose an adaptive form of frameless rendering with the potential to dramatically increase rendering speed over conventional interactive rendering approaches. Without the rigid sampling patterns of framed renderers, sampling and reconstruction can adapt with very fine granularity to spatio-temporal color change. A sampler uses closed-loop feedback to guide sampling toward edges or motion in the image. Temporally deep buffers store all the samples created over a short time interval for use in reconstruction and as sampler feedback. GPU-based reconstruction responds both to sampling density and space-time color gradients. Where the displayed scene is static, spatial color change dominates and older samples are given significant weight in reconstruction, resulting in sharper and eventually antialiased images. Where the scene is dynamic, more recent samples are emphasized, resulting in less sharp but more up-to-date images. We also use sample reprojection to improve reconstruction and guide sampling toward occlusion edges, undersampled regions, and specular highlights. In simulation our frameless renderer requires an order of magnitude fewer samples than traditional rendering of similar visual quality (as measured by RMS error), while introducing overhead amounting to 15% of computation time.

Keywords: I.3.3 [Computer Graphics]: Picture-Image Generation—Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics And Realism—Raytracing; Virtual reality.

Technical Report NWU-CS-05-07
Northwestern University, Evanston, IL 60201

Adaptive Frameless Rendering

Abhinav Dayal[1], Cliff Woolley[2], Benjamin Watson[1] and David Luebke[2]
[1]Northwestern University, [2]University of Virginia

Abstract
We propose an adaptive form of frameless rendering with the potential to dramatically increase rendering speed over conventional interactive rendering approaches. Without the rigid sampling patterns of framed renderers, sampling and reconstruction can adapt with very fine granularity to spatio-temporal color change. A sampler uses closed-loop feedback to guide sampling toward edges or motion in the image. Temporally deep buffers store all the samples created over a short time interval for use in reconstruction and as sampler feedback. GPU-based reconstruction responds both to sampling density and space-time color gradients. Where the displayed scene is static, spatial color change dominates and older samples are given significant weight in reconstruction, resulting in sharper and eventually antialiased images. Where the scene is dynamic, more recent samples are emphasized, resulting in less sharp but more up-to-date images. We also use sample reprojection to improve reconstruction and guide sampling toward occlusion edges, undersampled regions, and specular highlights. In simulation our frameless renderer requires an order of magnitude fewer samples than traditional rendering of similar visual quality (as measured by RMS error), while introducing overhead amounting to 15% of computation time.

Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture-Image Generation—Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics And Realism—Raytracing; Virtual reality

1. Improving Interactive Rendering

In recent years a number of traditionally offline rendering algorithms have become interactive or nearly so. The introduction of programmable high-precision graphics processors (GPUs) has drastically expanded the range of algorithms that can be employed in real-time graphics; meanwhile, the steady progress of Moore's Law has made techniques such as ray tracing, long considered a slow algorithm suited only for offline realistic rendering, feasible in real-time rendering settings [WDB*03]. These trends are related; indeed, some of the most promising interactive global illumination research performs algorithms such as ray tracing and photon mapping directly on the GPU [PBMH02]. Future hardware should provide even better support for these algorithms, bringing us closer to the day when ray-based algorithms are an accepted and powerful component of every interactive rendering system.

What makes interactive ray tracing attractive? Researchers in the area have commented on ray tracing's ability to model physically accurate global illumination phenomena, its easy applicability to different shaders and primitives, and its output-sensitive running time, which is only weakly dependent on scene complexity [WPS*03]. We focus on another unique capability: selective sampling of the image plane. By design, depth-buffered rasterization must generate an entire image at a given time, but ray-tracing can focus rendering with very fine granularity. This ability enables a new approach to rendering that is both more interactive and more accurate.

The topic of sampling in ray tracing may seem nearly ex-

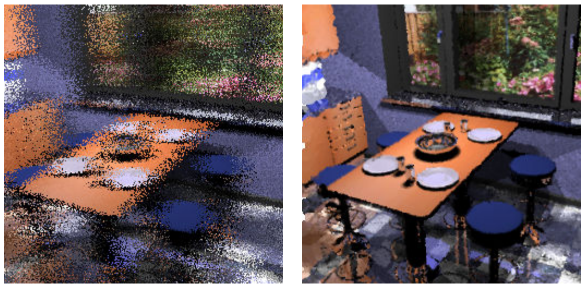


Figure 1: *Adaptive frameless rendering improves upon frameless rendering [BFMS94] (left) with adaptive sampling and reconstruction (right). Resulting imagery has similar visual quality to a framed renderer but is produced using an order of magnitude fewer samples per second.*

hausted, but almost all previous work has focused on *spatial sampling*, or where to sample in the image plane. In an interactive setting, the question of *temporal sampling*, or when to sample with respect to user input, becomes equally important. Temporal sampling in traditional graphics is bound to the frame: an image is begun in the back buffer incorporating the latest user input, but by the time the frame is swapped to the front buffer for display, the image reflects stale input. To mitigate this, interactive rendering systems increase the frame rate by reducing the complexity of the scene, trading off fidelity for performance.

In this paper we investigate novel sampling schemes for managing the fidelity-performance tradeoff. Our approach has two important implications. First, we advocate *adaptive temporal sampling*, analogous to the adaptive spatial sampling long employed in progressive ray tracing [BFGS86; M87; PS89]. Just as spatially adaptive renderers display detail *where* it is most important, temporally adaptive sampling displays detail *when* it is most important. Second, we advocate *frameless rendering* [BFMS94], in which samples are located freely in space-time rather than placed at regular temporal intervals forming frames, and with images reconstructed from a sampled space-time volume, rather than a coherent temporal slice. Frameless rendering decouples spatial and temporal sampling, enabling adaptive spatial and temporal sampling.

Our prototype adaptive frameless renderer consists of four primary subsystems. An *adaptive sampler* directs rendering to image regions undergoing significant change (in space and/or time). The sampler produces a stream of samples scattered across space-time; recent samples are collected and stored in *two* temporally *deep buffers*. One of these buffers provides feedback to the sampler, while the other serves as input to an *adaptive reconstructor,* which repeatedly reconstructs the samples in its deep buffer into an image for display, adapting the reconstruction filters to local sampling density and color gradients. Where the displayed scene is static, spatial color change dominates and older samples are given significant weight in reconstruction, resulting in sharper images. Where the scene is dynamic, only more recent samples are emphasized, resulting in a less sharp but correctly up-to-date image.

We describe an interactive system built on these principles, and show in simulation that this system achieves superior rendering accuracy and responsiveness. We compare our system's imagery to the imagery that would be displayed by a hypothetical zero-delay, antialiased renderer using RMS error. Our system outperforms not only frameless sampling (Figure 1), but also equals the performance of a framed renderer sampling 10 times more quickly.

2. Related work

Bishop et al.'s frameless rendering [BFMS94] replaces the coherent, simultaneous, double-buffered update of all pixels with samples distributed stochastically in space, each representing the most current input when the sample was taken. Pixels in a frameless image therefore represent many moments in time. Resulting images are more up-to-date than double-buffered frames, but temporal incoherence causes visual artifacts in dynamic scenes.

Inspired by frameless rendering, other researchers examined the loosening of framed sampling constraints. The just in time pixels scheme [OCMB95] takes a new temporal sample for each scanline. The address recalculation pipeline [RP94] sorts objects into several layered frame buffers refreshed at different rates. The Talisman system [TK96] renders portions of the 3D scene at different rates. Ward and Simmons [WS99] and Bala et al. [BDT99] store and reuse previously rendered rays. In work that is particularly relevant here, several researchers have studied sample reprojection, which reuses samples from previous frames by repositioning them to reflect the current viewpoint. Walter et al.'s Render Cache [WDP99; WDG02] reconstructs these temporally incoherent samples using depth comparisons and filtering that span small pixel neighborhoods. New samples are guided toward regions that have not been recently sampled, are sparsely sampled, or contain temporal color discontinuities. Simmons and Séquin [SS00] use a hardware interpolated 2.5D mesh to cache and reconstruct the samples, and guide new samples toward spatial color and depth discontinuities. Tolé et al.'s Shading Cache [TPWG02] stores samples in the 3D scene itself, performing reconstruction by rendering that scene in hardware. Sampling is biased toward spatial color discontinuities and toward specular and moving objects. Havran et al. [HDM03] calculate the temporal intervals over which a given sample will remain visible in an offline animation and reproject that sample during the interval. Shading is recalculated for reprojected samples in every frame. Although the images they produce combine samples created at many different moments, all of these systems sample time at regular intervals.

Woolley et al. [WLWD03] describe a fully framed but temporally adaptive sampling scheme called interruptible rendering. The approach adaptively controls frame rate to minimize simultaneously the error created by reduced rendering fidelity and by reduced rendering performance. A progressive renderer refines a frame in the back buffer until the error created by unrepresented input exceeds the error caused by coarse rendering. At that point, the front and back buffers are swapped and rendering begins again into the back buffer using the most recent input. Coarse, high frame-rate display results when input is changing rapidly, and finely detailed, low frame rate display when input is static.

Many advances in high-speed ray tracing have been made recently. These include clever software techniques to im-

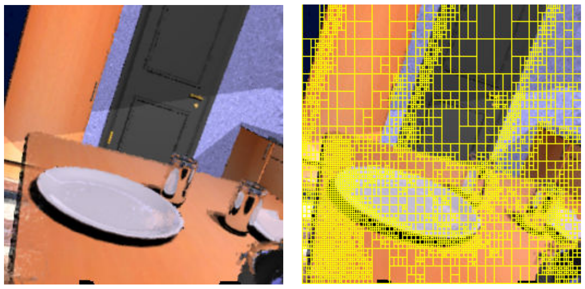


Figure 2: *A reconstructed image and an overlay showing the tiling used by the sampler at that moment in time. Note the finer tilings over object edges and occlusions.*

prove memory locality [PKGH97; TA98; WBWS01], as well as advances in hardware that enable interactive ray tracers on supercomputers [PMS*99], on PC clusters [WSB01; WBDS03], on the SIMD instruction sets of modern CPUs [WBWS01], and on graphics hardware [PBMH02; CHH02]. Wald et al. provide a good summary of the state of the art [WPS*03]. These advances will soon allow a very fine-grained and selective space-time sampling, in real time.

This real-time, selective sampling enables a new adaptive form of frameless rendering that incorporates techniques from adaptive renderers, reprojecting renderers, non-uniform reconstruction [M87], and GPU programming. The resulting system outperforms framed and traditional frameless renderers and offers the following advantages over reprojecting renderers:

Improved sampling response. Rather being clustered at each frame time, samples reflect the most up-to-date input available at the moment they are created. Further, closed-loop control guides samples toward not only spatial but temporal color discontinuities at various scales. These elements combine to reduce rendering latency.

Improved reconstruction. Rather than being non-adaptive or hardware-interpolated, reconstruction is adaptive over both space and time, responding to local space-time color gradients. This drastically improves image quality, eliminating the temporal incoherence in traditional frameless imagery without requiring framed sampling and its increased latency, and permitting antialiasing in static image regions. Moreover this reconstruction is already interactive and implemented on existing GPU hardware.

3. Adaptive frameless sampling

Previous importance sampling techniques [BFGS86; G95; M87; PS89] are spatially adaptive, focusing on regions where color changes across space. Our renderer is both spatially and temporally adaptive, focusing also on regions where color changes over time (Figure 2). Adaptive bias is added to sampling with the use of a spatial hierarchy of image-space *tiles*. However, while previous methods operated in the static context of a single frame, we operate in a dynamic frameless context. This has several implications. First, rather than operating on a frame buffer, we send samples to two temporally *deep buffers* that collect samples scattered across space-time (one buffer for the sampler, one for the reconstructor). Our tiles therefore partition a space-time volume using planes parallel to the temporal axis. We call each resulting subvolume a *block*. Second, as in framed schemes, color variation within each tile guides rendering bias, but variation represents change over not just space but also time. Moreover, variation does not monotonically decrease as the renderer increases the number of tiles, but rather constantly changes in response to user interaction and animation. Therefore the hierarchy is also constantly changing, with tiles continuously merged and split in response to dynamic changes in the contents of the deep buffer.

```
fill deep buffers non-adaptively
loop
    choose a tile to render and a pixel within it
    find last location sampled in pixel
    complete a crosshair of samples at last location
    update deep buffers and tile statistics
    repeat 5 times
        choose a tile crosshair and reproject it
        reevaluate tile gradients in crosshair
        check visibility of crosshair center sample
        if occluded then create new crosshair at same location
        update deep buffers and tile statistics
    end repeat
    choose a different pixel in tile to sample
    create sample, update last location sampled in pixel
    update deep buffers and tile statistics
    if one display time elapsed
        then send reconstructor view and tile information
    if another chunk of crosshairs has been completed
        then adjust tiling
end loop
```

Figure 3: *Pseudocode for the main loop in the sampler.*

The sampler's deep buffer provides it with important feedback. This deep buffer is a 3D array sized to match the number of image pixels in two dimensions, and a shallow buffer depth b in the third, temporal dimension (we use $b = 4$). Buffer entries at each pixel location form a queue, with new samples inserted into the front causing the removal of the sample in the back if the queue is full. Each sample is also sent to the reconstructor's buffer as soon as it arrives in the sampler's buffer, and is described by its color, position in world space, age, and a view-independent velocity vector. In addition to filling the reconstructor's deep buffer, the sampler sends the reconstructor regular updates describing the current view and tiling. This information is sent to the reconstructor 60 times per second, and includes each tile's image coordinates as well as the average temporal and spatial color gradients in the tile's block.

We implement our sampler's tiling hierarchy using a K-D tree. Given a target number of tiles, the tree is managed to

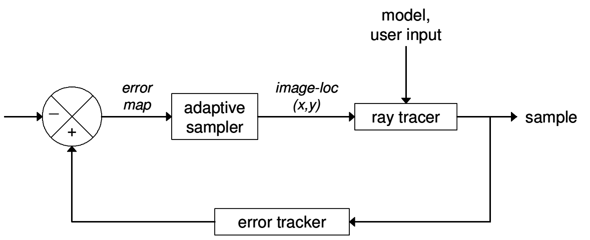


Figure 4: *Adaptive frameless sampling as closed loop control. Samples from the ray tracer (plant) are sent to an error tracker, which adjusts the tiling or error map. The adaptive sampler (compensator) then selects one location to render in a tile. Constantly changing user input (disturbance) makes it very difficult to track and limit error.*

ensure that the amount of color variation in each tile's block is roughly equal: the tile with the most color variation is split and the two tiles with the least summed variation are merged, until all tiles have roughly equal variation. We calculate variation across all of a tile's samples using the equation $v_{tile} = 1/n\ \Sigma_i\ (L_i - L_m)^2$, where L_i is a sample's luminance and L_m the mean luminance in the tile. We ensure prompt response to changes in scene content by weighting samples in the variance calculation using a function that declines exponentially as sample age increases. As a result, small tiles are located over dynamic or finely detailed buffer regions, while large tiles emerge over static or coarsely detailed regions (Figure 2). The tiling is updated after a *chunk* of *c* new samples has been generated (we set *c* = 150).

Sampling now becomes a biased probabilistic process (Figure 3). Since the current time is not fixed as it would be in a framed renderer, we cannot just iteratively sample the tile with the most variation—in doing so, we would overlook newly emerging motion and detail. At the same time, we cannot leave rendering unbiased and unimproved. Our solution is to select each tile with equal probability and select the sampled location within the tile using a uniform distribution. Because tiles vary in size, sampling is biased towards those regions of the image which exhibit high spatial and/or temporal variance. Because all tiles are sampled, we remain sensitive to newly emerging motion and detail.

This sampler thus constitutes a closed-loop control system [DTB97], capable of adapting to user input with great flexibility (Figure 4). In control theory, the *plant* is the process being directed by the *compensator*, which must adapt to external *disturbance*. Output from the plant becomes input for the compensator, closing the feedback loop. In a classic adaptive framed sampler, the compensator chooses the rendered location, the ray tracer is the plant that must be controlled, and disturbance is provided by the scene as viewed at the time being rendered. Our frameless sampler faces a more difficult challenge: view and scene state may change after each sample.

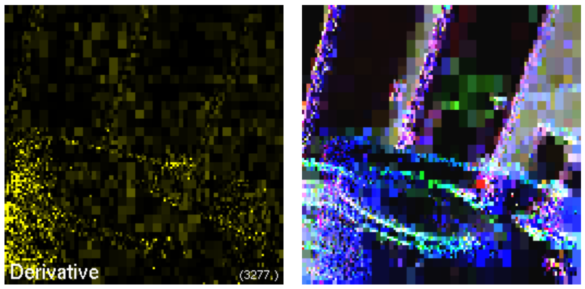


Figure 5: *Error derivatives (left) and the tile gradients (right) G_x, G_y, and G_t (shown here as red, green, and blue, respectively) in a scene corresponding to Figure 2.*

To meet this challenge, we apply two control engineering techniques. We first use a PD controller, in which control responds not only in proportion to error itself (*P*), but also to its derivative (*D*). In our sampler, error is color variation, and by biasing sampling toward variation, we are already responding in proportion to it. By responding to error's derivative, we bias sampling toward regions in which variation is changing such as the edge of the moving table in Figure 2, compensating for delay in our control system. We accomplish this by tracking variation change *d* and adding it to normalized variation *p* to form a new summed control error *e* in the tile:

$$e = kp + (1\text{-}k)d,$$

where

$$p = {v_{tile}\, s} \Big/ {\textstyle\sum_j^{|tiles|} v_j}$$

is the temporally weighted color variance in the tile block v_{tile} normalized by the sum of these variances over all tiles scaled by the tile size *s*, *d* is the absolute difference between *p*'s current value and its value *u* updates of the tile ago (we use *u* = 4) divided by the time between those updates, and *k* in the range [0,1] is the weight applied to the proportional term. The left image in Figure 5 visualizes *d* for each tile by mapping high *d* values to brighter colors.

Our prototype adaptive sampler will be less effective when the rendered scene is more dynamic, changing the desired image (or *target signal* in control theory) more rapidly. In such cases, fixed delays in response will make control increasingly ineffective. To address this problem we apply another control engineering technique: adjusting *gain*. We implement this by restricting or increasing the ability of the sampler to adapt to deep buffer content. Specifically, we adjust the number of tiles onscreen so that color change over space and time are roughly equal in all tiles by ensuring that $dC/ds\ S = i\ dC/dt\ T$, where dC/ds and dC/dt are spatial and temporal color gradients averaged over the entire image (Figure 5), *S* is the average width of the tiles, *T* the average age of the samples in each tile, and *i* is a constant adjusting the relative importance of temporal and spatial change in control. By solving for *S* we can derive the appropriate number of tiles.

We find current spatial gradients by sampling five tightly clustered image locations (xy, $x_{\pm1}y$ and $xy_{\pm1}$) in a crosshair pattern each time we add samples to the deep buffers, and averaging the horizontal and vertical absolute differences. To find a temporal gradient, we find the absolute difference between the center *xy* sample and a sample made at the same location the last time we visited the same pixel region, and divide by the time elapsed since that previous sample was made. We then produce and store a sample at a new location in the pixel region for pairing with the next crosshair made in the pixel region. This set of six samples forms a single entry in the *xy* queue of the sampler's deep buffer (they are not grouped when sent to the reconstructor's deep buffer). To determine average tile gradients, we reduce the weight of each sample gradient as a function of time using the same exponential scheme used to track color variation.

To permit antialiasing, we center sample crosshairs at random spatial locations. However when the scene is particularly dynamic and spatial sampling density is decreased, sharp edges may appear to "shimmer" in reconstructed imagery. Although adaptive reconstruction reduces these artifacts, we eliminate them by randomizing crosshair location only when the scene is locally static and temporal gradients approach zero.

4. Interactive space-time reconstruction

Frameless sampling strategies demand a rethinking of the traditional computer graphics concept of an "image", since at any given moment the samples in an image plane represent many different moments in time. The original frameless work [BFMS94] simply displayed the most recent sample at every pixel. This *traditional reconstruction* results in a noisy image that appears to sparkle when the scene is dynamic (see Figure 1). In contrast, we convolve the frameless samples in the reconstructor's temporally deep buffer with space-time filters to continuously reconstruct images for display. This is similar to the classic computer graphics problem of reconstruction of an image from non-uniform samples [M87], but with a temporal element: since older samples may represent "stale" data, they are treated with less confidence and contribute less to nearby pixels than more recent samples. The resulting images greatly improve over traditional reconstruction (see again Figure 1).

4.1. Choosing a filter

The key question is what shape and size filter to use. A temporally narrow, spatially broad filter (i.e. a filter which falls off rapidly in time but gradually in space) will give very little weight to relatively old samples, emphasizing the newest samples and leading to a blurry but very current image. Such a filter provides low-latency response to changes and should be used when the underlying image is changing rapidly. A temporally broad, spatially narrow filter will give nearly as much weight to relatively old samples as to recent samples; such a filter accumulates the results of many samples and leads to a finely detailed, antialiased image when the underlying scene is changing slowly. However, often different regions of an image change at different rates, as for example in a stationary view in which an object is moving across a static background. A scene such as this demands spatially adaptive reconstruction, in which the filter extent varies across the image. What should guide this process?

We use local sampling density (Figure 7) and space-time gradient information (Figure 5) to guide filter size. The reconstructor maintains an estimate of local sampling density across the image, based on the overall sampling rate and on the tiling used to guide sampling. We size our filter support—which can be interpreted as a space-time volume—as if we were reconstructing a regular sampling with this local sampling density, and while preserving the total volume of the filter, perturb the spatial and temporal filter extents according to local gradient information. A large spatial gradient implies an edge, which should be resolved with a narrow filter to avoid blurring across that edge. Similarly, a large temporal gradient implies a "temporal edge" such as an occlusion event, which should be resolved with a narrow filter to avoid including stale samples from before the event. This is equivalent to an "implicit" robust estimator; rather than searching for edges explicitly, we rely on the gradient to allow us to size the filter such that the expected contribution of samples past those edges is small.

Thus, given a local sampling rate R_l, expressed in samples per pixel per second, we define V_S as the expected space-time volume occupied by a single sample:

$$V_S = \frac{1}{R_l}.$$

The units of V_S are pixel-seconds per sample (note that the product of pixel areas and seconds is a volume). We then construct a filter at this location with space-time support proportional to this volume. For simplicity we restrict the filter shape to be axis-aligned to the spatial x and y and the temporal t dimensions. The filter extents e_x, e_y, and e_t are chosen to span equal expected color change in each dimension, determined by our estimates of the gradients G_x, G_y, and G_t and the total volume constraint V_s:

$$e_x G_x = e_y G_y = e_t G_t$$

$$V_S = e_x e_y e_z .$$

Thus the filter extents are given by

$$e_x = \sqrt[3]{\frac{V_S G_y G_t}{G_x^2}}, e_y = \sqrt[3]{\frac{V_S G_x G_t}{G_y^2}}, e_t = \sqrt[3]{\frac{V_S G_x G_y}{G_t^2}} .$$

What function to use for the filter kernel remains an open question. According to signal theory, a regularly sampled, band limited function should be reconstructed with a sinc function, but our deep buffer is far from regularly sampled

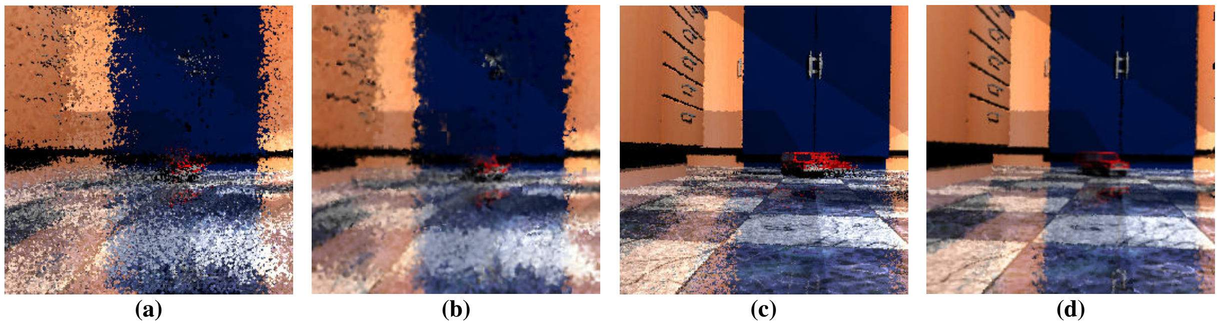


Figure 6: *Adaptive reconstruction illustrated in one moment of a scene with a moving view and car, sampled using our adaptive frameless techniques. In (a), traditional frameless reconstruction leaves many artifacts of the view motion in the image. In (b), adaptive reconstruction rejects many of the outdated samples, eliminating artifacts and clarifying edges. (c) shows the improvements possible by reprojecting samples as in [WDP99], even without adaptive reconstruction. When reprojection is combined with adaptive reconstruction as in (d), the car's motion and view-dependent reflections in the floor are clarified.*

and the underlying signal (an image of a three-dimensional scene) contains high-frequency discontinuities such as occlusion boundaries. We have experimented with a range of filters. Box and tent filters have poor bandpass properties but are extremely cheap to evaluate. A gaussian filter looks better but also requires more computation. The Mitchell-Netravali filter [M87] is considered among the best filters for nonuniform sampling, but is still more costly and requires more precision than is provided by our 16-bit GPU implementation. We have also experimented with a simple inverse exponential filter, which has the nice temporal property that the relative contribution of two samples does not change as both grow older; however, the bandpass properties of this filter are less than ideal. We are currently using a gaussian filter.

4.2. Scatter versus gather

We can consider reconstruction a *gather* process which loops over the pixels, looks for samples in the neighborhood of each pixel, and evaluates the contribution of those samples to that pixel. Alternatively, we can cast reconstruction as a *scatter* process which loops over the samples, projects each onto the image plane, and evaluates its contribution to all pixels within some footprint. We have experimented with both approaches.

We implemented the reconstructor initially as a gather process directly on the sampler's deep buffer. At display time the reconstructor looped over the pixels, then adjusted the filter size and extents at each pixel using gradient and local sample density as described above. The reconstructor gathered samples outwards from each pixel in space and time until the maximum possible incremental contribution of additional samples would be less than some threshold ε. The final color at that pixel was computed as the normalized weighted average of sample colors. This process proved expensive in practice—our unoptimized simulator required reconstruction times of several hundred ms for small (256×256) image sizes. It was also unclear how to efficiently implement hardware sample reprojection

We have therefore moved to a scatter-based implementation that stores the N most recent samples produced by the sampler across the entire image; the value of N is typically at least 4× the desired image resolution. This store is a distinct deep buffer for the reconstructor that organizes the samples as a single temporally ordered queue rather than a spatial array of crosshairs. At reconstruction time, the system splats each of these samples onto the image plane and evaluates the sample's affect on every pixel within the splat extent by computing the distance from the sample to the pixel center and weighting the sample's color contribution according to the local filter function. These accumulated contributions are then divided by the accumulated weight at each pixel to produce the final image (Figure 6).

We implement this scatter approach on the GPU, achieving real-time or near real-time performance and improving on the speed of our CPU-based gather implementation by almost two orders of magnitude. The GPU treats the samples in the deep buffer as vertices in a vertex array, and uses an OpenGL vertex program to project them onto the screen as splats (i.e., large GL_POINTS primitives). A fragment program runs at each pixel covered by a sample splat, finding the distance to the sample and computing the local filter shape by accessing tile information—local filter extent, precomputed from sample density and G_x,G_y,G_t gradients—stored in a texture. This texture is periodically updated by rasterizing the latest tiling (provided by the sampler) as a set of rectangles into an offscreen buffer. To reduce overdraw while still providing broad filter support in sparsely sampled regions, the vertex program rendering the samples adaptively adjusts point size. Section 5.2 describes this process in more detail.

The reconstructor uses several features of recent graphics hardware, including floating-point textures with blend support, multiple render targets, vertex texture fetch, dynamic branching in vertex programs, and separate blend functions for color and alpha. The results presented in this paper were obtained on a NVIDIA GeForce 6800 Ultra. In general, reconstruction occurs at 20 Hz rates when keeping a visually sufficient number of samples *N* (*N*=400K). This is remarkable considering that our use of the hardware differs greatly from the rendering task the hardware was designed to support.

5. Reprojection

Our adaptive frameless sampling and reconstruction techniques operate entirely in 2D image space and do not rely on information about sample depth or the 3D structure of the scene. However, because camera location and sample depth are easily available from our ray-tracing renderer, we also incorporate sample reprojection [WDP99; WDG02; BDT99; WS99] into our algorithms. During sampling, reprojection can help the sampler find and focus on image regions undergoing disocclusion, occlusion, view-dependent lighting changes, or view-independent motion. During reconstruction, sample reprojection extends the effective "lifetime" of a sample by allowing older samples to contribute usefully to imagery even after significant camera or object motion. This section describes our strategies for using reprojection with our sampler and reconstructor.

5.1. Reprojection in the sampler

It is not necessary to reproject every sample at fixed intervals, and indeed this would not be desirable since it would introduce periodicity into our frameless sampler. Instead, we reproject a small number of recent samples as we generate each new sample. When updates of tiling statistics (e.g. variation, gradients) are included, reprojecting a sample takes roughly 1/35th the mean time required to generate a new sample. We therefore reproject a small number (currently 5) of a tile's crosshairs each time the sampler visits a tile. In this way the same useful rendering bias that guides generation of new samples determines which samples are reprojected, focusing reprojections on important image areas.

Within each tile, we choose the corresponding pixels to reproject randomly and relocate the crosshairs from the front of each pixel's queue in the deep buffer. We apply both motion and viewing transformations to the samples in the crosshair. Despite being updated in some sense, reprojected samples continue to age normally and do not receive increased weight in variance and gradient calculations. On a local per-tile basis, every sample is treated similarly, and its age remains a good indicator of its utility. We determine a crosshair's new location in the buffer solely by its relocated center sample, and insert the crosshair sample at the back of its new queue, updating source and destination tile statistics if necessary. When updating tile gradients, spatial gradients for the crosshair are recalculated using the new spatial locations of the crosshair samples (after reprojection, they are no longer separated by one pixel length). We recalculate the crosshair temporal gradients by finding the absolute difference between the reprojected center sample and the newest sample in that pixel region, and dividing this difference by the age of this newest sample. Reprojection of a crosshair continues until it ages so much that it no longer affects tile variance and gradients, or until it is pushed out of its queue in the deep buffer by a newly arriving crosshair.

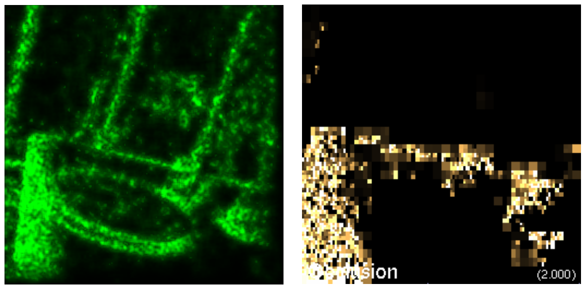


Figure 7: *A sample density map (left) used by the reconstructor to determine the expected local sample volume V_s, and occlusion detection (right) used direct sampling.*

Regions containing disocclusions will be undersampled as samples reproject to other image locations. We bias sampling toward these disocclusions with a new undersampling measure u_{tile}:

$$u_{tile} = 1 - \min\left(1, \frac{m\sum_{j=1}^{|tiles|}(whb - |buffer|) \big/ |tiles|}{sb - |tile|}\right).$$

Here the number of empty samples in a tile must be *m* times greater than the mean number of empty samples in all tiles to affect sampling. |*buffer*| and |*tile*| are the number of samples in the deep buffer and the current tile's block, while *whb* is the number of samples the deep buffer can hold (with image size *w*×*h*).

Regions undergoing occlusion will contain samples from multiple surfaces at differing view depths, leading to uncertainty about image content. To resolve this uncertainty, we increase sampling in occluded regions. We detect occlusions by casting rays from the eye to each reprojected sample. If the sample is no longer visible from the eye, we add a new sample at the occluded image location. We also increase sampling density in the occluded region by increasing error in tiles experiencing occlusion with an occlusion term o_{tile} = |*O*|/*sb*, where |*O*| is the number of occluded samples in a tile's block, tracked by our occlusion test. Figure 7 shows the occlusions that affect sampling. Proportional error *p* for the sampler then becomes:

$$p = s\left(\kappa^{v_{tile}} \Big/ \sum_j^{|tiles|} v_j + \lambda^{u_{tile}} \Big/ \sum_j^{|tiles|} u_j + (1-\kappa-\lambda)^{o_{tile}} \Big/ \sum_j^{|tiles|} o_j\right) \text{w}$$

ith κ, λ, and (κ + λ) all in [0,1].

5.2. Reprojection in the reconstructor

Unlike the sampler, the reconstructor operates in a framed context: to display an image on existing hardware, it scans out a traditional image (i.e., a uniform grid of pixels) at the regular intervals of the display refresh. Since each sample in the reconstructor's deep buffer stores the 3D hit point of the primary ray that generated that sample, reprojecting and reconstructing each of our renderer's images reduces to rendering the vertex array in the deep buffer with the current camera and projection matrices bound. Figure 6 shows the results of using reprojection in reconstruction.

Reprojection sometimes generates regions of low sample density, for example at disocclusions and near the leading edge of the screen during camera rotation. In such regions, the filter support for the few samples present must be quite large, requiring the reconstructor to rasterize samples with large splats. Rather than rasterizing all samples using large splats, we avoid overdraw with an adaptive point size scheme. All samples are accumulated into a *coverage map* during rendering that tracks the number and average splat size of all samples rendered to each pixel. To size splats, the sample vertex program binds the previous image's coverage map as a texture, computes the projected coordinates of the sample, and uses the coverage information at those coordinates to calculate the splat size at which the sample will be rasterized. Sample splats in a region are sized according to the average point size used in that region during reconstruction of the previous image, but point sizes in undersampled regions (defined currently as fewer than 4 samples affecting a pixel) are multiplied by 4 to grow rapidly, while point sizes in oversampled regions (more than 32 samples reaching a pixel) are multiplied by 0.7 to shrink gradually.

6. Evaluation

Using the *gold standard* validation described in [WLWD03], we find that our adaptive frameless renderer consistently outperforms other renderers that have the same sampling rates.

Gold standard validation uses as its standard an *ideal renderer I* capable of rendering antialiased imagery in zero time. To perform comparisons to this standard, we create n ideal images I_j (j in $[1,n]$) at 60 Hz for a certain animation A using a simulated ideal renderer. We then create n more images R_j for animation A using an actual interactive renderer R. We next compare each image pair (I_j,R_j) using an image comparison metric *comp*. Here we use root-mean-squared error (RMS).

We report the results of our gold standard evaluation in Table 1, which compares several rendering methods producing 256x256 images using various sampling rates. Two framed renderings either maximize Hz at the cost of spatial resolution (lo-res), or spatial resolution at the cost of Hz (hi-res).

Render Method	Animation/Sampling rate						
	Interactive			Bart		Toycar	
	100k	400k	800k	400k	800k	400k	800k
Framed: lo-res	92.7	71.8	60.9	112	100	47	42.8
Framed: hi-res	110	72.6	60.4	127	112	43.8	38.9
Traditional Frameless	80.8	48.8	39.3	92.3	74.8	35.3	32.5
Adaptive	34.4	24.1	23.6	50.1	51.9	20.4	18.5
hi-res 60Hz	28	28	28	30.7	30.7	29.4	29.4

Table 1: *Summary error analysis using the techniques of Figure 8, with some additional sampling rates.*

The traditional frameless rendering simply displays the newest sample at a given pixel. The adaptive frameless renderings use our system to produce the imagery. The hi-res 60Hz is a framed renderer that uses a sampling rate 10 times higher than other renderers to produce full resolution imagery at 60Hz. (The difference between the ideal renderer and this hi-res 60Hz renderer is that the latter suffers from double-buffering delay and does not use anti-aliasing). Rendering methods were tested in 3 different animations, all using the publicly available BART testbed [LAM00]: the viewpoint animation in the testbed itself (BART); a fixed viewpoint close-up of a moving car (toycar), and a recording of user viewpoint interaction (interactive).

Adaptive frameless rendering is the clear winner, with lower RMS error than all techniques using the same sampling rate and comparable error to the hi-res 60Hz rendering, which uses sampling rates 40, 10 and 5 times faster than the 100K, 400K and 800K adaptive frameless renderings.

Figure 8 offers a more detailed view that confirms this impression. The graphs here show frame-by-frame RMS error comparisons between several of these rendering techniques and the ideal rendering. Note the sawtooth pattern produced by the low-sampling rate hi-res renderer, due to double buffering error. In the interactive animation, the periodic increases in error correspond to periods of viewpoint change. Once more, adaptive frameless rendering has lower RMS error than all rendering techniques using equivalent sampling rates, and comparable error to the much more densely sampled hi res 60Hz renderer. The top right graph also depicts the advantage of using reprojection in the sampler. RMS error is considerably higher if reprojection is not used.

7. Discussion and future work

Frameless rendering and selective sampling have been criticized for sacrificing spatial coherence and thus memory locality, which can reduce sampling speed. We plan to experiment with increases in the number of samples we generate each time we visit a tile, increasing spatial coherence at the cost of slightly less adaptive sampling overall. However, exploiting spatial coherence has its limits: ultimately, it will limit our ability to take advantage of temporal coherence and force us to sample more often. Traditional renderers must sample every single pixel dozens of times each second; as displays grow in size and resolution, this ceaseless sampling

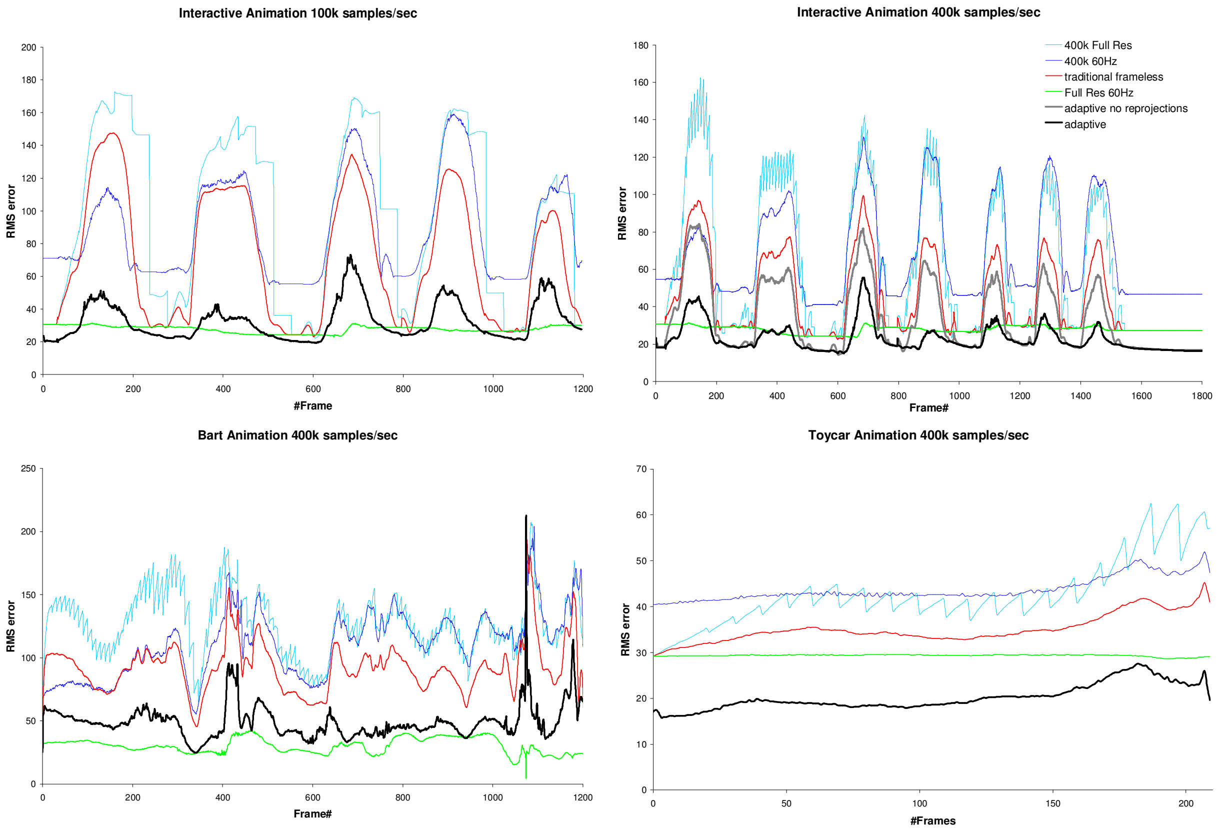


Figure 8: *An error analysis of rendering techniques for several animation sequences created using 100K or 400K samples/sec. Graphs show frame-by-frame RMS error between each technique's images and the ideal image that would be displayed by a hypothetical zero-delay, antialiased renderer at the same moment in time. Resolution is 256x256 pixels at 60 Hz.*

becomes wasteful of computation, power, and heat. With this work, we hope to shift the emphasis of interactive ray tracing research from spatial to temporal coherence, and from brute-force to selective sampling.

Good filter design for the adaptive space-time reconstruction of frameless sample streams remains an open problem. We have begun investigating the edge-preserving concepts of bilateral and trilateral filtering [DD02; CT03], which perform nonlinear filtering by weighting samples according to their difference in luminance as well as their distance in space. However, extending these approaches to include a third temporal dimension and to operate on a non-uniformly distributed samples presents a significant challenge. A related possibility is to exploit a priori information about the underlying model or animation, as do Bala et al. [BWG03].

We believe this work has great potential, and will continue this research in several longer-term directions. Extending our temporally adaptive methods to more sophisticated global illumination algorithms is one obvious avenue. With its ability to selectively alter sampling and reconstruction across both space and time, our adaptive frameless renderer is an ideal platform for experimenting with perceptually driven rendering in interactive settings [LRC02]. We are studying the possibility of extremely high resolution ("gigapixel") display hardware fed streams of frameless samples, with adaptive reconstruction performed in the display itself. This might be one solution to the immense bandwidth challenge posed by such displays. Such a rendering configuration would also enable a truly asynchronous parallelism in graphics, since renderers would no longer have to combine their samples into a single frame [MCEF94]. For this reason we are particularly interested in implementing these algorithms in graphics hardware.

8. Conclusion

In conclusion, we advocate a revival of frameless rendering, based on temporally adaptive sampling and reconstruction, and enabled by recent advances in interactive ray tracing. This approach improves traditional framed and frameless rendering by focusing sampling on regions of spatial and temporal change, and with adaptive reconstruction that emphasizes new samples when scene content is changing quickly and incorporates older samples when the scene is static. In testing, our prototype system displays greater accuracy than framed and frameless rendering schemes at comparable sampling rates, and comparable accuracy to a framed renderer sampling 10 times more quickly. Based on these results, we believe that a temporally adaptive frameless approach shows great promise for future rendering algorithms and hardware.

9. Acknowledgements

We would like to thank Edward Colgate and Kevin Lynch (Department of Mechanical Engineering, Northwestern University) for their fruitful discussions on control systems and the Saarland University Graphics Group for providing us with the OpenRT ray tracer. This work was supported in part by the NSF grants 0092973, 0093172, 0112937, and 0130869. The 3D model (kitchen) is the courtesy of the BART (A Benchmark for Animated Ray Tracing) project at Chalmers University of Technology, Sweden.

10. References

[BDT99] BALA, K., DORSEY, J., TELLER, S. 1999. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Trans. Graph*, 18, 3, 213-256.

[BWG03] BALA, K., WALTER, B., GREENBERG, D.P. 2003. Combining edges and points for interactive high-quality rendering. *ACM Trans. Graph., 22*, 3, 631–640 (*Proc. ACM SIGGRAPH*).

[BFGS86] BERGMAN, L., FUCHS, H., GRANT, E., SPACH, E. 1986. Image rendering by adaptive refinement. *Proc. ACM SIGGRAPH*, 29–37.

[BFMS94] BISHOP, G., FUCHS, H., MCMILLAN, H., SCHER ZAGIER, E.J. 1994. Frameless rendering: double buffering considered harmful. *Proc. ACM SIGGRAPH*, 175–176.

[CHH02] CARR, N.A., HALL, J.D., HART, J.C. 2002. The ray engine. *Proc. ACM SIGGRAPH/Eurographics Graphics Hardware*, 37–46.

[CT03] CHOUDHURY, P., TUMBLIN, J. 2003. The trilateral filter for high contrast images and meshes. *Proc. Eurographics Workshop on Rendering*, 186–196.

[DD02] DURAND, F., DORSEY, J. 2002. Fast bilateral filtering for the display of high-dynamic-range images. *ACM Trans. Graphics, 21*, 3, 257–266 (*Proc. ACM SIGGRAPH*).

[DTB97] DUTTON, K., THOMPSON, S., BARRACHLOUGH, B. 1997. *The Art of Control Engineering, 1st ed.* Addison-Wesley.

[G95] GLASSNER, A. 1995. *Principles of Digital Image Synthesis, 1st ed.* Morgan Kaufmann.

[HDM03] HAVRAN, V., DAMEZ, C., MYSZKOWSKI, K. 2003. An efficient spatio-temporal architecture for animation rendering. *Proc. Eurographics Symposium on Rendering*, 106-117.

[J01] JENSEN, H.W. 2001. *Realistic Image Synthesis Using Photon Mapping*. AK Peters.

[LAM00] LEXT, J., ASSARSSON, U., MOELLER, T. 2000. Bart: A benchmark for animated ray tracing. Tech. Rpt. 00-14, Dept. Computer Engineering, Chalmers Univ. Tech. http://www.ce.chalmers.se/BART.

[LRC*02] LUEBKE, D., REDDY, M., COHEN, J.D., VARSHNEY, A., WATSON, B., HUEBNER, R. 2002. *Level of Detail for 3D Graphics, 1st ed.* Morgan Kaufmann.

[M87] MITCHELL, D.P. 1987. Generating antialiased images at low sampling densities. *Proc. ACM SIGGRAPH*, 65–72.

[MCEF94] MOLNAR, S., COX, M., ELLSWORTH, D., FUCHS, H. 1994. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, *14*, 4, 23–32.

[OCMB95] OLANO, M., COHEN, J., MINE, M., BISHOP, G. 1995. Combatting rendering latency. *Proc. ACM Interactive 3D Graphics*, 19–24.

[PS89] PAINTER, J., SLOAN, K. 1989. Antialiased ray tracing by adaptive progressive refinement. *Proc. ACM SIGGRAPH*, 281–288.

[PMS*99] PARKER, S., MARTIN, W., SLOAN, P.-P.J., SHIRLEY, P., SMITS, B., HANSEN, C. 1999. Interactive ray tracing. *Proc. ACM Interactive 3D Graphics*, 119–126.

[PKGH97] PHARR, M., KOLB, C., GERSHBEIN, R., HANRAHAN, P. 1997. Rendering Complex Scenes with memory-coherent ray tracing. *Proc. ACM SIGGRAPH*, 101– 108.

[PBMH02] PURCELL, T.J., BUCK, I., MARK, W.R., HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Trans. Graphics, 21*, 3, 703–712 (*Proc. ACM SIGGRAPH*).

[RP94] REGAN, M.J.P., POSE, R. 1994. Priority rendering with a virtual reality address recalculation pipeline. *Proc. ACM SIGGRAPH*, 155–162.

[SS00] SIMMONS, M., SÉQUIN, C. 2000. Tapestry: A dynamic mesh-based display representation for interactive rendering. *Proc. Eurographics Workshop on Rendering*, 329–340.

[TA98] TELLER, S., ALEX, J. 1998. Frustum Casting for Progressive, Interactive Rendering. *Massachusetts Institute of Technology Technical Report LCS TR-740.* Available at http://graphics.csail.mit.edu/pubs/MIT-LCS-TR-740.ps.gz

[TPWG02] TOLE, P., PELLACINI, F., WALTER, B., GREENBERG, D.P. 2002. Interactive global illumination in dynamic scenes. *ACM Trans. Graphics, 21*, 3, 537–546 (*Proc. ACM SIGGRAPH*).

[TK96] TORBORG, J., KAJIYA, J. 1996. Talisman: Commodity Reality Graphics for the PC. *Proc. ACM SIGGRAPH*, 353-363.

[WBDS03] WALD, I., BENTHIN, C., DIETRICH, A., SLUSALLEK, P. 2003. Interactive distributed ray tracing on commodity PC clusters—state of the art and practical applications. *Lecture Notes on Computer Science*, *2790*, 499–508 (*Proc. EuroPar*).

[WBWS01] WALD, I., BENTHIN, C., WAGNER, M., SLUSALLEK, P. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, *20*, 153–164 (*Proc. Eurographics*).

[WPS*03] WALD, I., PURCELL, T.J., SCHMITTLER, J., BENTHIN, C., SLUSALLEK, P. 2003. Realtime ray tracing and its use for interactive global illumination. *Eurographics State of the Art Reports*.

[WSB01] WALD, I., SLUSALLEK, P., BENTHIN, C. 2001. Interactive distributed ray tracing of highly complex models. *Proc. Eurographics Workshop on Rendering*, 277– 288.

[WDG02] WALTER, B., DRETTAKIS, G., GREENBERG, D.P. 2002. Enhancing and optimizing the render cache. *Proc. Eurographics Workshop on Rendering*, 37–42.

[WDP99] WALTER, B., DRETTAKIS, G., PARKER S. 1999. Interactive rendering using render cache. *Proc. Eurographics Workshop on Rendering*, 19–30.

[WS99] WARD, G., SIMMONS, M. 1999. The Holodeck Ray Cache: An Interactive Rendering System for Global Illumination in Nondiffuse Environments, *ACM Trans. Graph.* 18, 4, 361-398.

[WLWD03] WOOLLEY, C., LUEBKE, D., WATSON, B.A., DAYAL, A. 2003. Interruptible rendering. *Proc. ACM Interactive 3D Graphics*, 143–151.

Multi-Level Ray Tracing Algorithm

Alexander Reshetov   Alexei Soupikov   Jim Hurley
Intel Corporation

Abstract

We propose new approaches to ray tracing that greatly reduce the required number of operations while strictly preserving the geometrical correctness of the solution. A hierarchical "beam" structure serves as a proxy for a collection of rays. It is tested against a kd-tree representing the overall scene in order to discard from consideration the sub-set of the kd-tree (and hence the scene) that is guaranteed not to intersect with any possible ray inside the beam. This allows for all the rays inside the beam to start traversing the tree from some node deep inside thus eliminating unnecessary operations. The original beam can be further sub-divided, and we can either continue looking for new optimal entry points for the sub-beams, or we can decompose the beam into individual rays. This is a hierarchical process that can be adapted to the geometrical complexity of a particular view direction allowing for efficient geometric anti-aliasing. By amortizing the cost of partially traversing the tree for all the rays in a beam, up to an order of magnitude performance improvement can be achieved enabling interactivity for complex scenes on ordinary desktop machines.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism — Ray Tracing, Global Illumination, Beam Tracing, Geometric Anti-Aliasing.

Keywords: ray-tracing, frustum occlusion culling, anti-aliasing

1 Introduction

A renewed interest in Ray Tracing (RT) algorithms is being fueled by the relentless progress of Moore's law in terms of raw compute power and various algorithmic discoveries resulting in significant performance improvements. This makes real-time Ray Tracing and Global Illumination (GI) attractive for implementation on desktop machines. Some of these new discoveries are summarized by Wald et al. [2003]. Table 1 provides comparison results for our implementation and those described by Wald. We were able to improve performance further by up to an order of magnitude. This is achieved by amortizing per-beam operations which would otherwise be performed for each ray in a group.

e-mail: {Alexander.Reshetov, Alexei.Soupikov, Jim.Hurley}@intel.com

In this paper, we focus on the most fundamental task in ray tracing, namely, finding the intersection of one or more rays with a given geometry. We consider reducing the average number of operations per ray as the most objective metric for comparing different algorithms. Generally speaking, there is no one guaranteed best ray tracing algorithm as performance depends on many factors such as: the overall scene complexity, the current view of the scene, and the characteristics of the host platform. Many of the techniques we use have been known for some time, although not specifically applied to the problem of efficient ray tracing. For example, a modified view frustum culling algorithm is used to reduce redundant operations for big groups of rays. In essence, we're building on top of the work of Heckbert and Hanrahan [1984], Teller and Alex [1998], Assarsson and Möller [2000], Kay and Kajiya [1986], and others. Another motivating factor is to address the issue of geometrical aliasing which is especially pronounced at interactive rates. One way to improve the quality of the resulting image is to cast multiple rays through each pixel. We propose a mechanism for dynamically measuring the geometrical complexity for a given view direction, which can be used for budgeting rays more cost-effectively.

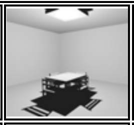
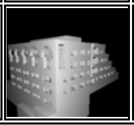
scene # of triangles and shader (+/-)		Framerate (FPS) @ 1024x1024 resolution	OpenRT @ 2.5 GHz P4 1 thread	MLRTA @ 2.4 GHz P4 1 thread	MLRTA @ 3.2 GHz P4 with HT 2 threads
Erw6 804		– shader	7.1	70.2	109.8
		+ shader	2.3	37.8	50.7
Confe- rence 274K		– shader	4.55	11.2	19.5
		+ shader	1.93	9.5	15.6
Soda Hall 2195K		– shader	4.12	21.1	35.5
		+ shader	1.8	15.3	24.1

Table 1: Framerate comparison results for 3 scenes. OpenRT data is taken from Wald et al. [2003]. Two sets of data are provided: one for a null shader, and the other for a simple shader (equivalent to placing a point light at the camera position).

We start by giving a brief overview of related work in section 2. Section 3 introduces the basic concepts which will be used throughout this article. By following relevant prior work, these concepts will be described and then refined and adapted for our approach. Then we describe the complete multi-level ray tracing pipeline in section 4 and discuss results in section 5. We conclude with some considerations for further research ideas in section 6, which will also describe limitations of the proposed method.

1.1 MLRTA Overview

The central contribution of this paper is a new robust approach to high performance ray-tracing which is achieved without any approximations or geometric simplifications. Instead of looking for ray/geometry intersections, we effectively *exclude* certain objects from consideration for any given group of rays. This is accomplished by providing a separate entry point into the global acceleration structure (kd-tree) for each group of rays. Instead of traversing each constituent ray from the top of the tree, we start at the group's entry node inside the tree. This reduces the average number of traversal steps by 2/3^{rds} for typical scenes.

Further insight came from analyzing the frustum culling algorithms used for axis-aligned bounded boxes (AABB) in the context of traditional raster graphics. Here, simple tests are used on AABB proxies for detailed geometries to purge objects which do not intersect the frustum, effectively reducing the amount of geometry submitted for rendering. These tests are designed to quickly detect the majority of cases where objects do not intersect with the view frustum. The objects can be trivially rejected if their AABBs can be separated from the frustum by one of the frustum's planes. This works well in those cases where small AABBs are culled by a relatively big frustum. A comparable approach may be used in ray tracing, where the AABBs represent the volumes associated within a kd-tree's nodes, and the frustum represents a beam of rays. Typically, this frustum will be much smaller than most of the AABBs in the kd-tree. As a result, using the trivial reject algorithm as described above would result in a higher percentage of redundant potential accepts (cases where we cannot trivially exclude the intersection for non-intersecting objects). Conversely, for RT applications we propose reversing the roles of the frustum and AABB by using the AABB faces as separation planes during depth-first traversal of the kd-tree.

Furthermore, we recommend new approaches to the kd-tree building process which make it more suitable for the Multi Level Ray Tracing Algorithm (MLRTA) proposed herein. We also introduce the new concept of an "empty occluder", which is basically a tagged empty box contained inside a watertight object. We use such empty occluders to stop further traversal of beams that completely intersect such occluders.

All these innovations together allow us to improve the performance of the ray-tracing algorithm by up to an order of magnitude compared with previous results in the literature. As an added bonus, the MLRTA provides a natural mechanism for measuring the geometric complexity of the portion of the scene visible to a given group of rays, which enables geometric anti-aliasing.

2 Related Work

A number of researchers have developed strategies for exploiting coherence between spatially adjacent rays. We took inspiration from the early work of Heckbert and Hanrahan [1984], in which polygonal beams were used instead of rays to improve the anti-aliasing properties of an image. The beam was annotated as it intersected with objects in the scene so that the edges of these interfering objects could be more effectively anti-aliased. Similar goals were pursued by different researchers through the introduction of cone tracing [Amanatides 1984], pencil tracing [Shinya at el. 1984], and ray bounds [Ohta and Maekawa 1990].

In [Heckbert and Hanrahan 1984], the beams are persistent with extra information accumulated during tracing to describe multiple beam/object intersections. Later, researchers switched to splitting beams whenever such intersections occurred. In [Ghazanfarpour and Hasenfratz 1998], this happens when a beam intersects multiple objects thus necessitating smaller sub-beams to precisely anti-alias a polyhedral scene. In [Genetti et al. 1998], "pyramidal rays" (pyrays) are split when any part of one intersects an object. Arvo and Kirk [1987] use a volume in a 5D space to represent a collection of rays (3D for origin and 2D for direction). The original 5D volume is then dynamically subdivided into hypercubes, each linked to a set of objects which are candidates for intersection. In [Heckbert and Hanrahan 1984], the beam tree which represents the surfaces intersected by the beam, is computed in object space and then passed to a polygon renderer for scan conversion. In [Teller and Alex 1998], frustum casting is proposed which samples discretely in image space, but operates in object space. In this algorithm, the frustum is recursively subdivided, while object space is organized linearly with indices identifying neighbors of a given current cell. In [Cho and Forsyth 1999], a visibility complex is incrementally constructed enabling efficient ray/geometry queries.

In [Havran and Bittner 2000], Longest Common Traversal Sequences are used to amortize common operations among multiple rays. In [Dmitriev et al. 2004], pyramidal shafts are used for the same purpose. This technique is also extended to secondary and shadow rays. In both of these works the convex hull of a group of rays is represented by a few boundary rays, which are traversed through the scene. It is then assumed that all interior rays will follow the same path. This is not always correct and we will provide examples illustrating this point in section 3.3. We believe that we have come up with the necessary mathematical apparatus which is geometrically accurate and achieves the same goals.

All these algorithms attempt to combine view-dependent culling in object space with some distance-based visibility determination in image space. This is generally achieved by the use of a spatially distributed recursive construct which initially encompasses multiple rays and is then progressively refined. We use the same approach, the fundamental difference between our method and these others is that we are not trying to find objects that this construct intersects. Instead, we eliminate those objects that *do not* intersect with the construct.

A different category of algorithms aims at minimizing the required number of intersection tests by budgeting rays diligently, sampling sparsely in areas of low geometric variation and super-sampling for geometrically complex or perceptually important parts of the image [Ramasubramanian et al. 1999]. Our approach facilitates this type of optimization by providing a natural measure of geometrical complexity for a specific viewpoint.

3 Basic Concepts

3.1 Acceleration Structures

To naively find an intersection of a ray with a scene, one could test this ray against all objects in the scene for an intersection and keep the one with the shortest distance from the ray origin. This algorithm might have the lowest memory footprint, but its execution time is prohibitive. A much better approach would be to organize the scene into some sort of data structure (usually called an "acceleration structure" – AS) and use this structure to zero in on the area of interest in a hierarchical fashion. An AS works by splitting 3D space into subsets containing a certain number of primitive objects (triangles if no other primitives are used). In addition to this spatial organization, specific traversal routines are defined as well. These routines are used to quickly decide which subsets to look into further for possible intersections. Different traversal routines may coexist for the same AS, for example, one for primary and another for shadow rays. One object may belong to multiple subsets and some subsets may be empty. Various types of ASs that lend themselves to different scene geometries and/or computer architectures are well known and described in the literature [Szirmay-Kalos et al. 2002]. In this work we use a kd-tree data structure; a systematic analysis of the kd-tree building algorithm was first given by Glassner [1984] and followed by numerous publications, in particular by Havran [2000].

The main operation in the kd-tree building process is to split an axis-aligned bounding box into two (potentially unequal) boxes by a plane orthogonal to one of the axes. The process is repeated recursively until some termination criteria are met. The splitting algorithm and termination criteria, in essence, define a particular flavor of a kd-tree building algorithm. The split position is chosen by minimizing a cost function over a set of candidate split positions, such as the coordinates of vertices inside the cell and the coordinates of triangle/cell intersections. Following MacDonald and Booth [1990], we use a surface area-based cost function which is computed by multiplying the area of the cell by the number of objects intersecting with it. We have modified the pure area-based approach to bias it in favor of creating large empty cells and also 2D cells (cells with a zero extent along one of the axes). The rationale for this adjustment is that a pure area-based cost function underestimates the importance of placing such cells closer to the top of the tree. This modification is based on three simple rules applied sequentially:

1. **We always create an empty cell if its volume with respect to the original cell is greater than some threshold (10% in our implementation).**
2. **If there is a possible split plane which is completely covered by co-planar triangles, it will be selected and these triangles will be included in the smaller sub-cell. This heuristic sits well with the axis-aligned nature of kd-trees.**
3. **In addition to termination criteria based solely on a cost function (cost of splitting > cost of non-splitting), we also avoid creating very small cells, as measured by the ratio of the cell area to the area of the bounding box of the scene.**

The motivation for rule (3) is to avoid cells that may be so small that it is unlikely to be hit even by one ray for a given camera position from which the whole scene is visible. These rules also speed up the kd-tree building process because, once conditions 1-3 are established, no further processing is necessary.

These rules have to be considered together with a traversal routine. For example, if a traversal routine cannot handle 2D cells, the second rule is not feasible. In our opinion, there is no guaranteed best kd-tree building or traversal algorithm suitable for all scenes or all computer architectures. For current PCs and our implementation of the traversal algorithm (see section 4.3), these rules yielded roughly a 50% improvement in traversal performance compared with pure area-based approaches.

In section 4.1 we will continue the discussion of kd-tree creation and traversal algorithms by assessing the validity of the basic concepts used and analyzing their drawbacks. This leads us naturally to the concept of multi-level ray tracing. But before doing that we have to discuss some additional concepts.

3.2 Grouping Rays Together

The evolution of desktop PCs (both CPUs and GPUs) is such that math operations are getting faster at a higher rate than memory operations. The Single Instruction Multiple Data (SIMD) capability of such devices makes it possible to perform calculations on four rays for the cost of one [Wald 2001; Benthin et al 2003]. This is possible if we carefully choose which rays to shoot together, as there is tremendous geometric spatial coherence to exploit (especially in primary rays). This makes the caching mechanism of modern computers very effective. The performance gains correlate with the size of the group, however, current SIMD hardware can only support four simultaneous operations. What we would like is an algorithm that works independently of hardware features and scales gracefully to a much larger number of concurrent rays.

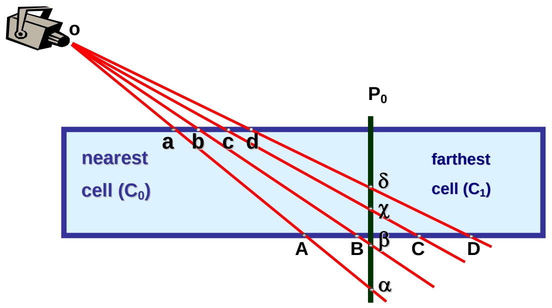


Figure 1: Tracing rays together: different rays go through different cells in the tree.

With grouping of rays comes the necessity to deal with the situation when rays go through different paths in the tree. Indeed, as evident from **Figure 1**, rays **oa** and **ob** travel only through the nearest cell (C$_0$), while other rays (**oc** and **od**) go through both cells. Obviously, as the number of rays in a group increases, so do the chances that these rays will diverge at some stage in the traversal process. Basically, we have 2 possibilities: we could work with a variable number of rays and regroup them every time some rays "get lost"; or we can mask the inactive rays in the group. Clearly, under such circumstances we can't realize the maximum benefit of processing multiple rays simultaneously, and worse, the overhead of tracking valid rays might even result in inferior performance of the concurrent implementation relative to one that process each ray individually. This also highlights another reason behind some aspects of the termination criteria (3) from the previous section. It doesn't make sense to continue splitting a cell for which only a few rays will still be active in most situations. Effective traversal of groups of rays is possible when all rays follow substantially the same path through the tree. Consequently, the rays must intersect any given split plane in one direction (either going from the negative to the positive direction or vice versa). This translates into the requirement that the coordinates of the direction vectors for all rays in a group have the same sign [Wald et al. 2003]. Groups of rays which do not possess this property must be split into multiple sub-groups which do, or into individual rays.

3.3 Frustum Culling

From the previous section, we see that there can be a penalty for grouping rays together (in the form of redundant operations), and this penalty increases with the size of the group. The desire to avoid or minimize this penalty led us to the development of the multi-level ray traversal algorithm. Even though it looks tempting, we cannot always use just a few specific rays to ascertain the behavior of the whole group. For example, we may consider using only the 4 corner rays in **Figure 3a** to represent all the rays by proxy. This is the approach utilized by [Havran and Bittner 2000] and [Dmitriev et al. 2004], where boundary rays are used to determine the behavior of internal rays. Unfortunately, this can lead to incorrect traversal choices. Consider the example in **Figure 2**. The top AABB is split into the red and blue sub-cells. We choose two points, **b** and **c** on two faces of the blue cell and one on the edge of the red cell (**a**) as shown below. If we place a camera **o** inside the plane passing through these 3 points, then all 4 of the corner rays **ob**, **oc**, **oe**, and **od** intersect only the blue cell. However, the ray **oa** (located inside the convex hull of the 4 corner rays) passes through both sub-cells. By scrutinizing this example, it is also obvious that even rays strictly inside the convex hull may not always follow the same path as boundary rays.

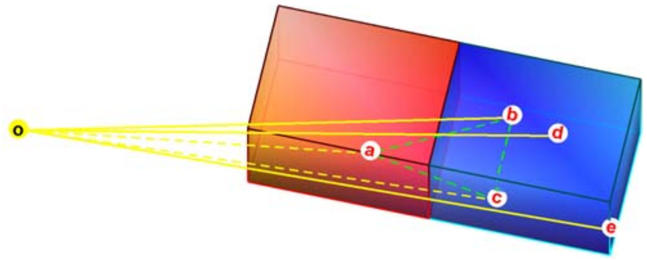


Figure 2: Problems using boundary rays as a proxy for the whole group. The ray **oa**, which is inside the convex hull of the 4 corner rays **obcde**, intersects both cells, while all 4 corner rays intersect only the blue cell.

What we can do instead, is to derive some inclusive properties to characterize the group as a whole, and then use these properties to determine the behavior of the whole group. One intuitive way would be to use the convex hull of all rays in the group, formed by a few given planes. The key operation will then be the determination of whether the convex hull intersects a particular axis-aligned box. This is analagous to the classic frustum culling algorithm used in raster graphics.

We will first describe the straightforward implementation of this algorithm as defined by Assarsson and Möller [2000] and then consider its applicability for ray-tracing purposes. In **Figure 3a**, a frustum is formed by 4 planes intersecting at one point. Each plane is defined by this intersection point and a normal, which we consider to be pointing outward from the frustum. We have to decide whether this frustum intersects with a given axis-aligned box. For each frustum plane, there are two vertices of interest belonging to the axis-aligned box: the one laying farthest in the positive direction of the plane's normal (p-vertex); and the other laying farthest in the negative direction of the normal (n-vertex). By inserting the n-vertex into the plane equation, we can decide whether the n-vertex and therefore the whole box is located outside this particular plane, and so on for each plane, hoping for a trivial rejection. In addition, we could use the p-vertices to determine whether or not the entire box is completely inside the frustum. This approach works for any number of planes forming the frustum.

In the special case where there are exactly 4 planes, the performance of this algorithm can be greatly enhanced by storing the positive and negative components of the 4 normal vectors separately in SIMD form for each of the x, y, and z components. As an axis aligned box can be represented by a pair of extremal vertices (one with the minimum x, y, z, and the other with the maximum x, y, z coordinate values), we can avoid having to find the n-vertices explicitly by noticing that for any frustum plane, the n-vertex has the lower coordinate value (among the 2 extremal vertex possibilities) for any positive normal coordinate, and vice versa for the negative normal direction. Consequently, using hardware with a 4-wide SIMD engine, the box can be culled against all 4 frustum planes using only 6 multiplications and 5 additions, this makes this approach very attractive for performance reasons. If we use "+" and "*" to represent 4-way SIMD operations, these calculations can be performed as follows:

```
__m128 nplane; // 4 plane values
nplane = (plane_normals[0][0] * bmin[0]) +
         (plane_normals[1][0] * bmax[0]);
nplane = (plane_normals[0][1] * bmin[1]) +
         (plane_normals[1][1] * bmax[1]) + nplane;
nplane = (plane_normals[0][2] * bmin[2]) +
         (plane_normals[1][2] * bmax[2]) + nplane;
```

plane_normals[0] and **[1]** contain positive and negative components of the 4 normal vectors and **bmin/bmax** – are the replicated x, y, and z coordinates of the min/max vertices of the current cell. Effectively, the **plane_normals** variable is used as a selector for the appropriate **bmin/bmax** values. With these 11 operations, we manage to find 4 n-vertices and compute all 4 plane values (for each frustum plane). The variable **nplane** now contains these 4 plane values. If any one of these is positive, then the frustum and the box are separated by the appropriate plane.

This trivial rejection mechanism is not perfect, as shown in **Figure 3b**. Here we see the axis-aligned box on the lower left of the frustum as viewed along its center axis. It fails the n-vertex outside test for each of the frustum's planes, implying that it may intersect the frustum, when, in fact, it is entirely outside the frustum. The proportion of these failed trivial rejects increases as the AABB becomes larger and larger (with respect to the frustum's cross section). This problem becomes acute when applied to ray-tracing traversal scenarios where the frustum is used as a proxy for a group of rays. It is often much smaller than the individual AABB cells that it is being tested against. One way to handle this is to reverse the roles of the frustum and AABB: we could use the AABB's planes to attempt to separate it from the frustum instead of the other way around. There is still the danger of failed trivial rejects, but their proportion will be lower due to the favorable ratio of the AABB's cross section to the frustum in the region where they pass one another. One circumstance that makes this tactic especially appealing and easy to implement is when the frustum contains only "coherent" groups of rays, that is, those in which all ray directions have the same sign.

This inverse approach is illustrated in **Figure 3**, (c) and (d). We are trying to decide whether the frustum intersects the red only, the blue only, or both sub-cells formed by the split plane **abcd**, with equation (**x = 1**). Suppose further that the frustum/plane intersection is bounded by the values [**p, n**] for the **y** coordinate, namely that all y-coordinates of the intersection lie in this range. If **n** is less than the y-value for edge **ad**, then we can conclude that:

1. **If all rays have negative Y direction components, the frustum does not intersect the blue sub-cell (Figure 3c). Since we know the frustum intersects the parent cell, we deduce that only the red sub-cell must be traversed.** (1)

2. **If all rays have positive Y direction components, the**

frustum does not intersect the red sub-cell (Figure 3d). Since we know the frustum intersects the parent cell, we know that only the blue sub-cell must be traversed.

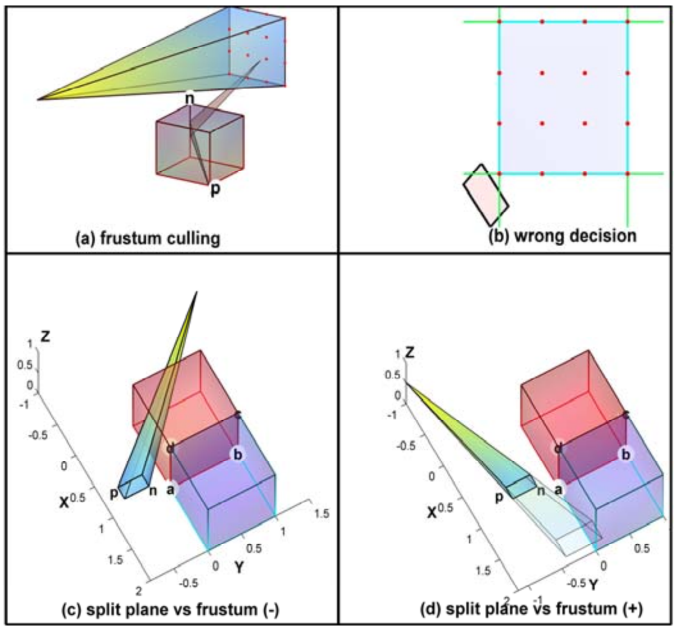

n
p
(a) frustum culling
(b) wrong decision
(c) split plane vs frustum (-)
(d) split plane vs frustum (+)
Z
X
Y
p
n
a
d
b
c

Figure 3: Direct (a,b) and inverse (c,d) frustum culling algorithms.

On architectures which allow it, we can execute these comparison operations in one command, i.e. by comparing the four **y** coordinates of frustum/plane intersection with the y-coordinate of the **ad** edge.

The inverse algorithm does not use frustum plane normals per se. What we need instead is the rectangular bounds for each axis-aligned plane which enclose frustum/plane intersection. This makes it possible to generalize this algorithm for groups of rays which do not have a common origin.

Another important property of the inverse frustum culling algorithm is that when used together with kd-tree traversal, it uses values only for one axis at a time. As we descend into the kd-tree, different axes will be processed at each level, thus allowing for effective culling of the current cell. As with the forward frustum culling algorithm, there may be situations where we erroneously conclude that the frustum might intersect a cell when in fact it does not. Any subsequent processing steps after this point will be wasted. We are primarily concerned with unnecessary intersection tests as they are quite expensive. Most of these extra tests can be avoided by using frustum/plane intersection data as specified in the inverse algorithm and executing additional clipping tests at the leaf nodes. At every leaf node, we perform robust clipping calculations for all 3 possible pairs of axes (**xy**, **yz** and **xz**) against the 6 AABB box faces.

We will illustrate this in **Figure 4** for the case of the **x** and **y** axes by projecting everything on the (**z = 0**) plane. There are 8 possible cases which can be easily detected. These cases differ by the direction of the frustum along the x and y axes and whether the frustum lies above or below the axis aligned box (using z projections). These tests enable us to exclude the great majority of the non-intersecting cases, and even though the remaining ones may cause some redundant calculations, they are tolerable. Amazingly enough, all 8 cases can be tested with only 2 comparisons. We will use terminology and ideas from Kay and Kajiya [1986].

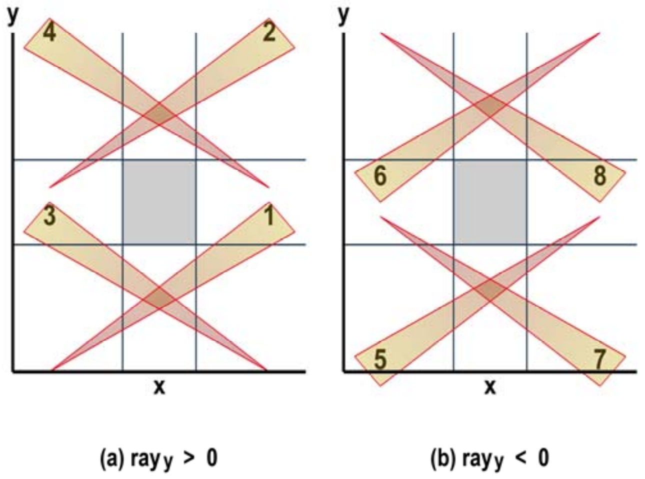


Figure 4: Frustum culling against leaf cell.

An axis-aligned box is defined as an intersection of 3 slabs, where a slab is the space between two parallel planes. For each ray, we may compute the entry and exit points for all 3 slabs which are represented as distances along the ray from the ray's origin. Accordingly, for the whole frustum we will need to know the ranges of these values. If either of the following two statements is true we conclude that the frustum and the box are separated:

1. **(minimum of y-entry values) > (maximum of x-exit values)**
2. **(minimum of x-entry values) > (maximum of y-exit values)** (2)

The first condition here describes cases 1, 3, 6, and 8; the second condition describes the rest. Instead of these two comparisons we could use the direct frustum culling algorithm or even an exact frustum/box intersection test, but this is expensive relative to the number of non-intersection cases that it eliminates. It is also possible to execute these 2 tests (for each pair of axes) at each traversal step, but it is not very effective. We have found that the best approach is to use assessments (1) for the internal nodes followed by (2) for the leaves.

4 Tracing Rays at Multiple Levels

Now we have the mathematical apparatus ready to describe the multi-level ray tracing algorithm (MLRTA). Any hierarchical acceleration structure imposes a certain spatial organization which is then stored in essentially linear memory. During a spatial query this structure is used to find ray/geometry intersection points. Processing is executed by sequentially narrowing the area of interest until final tests resolve the query. As an example from other field of study, we may look for a particular book in a catalog by narrowing the focus from 'science' to 'computer graphics' to 'ray tracing'.

One problem with using kd-trees for 3D scenes is that one doesn't necessarily end up with what might be expected. Consider **Figure 5a**, the box in the middle of the text objects ends up being sub-divided as shown in **Figure 5b**, and it ends up buried deep in the tree because of higher level split decisions made based on the surrounding geometry. If this box became the subject of interest for the camera we would have to traverse the entire kd-tree from its root for every ray we wish to trace until the box's cells are found (as depicted in **Figure 5c**). Wouldn't it be much better if we could dynamically find alternate entry points in the tree (depending on the current camera view) and start traversing at these nodes? If we are lucky, we might even find that the optimal entry point is right at the leaf node itself. Even better, if we use the beam concept described earlier, we can potentially "traverse" the kd-tree for all the rays in the beam, directly to the leaf node (in this optimistic example) in a single step.

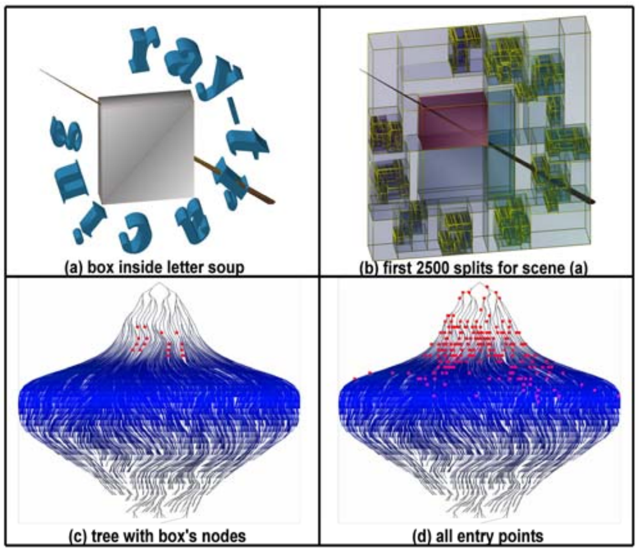

(a) box inside letter soup
(b) first 2500 splits for scene (a)
(c) tree with box's nodes
(d) all entry points

Figure 5: Simple object inside complex scene.

4.1 Finding Ideal Entry Points for Groups of Rays

Restating the results from section 3.3, the following information is sufficient to execute the inverse frustum culling algorithm:

1. **For any given axis-aligned plane, we compute a rectangle inside this plane, which contains all possible ray/plane intersection points. This rectangle does not have to be tight.** (3)
2. **All rays go in the same direction (i.e. x, y, and z projections of ray directions have the same sign).**

In this section we will describe a new algorithm which uses this information and the frustum culling algorithm to find an optimal entry point for all rays that satisfy condition (3). We will refer to this as an "entry point search", or EP search. It will be compared with the traditional depth-first search for intersection points for groups of rays using kd-trees [Wald 2004]. We will refer to this as an "intersection point search" or XP search. The main differences are that the new EP algorithm is not exhaustive and quickly aborts branches of the kd-tree which would not contribute to the final result. The EP search is used to find an optimal place to begin the traditional XP search.

Definition (3) does not presume any particular organization of the rays inside the group. Moreover, it does not even require that we know the specific rays, or even the number of rays, in advance, a fact which will be very handy later when we discuss adaptive tile splitting. The next algorithm describes the steps executed while looking for a common entry point. This is essentially a depth-first traversal of the visible nodes in the tree allowing for early escape from the traversal of branches that will not contribute further to the final result. It is achieved by maintaining a stack of nodes which can be potentially used as entry points (which we will call the "bifurcation stack").

1. **The tree is traversed in depth-first order by**
   - **Using the frustum culling algorithm.**
   - **Store all bifurcation nodes (those where both sub-cells are traversed) in a (last-in-first-out) stack structure until the first leaf node with potential intersections is found. This node is then marked as a candidate entry point node.**
2. **Continue depth-first walkthrough starting from the top-most node on the bifurcation stack. If another leaf with potential intersections is found, the node taken from the bifurcation stack will become the new candidate.**
3. **The algorithm ends and the current candidate node is returned as an entry point if** (4)
   - **The bifurcation stack is empty.**
   - **All potential rays end inside the current leaf node. Two cases are possible:**
     - **The leaf has some objects, and, all rays satisfying condition (3) intersect one of these objects inside the cell.**
     - **The leaf is empty, but is located *inside* some "watertight" object and all rays or groups satisfying condition (3) intersect the bounding box of this leaf.**

While executing this algorithm we are not interested in finding specific ray/object intersections and we may not even be able to do so since the rays in the group are not required to be defined at this point. What we are looking for is the *potential* for intersections. Specifically, if we cannot *exclude* an intersection with any ray satisfying condition (3) we will consider it as a potential intersection. We will illustrate this algorithm using the tree pictured in **Figure 6**.

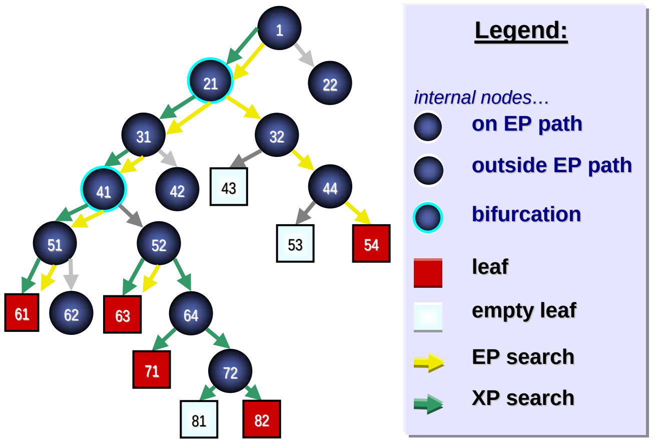


Figure 6: Two traversal algorithms: searching for an entry point (**EP**) and looking for intersections (**XP**).

Starting at node 1 and using group values described by condition (3), we realize that only the left sub-cell 21 has to be traversed. Both sub-cells of 21 have to be considered, so node 21 is stored in the bifurcation stack and processing continues with nodes 31 (split is ignored), 41 (stored in the stack), and 51 (split is ignored). While processing leaf 61, we conclude that there is a potential for intersections. Leaf 61 is then marked as a candidate and the bifurcation stack is frozen. The next node to consider, 41, is taken from the stack and we continue the depth-first traversal with nodes 52 and 63. Leaf 63 has a potential for intersections, so we mark node 41 as a candidate and move to the next node from the stack (21), abandoning the processing of the sub-tree starting at node 64. From node 21 we go to node 32 which has two children: 43 (ignored because it is empty); and 44 (taken). Node 44 has two leaves: 53 (ignored); and 54 (judged to have potential intersections). Therefore, we will mark node 21 as a candidate and return it as the entry point since the bifurcation stack is empty. All the rays which are bound by condition (3) may now start the tree traversal at node 21.

After the optimal group entry point is found we may split the group and continue looking for better entry points for each sub-group or perform intersection tests for all of the sub-groups to completion (XP search). Note the dissimilarities between the two traversal algorithms, one being a search for a common entry point (EP) and the other which is a search for intersection points (XP):

- At node 21, the XP algorithm is able to exclude node 32 from further processing since all the rays (in the beam) are now known and they intersect only with the cell of node 31. The EP algorithm uses only frustum properties that might intersect with node 32 and therefore traversal of both nodes on behalf of the group is required.
- The EP algorithm ignores node 64 when it reaches its parent, node 52, because it would have no effect on the selection of the group entry point. The XP algorithm however must continue the traversal of nodes 64, 71, 72, 81 & 82 because there may be some intersections to be found there.

This ability of the EP algorithm to disregard non-contributing branches and to do this on behalf of all the rays in a group helps it to greatly reduce the overall computations otherwise performed per ray.

4.2 Tile Splitting

The EP algorithm finds an optimal common entry point for all rays in a group by representing the group as a whole with the ranges of the individual ray directions. For beams representing broad ranges, the EP algorithm will most likely quickly detect that a good (i.e. deep) entry point is not available. In this case, it would be advantageous to split the ray direction ranges (and, accordingly, the actual underlying rays) in an attempt to get better separate EPs for individual sub-groups. Although the rays themselves do not have to be evenly distributed, in our implementation they are.

For primary rays it is practical to choose the initial groups of rays by splitting the image plane into equal tiles. This allows for trivial computation of bounding rectangles for the group/plane intersections as required by (3). Moreover, each tile may be assigned to a separate thread or task on a multi-processor or multi-threaded machine.

There are many possible approaches that could be taken here. In our experiments we used the simplest possible logic to decide what to do with a tile, basing the decision on 3 parameters:

1. **Initial Tile Size (ITS).**
2. **A Minimum Tile Size (MTS) which automatically triggers XP search.** (5)
3. **A Split Factor (SF), which defines how many pieces to split a tile into.**

Any tile larger than MTS is always split unless the chosen entry point is already a leaf. This is illustrated in the sequence of images in **Figure 7**. It is easy to see how the tile sizes change to adapt to the changing underlying geometry. Tiles marked in red are those which have diverging rays (going in different directions) and are therefore unsuitable for the EP search. For such the XP search is performed right away for all the coherent sub-groups of the original tile.

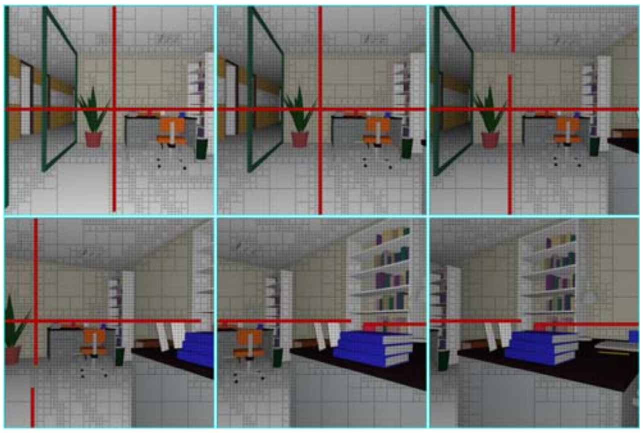


Figure 7: Sequence of frames from Soda Hall scene showing adaptive tile splitting (red color marks diverging packets).

Depending on a scene and/or the camera position, varying the values for the ITS/MTS/SF parameters produce different results. We tested various combinations of these parameters on a collection of over 2500 different models. We varied the Initial Tile Size parameter from 16x16 pixels to 256x256 pixels. Each initial tile was subsequently split either into 4 or 16 sub-tiles and we used a Minimum Tile Size between 4x4 and 64x64 pixels. The measured performance variations were not very large, mostly around 10%. This indicates that it is possible to derive a single set of parameters which would be roughly optimal for most scenes. Our best results were achieved by starting from tiles with 128x128 pixels and splitting them as needed into 16 sub-tiles directly. It is expected that for multi-threaded implementations, a smaller initial tile size might be better because the thread execution granularity will be a function of the time required to process one tile.

Another potential advantage of using the EP algorithm for such tiles is that it provides a natural way to measure geometric complexity. Given the way that the kd-tree creation algorithm works, the size of the sub-tree underneath any given node (measured by overall number of nodes in the sub-tree) serves as an indication of the geometrical complexity in this volume of space. In the extreme case, when an EP is a 2D leaf completely covered by triangles, we could cast rays sparingly and reserve more of the ray budget for more complicated regions. This approach is similar to one described by Ghazanfarpour and Hasenfratz [1998].

4.3 Interval Traversal Algorithm

The basic premise of MLRTA is that it works even if the exact rays in the group are not known. If, however, such information is available, MLRTA can be executed in parallel with the computation of ray/geometry intersection points. We can then use the found distances to intersection points to further purge traversal steps based on visibility culling. In this section we will describe one such implementation which we are currently using in lieu of low level traversal (XP search) in our system. It will be described together with a sample implementation which actually does not require SIMD instructions in the inner cycle.

The algorithm is based on computing and updating minimum and maximum distances to the traversed cell for all rays in the group. At the beginning of the traversal we will store minimum and maximum distances to the scene's bounding box in the float array **t_cell[2]**. In **Figure 1**, **t_cell[0] = oa** and **t_cell[1] = oD**. For each traversed cell, by using conditions (3) from section 4.1, we can compute two values: **t_plane[0]** – minimum distance to the split plane $\mathbf{P_0}$ in the cell ($\mathbf{o}\delta$) and **t_plane[1]** – maximum such distance ($\mathbf{o}\alpha$). Then

```
// If we can verify that all rays exit the cell
// before intersecting the plane...
if (t_cell[1] < t_plane[0]) {
    // Only the nearest cell is traversed.
    // Compute address of the nearest node and
    // continue traversal. t_cell is not changed.
    ...
}

// If all rays enter the cell
// only after intersecting the plane...
if (t_cell[0] > t_plane[1]) {
    // Only the farthest cell is traversed.
    // Compute address of the farthest node and
    // continue traversal. t_cell is not changed.
    ...
}

// We will have to traverse the nearest cell
// followed by the farthest cell. This branch also
// handles a small percentage of misdiagnosed cases
// (when only one cell is actually traversed).

// The next array will contain t_cell values
// for the farthest cell
float t_farthest[2];
t_farthest[0] = max(t_cell[0], t_plane[0]);
t_farthest[1] = t_cell[1];

// Store t_farthest values and the address
// of the farthest cell into the programming stack
// (it will be used after the subtree starting
// at the nearest node is processed).
...

// Modify the t_cell interval for the nearest cell
t_cell[1] = min(t_cell[1], t_plane[1]);
// Continue traversing nearest cell
...
```

It is evident that this algorithm has the same complexity as one for traversing an individual ray [Wald 2004]. However, it executes traversal of all rays in the group simultaneously by using proxy intervals. Groups cannot be very large as performance will suffer from too many inactive rays in the group (see discussion in section 3.2). In our experiments, we found the best group size to be 4x4, which compares favorably with the 2x2 groups used by Wald.

Even though the interval traversal algorithm sharply reduces the required number of operations compared to the diligent traversal of all 16 rays in the group, overall performance improvement is only about 20%. The reason is that the branches in the kd-tree traversal are data dependent with all 3 continuation scenarios (nearest, farthest or both sub-nodes) occurring with roughly equal probability.

5 Results and Discussion

The simplest way to evaluate the performance impact of a particular feature is to test two implementations, one with and one without the feature. The average performance difference of our MLRTA and non-MLRTA implementations for the 3 scenes in Table 1 is about 3.25X for primary rays and 2.75X for primary and shadow rays. As will be evident from the statistical data in this section, this is roughly equivalent to the reduction in the number of traversal steps executed by the algorithm. Compared with the best results reported elsewhere in the literature, our traditional implementation (without MLRTA) is still about 2X faster. We can only speculate that this is due primarily to different tree construction and somewhat different traversal and intersection methods, API overhead may also play a role. In this section we will provide a more formal quantification of the performance results based on measurements of the mathematical operations.

We will analyze the MLRTA results by providing data for 4 scenes which vary greatly in scene complexity and occlusion properties. For convenience, all results will be presented on a per packet basis - we use packets with 16 rays (4x4). If a total of **k** cells are traversed during the rendering of a 1024x1024 pixel image, the ratio **k/(1024*1024/(4*4))** will be used. We account for all traversal steps regardless of whether they are executed during the EP or XP search. Only those intersection tests which were not avoided through AABB culling are included in the statistics. If a triangle intersection test is avoided then no triangle data is accessed.

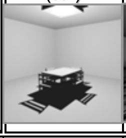
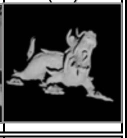
Scene # of triangles and view / Average measurements per 4x4 packet at 1024×1024 resolution		Erw6 (804)	Conference (274K)	Soda Hall (2195K)	Asian Dragon (7M)
					
number of traversed cells	1. MLRT	3.98	20.87	32.52	32.65
	2. no MLRT	13.00	49.98	71.37	42.18
3. EP search only		0.51	2.30	4.44	2.72
non-masked intersections	4. MLRT	1.09	2.48	1.59	19.97
	5. no MLRT	1.09	2.55	1.52	19.94

Table 2: For primary rays MLRTA significantly reduces the number of traversal steps (first row vs second) without adversely affecting the number of intersection tests.

number of traversed cells	6. MLRT	10.07	53.73	69.07	45.01
	7. no MLRT	24.83	101.06	117.22	58.41
non-masked intersections	8. MLRT	1.25	3.71	2.17	23.51
	9. no MLRT	1.22	3.75	2.09	23.48

Table 3: Corresponding measurements for primary + shadow rays (one light source).

By analyzing rows 1 and 2 of Table 2 we see that the MLRTA greatly reduces the number of traversal steps required. This ratio varies from ~3X for scenes with a lot of occlusion to 1.3X for the last scene which has limited occlusion. The MLRTA's goal is to minimize the number of operations in the most time-consuming part of the ray-tracing pipeline. The return on investment is quite high. A 10% investment in finding a good EP yields an overall performance improvement of 2.5X (in the conference scene).

By examining rows 4 and 5 we see that there is no significant change in the number of overall intersection tests performed, which is ideally what you would expect (ie finding good EPs helps you avoid redundant traversal of the upper parts of the kd-tree, but has no detrimental effect on the processing at the lower part of the tree). We have observed that the best RT results are achieved when roughly 2/3 of the time is spent traversing the kd-tree and 1/3 actually looking for intersections and that this ideal ratio increases with model size. Because the termination criteria do not depend explicitly on the kd-tree depth, the number of triangles in the leaf nodes remains roughly constant. In most cases individual triangles can show up in multiple leaf nodes. For those leaves some redundant intersection tests can be avoided by using a mailbox mechanism [Amanatides and Woo 1987].

The data in these tables were obtained for the inverse frustum culling algorithm introduced in section 3.3. The direct method requires about 20% more EP traversal steps. Also, a direct frustum culling test is more expensive than an inverse test. Accordingly, the inverse method clearly has an edge, at least for the coherent packages which we are currently using.

We have conducted preliminary tests on using MLRTA to facilitate adaptive geometric anti-aliasing as described above. Preliminary results show that for a given level of quality it results in a 50% reduction in the number of actual rays shot for a given scene (these results were evaluated using static images). In our experience, since we are now able to view most of these scenes at interactive rates on ordinary desktop machines, temporal aliasing artifacts are now more dominant. We are planning to revisit these issues in the future.

The results given in Table 2 were obtained for primary rays. Similar conclusions can be drawn also from analysis of secondary rays. Table 3 includes data for primary and shadow rays for the same 4 scenes (normalized for one primary packet of 4x4 rays). A significant portion of the intersection tests for non-occluded shadow rays can be avoided by excluding objects already hit by the parent primary rays. For this reason the ratio of traversal vs. intersection steps is even higher than for primary rays (compare the quotients of rows 1 to 4 and 6 to 8).

6 Limitations of MLRTA and Future Work

MLRTA does not require advance knowledge of the rays in the group and uses ranges of directions to traverse the whole group at once. Even the interval extension, as described in section 4.3, uses exact rays only during intersection tests and operates with inclusive intervals during traversal. Although this feature facilitates adaptive anti-aliasing of the image, it prevents direct utilization of MLRTA for very “wide” packages with small numbers of rays. In such cases we end up doing a lot of unnecessary speculative work on behalf of rays which will never materialize. This problem cannot be fixed merely by splitting the range data. In **Figure 8**, a big group of secondary rays is represented in some parametric space. If we just split the original voxel uniformly, some sub-voxels will have no rays at all and tracing them would be a waste of time.

If the size of the original group of rays is small compared with number of sub-voxels, it is very unlikely that any sub-voxel will include a large number of rays. In this situation, MLRTA or any other collective traversal mechanism will be ineffective. At the same time, for all secondary rays considered together there exists a partitioning of the parametric space for which there will be substantial amount of sub-voxels with a considerable number of coherent rays in each one. This draws a parallel with the Dirichlet Principle (if you try to place n+1 rabbits into n cells, there will be at least one cell with at least 2 rabbits). We have to select sub-voxels in such a way that they will be large enough to encompass big groups of rays yet small enough to be traversed mostly “together” through the tree.

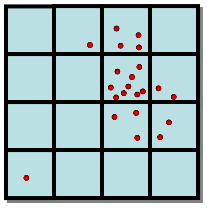


Figure 8: Distribution of secondary rays. Each red dot represents a ray in some parametric space (3D origin + 2D direction). Some voxels have none or very few rays, while others have a lot of coherent rays.

We are planning to research these issues, in particular exploring approaches for culling such 5D voxels first outlined by Arvo and Kirk [1987]. We assume that the 3D component of such a parametric space can be handled implicitly by associating rays with low-level cells in a kd-tree when they are traversed. These cells are usually small as this is one of the goals of kd-tree builder. We can then traverse those voxels with a larger number of constituent rays using the interval approach as described in section 4.3. All possible splits of the directions of the original group can be pre-computed using a simple binning technique to avoid tracing empty groups. Voxels with a small number of rays could be traversed on a per-ray basis.

This is, of course, speculation at this point and whatever approach eventually gets used will have to be compared against tracing individual rays sequentially. Presumably, by selecting the proper size of the original tile and tracing different levels of secondary rays separately (as suggested by Nakamaru and Ohno in [1997]), this could be effective for the majority of scenes.

Considering shadow rays for point lights, they can be handled by MLRTA directly by tracing them from the light sources to the hit points produced by the primary rays. Currently, we implemented a simplified version of this approach by using MLRTA for all the secondary rays which are reflected from flat surfaces (considering reflected and shadow rays). For shadow rays originating from secondary hits, it may be necessary to use partitioning schemes as outlined at the beginning of this section.

MLRTA can certainly also be used in photon mapping (for the final gathering step), area lights, and ambient occlusion schemes [Gritz et al. 2002]. In fact, area lights seem to be well suited for processing using a frustum formed between the hit point and polygonal area lights. We are planning to explore these issues in the near future.

7 Summary

MLRTA uses geometric properties of a large group of rays to find a common entry point into the kd-tree for all of the rays in the group, thus avoiding redundant operations. This approach enables us to find correct intersection points by using just 1/3 of the traversal steps which would otherwise be required.

The entry point search is carried out by identifying common group properties and using these properties in lieu of rays. We analyzed 2 different ways of defining such group properties. In one, a set of planes enclosing all the rays is created and traversed through the kd-tree using the direct frustum culling algorithm. This approach works well in traditional CG applications where the frustum is 'big' and objects are typically 'small' and can be effectively culled against the frustum by using the frustum's planes. For ray-tracing applications however, the opposite characterization is more likely. It allows us to "invert" the traditional frustum culling algorithm, that is to cull the frustum by using the faces of the AABBs. This new inverse frustum culling algorithm is broader in scope and does not include the notion of frustum bounding planes. Accordingly, it can be used for more general collections of coherent rays.

Another attractive property of the MLRTA algorithm is that it provides a natural measure of the geometric complexity of specific view directions. We intend to continue investigating these issues, paying particular attention to anti-aliasing in the temporal domain [Martin et al. 2002]. An appealing approach would be to track groups of rays through multiple time frames.

Acknowledgments

Many thanks to Ingo Wald, Philipp Slusallek, and Carsten Benthin for sharing their results and models. The Asian Dragon and Thai Statuette models are courtesy of the Stanford 3D Scanning Repository and room model from the accompanying video is courtesy of the Cornell University Graphics Group. The authors would like to thank the anonymous reviewers for their valuable comments and pointing out to missing references. We gratefully acknowledge discussions with and assistance from Radek Grzeszczuk and Gordon Stoll.

References

AMANATIDES, J. 1984. Ray Tracing with Cones, In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18, 4, ACM, 129-135.

AMANATIDES, J. and WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. *Eurographics Conference Proceedings 1987*, 3–10.

ARVO, J. and KIRK, D. 1987. Fast Ray Tracing by Ray Classification, In *Computer Graphics (Proceedings of ACM SIGGRAPH* 87), 21, 4, ACM, 55-64.

ASSARSSON, U. and MÖLLER, T. 2000. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools*, 5, 9-22.

BENTHIN, C., WALD, I., and SLUSALLEK, P. 2003. A Scalable Approach to Interactive Global Illumination, *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22(3), 621-630.

CHO, F.S. and FORSYTH, D. 1999. Interactive ray tracing with the visibility complex. *Computers and Graphics (Special Issue on Visibility - Techniques and Applications)*, 23(5), 703-717.

DAVIS, T. and DAVIS, E. 1999. Exploiting frame coherence with the temporal depth buffer in a distributed computing environment, *Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, 29-38.

DMITRIEV, K., HAVRAN, V., and SEIDEL, H.-P. 2004. Faster Ray Tracing with SIMD Shaft Culling, *Research Report, Max-Planck Institut Für Informatik*, MPI–I–2004–4–006.

GENETTI, J., GORDON, D., and WILLIAMS, G. 1998. Adaptive Supersampling in Object Space Using Pyramidal Rays. *Computer Graphics Forum*, 16(1), 29-54.

GHAZANFARPOUR, D. and HASENFRATZ, J-M. 1998. A Beam Tracing with Precise Antialiasing for Polyhedral Scenes. *Computer & Graphics*, 22(1), 103-115.

GLASSNER, A. 1984. Space Subdivision for Fast Ray Tracing. *IEEE Computer Graphics & Applications*, 4(10), 15-22.

GRITZ, L., APODACA, T., QUARONI, G., BREDOW, R., GOLDMAN, D., LANDIS, H., and PHARR, M. 2002. RenderMan in Production. *ACM SIGGRAPH 2002 Course Notes*, Course 16.

HAVRAN, V. and BITTNER, J. 2000. LCTS: Ray Shooting using Longest Common Traversal Sequences. *Computer Graphics Forum*, 19(3), C59-C70.

HAVRAN, V. 2000. *Heuristic Ray Shooting Algorithms*, Ph.D. Thesis, Czech Technical University.

HAVRAN, V., BITTNER, J., and SEIDEL, H.-P. 2003. Rendering: Exploiting temporal coherence in ray casted walkthroughs, *Proceedings of the 19th Spring Conference on Computer Graphics (SCCG 2003)*, 149-155.

HECKBERT, P. and HANRAHAN, P. 1984. Beam tracing polygonal objects. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18, 4, ACM, 119-127.

KAY, T. L. and KAJIYA, J. T. 1986. Ray Tracing Complex Scenes, In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, 20, 4, 269-278.

MACDONALD, J. and BOOTH, K. 1990. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6, 153-166.

MARTIN, W., REINHARD, E., SHIRLEY, P., PARKER, S., and THOMPSON, W. 2002. Temporally Coherent Interactive Ray Tracing. *Journal of Graphics Tools*, 7(2), 41-48.

NAKAMARU, K. and OHNO, Y. 1997. Breadth-First Ray Tracing Utilizing Uniform Spatial Subdivision, *IEEE Transactions On Visualization and Computer Graphics*, 3(4), 316-328.

OHTA, M. and MAEKAWA, M. 1990. Ray-bound tracing for perfect and efficient anti-aliasing. *The Visual Computer: International Journal of Computer Graphic*, 6(3), 125-133.

RAMASUBRAMANIAN, M., PATTANAIK, S., and GREENBERG, D. 1999. A perceptually based physical error metric for realistic image synthesis. In *Proceedings of ACM SIGGRAPH 1999*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 73-82.

SHINYA, M., TAKAHASHI, T., and NAITO, S. 1987. Principles and applications of pencil tracing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21, 4, ACM, 45-54.

SZIRMAY-KALOS, L., HAVRAN, V., BALAZS, B., and SZÉCSI, L. 2002. On the Efficiency of Ray-shooting Acceleration Schemes. *Proceedings of the 18th Spring Conference on Computer Graphics (SCCG 2002)*, 89-98.

TELLER, S. and ALEX, J. 1998. Frustum Casting for Progressive, Interactive Rendering. *Technical Report, Laboratory for Computer Science, Massachusetts Institute of Technology*, TR-740.

WALD, I., SCHMITTLER, J., BENTHIN, C., SLUSALLEK, P., and PURCELL, T.J. 2003. Realtime Ray Tracing and its use for Interactive Global Illumination, STAR, *Computer Graphics Forum (Proceedings of Eurographics 2002)*, 22(3).

WALD, I., 2004. *Realtime Ray Tracing and Interactive Global Illumination*, Ph.D. thesis, Saarland University.

WALD, I., SLUSALLEK, P., BENTHIN, C., and WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing, *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20(3), 153-164.

Ray Tracing Animated Scenes using Coherent Grid Traversal

Ingo Wald Thiago Ize Andrew Kensler Aaron Knoll Steven G. Parker
SCI Institute, University of Utah, 50 S Central Campus Dr., Salt Lake City, UT, 84112

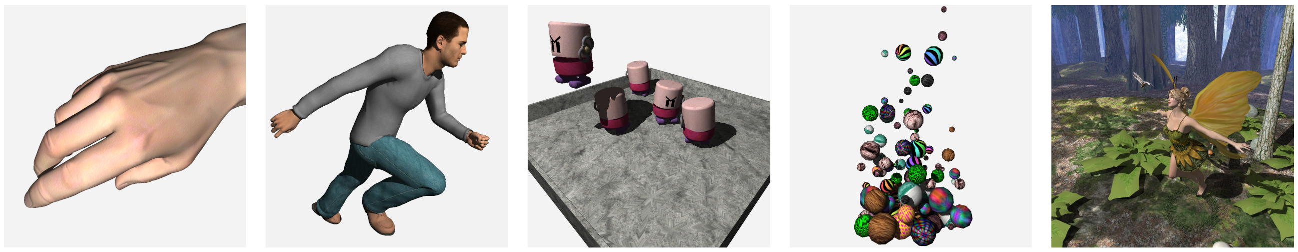


Figure 1: Several animated models ray traced using our coherent grid traversal: a) A gesturing hand of 16K triangles. b) An animated "Poser" model (78K triangles). c) Animated wind-up toys (11K triangles) walking and jumping incoherently around each other. d) A rigid-body dynamics simulation of marbles (8.8K triangles). e) A complex scene of 174K animated triangles, where a fairy and a dragonfly dance through an animated forest. Scenes are rebuilt from scratch every frame, allowing fully dynamic animation. Including shading, texturing, and hard shadows, as used in the above images, we can render these scenes at 1024×1024 pixels with 15.3, 7.8, 10.2, 26.2, and 1.4 frames per second on a dual 3.2 GHz Xeon. Excluding shading, texturing, and shadows, we achieve 34.5, 15.8, 29.3, 57.1, and 3.4 frames per second.

Abstract

We present a new approach to interactive ray tracing of moderate-sized animated scenes based on traversing frustum-bounded packets of coherent rays through uniform grids. By incrementally computing the overlap of the frustum with a slice of grid cells, we accelerate grid traversal by more than a factor of 10, and achieve ray tracing performance competitive with the fastest known packet-based kd-tree ray tracers. The ability to efficiently rebuild the grid on every frame enables this performance even for fully dynamic scenes that typically challenge interactive ray tracing systems.

1 Introduction and Related Work

Over the last 20 years, a number of different data structures have been proposed for accelerating ray tracing, such as Bounding Volume Hierarchies (BVH), Grids, Octrees [Glassner 1984], and Binary Space Partitioning (see, e.g., [Glassner 1989; Havran 2001]). Each of these data structures has its own strengths and weaknesses, and the effectiveness of each technique strongly depends on the scene, application, and efficiency of the actual implementation. Recent work in interactive ray tracing, however, has focused primarily on kd-trees [Wald 2004; Foley and Sugerman 2005; Reshetov et al. 2005; Woop et al. 2005] and grids [Purcell et al. 2002], or multilevel grids [Parker et al. 1999b; Reinhard et al. 2000].

While the first interactive ray tracers used grids [Parker et al. 1999b], algorithmic developments for kd-tree based ray tracers — most notably coherent ray tracing [Wald et al. 2001] and MLRT traversal [Reshetov et al. 2005] — have significantly improved the performance of kd-trees. Packet tracing creates groups of spatially coherent rays that are simultaneously traced *together* through a kd-tree, where all rays perform each traversal iteration in lock-step. This enables effective use of SIMD extensions on modern CPUs, increases the computational density of the code, and reduces strain on memory access. In turn, this gave rise to fast software implementations [Wald 2004], and to instruction-parallel special-purpose ray tracing hardware [Woop et al. 2005]. Exploiting the coherence in a packet of rays has yielded further improvements in "Multilevel Ray Tracing" (MLRT) [Reshetov et al. 2005], where a bounding frustum drives the kd-tree traversal of rays in bulk instead of considering each ray individually. Consequently, the cost of a traversal step becomes independent of the number of rays in the packet, encouraging larger packets with significantly lower cost per ray.

Unfortunately, these techniques are not directly applicable to grids. Thus, packet-enabled kd-trees have recently consistently outperformed grid-based ray tracers, and many believe that they are a superior acceleration structure (see, e.g., [Stoll 2005]).

Dynamic Scenes Although packet kd-tree traversals outperform grids for static scenes, animated scenes present a challenge due to the high cost of rebuilding a kd-tree as objects move. For the surface area heuristics required to build fast kd-trees [Wald and Havran 2006], building the acceleration structure effectively requires seconds to minutes for moderately complex scenes. This limitation to static scenes limits the utility of interactive ray tracing for many applications that would benefit from advanced lighting models, such as visual simulation, animations, and interactive games. While some efforts have focused on extending kd-trees to dynamic scenes [Wald et al. 2003; Günther et al. 2006], they are limited to mostly hierarchical motion or require advance knowledge of the scene, and therefore are unsuitable for most truly dynamic animations that require unstructured motion. For full generality, we propose rebuilding the acceleration structure from scratch every frame. For general scenes, with kd-trees this is currently infeasible.

A grid, in contrast, can be created and modified at interactive rates [Reinhard et al. 2000], at least for moderate sized scenes of up to a few hundred thousand triangles. Consequently, grids are attractive for dynamic scenes because of their faster build, even if they have a higher traversal cost than a kd-tree. Nevertheless, as kd-trees can be up to an order of magnitude faster than single-ray grids, grids will only be viable when their traversal can be performed with similar efficiency. Ultimately, this will require employing the same techniques for grids that made kd-trees as fast as they are today: coherent packets of rays, SIMD, and frusta. However, the 3D digital differential analyzer algorithms usually used for traversing a grid do not lend well to packetization, as we will explain below.

In this paper, we propose a new traversal scheme for grid-based acceleration structures that allows for traversing and intersecting packets of coherent rays using an MLRT-inspired frustum-traversal scheme. This algorithm is well-suited for SIMD implementation and provides dramatic speedup over a conventional grid traversal, yielding performance comparable to kd-tree based systems for static scenes. More importantly, this scheme facilitates animated scenes in a straightforward manner by interactively rebuilding the grid from scratch every frame. Using this technique on a fully animated teapot-in-a-stadium stress scene of 174K triangles, we achieve a ray tracing performance of around 1-2 frames per second (at 1024^2 pixels with hard shadows and simple shading) on a dual 3.2 GHz Xeon CPU; for a 16K triangle object, we achieve 15-16 fps (Figure 1). We mostly consider moderate scenes of up to a few hundred thousand triangles, and focus on only primary and shadow rays.

The importance of supporting dynamic scenes has recently been recognized by many different researchers, and several different approaches have been proposed concurrently to our work, for example [Wald et al. 2006; Stoll et al. 2006; Günther et al. 2006; Lauterbach et al. 2006]. We will compare to these approaches in Section 5.

2 Coherent Grid Traversal

Efficient ray-grid traversal has already received much attention [Cleary et al. 1983; Fujimoto et al. 1986; Amanatides and Woo 1987; Parker et al. 1999b; Spackman and Willis 1991], in aspects of both algorithm and implementation. Significant improvements cannot be expected from merely optimizing current implementations; we must explore new concepts to design an effective packetized traversal. Our new algorithm delivers to grids the same components that made kd-trees as fast as they are today: packets, SIMD extensions, and frustum traversal; while preserving the trivial computation of an incremental grid marching step.

In this section, we explain why these techniques have been successful for other acceleration structures and discuss the difficulties of applying the same concepts to a conventional grid traversal. Then, we derive our new packet traversal scheme, and show how it can benefit from known optimizations to achieve significantly higher performance than past grid implementations.

2.1 Issues with Packetized Grids

The basic idea of packet and frustum traversal is straightforward: rather than traverse each ray on its own, we exploit the intrinsic coherence between neighboring rays, and trace them together. If the rays are coherent, they will largely traverse the same regions of space, accessing identical nodes in an acceleration structure, and intersecting the same underlying triangles. Effectively, the cost of memory access becomes amortized over all the rays in a packet, ideally for both our acceleration structure and geometry data. In addition, traversing multiple rays through the same node of the acceleration structure allows us to perform SIMD operations on four rays at once, reducing the computation costs of both traversal and primitive intersection by up to a factor of four. Finally, frustum techniques determine intersection patterns of an entire packet, often replacing intensive per-ray branching with a single test; thus amortizing the computations over the entire packet.

The advantages of packets, SIMD, and frustum methods are beneficial to any acceleration structure. Spatially hierarchical structures, such as a kd-tree or BVH, typically exhibit little divergence at the upper levels of traversal, making them ideally suited for adaptation to ray packets. Packets are easily traversed through hierarchical acceleration structures where rays generally progress through identical cells; diverging only in finer nodes deep down in the hierarchy, if at all. Even when rays diverge, some rays just traverse a few cells that they would not have traversed otherwise, but do not interfere with traversal decisions in the remaining part of the subtree. Since the packet is never divided, those rays automatically are re-enabled as soon as the recursion returns from that subtree.

For a grid, in contrast, the situation is more complicated: traversal is always performed on the same fine level, where divergence is most likely. Moreover, grid based ray tracers typically use 3D digital differential analyzers (3DDDA) or Bresenham-like algorithms to iterate through the voxels traversed by the ray (e.g., [Fujimoto et al. 1986; Amanatides and Woo 1987; Spackman 1990]). These algorithms can chose only one cell at a time to step into, but different rays can disagree on the next cell to be traversed. For example, Figure 2 shows five rays diverging in cell B; some demand traversal to C, while others demand traversal to D. If the packet decides to go to C first, the 3DDDA state variables for those rays entering cell D become invalid (and vice versa). These invalid state variables break the 3DDDA algorithm in the next traversal step.

This disagreement could be solved by splitting the packet into subpackets with the same traversal decision. However, Figure 2 shows that the rays that have diverged in cell B still traverse other common cells (E and F) later on. If the packet were split at cell B, that coherence would be lost; in practice, packet splitting quickly deteriorates to single-ray traversal. Re-merging the packets after each step would solve that problem, but is prohibitively expensive.

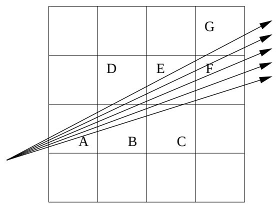


Figure 2: Five coherent rays traversing a grid. The rays are initially together in cells A and B, but then diverge at B where they disagree on whether to first traverse C or D in the next step. Even though they have diverged, they still visit common cells (E and F) afterwards.

2.2 A Slice-based Packet Traversal for Grids

As the above discussion has shown, the primary concern with packetizing a grid is that with a 3DDDA, different rays may demand different traversal orders. We solve this by abandoning 3DDDA altogether, and devise an algorithm that traverses the grid *slice by slice* rather than cell by cell. For example, we can traverse the rays in Figure 2 by traversing through vertical slices; from cell A in the first slice, we would traverse the rays to cells B and D in the second slice, then to C and E in the third, and so on. In each slice, we would intersect all rays with all of the slice's cells that are overlapped by any ray. This may traverse some rays through cells they would not have intersected themselves, but will keep the packet together at all times. In Figure 2, we would intersect 7 cells with 5 rays each, instead of 27 cell visits if the rays are traced individually. Though the packet now intersects only 7 instead of 27 cells, the total number of ray-cell intersection tests is $7 \times 5 = 35$. In practice, ray coherence easily compensates for this overhead.

We first transform the rays into the canonical grid coordinate system, in which a grid of $N_x \times N_y \times N_z$ cells maps to the 3D region of $[0..N_x) \times [0..N_y) \times [0..N_z)$. In that coordinate system, the cell coordinates of any $3D$ point p can be computed simply by truncating it. Then, we pick the dominant component (the $\pm X$, $\pm Y$, or $\pm Z$ axis) of the direction of the first ray. This will be the *major traversal axis* that we call $\vec{K}$; all rays are then traversed along this same axis; the remaining dimensions are denoted $\vec{U}$ and $\vec{V}$. In order to traverse the rays front to back, which allows early termination when all rays have intersected before the next slice, all rays must have the same sign along the traversal direction. For coherent packets, this is not a limitation; to violate this assumption, two rays would need to span an angle of more than $\frac{\pi}{2}$. Note that we do *not* demand that all rays in a packet have the same dominating axis, nor that their direction signs match along $\vec{U}$ or $\vec{V}$, as is usually required by kd-tree packet traversers [Wald 2004] as long as the rays are coherent.

Now, consider a slice k along the major traversal axis, $\vec{K}$. For each ray r_i in the packet, there is a point p_i^{in} where it enters this slice, and a point p_i^{out} where it exits. The axis aligned box $\mathcal{B}$ that encloses these points will also enclose all the $3D$ points — and thus, the cells — visited by at least one of of the rays. Once $\mathcal{B}$ is known, truncating its min/max coordinates yields the u, v extents of all the cells on slice k that are overlapped by any of the rays (Figure 3d).

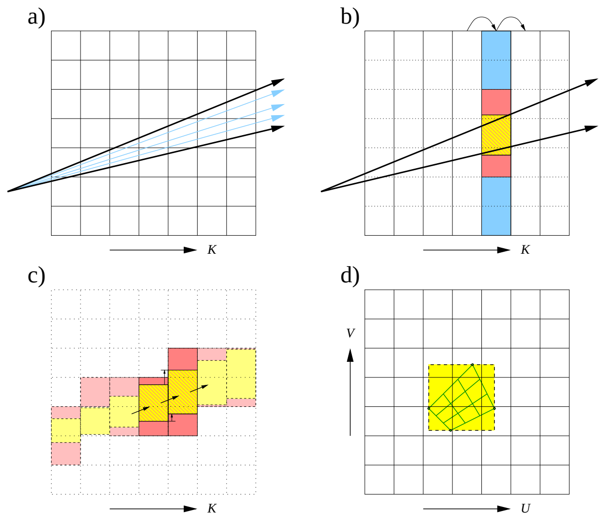


Figure 3: Given a set of coherent rays, our algorithm first computes the packet's bounding frustum (a) that is then traversed through the grid one slice at a time (b). For each slice (blue), we incrementally compute the frustum's overlap with the slice (yellow), which determines the actual cells (red) overlapped by the frustum. (c) Independent of packet size, each frustum traversal step requires only one four-float SIMD addition to incrementally compute the min and max coordinates of the frustum slice overlap, plus one SIMD float-to-int truncation to compute the overlapped grid cells. (d) Viewed down the major traversal axis, each ray packet (green) will have corner rays which define the frustum boundaries (dashed). At each slice, this frustum covers all of the cells covered by the rays.

Extension to Frustum Traversal Instead of determining the overlap $\mathcal{B}$ based on the entry and exit points of *all* rays, we can compute the four planes bounding the packet on the top, bottom, and sides. This forms a bounding frustum that has the same overlap box $\mathcal{B}$ as that computed from the individual rays[1]. Since the rays are already transformed to grid-space, we can determine our bounding planes based on the minima and maxima of all the rays' u and v slopes along $\vec{K}$. For a packet of $N \times N$ primary rays, we can simply compute these extremal planes using the four corner rays; however for more general (secondary) packets all rays must be considered.

Traversal Setup Once the plane equations are known, we can intersect the frustum with the bounding box of the grid; the minimum and maximum coordinates of the overlap determine the first and last slice that should be traversed. If this interval is empty, the frustum misses the grid, and we can terminate without traversing.

Otherwise, we compute the minimum and maximum u and v coordinates of the entry and exit points with the first slice to be computed. Essentially, these describe the lower left and upper right corner of an axis-aligned box bounding the frustum's overlap with the initial slice, $\mathcal{B}^{(0)}$. Note that we only need the u and v coordinates of each $\mathcal{B}^{(i)}$, as the k coordinates are equal to the slice number.

Incremental Traversal Since each slice's overlap box $\mathcal{B}^{(i)}$ is determined by the frustum's planes, the minimum and maximum coordinates of two successive boxes $\mathcal{B}^{(i)}$ and $\mathcal{B}^{(i+1)}$ will differ by a constant vector $\Delta\mathcal{B}$. With each slice being 1 unit wide, this $\Delta\mathcal{B}$ is simply $\Delta\mathcal{B} = (du_{min}, du_{max}, dv_{min}, dv_{max})$, where the $du_{min/max}$ and $dv_{min/max}$ are the slopes of the bounding planes in the grid coordinate space.

Given a slice's overlap box $B^{(i)}$, we can now incrementally compute the next slice's overlap box $B^{(i+1)}$ via $B^{(i+1)} = B^{(i)} + \Delta B$. This requires only four floating point additions, and can be performed with a single SIMD instruction. As mentioned above, once a slice's overlap box B is known, the range $[i_0..i_1] \times [j_0..j_1]$ of overlapped cells can be determined by truncating $\mathcal{B}$'s coordinates and converting them to integer values. This operation can also be performed with a single SIMD float-to-int conversion instruction. Thus, for arbitrarily sized packets of $N \times N$ rays, the whole process of computing the next slice's overlapped cell coordinates costs only two instructions: one SIMD addition, and one SIMD float-to-int conversion. The complete algorithm is sketched in Figure 3.

2.3 Efficient Slice and Triangle Intersection

Once the cells overlapped by the frustum have been determined, we intersect all of the rays in a packet with the triangles in each cell. Triangles may appear in more than one cell, and some rays will traverse cells that would not have been traversed without packets. Consequently, redundant triangle intersection tests are performed. The overhead of these additional tests can be avoided using two well-known techniques: SIMD frustum culling and mailboxing.

SIMD Frustum Culling A grid does not conform as tightly to the geometry as a kd-tree, and thus requires some triangle intersections that a kd-tree would avoid (see Figure 4). To allow for interactive grid builds, cells are filled if they contain the bounding boxes of triangles rather than the triangles themselves, further exacerbating this problem (see Section 3). However, as one can see in Figure 4, many of these triangles will lie completely outside the frustum; had they intersected the frustum, the kd-tree would have had to perform an intersection test on them as well.

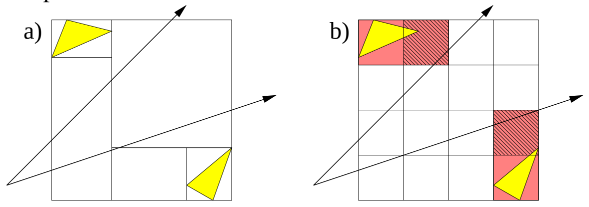


Figure 4: Since a grid (b) does not adapt as well to the scene geometry as a kd-tree (a), a grid will often intersect triangles (red) that a kd-tree would have avoided. These triangles however usually lie far outside the view frustum, and can be inexpensively discarded by inverse frustum culling during frustum-triangle intersection.

For a packet tracer, triangles outside the bounding frustum can be rejected quite cheaply using Dmitriev et al.'s "SIMD shaft culling" [2004]. If the four "corner rays" of the frustum miss the triangle on the *same* edge of the triangle, then all the rays must miss that triangle[2]. Using the SIMD triangle intersection method outlined in [Wald 2004], intersecting the four corner rays costs roughly as much as a single SIMD 4-ray-triangle intersection test. As such, for an N-ray packet, triangles outside the frustum can be intersected at $\frac{4}{N}$ the cost of those inside the frustum.

Mailboxing In a grid, large triangles may overlap many cells. In addition, since a single-level grid cannot adapt to the position of a triangle, even small triangles often straddle cell boundaries. Thus, most triangles will be referenced in multiple cells. Since these references will be in neighboring cells, there is a high probability that our frustum will intersect the same triangle multiple times. In fact,

[1]This is similar in spirit to beam tracing [Heckbert and Hanrahan 1984].

[2]Note that some (virtual) corner rays can also be computed for other than primary rays, by taking the four edges of the bounding frustum.

as shown in Figure 5 this is much more likely for our frustum traversal than for a single-ray traversal: While a single ray would visit the same triangle only along one dimension, the frustum is several cells wide, and will re-visit the same triangle in all three dimensions.

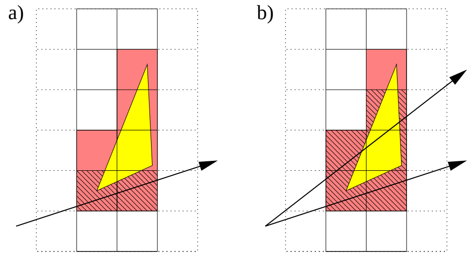


Figure 5: While one ray (a) can re-visit a triangle in multiple cells only along one dimension, a frustum (b) visits the same triangle much more often (even worse in 3D). These redundant intersection tests would be costly, but can easily be avoided by mailboxing.

Repeatedly intersecting the same triangle can be avoided by mailboxing [Kirk and Arvo 1991]. Each packet is assigned a unique ID, and a triangle is tagged with that ID before the intersection test. Thus, if a packet visits a triangle already tagged with its ID, it can skip intersection. Mailboxing typically produces minimal performance improvement in either a grid or a kd-tree for inexpensive primitive such as triangles; and may even reduce performance if gains from avoiding repeat intersection tests do not outweigh the costs of checking and updating the mailbox [Havran 2002].

As explained above, however, our frustum grid traversal yields far more redundant intersection tests than a single ray grid or kd-tree, and thus profits better from mailboxing. Additionally, the overhead of mailboxing for a packet traverser becomes insignificant; the mailbox test is performed *per packet* instead of per ray, thus amortizing the cost as we have seen before.

Impact of Mailboxing and Frustum Culling Mailboxing and frustum culling are both very useful in reducing the number of redundant intersection tests. In fact, both methods are much more powerful for our frustum grid traversal than for their original applications. Mailboxing is performed for multiple rays simultaneously, so the cost is amortized over the entire packet, and also avoids more redundant intersection tests. Similarly, due to the higher number of redundant triangle intersections in the packetized grid, SIMD frustum culling is more beneficial than in a kd-tree, where these intersections may have been avoided in the first place.

To quantify the magnitude of this impact, we have measured statistics on example scenes, using OpenRT's kd-tree system employing 4×4 packets, and our frustum grid also using 4×4 packets. For each of those, we have measured the total number of ray-triangle intersections that are performed if neither of these techniques are used, then the results when mailboxing and finally SIMD frustum culling are applied. As can be seen from Table 1, mailboxing alone reduces the number of tests by up to a factor of 2; for a kd-tree, it usually trims this by less than 10% [Havran 2002]. On top of the reductions achieved by mailboxing, frustum culling achieves yet another reduction by a factor of 4 to 9. With both techniques, the final number of intersection tests decreases by a factor of 8.5 to 14, and the absolute number of ray-triangle intersection tests roughly matches that of a kd-tree (see Table 1).

Together, mailboxing and frustum culling remedy the deficiencies of frustum traversal on uniform grids. Only one source of overhead cannot be avoided: when the bounding box of a triangle overlaps some cells traversed by a ray, but does not fall entirely outside the frustum. This scenario, however, is not limited to the grid; it also occurs in a packetized kd-tree.

scene	#tris	grid			grid ratio	kd-tree
MB/FC		n/n	y/n	y/y	n/n to y/y	
toys	11K	14.0M	8.7M	1.0M	14.0	0.82M
hand	15K	12.5M	6.0M	0.9M	13.9	0.85M
ben	78K	12.8M	6.0M	1.5M	8.5	1.1M
conf	274K	96.0M	54M	6.9M	13.9	3.7M

Table 1: Ray-triangle intersection tests for a 4×4 kd-tree and for our 4×4 frustum-grid traversal, and the impact of using mailboxing (MB) and frustum culling (FC). Mailboxing and frustum culling reduce the number of ray-triangle intersections by up to a factor of 14, to roughly as few as performed by a good kd-tree.

2.4 Extension to Hierarchical Grids

Our algorithm so far has been described for a single-level grid; however hierarchical grids generally achieve superior performance. There are several ways to organize grids hierarchically, including loosely nested grids [Cazals et al. 1995; Klimaszewski and Sederberg 1997], recursive or multiresolution grids [Jevans and Wyvill 1989], and macrocells or multigrids [Parker et al. 1999a]. Though these terms are ill-defined and often used ambiguously, they all share the same idea of subdividing some regions of space more finely than others, and thus traverse empty space more quickly than populated space. To demonstrate that our approach is not restricted to uniform grids, we have extended it with a single-level macrocell layer. Macrocells are a simple hierarchical optimization to a base uniform grid, often used to apply grids to scalar volume fields [Parker et al. 1999a]. Macrocells superimpose a second, coarser grid over the original fine grid, such that each macrocell corresponds to an $M \times M \times M$ block of original grid cells. Each macrocell stores a boolean flag specifying whether any of its corresponding grid cells are occupied.

Building the macrocell grid is trivial and cheap. Traversing it with our algorithm is rather simple: the macrocell grid in essence is just an $M \times M \times M$ downscaled version of the original grid, and many of the values computed in the frustum setup can be re-used, or computed by dividing by M. During traversal, we first consider a slice of macrocells, and determine all the macrocells overlapped by the frustum (usually but one in practice). If the macrocells in our slice are all empty, we can skip M traversal steps on our original fine grid. Otherwise, we perform these steps as usual.

Though the best value of M obviously depends on the scene, $M = 6$ has consistently shown to be a good choice for the test scenes in our system. For smaller resolutions, the savings for each macrocell step become too small to justify the additional computations; for larger resolutions the probability of finding empty regions decreases. Using macrocells yields a performance improvement of around 30%, which is consistent with improvents seen for single ray grids. Additional levels of macrocells could further improve performance for more complex models with larger grids. More robust varieties of hierarchical grids could speed up large scenes with varying geometric density, at the cost of higher build time. As our goal is to formulate a viable grid traversal for medium-size animated scenes, these have not yet been investigated.

3 Acceleration Structure Rebuild

With an animated scene, our acceleration structure is recreated every frame. Though schemes for incrementally [Reinhard et al. 2000] or hierarchically [Lext and Akenine-Möller 2001] updating a grid exist, we did not want to impose any restrictions on the kind of animations we support, and thus opted for the most general method by rebuilding the grid from scratch for every frame. We use the common scheme of choosing the number of cells to be a multiple, λ, of the number of triangles, N [Cleary et al. 1983]. Due to having the smallest surface area in relation to volume, cubically shaped

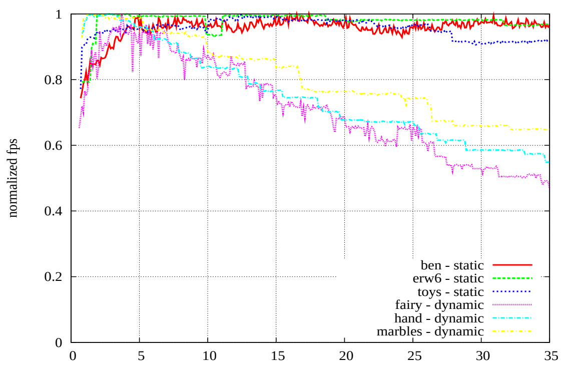


Figure 6: For several different models, this graph shows the framerate, normalized by the best time, in relation to grid size as determined by λ (on the X axis). Nearly all tested scenes, both static and dynamic, reach their optimum at approximately $\lambda \approx 5$.

cells minimize a grid's expected ray tracing cost. Thus, we choose the grid's resolution as:

$$N_x = d_x \sqrt[3]{\frac{\lambda N}{V}}, N_y = d_y \sqrt[3]{\frac{\lambda N}{V}}, N_z = d_z \sqrt[3]{\frac{\lambda N}{V}},$$

where $\vec{d}$ is the diagonal and V the volume of our grid. Fortunately, our experiments show that most scenes are insensitive to the parameter λ and achieved their best performance around $\lambda = 5$ (Figure 6), which we use for all the experiments throughout this paper.

Once the grid resolution is chosen, for each triangle we determine the cells overlapped by the triangle's bounding box and add a reference to the triangle to each of these cells. Since this is quite conservative, we also tested a more exact grid insertion scheme using an exact triangle-in-box test (e.g., [Akenine-Möller 2001]). Though the exact test could reduce the number of triangle references in the grid by more than one third, the number of ray-triangle intersection tests after mailboxing would shrink by only a few percent. For such a small gain, the significantly higher rebuild cost does not pay off, leading us to use the less accurate — but faster — bounding box test. For scenes with dominantly long, skinny, and diagonal triangles, a more accurate test may still pay off.

Since memory allocations are costly, we use a preallocated pooled-memory scheme that prevents per-cell memory allocations and fragmentation as the scene changes from frame to frame. We also use the macrocell information from the previous frame to reduce the number of cells we need to check for objects to clear. Memory layout techniques such as bricking [Parker et al. 1999b] have also been tested; but since the frustum traversal already amortizes memory accesses over the entire packet, these techniques did not result in a measurable performance difference for our scenes. Larger grids, however, may still benefit from these techniques.

In addition to rebuilding the grid, we also need to create the derived data for the triangle test described in [Wald 2004]. Though this could be avoided by storage-free triangle tests [Möller and Trumbore 1997], we found these to be slightly inferior in performance even after per-frame triangle rebuild time is taken into account; again, this could be different for much larger scenes than we tested. Furthermore, the triangle rebuild takes less time than the grid rebuild, and can be run in parallel with the grid rebuild.

4 Experiments and Results

In addition to the statistics presented above, we evaluated the performance of our algorithm on a working implementation. We first discuss the impact of the different governing parameters, and present performance for both static and dynamic scenes. If not mentioned otherwise, all experiments are performed at 1024×1024 pixels, without display, and on a dual 3.2 GHz Intel Xeon PC.

4.1 Impact of Grid and Packet Resolution

For any given scene, the performance of our frustum traversal algorithm is governed by four factors: The resolution of the grid, macrocell resolution, screen resolution, and ray packet size. As shown in the previous section, choosing the grid resolution via $\lambda = 5$ in practice works fine for the kind of moderate-sized scene we are targeting. Similar experiments show that a macrocell resolution of $6 \times 6 \times 6$ usually yields reasonable performance. Though tweaking these parameters can result in additional performance gains, these default parameters usually work well.

While grid and macrocell resolution do have an impact, screen resolution and packet size have the greatest impact on performance. For any given packet size, the cost of a traversal step is constant, but the cost for intersecting the cells in a slice increases with the number of cells that the frustum overlaps. Larger packets will benefit more from the constant cost traversal step, but are also more likely to overlap more cells. Thus, there is a natural crossover point where the savings in traversal steps from a larger packet are offset by the additional cell intersections. Obviously, this crossover point will be influenced by the model resolution, as larger models have finer grids and correspondingly smaller cells.

To find that crossover point — and thus determine the optimal packet size — we generated different resolutions of the Stanford Armadillo model and measured the rendering performance for packets of 2×2, 4×4, 8×8, 16×16, and 32×32 rays per packet. The results of these experiments are given in Figure 7. For 2×2 rays, the benefit of tracing packets is rather small, and the rendering times correspondingly high. Also not surprisingly, for packets of 32×32 rays, the frusta get very wide and performance deteriorates quickly as model complexity increases. Packets of 16×16 rays are better, but still deteriorate quite quickly. For small to medium sized models, 8×8 packets performed best until the crossover point of 250k triangles, at which point the smaller 4×4 packets begin to work better for large models. If a higher degree of coherence is given for a certain application — for example for higher resolutions, multiple samples per pixel for antialiasing or motion blur, or when computing soft shadows with lots of shadow rays to the same light source — even larger packets can still be beneficial.

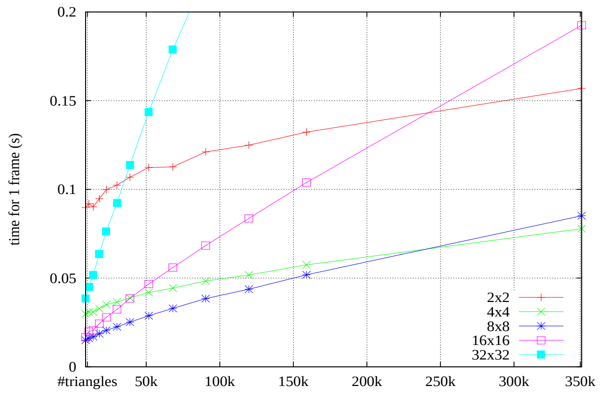


Figure 7: Static render time with varying packet sizes and different resolutions of the Stanford Armadillo. There is a crossover point around 250K triangles where 4×4 packets become more efficient than 8×8 packets. Nevertheless, both 4×4 and 8×8 show nearly the same performance over a wide range of model complexity.

4.2 Scalability with Screen Resolution

Obviously, the optimal packet size also depends on the screen resolution, as higher resolutions result in a higher density of rays, and thus allow for larger packet sizes. Given today's hardware constraints, we chose 1024×1024 pixels as a default resolution for all our experiments. In the future, high-resolution displays and supersampling will push demand for even larger images.

While the cost of ray tracing is usually considered to be linear in the number of pixels, this is not the case for our algorithm. Since higher resolutions enable larger packets, we generally see sublinear scaling in screen resolution: When increasing the screen resolution from 1024×1024 to 2048×2048 the frame rate usually drops by only a factor of 1.75-2.25, significantly less than the expected factor of 4. Weakening the linear dependence on pixel count helps overcome a major hurdle in interactive ray tracing systems.

4.3 Performance for Static Scenes

Though our main motivation was to enable ray tracing of dynamic scenes, the performance gains achieved by the packet traversal apply also to static models. To evaluate our raw ray tracing performance, we used several typical static test models for ray tracing, and rendered them with our system with the rebuild disabled. This lets us consider traversal time independently from grid build time, and facilitates a comparison between our algorithm and contemporary interactive ray tracing systems, namely OpenRT [Wald 2004] and Intel's MLRT system [Reshetov et al. 2005].

For this comparison, we chose the erw6, conference, and soda hall scenes of 800, 280K, and 2.2M triangles, respectively, as these are the only scenes for which numbers from both systems are available [Reshetov et al. 2005]. Though the axis-aligned features of these three architectural models strongly favor the kd-trees used in MLRT and OpenRT, Table 2 shows that our system, despite relatively little low-level optimization, is competitive even for these best-case scenarios for the other systems, usually being around 3-4× slower than MLRT, but consistently faster than OpenRT.

scene	#tris	OpenRT Pentium IV 2.5 GHz	MLRT Pentium IV 3.2 GHz w/ HT.	Frustum Grid Pentium IV 3.2 GHz w/ HT
erw6	804	2.3	50.7	18.3
conf	274k	1.93	15.6	4.0
soda hall	2.2m	1.8	24.1	8.0

Table 2: Static scene ray tracing performance for both the packetized grid, OpenRT, and MLRT. OpenRT and MLRT data are taken from [Reshetov et al. 2005]; all times are including simple shading, but without display. Though these three scenes are best-case examples for our competitors, we remain at least competitive.

4.4 Scalability with Model Resolution

As shown in Section 3, for moderate-sized scenes as targeted in our system, the optimal grid resolution is usually near $\lambda \approx 5$. For significantly larger models of up to several million triangles, however, the time for building a fine grid may no longer pay off for the constant number of rays shot, and a coarser grid may yield the higher aggregate performance if build time is taken into account. As shown in Figure 8, for the 10 million triangle Thai Statue, the grid rebuild for $\lambda = 5$ already takes three times longer than tracing the rays. In that case, trading grid resolution for lower rebuild times pays off, reaching the optimal aggregate performance around $\lambda = 1$. Though the thus reduced grid resolution increases the render time, this is more than made up for in saved rebuild time, resulting in a total rendering time *including* rebuild of less then 1.5s per frame. This time can be further reduced using a second thread for the rebuild, which we do not want to discuss here in detail.

For comparison, for the Soda Hall model the grid at $\lambda = 1$ can be rebuilt (using one build thread only) in a mere 110ms, and achieves a frame rate (including rebuild) of 3.5 frames/second; i.e., even including rebuild, the full 2.2M triangle model is still interactive. Though this shows that a coarser grid can pay off for much larger models than intended for our technique, in the remainder of this paper we will use the above-mentioned default resolution of $\lambda = 5$.

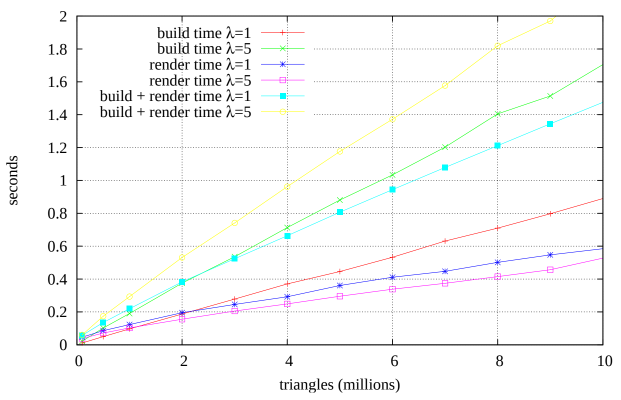


Figure 8: Rebuild and render times at $\lambda = 1$ and $\lambda = 5$ for different resolutions of the Stanford Thai Statue ranging from 100K to 10M triangles. For these large models, we use a packet size of 4×4.

4.5 Comparison to Single-Ray Grid Traversal

The somewhat surprising performance of our frustum grid on architectural models can be explained by the benefits of packetization. To illustrate this difference, we compare our approach to an optimized single-ray 3DDDA implementation of a hierarchical grid. Though this implementation uses a more sophisticated multilevel hierarchy, Table 3 shows that the packetized grid ranges from 6 to 21 times faster, depending on the scene and viewpoint. Though some of this improvement is due to our use of SIMD extensions that cannot easily be used with single-ray traversal, SIMD implementation alone usually gives only about a factor of two; the remainder is due to cost amortizations and the algorithmic improvements of the packet/frustum technique.

scene	ben	hand	toys	erw6	conf
single-ray	1.57	1.59	1.53	0.67	0.30
8×8 packets	10.6	16.1	20.0	14.0	3.2
ratio	6.75	10.1	13.1	20.9	10.6

Table 3: Static scene performance (in frames per second) for our system; and for an optimized 3DDDA single-ray grid, using a macrocell hierarchy if advantageous. Images rendered at 1024×1024 pixels on a Pentium IV 3.2 GHz CPU with 1 thread and simple shading. Our frustum traversal outperforms the single-ray variant by up to an order of magnitude.

This effect can best be explained by the number of cells visited during traversal: as we see in Table 4, compared to a single ray traversal, the frustum version visits roughly 10 to 20 times fewer cells for the 4×4 packets, and over 50 times fewer for the 8×8 packets. Due to efficient packetized slice and triangle intersection (Section 2.3), the frustum actually tests fewer triangle intersections as well; and can even do that in SIMD.

scene	ben	hand	toys	erw6	conf
# ray-triangle intersection tests (millions)					
single ray	2.96	3.58	1.97	8.90	15.70
packet 4 × 4	1.50	0.93	1.02	1.54	6.90
packet 8 × 8	5.74	2.54	2.23	2.00	20.70
# visited cells (millions)					
single ray	24.30	19.60	7.72	33.20	167.70
packet 4 × 4	2.91	0.95	0.80	2.18	16.54
packet 8 × 8	1.37	0.36	0.32	0.58	5.84
ratio 4 × 4	13.10	20.74	9.65	15.23	10.13
ratio 8 × 8	8.35	54.90	23.9	55.7	28.70

Table 4: Total number of triangles intersected and cells visited (in millions) for a single ray grid; a 4×4; and an 8×8 packet traversal. No macrocells are being used by either grid, and tests use identical dimensions for the same scene. Frustum traversal dramatically reduces both the numbers of cell visits and triangle intersection tests.

4.6 Performance for Animated Scenes

To support animation, the simplest mechanism for a grid is to rebuild the grid structure every time the geometry changes. For small to medium sized scenes, rebuilding the grid is fast; allowing the performance achieved for static scenes to be sustained during animation. For larger scenes, other techniques such as incremental or parallel rebuilds may be required to maintain interactive performance, although these techniques were not employed here. To demonstrate these performance characteristics, we used several animated scenes of various sizes and different dynamic behavior, and measured the rebuild time and rendering performance.

Animated meshes Some of the benchmark scenes are depicted in Figure 9: The "wood-doll" is a simple model with 5k triangles, resulting in a grid of 18 × 48 × 36 cells that can be built in 1ms. Without shading, this scene can be rendered at 67 frames per second; even including shading and shadows, 35 frames per second can be reached. However, consisting only of rigid body animation of its otherwise static limbs, the wood-doll could also be rendered using rigid-body animation schemes for kd-trees as proposed in [Wald et al. 2003].

To stress more complex kinds of animation, we also tested an animated "hand" model of 16K triangles, as well as "ben", a runner character of 80K triangles. Though the "ben" model is already non trivial in size, its grid of 48 × 108 × 78 cells can be rebuilt in only 14ms, resulting in a final performance of 16fps without shading, and 9fps with shading and shadows turned on. The "hand" (72 × 36 × 36 cells built in 5ms) can be rendered at 36 and 16 frames per second, respectively.

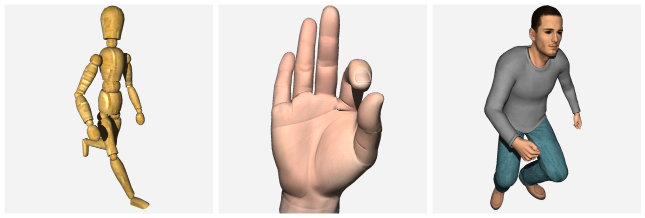


Figure 9: Some of the simpler animated models: a rigid-body wood-doll (5.3k triangles), a gesturing hand (16k triangles), and a running poser figure (78k triangles). Without shading and shadows, these scenes render at 66.9, 35.9, and 16.3 frames per second (including grid rebuild), and still at 35.1, 15.9, and 8.9 frames per second with shading, texturing, and shadows turned on.

Non-hierarchical animation Though differing in their forms of animation, both "wood-doll", "hand", and "ben" are individual models that are tightly enclosed by the grid. To demonstrate that our method is not limited to such models, the "toys" scene has a set of 5 individually animated wind-up toys that walk around incoherently, bump into each other, and even jump over each other (see Figure 10). With a total of 11K triangles, grid rebuild (for 66 × 18 × 66 cells) took 4ms, yielding a frame rate of 9-17 and 28-40fps with and without shading and shadows, respectively.

The grid's strongest advantage over other dynamic data structures is that it does not require any kind of a hierarchy to be present in the model. Thus, it can also be used for completely incoherent motion of triangles, such as explosions, physics-driven simulations, or particle sets. To demonstrate this, we modelled a scene where 110 "marbles" are dropped into a (invisible) glass box, where they participate in a rigid-body simulation (Figure 10). Since the grid does not depend on any kind of coherence in the motion, this kind of animation can be supported easily, taking just 2ms to rebuild (24 × 78 × 24 cells), and rendering at 20-24 respectively 42-50fps.

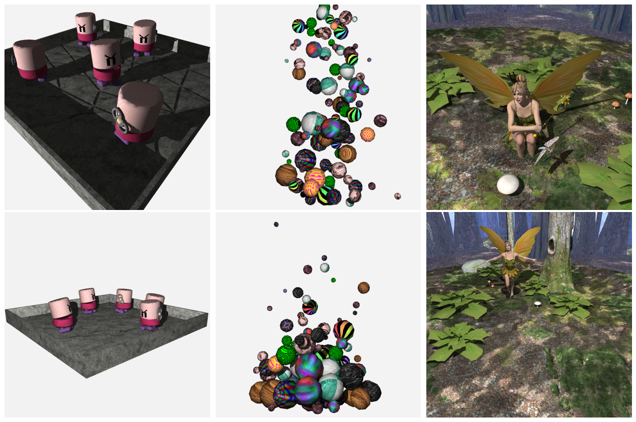


Figure 10: Examples of complex scenes composed of multiple individual objects: a) wind-up toys walking around and colliding with each other (11K tri); b) A simulation of 110 marbles dropping into an (invisible) box (8.8K tri). c) A complex scene of a typical game scenario: A skinned fairy and dragonfly dance through an animated forest (174K tri total). For the camera and light positions shown, these animations respectively run at 28.0/39.6, 41.5/50.2, and 3.3/4.3 fps without shading, and still at 9.4/17.3, 19.6/24.2, and 1.3/1.8 fps if shading, texturing, and shadows are turned on.

A real-world example While all these scenes are more or less artificial test models, the "fairy forest" scene (see Figure 10) has been chosen in particular because of its similarity to typical interactive scenarios: In this scene, a fairy and a dragonfly dance through an animated forest; both fairy and dragonfly are animated via a skinned skeleton. The scene incorporates both locally dense and largely empty regions; it is rather wide in spatial extent, requires complex shading, and consists of a total of 174K triangles, most of which are animated. Initially, we expected the high variation in scene density to be quite a challenge for our approach. However, the frustum traversal did surprisingly well, and still achieved some 3-4 and 1-2fps for shading and no shading, respectively. The fairy's grid of 150 × 42 × 150 cells can be rebuilt in 68ms.

The scenes discussed above were all modeled offline as animation sequences. This fact is not exploited at all by our traverser. The grid itself is built from a list of triangles and vertex positions every frame, neither knowing nor caring where they originate. It does not exploit the temporal coherence properties of sequenced animation, but therefore also does not depend on it. Thus, the system would work just as well for completely dynamic models. The number of triangles in the scene can easily be changed from frame to frame, and there is no restriction on the movement of existing triangles.

4.7 Shading, Shadows, and secondary Rays

So far all results have considered primary rays only. However, the true beauty of ray tracing — and its main advantage over algorithms like Z-Buffering — is that it can employ secondary rays to compute effects such as shadows, reflections, and refraction.

Among all kinds of secondary rays, shadow rays are arguably the easiest one, as they usually expose the amount of coherence that packet and frustum-based techniques like ours depend on. For most rendering algorithms, coherent shadow rays can be generated by connecting all of the primary rays' hit points to the same point light source [Wald et al. 2001]. Though certain algorithms like Monte Carlo path tracing [Kajiya 1986]), can exhibit incoherence even in shadow rays, most practical applications of ray tracing produce coherent shadow rays, and even global illumination has already been demonstrated with such rays [Benthin et al. 2003].

If we connect all surface hit points to the same point light source, the resulting shadow packets share a common origin just like primary rays, and differ from those only in that they have no concept of "corner rays". However, one can easily determine a principal march direction of the packet, and can then construct a frustum over the packet by determining the four planes that tightly bound the rays along that direction. The four edges of this frustum can then be determined quite cheaply, and can be used to perform the SIMD frustum marching and SIMD frustum culling.

Though shadow packets often *are* coherent, there is no guarantee that this is *always* the case. For example, if a primary packet hits an object's silhouette, the 3D hitpoints can be quite distant from each other, and connecting them to the same point light yields a wide frustum for which our method breaks down. In fact, for a frustum-based technique like ours the impact of some packets getting incoherent is much worse than for pure packet-based techniques, as all the triangles in the frustum would get intersected, which might comprise large parts of the scene.

Fortunately, however, this case can be detected and alleviated quite easily as already proposed in [Wald et al. 2001]: If the primary hit points are too far apart (measured, for example, by the minimum and maximum hit distances along the packet), the packet can be split into two more coherent subpackets. Without packet splitting, certain scene and light configurations can easily lead to severe performance degradation for shadows; while with splitting, shadow rays in practice work just as well as primary rays.

More general packets that do not even share the same origin would also be possible, as long as the rays are still coherent. Initial experiments have shown that this works quite well when, for example, computing soft shadows by connecting multiple surface samples with multiple light samples on the same light source. Though packet/frustum-based systems have shown that reflection and refraction rays often work surprisingly well in packet-based renderers [Woop et al. 2005; Mahovsky 2005], no experimental data are available for our coherent frustum traversal technique, yet.

5 Summary and Discussion

We presented a new approach to ray tracing with uniform grids. This algorithm elegantly allows for transferring the recent advantages in fast ray tracing — namely, ray packets, frustum testing and SIMD extensions — to grids, for which these techniques had previously not been available. The frustum based grid traversal has several important advantages. First, it has a simple traversal step, where a few SIMD operations allow for determining all the cells in a grid slice that are overlapped by the frustum. This operation has a constant cost for the entire frustum that is amortized over the entire packet of rays, and allows for a traversal step that is at least as cheap as that of a packet/frustum kd-tree. Using mailboxing and SIMD frustum culling (Section 2.3), our method performs roughly the same number of ray-triangle intersection tests as the kd-tree. Though our implementation is not *as* highly tuned as that of Intel's MLRT system [Reshetov et al. 2005], it is up to 21 times faster than known single-ray grid traversal schemes; competitive with kd-trees; and inherently supports fully-dynamic animated scenes.

Our method does possess several limitations. The very nature of using a uniform grid makes the method ill-suited for highly complex scenes with a high variation in size and density of geometry; for example, the Boeing 777 data set or the classic teapot-in-a-stadium. Though our macrocell technique works for most cases, for highly complex scenes multiresolution grids [Parker et al. 1999b], multilevel techniques [Wald 2004; Lext and Akenine-Möller 2001], or separation of static and dynamic objects [Reinhard et al. 2000], as well as mechanisms to incrementally rebuild the grid data structure may be advantageous.

Grids still suffer from common pathological cases such as large flat areas (i.e., from architectural models) where geometry overlaps numerous cells. These situations can be handled more efficiently by today's kd-tree based ray tracers and therefore, kd-trees are likely to remain somewhat more efficient for many scenes. It is also not guaranteed that our technique will perform similarly well for other kinds of secondary rays like reflection and refraction, for which the coherence can be lower than for primary and shadow rays.

Our technique may be very appropriate for special-purpose hardware architectures such as GPUs and the IBM Cell processor [Minor et al. 2005] that offer several times the computational power of our current hardware platform. Though kd-trees have been realized on both architectures, they are limited by the streaming programming model in those architectures. In contrast, a grid-based iteration scheme is a better match to these architectures, and may be able to achieve a higher fraction of their peak performance. The current method may also be appropriate for a hardware-based implementation, similar to Woop et al. [2005].

The primary motivation of this approach is to enable ray tracing of dynamically deforming models. Rebuilding an acceleration structure on each frame enables ray tracing these models without placing any constraints on the motion. As this update cost is — like rasterization — linear in the number of triangles, it introduces a natural limit for the size of models that can be rebuilt interactively. The rebuild cost is manageable for many applications such as visual simulation or games, where moderate scene sizes with several thousand to a few hundred thousand polygons are common.

Comparison to Alternative Approaches In this paper, we have shown that uniform grids are a viable option for interactively ray tracing animated scenes. Nevertheless, other alternatives exist: Even without any assumptions on the scene structure, today's $O(N \log N)$ kd-tree construction schemes in practice exhibit near-linear complexity [Wald and Havran 2006], albeit with higher constants; thus, kd-tree construction could eventually be optimized to achieve interactive rebuilds. As soon as some assumptions on the scene can be made, even more alternatives become available: If information from the scene graph could be exploited to steer the building process [Stoll et al. 2006], interactive kd-tree rebuilds may become feasible. For the case of locally smooth animations whose deformations are known in advance, Günther et al. [2006] have proposed to cluster the triangles into groups of similarly moving triangles, the motion of which is decomposed into a rigid-body transform and a residual motion, which are then handled separately. For a similar class of scenes — albeit with fewer a-priori knowledge of the deformation — Wald et al. have also proposed an approach based on merely deforming a bounding volume hierarchy [Wald et al. 2006]: Using a specially designed traversal scheme their dynamic BVH performs competitively to both grids and kd-trees. All these approaches allow for interactively ray tracing animated scenes, and more competitors are likely to appear. Hybrid approaches (e.g., a kd-tree for static content and one separate grid for each animated character) may be possible, but this has not yet been investigated.

From the performance and efficiency standpoint, among all these approaches our coherent grid traversal is arguably the most extreme one in that it is a *pure* frustum-based technique, while all other approaches are mixed packet/frustum-traversal techniques (i.e., [Reshetov et al. 2005; Günther et al. 2006; Wald et al. 2006]). Compared to a packet-based technique, a pure frustum traversal can take even better benefit from coherence *if* it exists; for example, though doubling the number of rays in the packet would increase the total number of ray-triangle intersections, the *traversal* cost would not change at all. On the other hand, with rising incoherency a pure frustum based technique will deteriorate much more quickly than the other techniques — a single incoherent ray in a packet can significantly widen a frustum, and lead to painfully degraded performance. Similarly, the frustum alone is prone to suffer worse from triangles becoming smaller: In the worst case, the frustum will intersect all the triangles in the frustum, even if those become as small as to fall in between the raster of rays in the packet. Consequently, when comparing our approach to, for instance, the BVH-based packet/frustum technique described in [Wald et al. 2006] we typically see that both techniques usually are within a factor of $\sim 2\times$ within each others performance; the BVH usually has a slight advantage — in particular for increasingly complex scenes — but can suffer worse for intentionally designed worst-case scenes, and in addition is less general in the kind of scenes it can handle.

In summary, we believe our approach to be at least competitive with other data structures and traversal algorithms known today, while at the same time being the most general of these techniques, supporting any incoherent deformation to the scene.

Acknowledgments

The fairy animation has been created using DAZ Studio; the software and base models have been graciously provided by DAZ Productions (www.daz3d.com). The Thai Statue and Armadillo are available through the Stanford Scanning Repository. We would like to thank Alexander Reshetov, Carsten Benthin, Solomon Boulos, Peter Shirley, Johannes Günther, and Heiko Friedrich for insight into their respective systems. This work was supported by the State of Utah Center of Excellence program and the U.S. Department of Energy through the Center for the Simulation of Accidental Fires and Explosions under grant W-7405-ENG-48.

References

AKENINE-MÖLLER, T. 2001. Fast 3D triangle-box overlap testing. *J. Graph. Tools 6* (1), 29–33.

AMANATIDES, J., AND WOO, A. 1987. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics '87*. Eurographics Association, 3–10.

BENTHIN, C., WALD, I., AND SLUSALLEK, P. 2003. A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum 22* (3), 621–630. (Proceedings of Eurographics).

CAZALS, F., DRETTAKIS, G., AND PUECH, C. 1995. Filtering, Clustering and Hierarchy Construction: a new solution for Ray Tracing very Complex Environments. In *Proceedings of Eurographics '95*.

CLEARY, J., WYVILL, B., BIRTWISTLE, G., AND VATTI, R. 1983. A Parallel Ray Tracing Computer. In *Proceedings of the Association of Simula Users Conference*, 77–80.

DMITRIEV, K., HAVRAN, V., AND SEIDEL, H.-P. 2004. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

FOLEY, T., AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of HWWS*, 15–22.

FUJIMOTO, A., TANAKA, T., AND IWATA, K. 1986. ARTS: Accelerated ray tracing system. *IEEE CG&A 6* (4), 16–26.

GLASSNER, A. S. 1984. Space subdivision for fast ray tracing. *IEEE CG&A 4* (10), 15–22.

GLASSNER, A. 1989. *An Introduction to Ray Tracing*. Morgan Kaufmann.

GÜNTHER, J., FRIEDRICH, H., WALD, I., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Ray tracing animated scenes using motion decomposition. In *Proceedings of Eurographics 2006*. (to appear).

HAVRAN, V. 2001. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.

HAVRAN, V. 2002. Mailboxing, Yea or Nay? *Ray Tracing News 15* (1).

HECKBERT, P. S., AND HANRAHAN, P. 1984. Beam tracing polygonal objects. In *Proceedings of SIGGRAPH*, 119–127.

JEVANS, D., AND WYVILL, B. 1989. Adaptive voxel subdivision for ray tracing. *Proceedings of Graphics Interface '89* (June), 164–172.

KAJIYA, J. T. 1986. The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, D. C. Evans and R. J. Athay, Eds., vol. 20, 143–150.

KIRK, D., AND ARVO, J. 1991. Improved ray tagging for voxel-based ray tracing. In *Graphics Gems II*, J. Arvo, Ed. Academic Press, 264–266.

KLIMASZEWSKI, K. S., AND SEDERBERG, T. W. 1997. Faster ray tracing using adaptive grids. *IEEE CG&A 17* (1) (Jan./Feb.), 42–51.

LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. Tech. Rep. 06-010, Department of Computer Science, University of North Carolina at Chapel Hill.

LEXT, J., AND AKENINE-MÖLLER, T. 2001. Towards rapid reconstruction for animated ray tracing. In *Eurographics Short Presentations*, 311–318.

MAHOVSKY, J. 2005. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary.

MINOR, B., FOSSUM, G., AND TO, V. 2005. TRE : Cell broadband optimized real-time ray-caster. In *Proceedings of GPSx*.

MÖLLER, T., AND TRUMBORE, B. 1997. Fast, minimum storage ray triangle intersection. *JGT 2* (1), 21–28.

PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P.-P., HANSEN, C., AND SHIRLEY, P. 1999. Interactive ray tracing for volume visualization. *IEEE Trans. on Computer Graphics and Visualization 5* (3), 238–250.

PARKER, S. G., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B. E., AND HANSEN, C. D. 1999. Interactive ray tracing. In *Proceedings of Interactive 3D Graphics*, 119–126.

PURCELL, T., BUCK, I., MARK, W., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *Proceedings of SIGGRAPH*, 703–712.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering*, 299–306.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *Proceedings of SIGGRAPH*, 1176–1185.

SPACKMAN, J., AND WILLIS, P. 1991. The SMART navigation of a ray through an oct-tree. *Computers and Graphics 15* (2), 185–194.

SPACKMAN, J. 1990. *Scene Decompositions for Accelerated Ray Tracing*. PhD thesis, The University of Bath, UK. Available as Bath Computer Science Technical Report 90/33.

STOLL, G., MARK, W. R., DJEU, P., WANG, R., AND ELHASSAN, I. 2006. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Tech. Rep. 06-21, University of Texas at Austin Dep. of Comp. Science.

STOLL, G. 2005. Part II: Achieving Real Time - Optimization Techniques. In *SIGGRAPH 2005 Course on Interactive Ray Tracing*, P. Slusallek, P. Shirley, I. Wald, G. Stoll, and B. Mark, Eds.

WALD, I., AND HAVRAN, V. 2006. On building good kd-trees for ray tracing, and on doing this in O(N log N). Tech. Rep. UUSCI-2006-009, SCI Institute, University of Utah.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20* (3), 153–164. (Proceedings of Eurographics).

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2006. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics (conditionally accepted, to appear)*.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: A programmable ray processing unit for realtime ray tracing. In *Proceedings of SIGGRAPH*, 434–444.

Razor: An Architecture for Dynamic Multiresolution Ray Tracing

Gordon Stoll*, William R. Mark**, Peter Djeu**, Rui Wang***, Ikrima Elhassan**

University of Texas at Austin Department of Computer Sciences
Technical Report #06-21
April 26, 2006

* = Intel Research, ** = University of Texas at Austin, *** = University of Virginia

Abstract

Rendering systems organized around the ray tracing visibility algorithm provide a powerful and general tool for generating realistic images. These systems are being rapidly adopted for offline rendering tasks, and there is increasing interest in utilizing ray tracing for interactive rendering as well. Unfortunately, standard ray tracing systems suffer from several fundamental problems that limit their flexibility and performance, and until these issues are addressed ray tracing will have no hope of replacing Z-buffer systems for most interactive graphics applications.

To realize the full potential of ray tracing, it is necessary to use variants such as distribution ray tracing and path tracing that can compute compelling visual effects: soft shadows, glossy reflections, ambient occlusion, and many others. Unfortunately, current distribution ray tracing systems are fundamentally inefficient. They have high overhead for rendering dynamic scenes, use excessively detailed geometry for secondary rays, perform redundant computations for shading and secondary rays, and have irregular data access and computation patterns that are a poor match for cost-effective hardware.

We describe Razor, a new software architecture for a distribution ray tracer that addresses these issues. Razor supports watertight multiresolution geometry using a novel interpolation technique and a multiresolution kD-tree acceleration structure built on-demand each frame from a tightly integrated application scene graph. This dramatically reduces the cost of supporting dynamic scenes and improves data access and computation patterns for secondary rays. The architecture also decouples shading computations from visibility computations using a two-phase shading scheme. It uses existing best-practice techniques including bundling rays into SIMD packets for efficient computation and memory access. We present an experimental system that implements these techniques, although not in real time. We present results from this system demonstrating the effectiveness of its software architecture and algorithms.

Outline of this document

Pages 4-15 of this document constitute the paper submitted to the SIGGRAPH 2006 conference on January 25, 2006. We have not made any changes to the document since that date, other than to de-anonymize the author list and change the page header to indicate that it is now a technical report. The paper was not accepted to SIGGRAPH and so we expect to submit a future version of the work for publication, but we wanted to make this snapshot description of our work available to the research community now.

Pages 1-3 of this document provide some updated information that did not appear in the original document, including some missing references to previous work, a list of concurrent work, and acknowledgements.

Additional previous work

BENTHIN, C., WALD, I., AND SLUSALLEK, P. Interactive ray tracing of free-form surfaces, 2004, Proceedings of Afrigraph 2004.

This paper describes a system for interactive ray tracing of cubic Bezier patches and Loop subdivision surfaces. It uses a fixed subdivision depth in contrast to Razor which subdivides adaptively. By using a fixed subdivision depth, Benthin et al.'s system avoids the need to address many of the issues with surface cracking and tunneling that Razor must address.

Concurrent work on ray tracing dynamic scenes

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. Ray tracing animated scenes using coherent grid traversal. Technical Report, SCI Institute, University of Utah, No UUSCI-2005-014, 2006. (conditionally accepted to ACM SIGGRAPH 2006).

This paper uses a grid acceleration structure for ray tracing arbitrary dynamic scenes of moderate complexity. By adapting and extending packet-tracing and frustum culling techniques originally developed for kd-trees, the system achieves performance for primary rays and shadow rays that is reasonably close to that of a cost-optimized kd-tree. No results are reported for other types of secondary rays.

WALD, I., BOULOS, S., SHIRLEY, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. Technical Report, SCI Institute, University of Utah, No UUSCI-2005-014, 2006. (conditionally accepted to ACM Transactions on Graphics).

This paper uses a bounding-volume acceleration structure for ray tracing dynamic scenes, and more specifically deformable objects. To achieve high performance, the acceleration structure must be pre-built in a cost-optimized manner for the expected object deformations.

LAUTERBACH, C., YOON, S., TUFG, D., AND MANOCHA, D. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs, 2006. Available online at http://gamma.cs.unc.edu/BVH.

This paper also directly uses a bounding-volume hierarchy as a ray tracing acceleration structure. The BVH is incrementally updated as objects deform. Since the quality of the BVH degrades with time due to the incremental updates, the system rebuilds the BVH from scratch once its efficiency drops below a pre-set threshold.

WALD, I. On building fast kd-trees for ray tracing, and on doing that in O(N log N). Technical Report, SCI Institute, University of Utah, No UUSCI-2006-009, 2006.

This paper presents a nice overview of algorithms and implementation details for constructing and traversing cost-optimized kd-tree acceleration structures. The paper also describes an algorithmic change to improve the efficiency of building cost-optimized kd-trees.

Concurrent work on ray tracing with multiple levels of detail

YOON, S. LAUTERBACH, C., AND MANOCHA, D. R-LODs: Fast LOD-based ray tracing of large models, University of North Carolina at Chapel Hill, Department of Computer Sciences Technical Report #TR06-009, 2006.

This paper describes a simple mechanism for supporting LOD in a ray tracer for static scenes. As in Razor, a single kd-tree is used to hold both original and simplified representations. The simplification used for LOD does not preserve topology, and the LOD transitions are *discrete. This approach has the advantages that the implementation is fast and that drastic simplification is possible, but the disadvantage that artifacts such as cracking and popping occur. The system provides some control over LOD artifacts by suppressing the LOD transition for a particular kd-tree node until the screen-space projection of the kd-node is smaller than a user-specific threshold measured in pixels.*

Acknowledgements

Don Fussell participated in much of our early thinking about the system design and in particular about the importance of integrating the scene graph with the acceleration structure. Okan Arikan provided several useful suggestions and helped us compare Razor's approach to that of batch rendering systems. Jim Hurley, Bob Liang, and Stephen Junkins at Intel have strongly supported this research effort. This work was funded by Intel, Microsoft Research, and the University of Texas.

University of Texas at Austin, Dept. of Computer Sciences Technical Report #TR-06-21

Razor: An Architecture for Dynamic Multiresolution Ray Tracing

Gordon Stoll* Intel Corporation
William R. Mark† UT Austin
Peter Djeu‡ UT Austin
Rui Wang§ U Virginia
Ikrima Elhassan¶ UT Austin

Abstract

Rendering systems organized around the ray tracing visibility algorithm provide a powerful and general tool for generating realistic images. These systems are being rapidly adopted for offline rendering tasks, and there is increasing interest in utilizing ray tracing for interactive rendering as well. Unfortunately, standard ray tracing systems suffer from several fundamental problems that limit their flexibility and performance, and until these issues are addressed ray tracing will have no hope of replacing Z-buffer systems for most interactive graphics applications.

To realize the full potential of ray tracing, it is necessary to use variants such as distribution ray tracing and path tracing that can compute compelling visual effects: soft shadows, glossy reflections, ambient occlusion, and many others. Unfortunately, current distribution ray tracing systems are fundamentally inefficient. They have high overhead for rendering dynamic scenes, use excessively detailed geometry for secondary rays, perform redundant computations for shading and secondary rays, and have irregular data access and computation patterns that are a poor match for cost-effective hardware.

We describe *Razor*, a new software architecture for a distribution ray tracer that addresses these issues. Razor supports watertight multiresolution geometry using a novel interpolation technique and a multiresolution kD-tree acceleration structure built on-demand each frame from a tightly integrated application scene graph. This dramatically reduces the cost of supporting dynamic scenes and improves data access and computation patterns for secondary rays. The architecture also decouples shading computations from visibility computations using a two-phase shading scheme. It uses existing best-practice techniques including bundling rays into SIMD packets for efficient computation and memory access. We present an experimental system that implements these techniques, although not in real time. We present results from this system demonstrating the effectiveness of its software architecture and algorithms.

Keywords: ray tracing, level of detail, rendering

1 Introduction

It has been a longstanding goal in computer graphics to synthesize images interactively that are indistinguishable from those we observe in the real world. Despite much progress over the past thirty years, current interactive graphics systems are still far from that goal.

It is becoming increasingly clear that the Z-buffer algorithm used in today's interactive graphics systems is likely to fundamentally limit progress towards photorealism. Within the next 5-10 years, we believe that the Z-buffer algorithm will need to be augmented or replaced with algorithms such as ray tracing that efficiently support a more general class of visibility queries. This transition to ray tracing is already well under way in offline rendering [Tabellion and Lamorlette 2004].

Recently developed interactive ray tracing systems [Parker et al. 1999; Woop et al. 2005; Reshetov et al. 2005] compellingly demonstrate that it is no longer possible to dismiss interactive ray tracing as computationally infeasible. Yet these existing systems have serious limitations that make them impractical for most mainstream interactive applications. In particular, these systems perform poorly for large dynamic scenes, and especially for scenes containing deformable objects such as human characters. Furthermore, when these systems are running at interactive rates on practical hardware they typically implement classical Whitted ray tracing, which for most applications does not provide a compelling improvement in visual quality over state-of-the-art Z-buffer rendering.

The true advantages of ray tracing visibility algorithms only become apparent with the addition of effects that are produced using distribution ray tracing [Cook et al. 1984]. These effects include soft shadows, glossy reflections, diffuse reflections, ambient occlusion, subsurface scattering, final gathering from photon maps and others. But current distribution ray tracing systems are fundamentally inefficient, particularly for dynamic scenes. Until these inefficiencies are resolved, ray tracing will not be able to replace Z-buffer rendering for most interactive applications.

In this paper, we explain why current distribution ray tracing systems are inefficient, and propose a new rendering-system architecture that reduces or eliminates the various inefficiencies. Our approach is explicitly designed to be appropriate for future interactive use. We also present an experimental system that implements our approach in testbed form. Although this system is not parallelized and performance-tuned as would be necessary to achieve interactive performance, it demonstrates the viability of the core ideas in our new rendering architecture.

It is important to understand that our motivation for this work is to develop a better understanding of how to build *future* interactive rendering systems that support the *full set of functionality* that one would want in an interactive ray tracing system. This strategy contrasts with most other recent work on interactive ray tracing, which takes the opposite approach of either restricting functionality (e.g. dynamics) or image quality (e.g. resolution, visual effects, shading) or simply running on impractical hardware (large clusters) so that the system can run at interactive rates today.

The most important new ideas in this paper are:

- The system architecture as a whole.
- A novel algorithm for representing and intersecting continuous level-of-detail surfaces in a ray tracer.
- A practical technique for lazily building a multiresolution kD-

*e-mail: gordon.stoll@intel.com
†e-mail:billmark@cs.utexas.edu
‡e-mail:djeu@cs.utexas.edu
§e-mail:rui.wang@gmail.com
¶e-mail:ikrima@mail.utexas.edu

tree each frame from a tightly-integrated scene graph holding a dynamic scene. All major system data structures except the original scene graph are rebuilt every frame.

- An approach to surface shading that partially decouples shading computations from visibility computations. This approach extends the grid-based shading approach pioneered in the REYES system [Cook et al. 1987] to a ray tracing framework.

2 The Challenges

There are several challenges to building an efficient distribution ray tracing system:

Overall system performance:

Distribution ray tracing is computationally expensive, so systems must use a variety of best-practice techniques to achieve high performance at reasonable cost. First, geometry must be tessellated into triangles before intersection testing (see e.g. [Christensen et al. 2003]). Second, the system must use an efficient acceleration structure such as a cost-optimized kD-tree [Havran and Bittner 2002] [1]. Third, the system must support aggregation of rays into packets [Wald et al. 2001]. By bundling rays into packets, cache hit rates are improved, branch mis-predict penalties are reduced, and use of register SIMD hardware such as SSE is improved. These practical considerations constrain other aspects of the system design.

Dynamic scenes:

If objects are moving within the scene, it is not possible to treat the construction of a spatial-acceleration structure as a "free" preprocessing step – part or all of the work must be performed each frame. Furthermore, if the objects undergo non-rigid motion such as deformation (as is common in skinned characters used in computer games), then it is not even possible to use the common optimization of pre-building acceleration structures for individual objects.

If the scene is complex with many occlusions (such as an entire building with occupants), then it is unacceptably expensive to build the entire acceleration structure every frame. This problem is even more acute if we want to represent each object at multiple levels of detail; in this case the finer levels of detail will cause the system to run out of memory if we store tessellated geometry in the acceleration structure.

Distribution-sampled secondary rays:

Distribution ray tracing systems cast large numbers of secondary rays. For example, many rays are cast to sample area light sources, to sample incoming BRDF directions, and for ambient occlusion computations. There are many more secondary rays than primary rays, so the cost of tracing the secondary rays and tessellating the geometry they hit dominates the ray tracing time.

Redundant shading computations:

Most ray tracers perform shading computations at each ray hit point. At high screen-space super-sampling rates, most of these shading computations are redundant. The situation is even worse for shaders that require arbitrary differential computations, since these shaders must be run three times at each hit point to compute discrete differentials [Gritz and Hahn 1996]. Redundant shading computations severely degrade overall system performance, since it is common for a renderer's surface shading costs to be greater than than that of all other rendering costs combined.

[1]This data structure is perhaps more accurately an axis-aligned BSP tree, but we use the common ray tracing parlance here

3 High-level solutions

Once the challenges above are understood, a set of potential solutions emerges. At the conceptual level these solution strategies are simple, but they each uncover more detailed challenges. In this section we explain these solution strategies and corresponding detailed challenges.

Use multiresolution surfaces to reduce the cost of tracing secondary distribution rays:

As [Christensen et al. 2003] and [Tabellion and Lamorlette 2004] have demonstrated, most secondary rays can be traced using a very coarse geometric representation of the scene. Mathematically the reason for this is that most secondary rays have large ray differentials [Igehy 1999] – i.e. they diverge strongly from each other as they progress away from their origins (Figure 1).

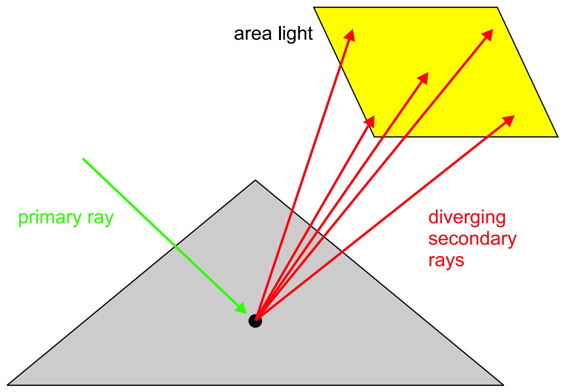


Figure 1: Distribution-sampled secondary rays diverge rapidly as they leave a surface. As Christensen *et al.* demonstrated, the ray tracing system must use a multiresolution surface representation to minimize the cost of tracing these secondary rays.

Thus, efficient distribution ray tracing for large scenes requires a multiresolution scene representation. Without this capability, the cost of generating and accessing the geometry touched by the secondary rays becomes prohibitive, particularly if this geometry is dynamic. In addition to improving memory performance, and reducing the cost of tessellation and shading, these techniques potentially improve SIMD packet tracing efficiency for the same reasons.

Multiresolution and level-of-detail techniques are well understood for Z-buffer systems, but using them in a ray tracing system presents additional challenges. Most importantly, there is no longer a single reference point (the eye point) with which to set the resolution of each surface in the scene. Instead, each ray – including secondary rays – may request an LOD that is essentially unrelated to that requested by any other ray. An important implication of this situation is that any particular surface region may be accessed at multiple levels of detail by different rays. Under these conditions, the problem of guaranteeing that surfaces are watertight is much harder than it is in a Z-buffer system. This guarantee is important to insure that reflections, refractions, and shadows do not have crack artifacts. It is unclear how or whether the multiresolution ray tracing system described by [Christensen et al. 2003] solves this problem. In future interactive systems these guarantees must operate automatically; it will be unacceptable to rely on manual per-shot tuning of LOD parameters as is done in some offline ray tracing systems [Tabellion and Lamorlette 2004].

Adding multiresolution capability to a ray tracing system makes the design of the acceleration structure more complicated. Standard space-partitioning data structures represent each surface once at a single level of detail. To store each surface at multiple resolutions, the system must use multiple acceleration structures or be able to represent the same surface more than once in a single acceleration structure. Similarly, the ray traversal algorithm must be able to select the appropriate representation of a surface for intersection tests with the ray.

These challenges are more serious in a system that builds its acceleration structure on demand from dynamic geometry. In particular, solutions that require extensive preprocessing of geometry or that require global topological knowledge are unlikely to be acceptable.

Thus the challenges are: 1) How do we provide multiresolution surfaces that are watertight for ray tracing? 2) How should an acceleration structure store multiresolution surfaces so that the overall design is efficient for dynamic geometry?

Support dynamic scenes by lazily building the acceleration structure each frame:

The most straightforward approach to supporting arbitrary dynamic scenes is to dispense with the idea of pre-building an acceleration structure, and instead build the acceleration structure each frame. To avoid unnecessary work, the acceleration structure is built lazily, so that only the portions of it needed for a particular frame are built. At the end of the frame, the acceleration structure is discarded.

This conceptually simple idea presents three major challenges: First, how do we efficiently find the subset of the scene geometry that we need to insert into the acceleration structure in any particular frame? Second, how does a system like this interface with the rest of an interactive graphics application? Third, how do we keep the cost of lazy kD-tree construction low enough to do it every frame?

Decouple shading from visibility to eliminate redundant shading computations:

In a system that uses super-sampling the desired rate for visibility computations is usually higher than that for shading computations. The obvious solution to this mismatch is to decouple the visibility computations from the shading computations in some manner.

This is exactly the approach used by the REYES system [Cook et al. 1987] and by the multi-sampling technique used in modern Z-buffer graphics systems [Akenine-Moller and Haines 2002]. However, both of these systems are designed exclusively for eye rays. A ray tracer cannot pre-shade for a single viewpoint as the REYES system does. A ray tracer also cannot assume a regular pattern for all rays as the multi-sampling technique does.

Worse yet, the goal of decoupling visibility from shading interacts in difficult ways with the goal of using multiresolution surfaces. We now have a situation where shading may need to be performed at multiple resolutions for any particular surface. This is straightforward when visibility is coupled to shading, but less so once we decouple them. How do we solve this problem?

4 System architecture

It is clear that these various individual strategies for building an efficient distribution ray tracing system interact in complex ways. We will show how to combine these strategies so that they are compatible with each other and form a single integrated system. While some pieces of our system adapt well-known approaches, other portions of the system are individually novel and require more detailed explanation. Fortunately, the major components are familiar from any standard ray tracer: the ray/surface intersection technique, the acceleration structure, and the shading system.

4.1 Multiresolution ray/surface intersection

As summarized earlier, the problem of managing geometric level of detail [Luebke et al. 2003] is considerably more challenging in a ray tracer than it is in systems such as a Z-buffer that only use eye rays or their equivalent. This difficulty is caused by the fact that it is no longer possible to choose a single level of detail for each object or surface region based on its distance from the eye point. We must switch from thinking about level-of-detail in an geometry-centric manner to thinking about it in a ray-centric manner. The level of detail required by each individual ray is a unique function of the location along that ray. Each surface region may be accessed at multiple levels of detail by different rays [Christensen et al. 2003]. This raises the question of how to generate and manage surface tessellations at different levels of detail such that each ray can be intersected with the unique geometry that it requires in a robust and efficient fashion.

Our solution to this problem applies to adaptive surface tessellation, rather than more aggressive topology modifying LOD or non-surface primitives (volumes, point clouds, etc.) In other words, the question is reduced to one of how to robustly and efficiently intersect every ray in the system with surfaces tessellated to an appropriate level of detail. There are three important requirements that constrain the solution space. First, the technique should guarantee that there will be no cracks or pinholes in the surface. Second, the technique must be entirely local in nature. This second requirement is important because our system computes everything on demand in an unspecified order, and so we cannot rely on the availability of information about a large local neighborhood or about global surface topology. Third, the technique must allow the system to cache and reuse tessellations and shading computations at tessellation vertices.

In order to generate and cache tessellations, it seems necessary to discretize the levels of detail in the system. Conventional continuous-LOD tessellation would have to generate unique geometry for every ray and thus would not allow reuse of tessellations or associated shading computations.

Unfortunately, in a ray tracer, discrete level-of-detail approaches suffer from what we call the *tunneling* problem. Figure 2 illustrates this problem, in which a ray with a series of discrete scales passes through a surface without the intersection being detected, due to the abrupt transition from one discrete scale to another at a point along the ray. The result is cracking artifacts in the image. A key challenge in ray tracing multiresolution surfaces is to design a technique that avoids tunneling while satisfying other system constraints.

Our solution is to use a hybrid scheme, in which tessellation and shading are performed at discrete levels of detail, but the system interpolates between adjacent discrete levels to produce a unique continuous surface for intersection testing against each ray. Figure 3 illustrates this scheme. We refer to the adjacent discrete levels

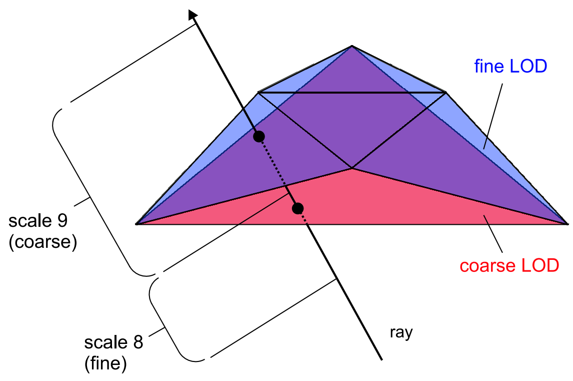


Figure 2: With discrete LODs, a ray may miss a surface completely if it changes the LOD that it is requesting at a point along the ray that is in between the surfaces produced by two discrete LODs.

of detail as the *fine* mesh and the *coarse* mesh. The meshes in our system are generated by subdivision, and each triangle in the fine mesh maps to a portion of a single triangle in the coarse mesh. The system is capable of corresponding each vertex of the finer triangle with a point on the corresponding triangle in the coarse mesh.

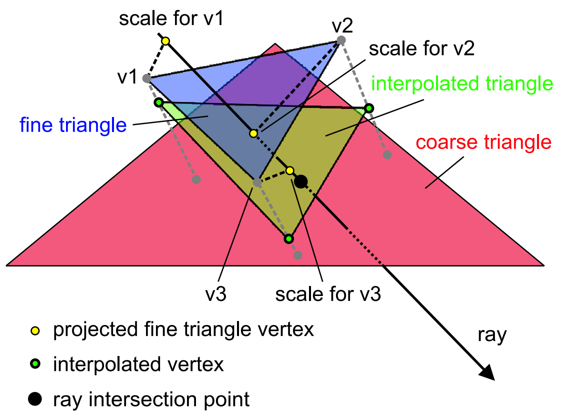


Figure 3: For each ray/triangle intersection test, the system generates a customized triangle that is specific to that ray. This customized triangle (shown in green) is generated by interpolating between triangles from two discrete levels of detail (shown in blue and in red). There is a separate interpolation weight for each vertex of the customized triangle. The weight for a vertex is determined by projecting the corresponding fine-triangle vertex (e.g. V1) onto the ray, and computing the weight from the scale value at that point on the ray (shown in yellow).

The system produces the in-between surface by interpolating between vertex positions in the fine mesh, and the corresponding points on the coarse surface. This interpolation is performed independently for each vertex in the fine mesh, with a separate interpolation weight used for each of the three vertices in a triangle. The interpolation weight for each vertex in the fine mesh is found by projecting the vertex onto the ray, and computing the weight from a continuous scale function defined on the ray. This projection and interpolation step reduces the problem to normal ray/triangle intersection, and is thus very efficient (various direct solution alternatives involve multiple cubic equations). One minor alternative would be to use distance from the origin of the ray to the vertex rather than projection of the vertex onto the ray, which might have advantages when multiple rays share an origin (such as within a SIMD packet). The interpolation weights in this scheme are associated with vertices, not triangles, so if both the fine and the coarse meshes are watertight, the interpolated mesh is as well. Note that this guarantee is for a single ray, and that we currently make no guarantees about the relation between what geometry will be "seen" by one ray versus another. We also cannot guarantee that a surface will not "misbehave" under interpolation (e.g. folding on itself, etc.). There is some commonality between this approach and eye-ray LOD techniques for terrain [Luebke et al. 2003].

The technique we have just described allows us to intersect a ray with a blend of geometry from two adjacent discrete levels of detail. The blend weights are computed from a continuous scale function along the ray. The remaining questions are how to compute the continuous scale function and how to manage transitions from using one pair of levels to using another. The continuous scale function is calculated using ray differentials [Igehy 1999], as described below. We manipulate this scale function so that the abrupt switch from using one pair of levels to using another pair occurs in a region of flat (constant) scale. These constant-scale regions are made (provably) large enough that any individual vertex will always be "seen" consistently by the ray. Space limitations prevent us from discussing this mechanism in detail, but we hope to report on it in a future publication that focuses on the LOD mechanism.

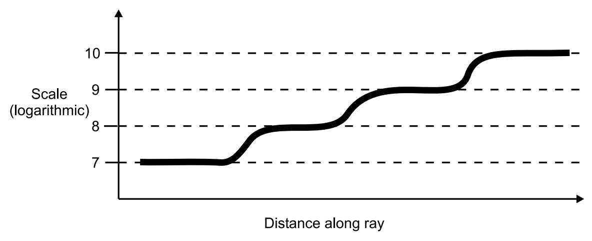


Figure 4: The system manipulates the scale values along the ray to insure that regions of varying scale are separated from each other by regions of constant scale. These regions of constant scale correspond exactly to one of the discrete levels of detail.

4.1.1 Computing scale values for rays

Each ray in our system has an associated scale that varies continuously with position along the ray. As explained earlier, this scale is used to decide which surface resolution to use for intersection testing. In this section we explain briefly how this scale is computed.

Our approach builds on the concepts of ray differentials [Igehy 1999] and path differentials [Suykens and Willems 2001], which we will summarize here. Each ray carries information with it sufficient to compute the origin and direction of its immediate neighbor. For example, the image-plane differentials provide the origin and direction of ray that is one pixel to the right and one pixel down on the image. These differentials are propagated through events such as reflections so that they continue to indicate the behavior of the neighbor ray at that point in the ray tree. Additional differentials are introduced each time the ray tree forks; for example, the system generates an additional pair of differentials for a ray when an area light source is sampled.

Each ray is best thought of as a beam with a finite cross-section. At any point on the ray, the ray differentials specify the area and geometry of the beam cross section. Most systems project this cross section onto a hit surface to compute a texture footprint.

Our system uses the differentials in a different manner, to compute a *single, isotropic* world-space scale value at each point on the ray. The scale is computed such that it is proportional to the width of the beam footprint. In the case of an anisotropic beam cross-section, the minimum width is used. By choosing the minimum width we guarantee that we tessellate and shade at a rate in each dimension equal to or greater than the desired rate.

Our system currently simplifies the problem of computing footprints from arbitrary path differentials by retaining just the most important differential pair along with the scale value used at the last intersection point. Area light rays provide an example of how this simplification works: as they first leave the surface, their footprint is a constant determined by the spacing on the surface, but as they move further away from the surface, the area-light differential pair takes over, allowing the footprint to grow rapidly thereafter. For some effects, it might be necessary to track more differentials.

Before tracing rays, the system must partition each ray into a series of segments. Each segment represents the portion of the ray that can be intersected with a single pair of our discrete geometry levels. To determine each cut point between segments, the system must invert the equation that computes the scale value from the differentials as a function of position along the ray. In the general case this inversion requires solving a quadratic equation, although in common cases such as eye rays and area-light shadow rays the equation is linear. Our system uses division to solve the linear equation and otherwise uses the quadratic formula.

4.1.2 Subdivision implementation

The geometry for each discrete scale is generated by adaptive tessellation of subdivision patches. We currently use a very simple implementation of the Loop subdivision scheme for triangles [Loop 1987], with support for crease edges [Hoppe et al. 1994] and texture coordinates [DeRose et al. 1998]. Our implementation of subdivision operates on vertex grids formed from triangles pairs [Pulli and Segal 1996]. Vertex grids larger than a specified threshold are broken up into smaller grids to allow for adaptivity and lazy evaluation in both tessellation and shading. Currently, the target grid size is 5x5 vertices (32 triangles). Once subdivision has been applied twice to reach this 5x5 size, all further grids will be of this size (i.e. the vast majority of the grids in the system). The system computes bounds on the limit surface for each patch and sub-patch using the technique described by Kobbelt [Kobbelt 1998]. These bounds are used during the kD-tree construction.

As in any adaptive tessellation system, there is the possibility of cracks forming between adjacent patches. In our system, it is easiest to consider the patch cracking problem for the case of a single discrete scale applied to every patch on a surface. It turns out that solving the patch cracking problem for this single-scale case is sufficient to solve the problem for the general case as well, since our multiresolution geometry-interpolation scheme will work correctly if the geometry for each discrete scale is watertight. We use a simple local crack fixing technique [Owens et al. 2002] to insure that each discrete scale is watertight.

Our current subdivision system has serious shortcomings for our application in that it cannot actively target a specific edge length (our world space scale threshold) and it cannot actively control patch aspect ratios. It simply subdivides each patch into four pieces, roughly evenly in each parametric direction. We initially chose explicit Loop subdivision for its simplicity and to allow the system to be tested with existing triangle-mesh content. Using Catmull-Clark patches instead [Catmull and Clark 1978; DeRose et al. 1998] would facilitate independent and variable subdivision in both parametric directions, be a better match for modern animated content, and generally be a better long-term choice.

4.2 Dynamic Multiresolution Acceleration Structure

The system utilizes two primary data structures: a scene graph and a multi-scale kD-tree acceleration structure. The upper levels of the scene graph contain the original geometric primitives comprising the scene (subdivision surface patches) and are relatively persistent, updated from frame to frame according to animation or interaction as with any typical scene graph system. All other data in the system is rebuilt from scratch every frame. The lower levels of the scene graph are built out during the course of rendering a frame using the results of subdivision operations applied to the original patches. Hierarchical bounding volumes are maintained throughout this extended scene graph.

The multi-scale kD-tree acceleration structure must support the interpolating intersection technique described earlier. This technique breaks individual rays into segments, each of which is intersected against geometry generated from adjacent discrete levels of detail. Conceptually, we could build a separate kD-tree for every pair of adjacent discrete levels. The geometric primitive at the leaf nodes in each such tree would be a triangle pair consisting of a finer-level triangle paired with the corresponding portion of a coarser-level triangle. We elaborate on this basic scheme in three ways: 1) the kD-trees for all of the level pairs are merged into a single data structure, 2) this merged data structure is built lazily from the scene graph, and 3) the merged data structure stores grids (small regular meshes) of vertices at its leaf nodes rather than storing individual triangle pairs.

4.2.1 Merged kD-trees

Figure 5 illustrates our kD-tree. The multiresolution capability is provided by allowing each node to fill a dual role: when traversed at a particular scale the node acts as a leaf node containing geometry at that scale, but when traversed at a finer scale the node acts as an interior node with a split plane and child nodes. This multi-scale kD-tree is similar to that described by [Wiley et al. 1997] for a multiresolution BSP tree, although our system uses a hierarchical nesting of LODs whereas theirs used n-ary LOD-selection nodes. Also, our approach does not restrict the location of cut planes with respect to the geometry as theirs did.

The multi-scale kD-tree acceleration structure can be thought of as numerous separate kD-trees, each built for a different discrete scale pair, layered on top of each other. The leaves of a kD-tree built for a single pair become a frontier of internal nodes in the combined tree. If we set aside the laziness of the building process for now, the algorithm for building the tree is as follows:

```
1) Create a root node for the kD-tree with the
   scene bounding box and the scene graph root node.
2) Set the current node to be the root.
3) Set the current discrete LOD level to be the
   coarsest supported level.
4) Subdivide the geometry at the current node until
   it satisfies the current discrete LOD criteria.
5) Build out the kD-tree from this node until the
```

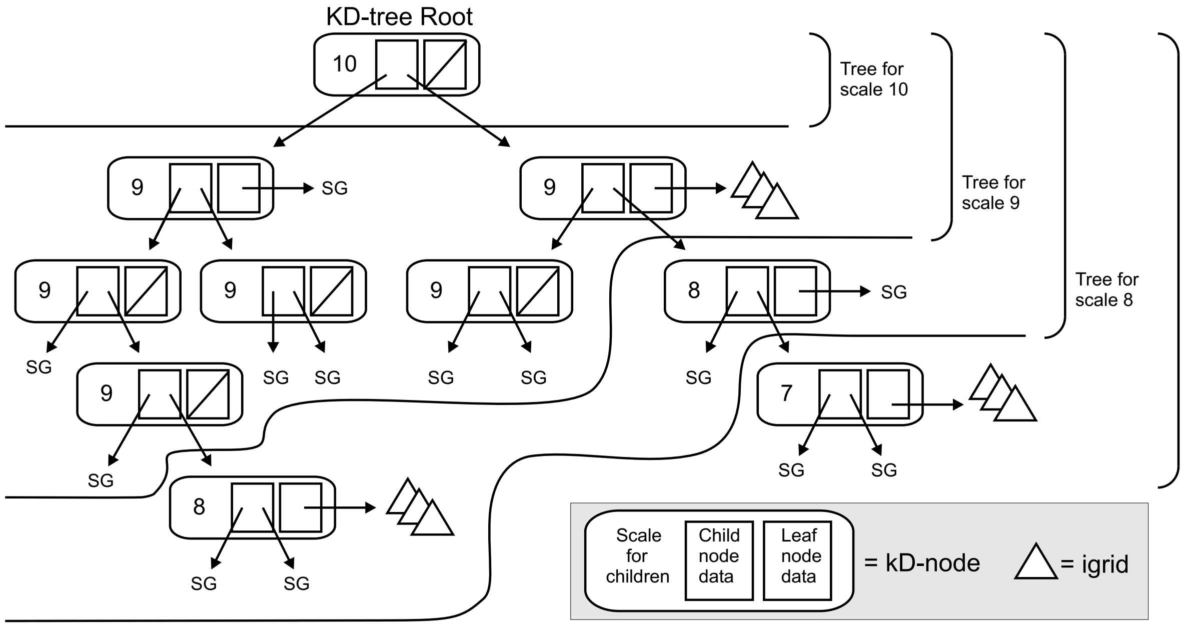


Figure 5: Multi-scale dynamic kD-tree. 'SG' designates a pointer into the scene graph.

```
   tree termination criteria are satisfied.
6) Retain the current geometry
   (these nodes are effectively leaves for
   the current discrete LOD level).
7) Set the current discrete LOD level to the next
   finer level.
8) Goto 4.
```

As mentioned, we perform traversal for a single ray segment (and thus a single discrete level pair) at a time. Traversal is very nearly identical to normal kD-tree traversal, and thus is similarly efficient. Our kD-tree data structure is specifically designed to utilize known best practices for high-performance kD-tree traversal [Wald et al. 2001; Reshetov et al. 2005], including nearly identical SIMD packet traversal code and an eight-byte internal node record. Rays simply descend through the merged tree treating all nodes as internal (split) nodes until they reach either an empty leaf or a node which is a leaf for the segment's discrete level pair (i.e. from step 6 above). Note that we have not yet attempted to merge the traversal of the individual segments of the rays into a single continuous traversal operation. We simply break rays up into segments and then process each segment against the merged data structure in order along the rays. This is a significant inefficiency which we intend to address in the future.

Split planes in our tree are chosen using a simple surface area cost metric [Havran and Bittner 2002], using bounding boxes for split candidate determination (as opposed to more exact geometry).

4.2.2 Lazy Construction

The basic idea of lazily tessellating and storing geometry has been used for a long time. Arvo and Kirk lazily build a 5D acceleration structure for a ray tracer [Arvo and Kirk 1987]. The RenderMan interface [Pixar 2000] supports a callback to user code for on-demand generation of geometry within a bounding box at the needed resolution, and there are now several ray-tracing implementations of the RenderMan interface (e.g. [Gritz and Hahn 1996]). [Pharr and Hanrahan 1996] builds displacement maps on demand in a ray tracer.

But in addition to being desirable for efficiency in large or highly occluded scenes, laziness is required in order to support multiresolution geometry. Building out the entire data structure across the entire range of interesting levels of detail would be prohibitive.

Thus, our system builds its kD-tree lazily. A node encountered in our tree during traversal may have been previously marked as "lazy". Such a node has no children or geometry. Instead, it has a pointer to a linked list of as-yet unprocessed nodes in the scene graph. Conceptually these scene-graph nodes can be any node in the scene graph: an original interior node; an original leaf node (base patch); or a per-frame temporary node consisting of a sub-patch produced by earlier subdivision and patch-splitting steps. However, our current implementation only uses the last two cases. The information in the lazy kD node's linked list is sufficient to build the missing portion of the kD-tree if it is needed. This mechanism is similar to one used by Ar et al to build BSP trees for collision detection [Ar et al. 2002].

At the beginning of every frame, kD-tree construction is initialized with a single root kD-tree node containing the bounding box of the entire scene and a single pointer to the root of the scene graph. All further kD-tree building is triggered by traversal operations during ray tracing.

4.2.3 Low-Level Grid Intersection Structures

The geometry in the system is managed in grids (small regular meshes) rather than individual triangles, and the system also performs lazy evaluation at the granularity of a grid. A kD-tree node that serves as a leaf node at a particular scale may have the associated geometry marked as "lazy". Such a node has a linked list of geometry (patches and sub-patches), but the final grid data structures have not been constructed yet. When such a node is intersected, the final vertex data is computed. In addition, a simple bounding volume hierarchy is constructed based on the internal structure of the tessellation. This low-level acceleration structure (the "igrid" in figures 8 and 9) avoids computation of several levels of kD-tree splits at the bottom of the tree and likely has better computational regularity and coherence properties as well. This data structure is traversed with a non-recursive fixed order ("flattened") traversal scheme as per [Smits 1998]. As described above in Section 4.1.2, the target grid size (and in fact the size of the vast majority of grids in the system)is 5x5 vertices or 32 triangles.

4.2.4 A note on efficiency

This lazy kD-tree-building mechanism is extremely effective. As mentioned above, laziness is required in order to efficiently support multiresolution geometry. What is less obvious is the fact that multiresolution geometry, or some other form of hierarchical clustering, makes lazy evaluation much more effective.

Standard kD-tree build algorithms build top-down starting from the full geometry description of the scene and the scene's bounding box. Unfortunately this leads to a situation analogous to sifting through individual grains of sand to figure out where to split a beach in half. The time to compute the single split at the root node is linear in the amount of geometry in the scene. This is the case even for an "optimal" n log n build algorithm. The kD-tree is heavily "top-loaded" in computational cost, greatly impairing the benefits of lazy evaluation (you always touch the root, obviously).

Building a merged multiresolution tree as described above makes the cost of the root node split proportional to the amount of geometry at the coarsest supported level of detail, and similarly removes the top-loading of computational cost from the entire build process. Our results for tree building performance clearly demonstrate the advantages of this technique. We currently only utilize the natural "clustering" provided by repeatedly subdividing and breaking up patches. Further efficiency could be achieved by utilizing the clustering information inherent in a well-structured scene graph. We expect that the observed performance of kD-tree building for a well-structured scene graph using these techniques (including lazy evaluation) will be linear in the amount of geometry actually intersected by rays.

4.3 Split-phase shading

The design of our shading system was driven by the desire to decouple shading from visibility. The REYES system [Cook et al. 1987] accomplishes this goal, but in a system that only supports eye rays. Our goal was to extend the REYES approach to a ray tracing framework. Like REYES, our goal is to perform shading computations at the vertices of a finely tessellated polygon mesh and then interpolate to specific hit points, rather than shading at the hit points themselves. The REYES algorithm has amply demonstrated the benefits of this technique: shading calculations can be performed in highly regular and coherent batches in their natural coordinate space on the surface, and a variety of otherwise tricky operations (arbitrary differential calculations, displacement shading) are simplified.

Another critical performance characteristic is that this technique creates a separation between functions which can be band-limited *a priori* from functions which cannot. In REYES, this means that procedural shaders (expected to band-limit themselves) are separated from visibility calculations. The extremely expensive procedural shading operations can be performed less frequently, at the vertices of the grid, while the cheaper-to-evaluate but ill-behaved visibility function is super-sampled.

Our system uses this concept by leveraging the system's multiresolution representation of geometry. Shading is explicitly factored into two phases. Operations in the first phase are performed at the vertices of grids. The functions calculated in phase one are expected to be band-limited to the frequency of the sampling implied by the tessellation of the grid. Additionally, as the results are cached and reused by the system, these values must be independent of viewing direction. The first phase of shading is calculated lazily the first time that a ray strikes the given grid and requires the results.

The second phase of shading is more typical of a ray tracer. When a ray strikes a grid, the results of the first phase are fetched (following lazy evaluation of the first phase if necessary) and interpolated to the hit point. These values are available as parameters to phase two. Shading in this phase is as flexible as shading in any typical ray tracer. In typical use a BRDF function would be generated from the results available from phase one, and distribution sampling of the BRDF would be performed by casting secondary rays as necessary.

A similar split-phase shading model has been applied previously in physically-based rendering systems [Pharr and Humpreys 2004] in order to enforce properties such as BRDF reciprocity. The separation in our system is more pragmatic and performance-oriented. Shading operations should be factored into phase one as much as possible, with the remainder in phase two, without necessarily considering physical interpretations. Creative abuse of the shading system is certainly an option, such as using various mapping tricks in either of the phases, or casting various physical-or-otherwise secondary rays in phase one. Variants on irradiance caching based on casting rays in phase one are certainly possible.

Altogether, there are four sources of performance improvement in this shading system. First, redundant shading computations caused by visibility super-sampling are reduced. Second, phase one is performed on a grid, so that shading "derivative" computations may be computed by discrete differences with neighbors, rather than by executing the shader three times for each hit point as is standard in ray tracers [Gritz and Hahn 1996]. Third, the grid structure of phase one shading makes it amenable to acceleration by SIMD mechanisms like x86 SSE. Grid-based shading also improves memory-access locality. Fourth, the scheme improves the efficiency of SIMD ray packets because there are fewer distinct kinds of phase two shaders than kinds of combined shaders.

Our experimental system uses simple phase one shaders that read and filter surface colors from a texture map and compute normal vectors from a bump map. Our phase two shading currently includes area light source sampling, mirror reflection, hemisphere sampling of ambient occlusion, and simple diffuse and Schlick [Schlick] BRDF evaluation. It remains to be seen how well programmable shading can be adapted to this shading scheme.

5 Results

We have evaluated our prototype implementation using a courtyard scene with several animated skinned characters and two area lights, as shown in the accompanying video. Our rendering and timings were performed using single-threaded code running on a single 3.2GHz Intel Pentium 4 Processor with 2GBytes of memory.

The courtyard scene contains over 31,000 Loop subdivision patches, with 2,150 patches in each of the characters. Figure 6 shows a single frame from the animation, rendered at 512x512 resolution with 4x image-space super-sampling and 4x sampling of each area light from each of the four image-space samples. Figure 7 shows the elapsed time for rendering this frame with a range of tessellation rate settings. At the coarsest setting, the maximum on-screen area of a triangle is 37.5 pixels, and roughly 9,500 32-triangle grids are actually hit by rays and shaded. At the finest

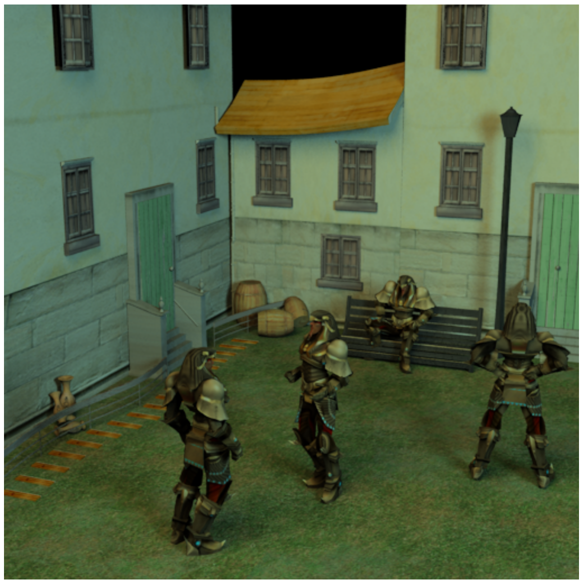


Figure 6: Courtyard Scene

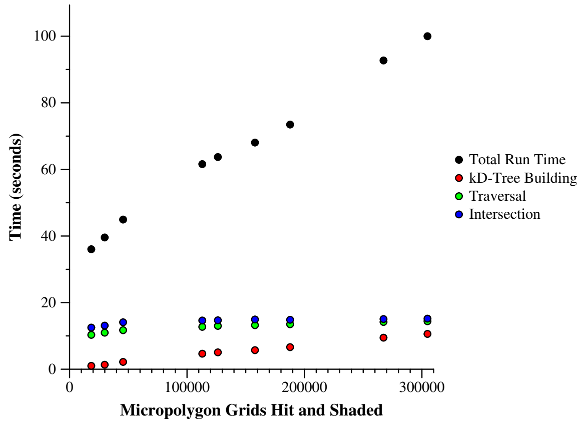


Figure 7: Performance on the Courtyard Scene. The scene was rendered at a range of tessellation rate settings, resulting in 9,500 to 300,000 visible micropolygon grids (each containing 32 triangles).

setting the maximum triangle area is one pixel, and over 300,000 grids are hit and shaded.

As can be seen in the figure, traversal and intersection consume from 10-15 seconds each, and are fairly insensitive to the amount of geometry. The time spent on calculating kD-tree splits is roughly linear with respect to the amount of geometry and is remarkably small, at roughly 10 seconds for over 300,000 grids (containing over nine million triangles). Because of the merged kD-tree and build algorithm, the maximum number of candidates considered for any single split (the split at the root) is only proportional to the number of grids at the coarsest scale, rather than the finest. Lazy building provides additional efficiency. As a result, on-the-fly conversion from the scene graph into a kD-tree is clearly not a bottleneck.

A large fraction of the run time is not being spent in any of these three fundamental ray tracing operations. The bulk of the rest of the time is being consumed by subdivision surface calculations, ray differential calculations, and MIPmap filtering, all of which are completely unoptimized in the prototype. The subdivision calculation and MIPmapping costs in particular are exacerbated by over-tessellation, addressed below.

The ambient occlusion sequence in the accompanying video was rendered with 6x image-space super-sampling and 26x hemisphere occlusion sampling at each image-space sample for a total of 162 rays cast per pixel. On a similar machine, these frames are roughly three times as expensive as the multiple-area-light frames, averaging 292 seconds each.

5.1 Over-Tessellation

The prototype implementation suffers from severe over-tessellation, producing approximately thirty times the number of micropolygons that would be expected in the ideal case (i.e. simply the screen area divided by the requested micropolygon area). There are four primary factors that cause this over-tessellation:

Non-Uniform Edge Lengths in a Single Grid Our simple Loop subdivision system cannot adapt to varying edge lengths within a single grid. Large variation in edge lengths can be caused by highly elongated triangles in the initial mesh, or by pairing a small triangle with a large triangle in the base grid. As a result, a single grid may have many edges that are much shorter than the maximum length edge which drives tessellation.

Subdivision Occurs in Discrete Steps Each iteration of our subdivision scheme reduces edge lengths by about a factor of two and triangle area by about a factor of four. This discretization is too coarse to precisely target a desired maximum edge length.

Shading-Grid Scales are Discrete For the two-level intersection scheme, a ray requires geometry grids for the two discrete scales that bracket the continuous scale that the ray actually wants. One of these geometry grids is tessellated at a finer scale than is strictly necessary for the continuous scale wanted by the ray.

Viewing Angle We use an isotropic world-space scale metric to control tessellation. Rays that strike surfaces at shallow angles may request geometry that is over-tessellated with respect to projected area.

The first two causes could be largely eliminated with a more sophisticated subdivision surface system capable of tessellating at varying rates in each parametric direction. Such a system could also ameliorate the third cause. If the subdivision system can consistently generate discrete scale levels which differ by a factor of two in area rather than four, then the system's discrete scale values can be set correspondingly, and the finer-level geometry needed by the ray will similarly be off by a factor between one and two rather than between one and four. We believe that this third cause can also be ameliorated by adjustments to the mechanism for breaking rays into segments. The fourth and final cause (viewing angle) is significantly more difficult to address, as the isotropic world-space scale metric is a basic component of the architecture.

We have measured the separate impact of each of these causes for the courtyard scene at a requested tessellation rate of one pixel per triangle. The breakdown is as follows: non-uniform edges = 3.47x, subdivision discretization = 2.26x, grid-scale discretization = 2.19x, and off-axis viewing = 1.84x. Combining these four measurements yields 31.6x over-tessellation, which closely matches our observed total deviation from the ideal. For this scene, at least, the breakdown of the various causes of over-tessellation indicates that we can eliminate much of the tessellation problem with more implementation work – the first three causes combined account for 17.2x over-tessellation and can be largely eliminated.

5.2 Memory Consumption

Because of the over-tessellation the prototype implementation suffers from high memory consumption at the desired shading rates. As an interim measure, we use a workaround that takes advantage of the fact that the system computes most data structures on demand. We set a maximum memory consumption level and when that level is hit all data structures other than the persistent top-level scene graph are simply discarded. Rendering continues, with needed portions of the data structure being built or re-built on demand. This scheme is a primitive version of a caching scheme that we plan to implement later; the discard operation is equivalent to a complete cache flush.

For the courtyard scene rendering described above, at the finest tessellation setting, this memory flush operation occurs 11 times during the course of the frame. This cost is included in the times shown in figure 7; i.e. the total rendering time at the finest setting was approximately 100 seconds, and the aggregate kD-tree build time was approximately 10 seconds, even though the kD-tree and all tesselated and/or shaded geometry was simply thrown away eleven times during the course of the frame. It is difficult to precisely measure the impact of this flushing, but our best estimate is that the impact on total run time is less than 10%. This estimate is based on experiments where we varied both the tessellation settings and the memory flush thresholds.

The minimal performance impact of this crude mechanism indicates that a more sophisticated software caching scheme is likely to be very effective. A variant of this mechanism would also provide a simple but efficient coarse-grained parallelism technique. Rather than dealing with the synchronization issues inherent in the lazy construction of the data structures, each thread would simply build its own data structures.

6 Related work

Our work builds on five major foundations: 1) The basic principles of ray tracing and distribution ray tracing [Appel ; Whitted 1980; Cook et al. 1984; Igehy 1999], summarized nicely in [Pharr and Humpreys 2004]; 2) The REYES system for efficient, high-quality rendering of eye rays [Cook et al. 1987]; 3) Work on multiresolution ray tracing [Christensen et al. 2003] and related data structures [Wiley et al. 1997]; 4) Work on efficient ray tracing acceleration structures [Havran and Bittner 2002; Reshetov et al. 2005; Wald et al. 2001]; 5) Work on subdivision surface representations [Loop 1987; Hoppe et al. 1994; DeRose et al. 1998].

In this section we compare various aspects of our system design to alternative approaches.

6.1 Caching schemes for shading, irradiance, and radiance

Razor's mechanism for partially decoupling shading from visibility has two characteristics: First, it *interpolates* values computed at nearby points on the surface. Second, these values computed at nearby points are computed on demand and reused; that is, they are *cached*. Razor currently caches and interpolates just material properties (i.e. the BRDF), although the architecture would easily support caching of irradiance [Ward et al. 1988; Ward and Heckbert 1992] or a compact representation of radiance [Arikan et al. 2005], and we plan to implement this capability in the near future.

Our caching and interpolation mechanism was inspired by REYES [Cook et al. 1987]. REYES assumes a single viewing-ray direction, and thus can evaluate, cache, and interpolate the *entire* shading computation rather than just the BRDF. Both Razor and REYES cache samples on a grid associated with the surface and use regular data interpolation. This explicit association of samples with a surface neighborhood has the potential to facilitate a large class of interesting optimizations. REYES explicitly generates and caches results for just a single resolution of each surface, whereas Razor can cache results for several several different resolutions of a single surface. In both systems, each cached sample is associated with a particular resolution and may thus be pre-filtered.

Irradiance caching [Ward et al. 1988; Ward and Heckbert 1992; Tabellion and Lamorlette 2004] and radiance caching [Arikan et al. 2005] systems cache just irradiance or radiance, rather than caching the results of the full shading computation. Photon mapping systems [Wann Jensen 2001] behave similarly. All of these systems typically cache data as individual points in a global 3-D data structure such as an octree or kD-tree, and thus do not explicitly associate cached points with a particular 2-D surface. This has both the advantage and disadvantage that points from nearby surfaces or from nearby patches on the same surface may be accessed during retrieval, which is not done in our system. These systems also use scattered data interpolation rather than regular interpolation, and treat each sample as a true point rather than as a filtered sample associated with a particular surface resolution as Razor does.

6.2 Ray tracing dynamic scenes

A variety of techniques have been proposed for ray tracing dynamic scenes. We discuss these techniques in turn and compare them to our approach.

For the special case of rigid objects, it is possible to pre-build an acceleration structure for each object and transform rays into the object coordinate system during ray tracing [Lext and Akenine-Moller 2001; Wald et al. 2003]. A top-level acceleration structure is still required; some systems use a bounding volume hierarchy, and others rebuild a complete top-level kD-tree every frame [Wald et al. 2003].

It is more difficult to efficiently support unstructured motion (also referred to as non-rigid motion). Several systems rely on building a complete kD-tree for these objects [Wald et al. 2003], but this approach performs unnecessary work for occluded objects. It is also possible to directly trace rays through the scene graph since it is a bounding volume hierarchy, which may be used directly as an acceleration structure [Rubin and Whitted 1980]. However, this approach is less efficient than using a kD-tree for ray tracing acceleration.

Several systems [Torres 1990; Chrysanthou and Slater 1992; Reinhard et al. 2000; Luque et al. 2005] dynamically update an acceleration structure rather than lazily rebuilding it each frame as we do. However, we believe that it is simpler and more efficient to lazily re-build the tree, especially since it appears to be difficult to guarantee that a kD-tree remains optimized for traversal cost [Havran and Bittner 2002] when it is incrementally modified.

6.3 Interface between scene graph and ray tracer

Our system closely couples the scene graph to the ray tracing acceleration structure, as proposed by [Mark and Fussell 2005]. This system organization enables the system to lazily build the acceleration structure every frame. This organization is very different from the classical one in which the two data structures are separated by an API layer such as OpenGL [OpenGL Architectural Review Board 2003] (for Z-buffers) or OpenRT [Dietrich et al. 2003] (for ray tracers), and has implications for the design of ray tracing hardware which are discussed in [Mark and Fussell 2005].

7 Discussion and Future Work

Razor's high-level system architecture and algorithms are *explicitly designed* for future interactive use, even though the performance of our current implementation is multiple orders of magnitude away from interactive performance for our target imagery. As with any complex new system design, we expect a rapid ramp in performance as we address issues that we have identified in the first working implementation. As our performance results show, most of our execution time is spent in parts of our system that are unoptimized and whose execution time grows linearly with micropolygon count. By addressing issues with over-tessellation and by aggressively tuning all aspects of system performance, we believe that we can improve performance by 10-20x. An additional 5x or more in performance should be possible by parallelizing our system for multi-threaded, multi-core processors, even using the simple scheme mentioned above. Thus, we believe that our system will soon be 50-100x faster on commonplace desktop hardware without any fundamental changes to the system architecture.

Our experimental implementation current lacks several features that the overall system architecture would easily support. Displacement mapping and depth-of-field would be easy to add and virtually free, just as they are in REYES. For diffuse surfaces, it would be simple to cast hemisphere-sampling secondary rays in phase one of shading, yielding a capability similar to irradiance caching.

Our experimental system also lacks some useful features that would require more effort to support, including motion blur and more aggressive topology-modifying LOD.

Working within our system feels qualitatively different from working within any other ray tracing framework we've used. In particular, the notion that almost all operations are performed with respect to a specific spatial scale is very powerful. For example, most "epsilon" values within our system are set relative to the current scale, rather than to fixed global values.

8 Conclusion

We have presented a new software architecture for a dynamic-scene ray tracer. The architecture represents surfaces at multiple resolutions, integrates scene management with ray tracing, builds most of its per-frame data structures lazily, and partially decouples shading computations from visibility computations. The architecture is designed to efficiently support the needs of distribution ray tracing, including future interactive systems.

We believe that the goal of building an efficient distribution ray tracer for dynamic scenes leads almost inevitably to a design using principles similar to ours. Efficient support for distribution-sampled secondary rays requires multiresolution surfaces, and efficient support for multiresolution surfaces requires a lazily-built acceleration structure. Allowing shading operations to be performed on surface neighborhoods is in many respects more natural than performing them at intersection points and will likely prove to be more efficient in an optimized implementation.

The experimental system that we have built is not a product-quality system, and in its current form leaves some important questions unanswered. However, our implementation clearly illustrates the potential of our system architecture by successfully integrating a complex set of ideas into a working system with powerful new capabilities.

We believe that many of the principles used in our system will be important to the design of future interactive rendering systems, and we hope that others in the graphics community can benefit from learning about our ideas and the results from our experimental system.

9 Acknowledgments

Removed for review.

References

AKENINE-MOLLER, T., AND HAINES, E. 2002. *Real-Time Rendering*, 2nd ed. AK Peters.

APPEL, A. Some techniques for shading machine renderings of solids. In *AFIPS 1968 spring joint computer conf.*, vol. 32, 37–45.

AR, S., MONTAG, G., AND TAL, A. 2002. Deferred, self-organizing bsp trees. In *Eurographics 2002*.

ARIKAN, O., FORSYTH, D. A., AND O'BRIEN, J. F. 2005. Fast and detailed approximate global illumination by irradiance decomposition. *ACM Trans. Graph. 24*, 3, 1108–1114.

ARVO, J., AND KIRK, D. 1987. Fast raytracing by ray classification. *SIGGRAPH 87 21*, 4 (July), 55–64.

CATMULL, E., AND CLARK, J., 1978. Recursively generated B-spline surfaces on arbitrary topological meshes.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Eurographics 2003*.

CHRYSANTHOU, Y., AND SLATER, M. 1992. Computing dynamic changes to BSP trees. In *Proc. of Eurographics 1992*.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 137–145.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The REYES image rendering architecture. *SIGGRAPH 87 21*, 4 (July), 95–102.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision surfaces in character animation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 85–94.

DIETRICH, A., WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. The OpenRT application programming interface – towards a common API for interactive ray tracing.

GRITZ, L., AND HAHN, J. K. 1996. BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools 1*, 3, 29–47.

HAVRAN, V., AND BITTNER, J. 2002. On improving KD-trees for ray shooting. In *Proc. of WSCG 2002 Conference*.

HOPPE, H., DEROSE, T., DUCHAMP, T., HALSTEAD, M., JIN, H., MCDONALD, J., SCHWEITZER, J., AND STUETZLE, W. 1994. Piecewise smooth surface reconstruction. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 295–302.

IGEHY, H. 1999. Tracing ray differentials. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, 179–186.

KOBBELT, L. 1998. Tight bounding volumes for subdivision surfaces. In *PG '98: Proceedings of the 6th Pacific Conference on Computer Graphics and Applications*, IEEE Computer Society, Washington, DC, USA, 17.

LEXT, J., AND AKENINE-MOLLER, T. 2001. Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001*.

LOOP, C. T., 1987. Smooth subdivision surfaces based on triangles.

LUEBKE, D., REDDY, M., COHEN, J., VARSHNEY, A., WATSON, B., AND HUEBNER, R. 2003. *Level of Detail for 3D Graphics*. Morgan Kaufmann.

LUQUE, R. G., COMBA, J. L. D., AND FREITAS, C. M. D. S. 2005. Broad-phase collision detection using semi-adjusting BSP-trees. In *Proc. of 2005 Conf. on Interactive 3D graphics*.

MARK, W. R., AND FUSSELL, D. 2005. Real-time rendering systems in 2010. *UT-Austin Computer Sciences Technical Report TR-05-18* (May).

OPENGL ARCHITECTURAL REVIEW BOARD. 2003. OpenGL 1.5 specification.

OWENS, J. D., KHAILANY, B., TOWLES, B., AND DALLY, W. J. 2002. Comparing Reyes and OpenGL on a stream architecture. In *2002 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 47–56.

PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *Symposium on interactive 3D graphics*.

PHARR, M., AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. In *1996 Eurographics workshop on rendering*.

PHARR, M., AND HUMPREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann.

PIXAR. 2000. *The RenderMan interface version 3.2*, July.

PULLI, K., AND SEGAL, M. 1996. Fast rendering of subdivision surfaces. In *Proc. of Eurographics Rendering Workshop*.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the 11th Eurographics Workshop on Rendering*, Eurographics Association, 299–306.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *SIGGRAPH '05: Proceedings of the 32nd annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA.

RUBIN, S. M., AND WHITTED, T. 1980. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 110–116.

SCHLICK, C. An inexpensive BRDF model for physically-based rendering. *Computer graphics forum 13*, 3, 233–246.

SMITS, B. 1998. Efficiency issues for ray tracing. *J. Graph. Tools 3*, 2, 1–14.

SUYKENS, F., AND WILLEMS, Y. 2001. Path differentials and applications. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 257–268.

TABELLION, E., AND LAMORLETTE, A. 2004. An approximate global illumination system for computer generated films. *ACM Transactions on Graphics 23*, 3, 469–476.

TORRES, E. 1990. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In *Proc. of Eurographics 1990*.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. In *Proc. of Eurographics 2001*.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed interactive ray tracing of dynamic scenes. In *Proc. IEEE symp. on parallel and large-data visualization and graphics*.

WANN JENSEN, H. 2001. *Realistic image synthesis using photon mapping*. AK Peters.

WARD, G. J., AND HECKBERT, P. 1992. irradiance gradients. In *Proc. 3rd Eurographics Workshop on Rendering*, 85–98.

WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. 1988. A ray tracing solution for diffuse interreflection. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 85–92.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June), 343–349.

WILEY, C., A. T. CAMPBELL, I., SZYGENDA, S., FUSSELL, D., AND HUDSON, F. 1997. Multiresolution bsp trees applied to terrain, transparency, and general objects. In *Proceedings of the conference on Graphics interface '97*, Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 88–96.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: a programmable ray processing engine. In *SIGGRAPH '05: Proceedings of the 32nd annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA.

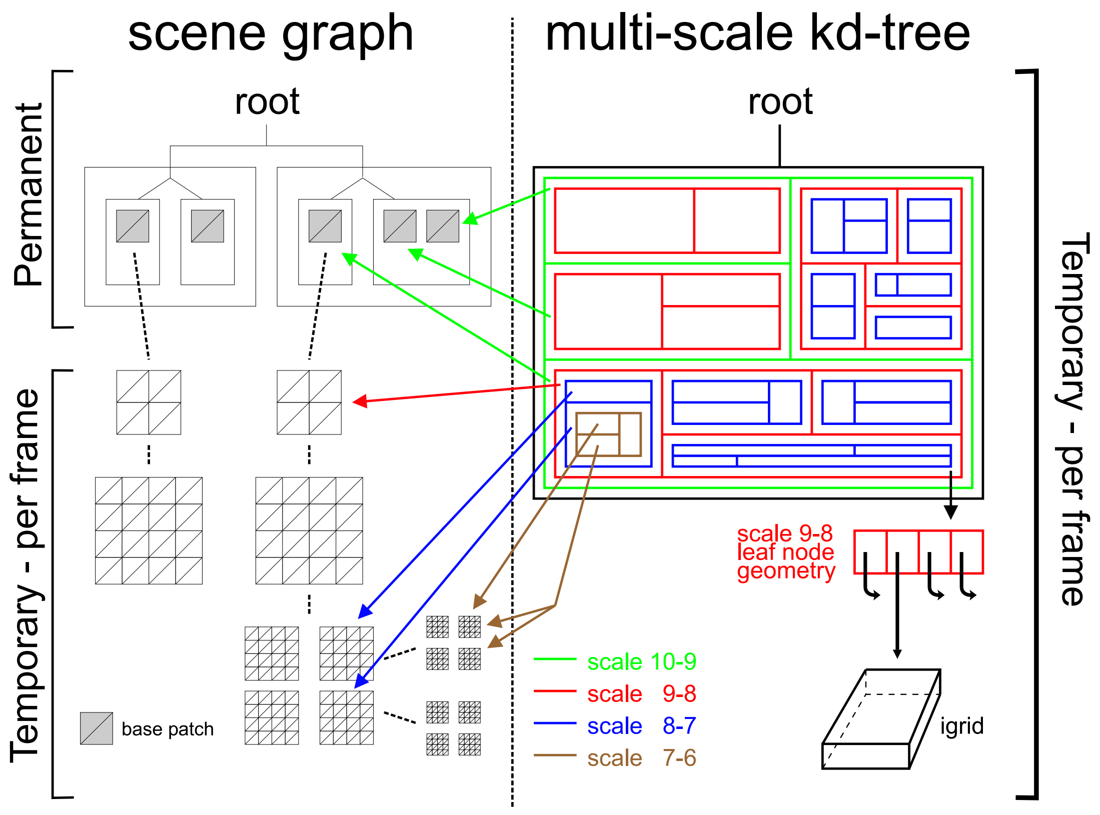


Figure 8: The key data structures in our system. The multi-scale kD-tree is closely coupled to the scene graph by “lazy” pointers. Regular (non-lazy) leaf nodes of the kD-tree point to a grid of geometry called an igrid.

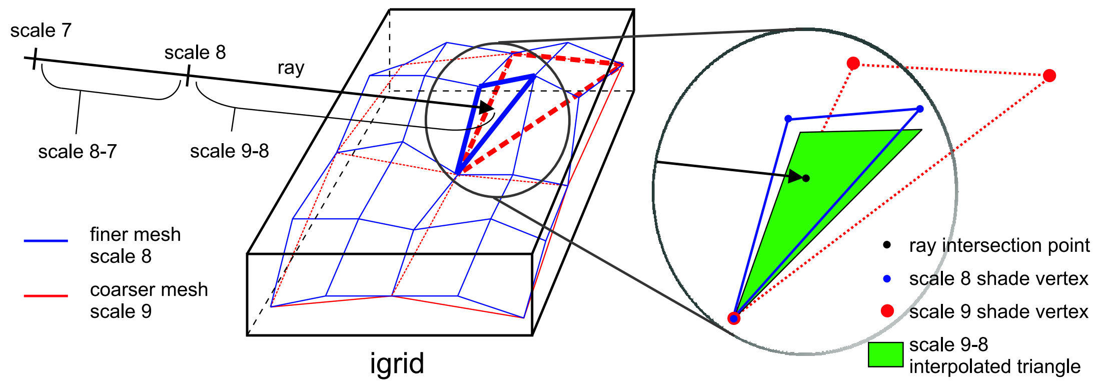


Figure 9: An igrid holds vertices for a pair of discrete scales. One set of vertices comes from a finer scale of geometry and the other set of vertices comes from a coarser scale of geometry. The igrid contains information associating each fine-scale vertex with a point on a coarse-scale triangle. The information in the igrid is used to generate interpolated triangles (shown in green) that are customized for particular rays. The igrid also contains (not pictured) a simple bounding volume acceleration structure based on the structure of the tessellation.

Cache-Efficient Layouts of Bounding Volume Hierarchies

Sung-Eui Yoon[1] Dinesh Manocha[2]
[1]Lawrence Livermore National Laboratory [2]University of North Carolina at Chapel Hill

LLNL Tech. Report: UCRL-ABS-219070-DRAFT
Feb - 16, 2006
{sungeui,dm}@cs.unc.edu

Abstract

We present a novel algorithm to compute cache-efficient layouts of bounding volume hierarchies (BVHs) of large polygonal models. Our approach works well with various types of BVHs and does not make any assumptions about the cache parameters or block sizes of the memory hierarchy. We introduce a new probabilistic model to predict the runtime access patterns of a BVH. Our layout computation algorithm utilizes parent-child and spatial localities between the accessed nodes to reduce both the number of cache misses and the size of working set. We use our algorithm to compute layouts of BVHs of large models composed of millions of triangles. We also compare our cache-efficient layouts with other layouts in the context of collision detection and ray tracing. In practice, our layouts can improve the performance of these applications by 15%–400% without any modification of the underlying algorithms or runtime applications.

1 Introduction

Bounding volume hierarchies (BVHs) are widely used to accelerate the performance of geometric processing and interactive graphics applications. The applications include ray tracing, visibility culling, collision detection, and geometric computations on large datasets. Most of these algorithms precompute a BVH and traverse the hierarchy at runtime to perform intersection tests or culling.

The leaf nodes of a BVH correspond to the triangles of the original model. The intermediate nodes are the bounding volumes (BVs) such as spheres, axis-aligned bounding boxes (AABBs), oriented bounding boxes (OBBs), and convex polytopes. The memory requirements of BVHs can be high for large datasets. For example, the storage cost of a hierarchy of OBBs, i.e., OBB-tree, is approximately 64 bytes per node. As a result, BVHs of large datasets composed of tens of millions of triangles can require gigabytes of space.

Our goal is to compute cache-efficient layouts of BVHs to reduce the number of cache misses and improve the performance of BVH-based algorithms. As the gap between the processor speed and main memory speed widens, system designers increasingly use caches and memory hierarchies to reduce memory latency. The access times of different levels of a memory hierarchy vary by orders of magnitude. As a result, the running time of an algorithm varies as a function of its cache access pattern. We would like to use data layout optimization techniques to place the nodes of a BVH in the memory and reduce the number of cache misses at runtime.

Many mesh representations and algorithms have been proposed to improve the cache access patterns of geometric models for specific applications. These representations and algorithms include rendering sequences (e.g., triangle strips), processing sequences (e.g., streaming meshes), layouts computed using space filling curves, and minimum linear arrangement (MLA). However, these representations and algorithms are not general enough to handle all the kinds of BVHs that are used in various geometric applications.

Main Results: We present a novel algorithm to compute cache-efficient layouts of BVHs of large models. Our approach is cache-oblivious as it does not require any knowledge of cache parameters or block sizes of the memory hierarchy and is applicable to all kinds of BVHs that can be represented as a tree. We represent a BVH as two separate linear sequences of BVs and triangles. Then, our problem reduces to computing cache-efficient layouts of the BVs and the triangles. We introduce a new probabilistic model to predict the runtime access patterns of BVHs based on localities. Specifically, we utilize two types of localities during traversal of a BVH: parent-child and spatial localities between the accessed nodes. Our algorithm also uses a tree decomposition algorithm [GI99] and cache-oblivious mesh layout [YLPM05] to compute a layout that reduces the number of cache misses and the size of the working set.

We use our algorithm to compute layouts of OBB trees and kd-trees of large models composed of millions of triangles. Based on these layouts, we accelerate the performance of collision detection and ray tracing without any modifications to the underlying algorithms or runtime application. We also compare the performance of our layouts with other layouts including depth-first layout, breadth first layout, van Emde Boas layout, cache-oblivious mesh layout, and cache-aware layouts. We have observed up to a 4 times improvement in performance based on our cache-efficient layouts. Moreover, performance of our cache-oblivious layouts is comparable to that of cache-aware layouts. Overall, our approach offers the following benefits:

1. **Generality:** Our algorithm is general and applicable to a wide range of BVHs. It does not require any knowledge of cache parameters or block sizes of a memory hierarchy.

2. **Applicability:** Our algorithm does not require any modification of BVH-based algorithms or the runtime application. We simply compute efficient layouts of BVHs and the same application can run on our layouts.

3. **Improved performance:** Our layouts reduce the number of cache misses during traversals of BVHs. We are able to improve the performance of standard collision detection and ray tracing algorithms.

Organization: The rest of the paper is organized in the following manner. We give a brief survey of related work in Section 2 and give an overview of memory hierarchies and BVHs in Section 3. Section 4 describes the localities that are used by our algorithm. We present a novel probabilistic model to predict the runtime access patterns of BVHs in Section 5 and describe our layout algorithm in Section 6. We highlights its performance in Section 7 and compare its performance with prior approaches in Section 8.

2 Related Work

We give a brief overview of the related work on cache-efficient algorithms and layouts of bounding volume hierarchies and geometric models.

Figure 1: **Ray Tracing the Lucy model:** *We apply a standard kd-tree based ray tracing algorithm to the Lucy model consisting of* 28 *million triangles. A reflective plane is placed behind the Lucy model and the scene also has shadows. We compute a cache-efficient layout of the kd-tree of the Lucy model using our algorithm. Our layout improves the performance of ray tracing by up to two times over tested layouts, without any change to the underlying algorithm.*

2.1 Cache-Efficient Algorithms

Cache-efficient algorithms have received considerable attention over last two decades in the theoretical computer science and compiler literature. These algorithms include theoretical models of cache behavior [Vit01, SCD02] , and compiler optimizations based on tiling, strip-mining, and loop interchanging to minimize cache misses [CM95].

At a high level, cache-efficient algorithms can be classified as either cache-aware or cache-oblivious. Cache-aware algorithms utilize knowledge of cache parameters, such as cache block size [Vit01]. On the other hand, cache-oblivious algorithms do not assume any knowledge of cache parameters [FLPR99]. There is considerable literature on developing cache-efficient algorithms for specific problems and applications [ABF04, Vit01].

2.2 Layouts of BVHs

The impact of different layouts of tree structures has been widely studied. There is considerable work on cache-coherent layouts of tree-based representations including work on accelerating search queries. Given the cache parameters, Gil and Itai [GI99] cast cache-coherent layout computation as an optimization problem. They propose a dynamic programming algorithm to minimize the number of cache misses during traversals of search queries. However, the computed layout may not be storage efficient. Alstrup *et al.* [ABFC 03] propose a method to compute cache-oblivious layouts of search trees by recursively partitioning the trees.

There is relatively less work on cache-coherent layouts of BVHs. Opcode[1] uses a blocking method that merges several bounding volumes nodes together to reduce the number of cache misses. The blocking is based on *van Emde Boas* layout of complete trees [vEB77]. However, it is not clear whether van Emde Boas layouts can minimize the number of cache misses during traversal of general BVHs. Havran analyzes various layouts of BVHs in the context of ray tracing and improves the performance by using a compact layout representation of BVHs [Hav97]. Yoon et al. [YLPM05] propose a cache-oblivious mesh layout algorithm to compute layouts of geometric meshes and bounding volume hierarchies. We compare our approach with these algorithms in Section 8.2.

[1]http://www.codercorner.com/Opcode.htm

Layouts of geometric meshes: Many algorithms and representations have been proposed to compute coherent layouts for specialized applications. Rendering sequences (e.g., triangle strips) [Dee95, Hop99] are used to improve rendering throughput by optimizing the vertex cache hits in the GPU. Isenburg and Gumhold [IG03] propose processing sequences, including streaming meshes [IL04], as an extension of rendering sequences for large-data processing. In these cases, global mesh access is restricted to a fixed traversal order. Many algorithms use space filling curves [Sag94] to compute cache-friendly layouts of volumetric grids or height fields. These layouts are widely used to improve performance of image processing [VG91] and terrain or volume visualization [PF01, LP01]. However, it is unclear whether space filling curves would extend to compute layouts of unstructured models and their hierarchies. In graph theory, minimum linear arrangement (MLA) [DPS02] has been widely researched to minimize the sum of edge lengths of all the edges in a graph layout. However, there may be no direct relationship between reducing the sum of edge lengths and minimizing the number of cache misses.

3 Memory Hierarchies and BVH Layouts

In this section, we give an overview of memory hierarchies, BVHs, and their layouts. We also introduce some of the terminology used in the rest of the paper.

3.1 Memory Hierarchy and Caches

Most modern computer architectures use hierarchies of memory levels, where each level of memory serves as a *cache* for the next level. The memory hierarchies have two main characteristics [AV88]. First, higher levels in the hierarchy are larger in size, farther away from the processor, and have slower access times. Second, data is moved in large blocks between different memory levels. The BVH layout is initially stored in the highest memory level, typically the disk. The portion of the layout accessed by the application is transferred in large blocks into the next lower level, such as main memory. A transfer is performed whenever there is a cache miss between two adjacent levels of the memory hierarchy. The number of cache misses depends on the layout of the BVH in memory and the access patterns of the application. If nodes of BVHs are packed in blocks in a cache-coherent manner, the number of cache misses can be reduced.

3.2 Bounding Volume Hierarchies

BVHs are widely used in various applications to accelerate the performance of intersection or culling tests. The leaf node of a BVH corresponds to the triangulated primitives and the intermediate nodes are the bounding volumes (BVs). Each BV encloses its children nodes and their descendants. In the rest of this paper, we use collision or intersection queries as the driving application to explain the concepts behind computing cache-efficient layouts of BVHs. These algorithms typically take two inputs: two 3D objects or one 3D object and a ray. The runtime algorithm traverses the BVHs of each object using a depth-first or a breadth-first order. The depth-first order is typically used when we need to check for ray-object intersection or to check whether two objects overlap. The breadth-first traversal order is preferred when the runtime algorithm can be interrupted and may return approximate results, e.g. time-critical computations or constant frame-rate rendering of large models.

Extensive work has been done on evaluating the performance of different BVHs for ray-tracing and proximity queries. These include the cost equations for ray-tracing [WHG84] and collision detection [GLM96, KHM 98]. These cost equations take into account the tightness of fit for a BV and the relative cost of computing intersections or overlaps with those BVs based on the traversal pattern. However, these formulations do not take into account the cost of memory accesses or cache misses while traversing the BVHs. If the underlying model and its BVH cannot fit into the main memory, the cost of memory accesses and cache misses can become a significant factor.

3.3 Layout of BVH

We use the following notation to represent the BVs of a BVH. We define n_i^1 as the ith BV node at the leaf level of the hierarchy and n_i^k as a BV node at the kth level of the hierarchy. We also define $Left(n_i^k)$ and $Right(n_i^k)$ to be the left and right child nodes of the n_i^k. A parent node and a grandparent node of the n_i^k are denoted by using $Parent(n_i^k)$ and $Grand(n_i^k)$.

Formally speaking, a BVH is a directed acyclic graph, $G(N, A)$, where N is a set of BV nodes, n_i^k, and A is a set of directed edges from a node, n_i^k, to each child node, $Left(n_i^k)$ and $Right(n_i^k)$, in the BVH. A layout of a BVH is composed of two parts: a BV layout and a triangle layout. A BV layout of a BVH, $G(N, A)$, is a one-to-one mapping of BVs to positions in the layout, $\varphi : N \rightarrow \{1, \ldots, |N|\}$. Our goal is to compute a mapping, φ, that minimizes the number of cache misses and the size of working set during the traversal of the BVH at runtime. Similarly, we also compute a triangle layout to minimize both cache misses and the working set size during BVH traversals.

4 Localities in BVH Traversal

In this section, we define two localities that are used to compute a cache-efficient layout of a BVH. We also give a brief overview of prior work on packing trees and cache-oblivious mesh layout algorithms, which are used by our layout algorithm.

4.1 Access Patterns during BVH Traversal

Collision queries traverse BVHs as long as each query between two BVs reports a collision between them. Our goal is to minimize the number of cache misses and the size of working set during the traversal.

We decompose the access pattern during a traversal into a set of search queries. We define a *search query*, $S(n_i^k)$, to be the traversal from the root node of the BVH to the node, n_i^k, which can be either a leaf or an intermediate node. Let us assume that the traversal of a collision query starts from the root node and ends at nodes, $n_{i(1)}^{k(1)}, \ldots, n_{i(m)}^{k(m)} (= BV_1, \ldots, BV_m)$. In this case, the nodes, $(BV_1, \ldots, BV_m)$, define a front of the BVH for this traversal. We represent this traversal as the union of traversals of m different search queries, $S(BV_j)$. An example of an access pattern between two colliding objects is shown in Fig. 2. In frame i, the collision query ends at n_1^3 and n_5^3 starting from the root node, n_1^4, of the BVH of object 1. We can represent the access patterns of this collision query with two search queries ending at n_1^3 and n_5^3.

There are two different localities, parent-child locality and spatial locality, which arise during the traversal.

1. **Parent-child locality:** Once a node of a hierarchy is accessed by a search query, it is likely that its child nodes will be accessed soon. For example, in frame i of Fig. 2, if the root node of the BVH is accessed, its two child nodes, n_1^3 and n_5^3, are likely to be accessed soon. Moreover, after n_1^3 is accessed during frame i, its child nodes are likely to be accessed in the next frame.

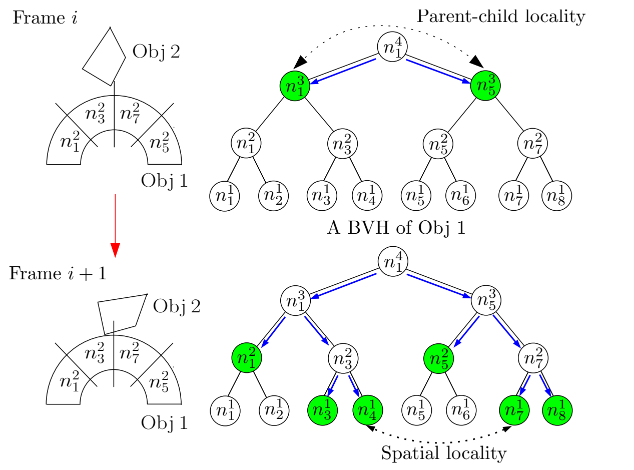


Figure 2: **Two localities within BVHs:** *We show two successive frames from a dynamic simulation and the change in access patterns (shown with blue arrows) of a BVH. In this simulation, object* 2 *drops on object* 1*, as shown on the left. The access pattern of the BVH of object* 1 *during each frame is shown on the right. The BVs from the* 2*nd level in the BVH are shown within object* 1 *on the left. We also illustrate the front traversed within each BVH during each frame in green. The top BVH shows the parent-child locality, when the root node,* n_1^4*, of the BVH of object* 1 *collides with the BVs of objects* 2*. During frame* $i+1$*, object* 2 *is colliding with object* 1*. In this configuration, the BVs* n_3^2 *and* n_7^2 *(and their sub-nodes) are accessed due to their close spatial locality.*

2. **Spatial locality:** Whenever a node is accessed by a search query, other nodes in close proximity are also highly likely to be accessed by other search queries. For example, collisions or contacts between two objects occur in small localized regions of a mesh. Therefore, if a node of a BVH is accessed, other nearby nodes are either colliding or are in close proximity and may be accessed soon. In frame $i+1$ of Fig. 2, if one of two nodes, n_4^1 and n_7^1, is accessed, the other node is also likely to be accessed during that frame or subsequent frames.

We separately consider each of these two localities and use them to compute the layout of a BVH. In the remainder of this section, we briefly summarize several known results related to these localities.

4.2 Parent-Child Locality

We use several results presented by Gil and Itai [GI99] to compute a cache-coherent layout of a BVH. Gil and Itai address the problem of computing a good layout for search queries on a tree. They define two different measures for the cache-coherence of a layout of a tree. The two measure are:

1. **The number of cache misses (or page faults):** $PF^1(BV_i)$ is defined as the number of cache misses, given a cache that can hold only single cache block during the traversal of a search query ending at BV_i.

2. **The size of working set:** The working set during the traversal of the search query is a set of different cache blocks that are accessed. $WS(BV_i)$ is defined as the size of the working set.

Intuitively speaking, $PF^1(BV_i)$ measures the number of times that accessing BVs crosses boundaries of cache blocks of the layout during the traversal. Moreover, [GI99] introduces a virtual probability function, $Pr(BV_i)$, that can measure how many times BV_i is accessed during any search query on the tree. The expected size of working set, WS, of the layout can be formulated as:

$$WS = \sum_{BV_i \in \text{BVH}} Pr(BV_i) WS(BV_i),$$

for all nodes BV_i in the hierarchy. Similarly, we can define the expected number of cache misses, PF^1, of a layout by multiplying $Pr(BV_i)$ with $PF^1(BV_i)$ for all nodes BV_i in the tree. If a tree layout is optimal given the PF^1 or WS measure, the tree layout is defined as PF^1-optimal or WS-optimal, respectively.

Lemma 1 **(Convexity):** *If a layout of a tree is PF^1-optimal or WS-optimal, the layout is convex [GI99].*

The layout of a tree is convex if all the intermediate BVs between BV_0 and BV_k are stored in the same block when a node BV_0 and its descendant BV_k are stored in the same cache block.

Lemma 2 **(Equivalence):** *A layout of a tree is PF^1-optimal if and only if it is WS-optimal [GI99].*

Lemma 3 **(NP-Completeness):** *Computing a layout of a tree that is a WS-optimal with a minimum storage is NP-Complete [GI99].*

We use these properties and lemmas to design our layout algorithm that considers parent-child locality during the traversal of search queries.

4.3 Spatial Locality

We use the layout technique proposed by Yoon et al. [YLPM05] to compute cache-oblivious layouts of geometric meshes. They construct an undirected graph to represent cache-coherent access patterns. Each vertex of the graph represents a data element (e.g. a vertex of the mesh) that an application accesses. An edge exists between two vertices of the graph if their representative data elements are likely to be accessed sequentially by the application at runtime. Depending on the spatial locality between two elements, an edge connecting two vertices is created with a weight that is inversely proportional to the spatial locality. This method will be used as a part of our layout algorithm with our probabilistic model to compute weights.

5 Probability Model

In this section we present our probability model that is used to predict the runtime access patterns on BVHs. We derive our formulation based on the geometric relationship between the nodes of BVHs.

We assign a probability, $Pr(n_i^k)$, to a BV, n_i^k, that the node would be accessed during the traversal as part of a search query. Suppose the parent node, $Parent(n_i^k)$, of a node, n_i^k, of an object collides with a BV node, BV_{Obj2}, of another object. In this case, the two children of $Parent(n_i^k)$ are fetched and tested to further localize the colliding region. Therefore, $Pr(n_i^k)$ can be computed by multiplying two factors: 1) the probability that $Parent(n_i^k)$ is accessed, and 2) the probability that $Parent(n_i^k)$ collides with BV_{Obj2}. If there is a collision between the two nodes, each node is further refined with its two child nodes. Thus, the second probability can be computed by assuming that there was also a collision between $Grand(n_i^k)$ and BV_{Obj2}. The probability that n_i^k is accessed during the traversal can be recursively formulated as following:

$$Pr(n_i^k) = Pr(Parent(n_i^k)) Pr(n_i^k, X_p = 1 | X_g = 1), \quad (1)$$

where X_p and X_g are two binary random variables indicating whether there are collisions between a $Parent(n_i^k)$ and BV_{Obj2} and between $Grand(n_i^k)$ and BV_{Obj2}, respectively.

5.1 Probability Computation

Our goal is to efficiently compute $Pr(n_i^k, X_p = 1 | X_g = 1)$ given the recursive probability formulation presented in Eq. (1). Since we compute probabilities for nodes of the BVH as a preprocess, we do not know anything about the size or BV type of BV_{Obj2}, a BV node of another object. Instead of assuming any particular BV for BV_{Obj2}, we enumerate all possible configurations of BVs for BV_{Obj2} and compute the probability. Let $S_g(n_i^k)$ be the set that represent all possible configurations of BV_{Obj2} that collide with $Grand(n_i^k)$. We can similarly define $S_p(n_i^k)$. For example, if BV_{Obj2} is a sphere, $S_p(n_i^k)$ can be constructed by *Minkowski sum*: $S_p(n_i^k, r) = Parent(n_i^k) \bigoplus Sphere(r)$ where $Sphere(r)$ is a sphere with a radius, $r \in [0, \infty)$ and $\bigoplus$ is the Minkowski sum operator. In the more general case, BV_{Obj2} would correspond to a box or a convex shape and can have arbitrary orientation. As a result, both $S_g(n_i^k)$ and $S_p(n_i^k)$ can be represented as high dimensional *configuration-space*. Given the formulation of $S_p(n_i^k)$ and $S_g(n_i^k)$, $Pr(n_i^k, X_p = 1 | X_g = 1)$ can be defined as:

$$\begin{aligned} Pr(n_i^k, X_p = 1 | X_g = 1) &= \frac{Pr(X_p = 1 \cap X_g = 1)}{Pr(X_g)} \\ &= \frac{Vol(S_p(n_i^k) \cap S_g(n_i^k))}{Vol(S_g(n_i^k))} \end{aligned} \quad (2)$$

where $Vol(A)$ represents the volume of A. Intuitively speaking, the probability is the ratio of the volume of the intersected space between $S_p(n_i^k)$ and $S_g(n_i^k)$ to the volume of $S_g(n_i^k)$. We refer to the intersected volume ratio between $S_p(n_i^k)$ and $S_g(n_i^k)$, $V_{intersected}(n_i^k)$.

It is, however, complex and expensive to construct the Minkowski sum or the configuration space in general [DHKS93]. The combinatorial complexity is high and the resulting algorithms are susceptible to degeneracies and robustness problems. As a result, exact computation of $Pr(n_i^k)$ is non-trivial.

5.2 Approximate Probability Computation

We propose a simple method to approximate the probability function described in Eq. (2). We observe that the intersected volume ratio computed when BV_{Obj2} is considered to a point–therefore, BV_{Obj2} has zero extent–is a good approximation of the probability, which is the intersected volume ratio, $V_{intersected}(n_i^k)$, between $S_p(n_i^k)$ and $S_g(n_i^k)$. In other words, we use an intersected volume ratio, $V_{intersected}(n_i^k, 0)$, between $Parent(n_i^k)(= S_p(n_i^k, 0))$ and $Grand(n_i^k)(= S_g(n_i^k, 0))$ as the probability defined in Eq. 2. This approximation is based on the following observations:

- **Relative importance of probabilities during layout computation:** Suppose that our layout algorithm considers two nodes, n_1 and n_2, to decide which node should be ordered first. Our layout algorithm will choose a node that has a higher probability.

- **Importance of $S_p(n_i^k, 0)$ and $S_g(n_i^k, 0)$ for probability computation:** Suppose that an intersected volume ratio, $V_{intersected}(n_1, 0)$, between $Parent(n_1)$ and $Grand(n_1)$ is bigger than its counterpart, $V_{intersected}(n_2, 0)$, of $Parent(n_2)$ and $Grand(n_2)$ when r is zero. Then, it is likely that the intersected volume ratio, $V_{intersected}(n_1, r)$, between $S_p(n_1, r)$ and $S_g(n_1, r)$ is also bigger than its counterpart, $V_{intersected}(n_2, r)$, of n_2 when r is non-zero. Therefore, we can approximate the relative importance between the intersected volume ratio, $V_{intersected}(n_i^k)$, between $S_p(n_i^k)$ and $S_g(n_i^k)$ as the relative importance between the intersected volume ratio, $V_{intersected}(n_i^k, 0)$, between BVs of $Parent(n_i^k)$ and $Grand(n_i^k)$.

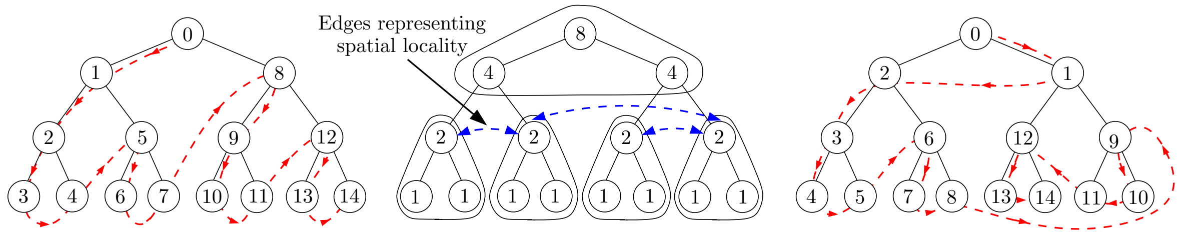

Edges representing spatial locality
Depth-first layout
Clustering and undirected graph
Cache-oblivious layout

Figure 3: **Layout computation of a BVH:** *A depth-first layout of a BVH is shown in the leftmost figure and a cache-oblivious layout of the same tree is shown in the rightmost figure. The number within each BV node in the leftmost and the rightmost figures is an index of the ordering of BVs in the layout. The middle figure shows the output of the clustering step. The topmost cluster is the root cluster and the rest are child clusters. Edges (shown in blue) are drawn between the root BVs of the child clusters that are nearby according to spatial relationships shown in Fig. 2.*

In order to quantitatively verify our approximation, we selected two nodes, n_1 and n_2, during layout computation of the dragon model and measured $V_{intersected}(n_1, r)$ and $V_{intersected}(n_2, r)$ as r geometrically increases from zero. We observed that less than 5% of relative importance between $V_{intersected}(n_1, 0)$ and $V_{intersected}(n_2 2, 0)$ is reversed as compared to $V_{intersected}(n_1, r)$ and $V_{intersected}(n_2, r)$, when we used a higher radius on BV_{Obj2} as a sphere.

Discretization: In order to approximate the volume ratio of the intersected area between $Parent(n_i^k)$ and $Grand(n_i^k)$ to $Grand(n_i^k)$, we overlay a uniform grid on the BV of $Grand(n_i^k)$ and measure the number of cells of the grid that are contained in the BV of $Parent(n_i^k)$. We generate a sample point in each cell to perform this containment test. In practice, we found that using 64 samples to compute the probability is sufficient.

6 Layout Computation

In this section, we present a simple greedy algorithm to compute a cache-efficient layout of a BVH. Our layout is designed to reduce the number of cache misses at runtime. We use the properties, lemmas, and the probability model described in the previous sections to compute cache-efficient layouts of BVHs.

6.1 Overall Algorithm

At the top level, our algorithm decomposes a BVH into clusters. The goal is to compute clusters whose size is the same as the cache block. If we knew the cache parameters and the block size, we can compute how many BV nodes fit into the cache block. Given this information, we could decompose the BVH into a set of clusters, such that the size of each cluster is equal to the size of the cache block. However, our algorithm does not assume any particular cache size and constructs a layout that works well with any cache parameter. In order to achieve this goal, we recursively compute the clusters. We first decompose the BVH into a set of clusters and recursively decompose each cluster. In this case, the cache block boundaries can lie anywhere within a layout that corresponds the nodes of these clusters. Therefore, we need to compute a cache-efficient ordering of the clusters computed at each level of recursion.

Our algorithm has two different components that separately handle parent-child and spatial localities. In particular, the first part of the algorithm decomposes a BVH into a set of clusters that minimizes the cache misses for parent-child locality. The clusters are classified as a root cluster and child clusters. The root cluster contains the root node of the BVH and child clusters are created for each node outside the root cluster whose parent node is in the root cluster (See the middle image in Fig. 3). The second part of the algorithm computes an ordering of the clusters and stores the root cluster at the beginning of the ordering. The ordering of child clusters is computed by considering their spatial locality and relying on the cache-oblivious mesh layout algorithm described in [YLPM05]. We recursively apply this two-fold procedure to compute an ordering of all the BVs in the BVH.

Cluster size: For each level of recursion, we decompose the BVH into a set of clusters that have approximately the same number of BV nodes. Suppose that a root cluster has B BV nodes. Then, the root cluster has $B + 1$ child clusters and we decompose the BVH into $B + 2$ clusters. Assuming that each cluster is reasonably balanced in terms of the number of BV nodes belonging to each cluster, $B \times (B + 2)$ should be bigger than n, the number of nodes in the BVH, to contain all the nodes in the BVH. Therefore, B is set to be $\lceil \sqrt{n + 1} - 1 \rceil$.

6.2 Cluster Decomposition

Before computing clusters from the BVH, we first compute and assign a probability, $Pr(n_i^k)$, to a BV, n_i^k, as described in the previous section. Then, we partition the BVH into $B + 2$ clusters, where B is the number of nodes in the root cluster.

Our goal in this step is to store the BV nodes, which are accessed together due to the parent-child locality, into the same cluster in order to minimize the number of cache misses. According to our probability model shown in Eq. (1), the probability assigned to each node can also be considered the probability that the node is accessed, given that a root node of a cluster is accessed. Therefore, we can achieve our goal by maximizing the sum of probabilities of BVs belonging to the root cluster. Moreover, maximizing this sum to the root cluster also minimizes the probability to access the nodes belonging to the child clusters. This formulation also minimizes the number of times that a search query accesses the data across the boundaries of cache blocks of the layout, which is quantified by PF^1 measure. According to Lemma 2, computing an optimal layout for the PF^1 metric is equivalent to computing an optimal layout that minimizes the expected size of working set, WS. Therefore, maximizing the sum of probabilities of BVs belonging to the root cluster minimizes the expected size of the working set during collision queries in the end.

However, computing a layout minimizing the working set and the number of cache misses for all possible search queries with minimum space of a layout is NP-complete (as per Lemma 3). As a result, we employ a greedy algorithm to efficiently compute a cache-oblivious layout of the BVH. Our algorithm greedily traverses the BVH and merges nodes from the root node of the BVH into the root cluster by locally choosing a node that has the highest probability. Once the root cluster has B nodes, we stop merging the nodes into the root cluster. Then, each child node of the nodes inside the root cluster whose child nodes are outside the root cluster consists of a child cluster containing all the nodes of its sub-tree. The layout computed by this greedy approach also maintains the convexity of the layout as

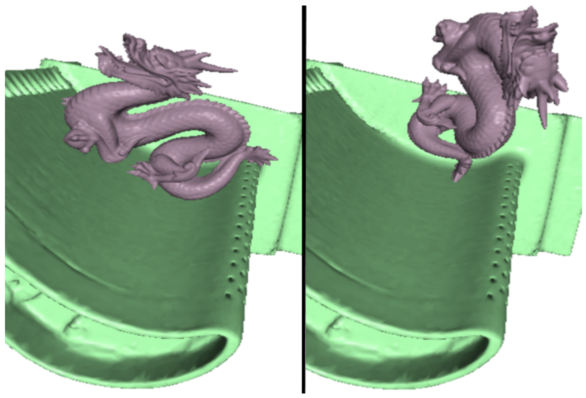


Figure 4: **Dynamic Simulation between Dragon and Turbine Models:** *This image sequence shows discrete positions from our dynamic simulation between dragon and CAD turbine models. We are able to achieve* 17%–319% *performance improvement in collision detection by using our cache-efficient layouts of the OBBTree over other tested layouts.*

defined by Lemma 1.

6.3 Layouts of Clusters

Given the computed clusters at each level of recursion, we compute a cache-oblivious ordering of the clusters by considering their spatial locality. During each recursive step of the algorithm, the number of BV nodes belonging to each cluster roughly reduces by a factor of $B+2$, based on our cluster computation algorithm. This causes considerable differences between the sizes of clusters created during the previous level of the recursion and the current level of the recursion. Therefore, it is important to compute a cache-coherent ordering of the clusters in order to further reduce the cache misses. This is because there is high likelihood that the size of a cache block may lie between the cluster size of the previous level and the current level of recursion.

We place the root cluster at the beginning of the ordering of clusters since the traversal typically starts at the root node of the BVH. In order to compute an ordering of child clusters, we construct an undirected graph with the child clusters as the nodes of the graph. We construct an edge between two clusters if they are in close proximity, that is, if their BVs overlap. Then, we compute a probability that a BV of a cluster has collided given that a BV of another cluster has collided based on the probability formulation described in Eq. 2. An example of an undirected graph between two child clusters is shown in the middle BVH of Fig. 3.

Once the graph is computed, we compute a cache-oblivious layout from the graph that represents the access patterns between the child clusters. This is performed using the cache-oblivious mesh layout algorithm [YLPM05]. An example of a cache-oblivious layout of a complete tree is shown in the rightmost figure of Fig. 3.

6.4 Triangle Layout

Once a set of BV pairs is computed during the runtime traversal of the BVHs of two objects, exact query computation based on the triangles of leaf nodes is performed. We extract a triangle layout from the BV layout of the BVH for efficient layout computation. If we encounter leaf nodes of the BVH during layout computation, we sequentially order the triangles stored in the BVs into the triangle layout since we perform the overlap tests at runtime in a sequential manner based on the stored order of the triangles within a leaf node.

Model	Triangles (M)	Size of BVH (MB)	Avg. and std of depth of leaves	Comp. time (min)
Bunny	0.06	4	$17, 0.8$	0.06
Dragon	0.8	54	$21, 1.6$	0.88
Turbine	1.7	110	$22, 0.7$	2
Lucy	28	$4,811$	$37, 3.4$	34

Table 1: **Benchmark Models:** Model complexity, sizes of BVHs, avg. and standard deviation(std) of depth of leaf nodes, and computation time to compute cache-oblivious layouts are shown.

7 Implementation and Performance

In this section we describe our implementation and highlight the performance of cache-oblivious layouts on different BVHs. These include the kd-tree used by a ray tracing algorithm and OBBTree used to perform collision queries in a dynamic simulation.

7.1 Implementation

We have implemented our cache-oblivious layout computation algorithm as well as the two applications on a 2.4GHz Pentium-IV PC with 1GB of RAM. All our implementations can handle very large datasets in an out-of-core manner. Our system runs on Windows XP and uses the operating system's virtual memory through memory mapped files. Windows XP imposes a 2GB limitation for mapping a file to user-addressable address space. We overcome this limitation by mapping a 1MB portion of the file at a time and remapping when data are required from outside this range. We also use *OpenCCL* library [2] to compute a cache-oblivious layout between child clusters during our layout algorithm.

7.2 Benchmark Models

Our algorithm has been applied to different polygonal models. These include the Lucy model composed of more than 28 million polygons (Fig. 1), the CAD turbine model consisting of a single object with 1.7 million triangles (Fig. 4), the dragon model consisting of 800K polygons, and the Stanford bunny model consisting of 67K polygons (Fig. 6). The details of these models are shown in Table 1.

7.3 Performances

We applied our out-of-core algorithm to compute cache-oblivious layouts of BVHs of the models. Table. 1 presents the layout time for each model. An unoptimized implementation of our out-of-core algorithm can process 14K triangles per second.

7.3.1 Collision Detection

We have implemented an impulse based rigid body simulation [MC95] for dynamic simulation. We use OBBTrees [GLM96] to perform collision queries. We compute cache-efficient layouts of the OBBTrees of different models and use these layouts with the same underlying algorithm, i.e. RAPID [GLM96], to perform collision queries.

We compared the performance of our cache-oblivious layout of BVHs (COLBVH) with different layouts including depth-first layout(DFL) of the BVH, breadth-first layout(BFL), van Emde Boas layout (VEB) [vEB77], and cache-oblivious mesh layout (COML) [YLPM05], and cache-aware layout obtained by explicitly setting cache size into our cache-oblivious layout algorithm (CALBVH). The OBBs are precomputed and only the ordering of the hierarchy

[2] http://gamma.cs.unc.edu/COL/OpenCCL

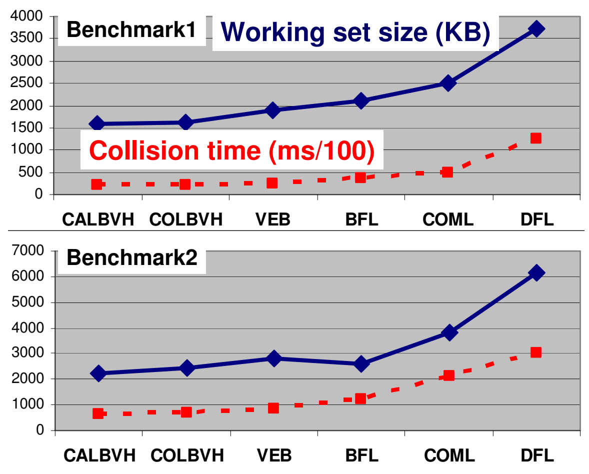


Figure 5: **Performance of Collision Detection:** *Average collision query time and the size of working set for collision detection in the two benchmarks. We highlight the performance of other layouts (i.e. VEB, DFL and BFL) and compare them with our layouts (COLBVH and CALBVH). VEB is the van Emde Boas layout, DFL and BFL are the depth-first and breadth-first layouts, respectively. The top and bottom graphs correspond to the first and second benchmarks, respectively. We obtain* 13%–468% *improvement in the performance of collision queries based on reduced working set size and fewer cache misses. Moreover, the performance of cache-oblivious layout (COLBVH) is comparable to that of cache-aware layouts (CALBVH).*

is modified. The COML, explained in Sec. 4.3, is computed by constructing an undirected graph. This is accomplished by generating edges between parent and child nodes and between nearby nodes on the same level of the BVH. The VEB layout is computed recursively. The tree is partitioned with a horizontal line so that the maximum height of the tree is divided into half. The resulting sub-trees are linearly stored by first placing the root sub-tree followed by other sub-trees from leftmost to rightmost. This process is applied recursively until it reaches a single node of the tree.

We have tested the performance of the OBBTree collision detection algorithm with our layouts in a rigid body simulation with two different benchmarks:

1. **Bunny and Dragon:** A bunny moves towards a dragon (Fig. 6).

2. **Dragon and Turbine:** A dragon drops onto the CAD turbine model(Fig. 4). This benchmark has much larger contacting area compared to the first benchmark

We collected timing data after making sure that there is no loaded data in main memory. Moreover, we also made sure there is no file fragmentations since the fragmentations can slow down the performance of I/O accesses. Dynamic simulations of the two benchmarks are shown in the accompanying video. In this case, we are able to achieve a 13%–468% improvement in the performance of collision queries by using COLBVHs over other layouts on our benchmarks. This improvement is achieved by reducing the working set during collision queries and fewer cache misses. Moreover, the performance of cache-oblivious layout is comparable to that of cache-aware layout. In Fig. 5, we report the average collision query times and working set size in the two benchmarks.

In the two benchmarks, VEB layout has slightly worse performance over our layouts. This is mainly because the OBBTrees are almost balanced trees and the ordering of child clusters from left to right during VEB layout computation maintains reasonably

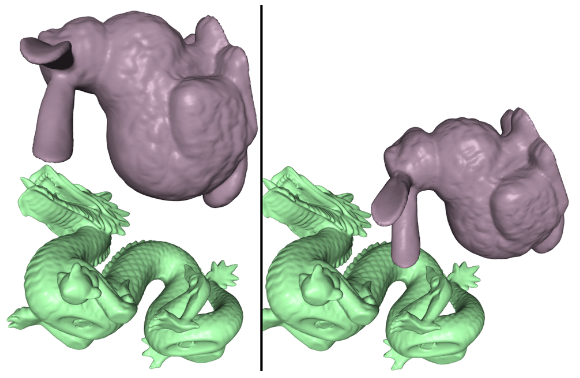


Figure 6: **Dynamic Simulation between Bunny and Dragon Models:** *This image sequence shows discrete positions from our dynamic simulation between bunny and dragon models. We are able to achieve* 13%–468% *performance improvement by using our cache-oblivious layouts of OBBTrees as compared to other layouts. Moreover, the cache-oblivious layout has only* 10% *lower performance as compared to the cache-aware layout.*

good cache-coherence. Another thing to note is that BFL layout has smaller working set size compared to VEB in the second benchmark, which has much larger contact area as compared to the first benchmark. Therefore, BFL is more suitable in that case.

7.3.2 Ray Tracing

We implemented an interactive ray tracer based on kd-trees [Wal04]. We applied our layout algorithm to compute cache-oblivious layouts of kd-trees. Since the BV of each kd-node is fully contained in its parent BV and some BVs can have zero volumes, we use the surface areas of BVs as the volume for probability computation. Such techniques have also been used by kd-tree construction algorithms [MB90]. We compute the probability based on the ratio of surface areas and use our layouts of kd-trees without any modification of runtime ray tracer.

We tested different layouts of the Lucy model consisting of 28 million triangles. Please note that the kd-tree of the Lucy model is somewhat unbalanced since the standard deviation of depth of leaf nodes is about 3, which is almost 10% of the average depth of the tree. We also ensure that there is no fragmentations in the data files.

We are able to achieve 77%–180% improvement in the performance of ray tracing and able to achieve 7%–55% reduction in the size of working set compared to other layouts. In this case, the performance improvement cannot be directly measured by reduction in the working set size since the I/O access time is also affected by other factors including disk I/O sequential prefetching. Since the cache-oblivious layout stores coherent data in spatially close region on the disk, it is likely that its layout is well suited to reducing disk I/O access times. We report the rendering time and the working set size in Fig. 7. The ray tracing traverses the kd-tree in the depth-first order and performs intersection tests between the BVs of kd-tree and the rays. Moreover, there is no overlap between BVs of kd-nodes that are not descendant to each other. Therefore, depth-first layout is likely to be more coherent to the runtime traversal compared to van Emde Boas (VEB) layout and the breadth-first layout (BFL). Our experimental results also support this conjecture.

8 Analysis and Limitation

In this section, we analyze the performance of our algorithm and discuss some of its limitations.

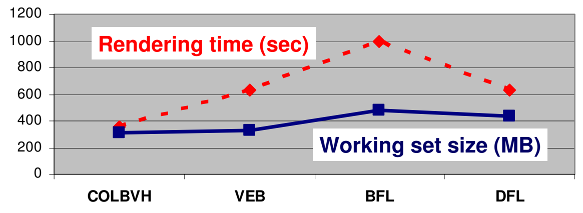


Figure 7: **Performance of Ray Tracing:** *Average render time and the size of working set during ray tracing of the Lucy model with* 28 *million triangles are shown with different layouts. We are able to achieve* 77%–180% *improvement in the performance of ray tracing and reduce the working set size by* 7%–55%.

8.1 Performance Analysis

Our layout algorithm computes cache-efficient layouts of BVHs to reduce cache misses during runtime applications. Since many current caching architectures employ a block fetching mechanism [AV88], we can get performance improvement by storing related data into one block. Moreover, the performance of the cache-oblivious layouts of BVHs strongly depends on the size of each BV as compared to the size of cache block. We observe higher performance improvement when we have many disk I/O accesses with block size of 4KB. On the other hand, we do not achieve significant improvement in terms of reducing L1/L2 cache misses, which have block size of 64 bytes. In the extreme case, when the block size is exactly same as the size of each element used in the layout computation, there is very little improvement based on our layouts.

8.2 Comparison with Cache-Oblivious Mesh Layouts

Yoon et al. [YLPM05] presented a cache-oblivious mesh layout (COML) to minimize the number of cache misses during runtime accesses on the mesh. They compute the layouts by representing access patterns of applications as an undirected graph and deriving a cache-oblivious metric which decides whether a local permutation would decrease cache misses or not. We were able to achieve 50%–200% increased performance over the cache-oblivious mesh layout (COML). We attribute the improvement of our new algorithm to the following reasons:

- **Clustering method:** The COML method uses a graph partitioning to compute layouts for any graph that may correspond to a polygonal mesh or a BVH. However, there is no guarantee that the clustering outputs of the graph partitioning on the input graph satisfy the convexity property, which is very important to compute cache-coherent layouts of trees. Therefore, the constructed layout of the BVH may be far from the optimal layout that minimizes the size of the working set during traversal of collision queries. Instead, our layout algorithm optimized for BVHs always guarantees that the clustering output satisfies the convexity property. At the same time, our layout maximizes the probabilities that BVs, which are accessed together due to parent-child locality, are stored in the same cluster.

- **Probability computation:** In order to construct an input graph for the COML algorithm, edges should be created to represent access patterns of traversals of collision queries. However, it is difficult to consistently compute weights of edges that represent parent-child or spatial localities in the graph. The edge creation methods for BVHs described in [YLPM05] do not adequately represent access patterns of the traversals. On the other hand, our algorithm (COLBVHs) considers the two different localities separately and quantifies the probability that a node is accessed during runtime traversal based on the geometric relationship between the BVs. As a result, we are able to compute more accurate weights for layout computation.

8.3 Limitations

Our algorithm works well in our current set of benchmarks. However, it has certain limitations. Our greedy algorithm is based on several heuristics to compute cache-coherent layouts based on parent-child locality. Therefore, there is no guarantee that our cache-oblivious layouts of BVHs always reduce the number of cache misses or the size of the working set. Moreover, our current layout algorithm assumes that traversals of collision queries start from the root node of the BVH. However, some implementations of collision queries (e.g. front-tracking) may take advantage of temporal coherence and start from the front of BV nodes from the previous frame as opposed to the root.

9 Conclusion and Future Work

We have presented a novel algorithm to compute cache-efficient layouts of BVHs. We do not make assumptions about the cache parameters or the memory hierarchy and take advantage of coherent data access patterns on BVHs. We describe a new probabilistic model to predict the runtime access patterns of applications on BVHs. We decompose the access patterns during the traversal of BVHs into a set of search queries and utilize parent-child and spatial localities between the accessed nodes. Our layout algorithm separately considers these two localities and reduces the number of cache misses and the size of working set. We have applied our cache-oblivious layouts of BVHs to collision detection between complex models and ray tracing of massive models. We were able to achieve 15%–400% improvements on the performance over different layouts.

There are several areas for future work. We would like to extend our probability formulation that predicts runtime data access patterns of collision queries to consider other proximity queries such as minimum separation distance. We also would like to compute cache-coherent layouts of other hierarchical representations such as multiresolution meshes (e.g. vertex hierarchies) by extending our layout algorithm. Finally, we would like to design layout algorithms for deformable models.

References

Arge L., Brodal G., Fagerberg R.: Cache oblivious data structures. *Handbook on Data Structures and Applications* (2004).

Alstrup S., Bende M. A., Farach-Colton E. D. D. . M., Rauhe T., Thorup M.: Efficient tree layout in a multilevel memory hierarchy. *Computing Research Repository (CoRR)* (2003).

Aggarwal A., Vitter J. S.: The input/output complexity of sorting and related problems. *Commun. ACM 31* (1988), 1116–1127.

Coleman S., McKinley K.: Tile size selection using cache organization and data layout. *SIGPLAN Conference on Programming Language Design and Implementation* (1995), 279–290.

Deering M. F.: Geometry compression. In *SIGGRAPH 95 Conference Proceedings* (Aug. 1995), Cook R., (Ed.), Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 13–20. held in Los Angeles, California, 06-11 August 1995.

Dobkin D., Hershberger J., Kirkpatrick D., Suri S.: Computing the intersection-depth of polyhedra. *Algorithmica 9* (1993), 518–533.

Diaz J., Petit J., Serna M.: A survey of graph layout problems. *ACM Computing Surveys 34*, 3 (2002), 313–356.

Frigo M., Leiserson C., Prokop H., Ramachandran S.: Cache-oblivious algorithms. *Symposium on Foundations of Computer Science* (1999).

Gil J., Itai A.: How to pack trees. *Journal of Algorithms* (1999).

GOTTSCHALK S., LIN M., MANOCHA D.: OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96* (1996), 171–180.

HAVRAN V.: Cache sensitive representation for the bsp tree. *Proc. of Compugraphics* (1997).

HOPPE H.: Optimization of mesh locality for transparent vertex caching. *Proc. of ACM SIGGRAPH* (1999), 269–276.

ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. In *ACM Trans. on Graphics (Proc. of ACM SIGGRAPH)* (2003), vol. 22, pp. 935–942.

ISENBURG M., LINDSTROM P.: *Streaming Meshes*. Tech. Rep. UCRL-CONF-201992, LLNL, 2004.

KLOSOWSKI J., HELD M., MITCHELL J., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics 4*, 1 (1998), 21–37.

LINDSTROM P., PASCUCCI V.: Visualization of large terrains made easy. *IEEE Visualization* (2001).

MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer* (1990).

MIRTICH B., CANNY J.: Impulse-based simulation of rigid bodies. In *Proc. of ACM Interactive 3D Graphics* (Monterey, CA, 1995).

PASCUCCI V., FRANK R. J.: Global static indexing for real-time exploration of very large regular grids. *Super Computing* (2001).

SAGAN H.: *Space-Filling Curves*. Springer-Verlag, 1994.

SEN S., CHATTERJEE S., DUMIR N.: Towards a theory of cache-efficient algorithms. *Journal of the ACM 49* (2002), 828–858.

VAN EMDE BOAS P.: Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.* (1977).

VELHO L., GOMES J. D.: Digital halftoning with space filling curves. In *Computer Graphics (SIGGRAPH '91 Proceedings)* (July 1991), Sederberg T. W., (Ed.), vol. 25, pp. 81–90.

VITTER J.: External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys* (2001), 209–271.

WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

WEGHORST H., HOOPER G., GREENBERG D.: Improved computational methods for ray tracing. *ACM Transactions on Graphics* (1984), 52–69.

YOON S.-E., LINDSTROM P., PASCUCCI V., MANOCHA D.: Cache-Oblivious Mesh Layouts. *Proc. of ACM SIGGRAPH* (2005).

R-LODs: Fast LOD-based Ray Tracing of Massive Models

Sung-Eui Yoon[1] Christian Lauterbach[2] Dinesh Manocha[2]
[1]Lawrence Livermore National Laboratory [2]University of North Carolina at Chapel Hill
{sungeui,cl,dm}@cs.unc.edu
http://gamma.cs.unc.edu/RAY

UNC Tech. Report: T06-009, Jan 2006

Abstract

We present a novel LOD (level-of-detail) algorithm to accelerate ray tracing of massive models. Our approach computes drastic simplifications of the model and the LODs are well integrated with the kd-tree data structure. We introduce a simple and efficient LOD metric to bound the error for primary and secondary rays. The LOD representation has small runtime overhead and our algorithm can be combined with ray coherence techniques and cache-coherent layouts to improve the performance. In practice, the use of LODs can alleviate aliasing artifacts and improve memory coherence. We implement our algorithm on both 32bit and 64bit machines and able to achieve up to 2–20 times improvement in frame rate of rendering models consisting of tens or hundreds of millions of triangles with little loss in image quality.

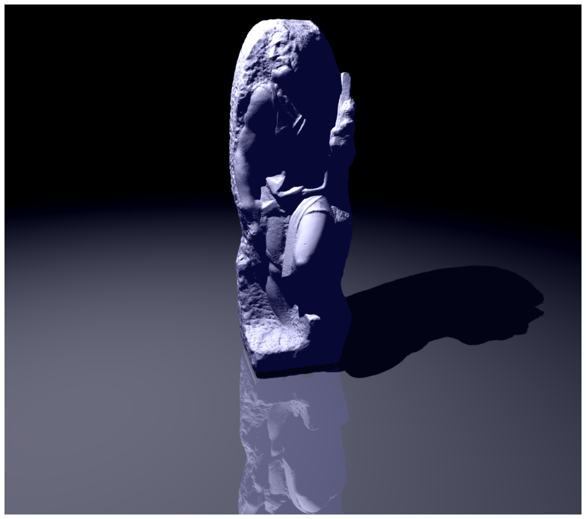


Figure 1. **St. Matthew Model:** *We use our LOD-based algorithm to accelerate ray tracing of the St. Matthew model with shadows and reflections. We render the $128M$ triangle model at 512×512 resolution with 2×2 anti-aliasing and pixels-of-error (PoE) $= 4$. We are able to achieve $2-3$ frames per second on two dual-core Xeon processors with 4GB of memory. We observe a $2-20$ times increase in the frame rate due to R-LODs with very little loss in image quality.*

1 Introduction

In recent years, there has been a renewed interest in real-time ray tracing for interactive applications. This is due to many factors: firstly, processor speed has continued to rise at exponential rates as predicted by Moore's Law and is approaching the raw computational power needed for interactive ray tracing. Secondly, ray tracing algorithms can be parallelized on shared memory and distributed memory systems. Therefore, the current hardware trend towards desktop systems with multi-core CPUs and programmable GPUs can be used to accelerate ray tracing. Finally, recent algorithmic improvements that exploit ray coherence can achieve a significant improvement in rendering time [22, 31].

Our goal is to perform interactive ray tracing of massive models consisting of tens or hundreds of millions of triangles on current desktop systems. Such gigabyte-sized models are the result of advances in model acquisition, computer-aided design (CAD), and simulation technologies and their complexity makes interactive visualization and walk-throughs a challenging task. In the context of rendering massive models, ray tracing has an important property: its asymptotic performance is logarithmic in the number of primitives for a given resolution. This is due to the use of hierarchical data structures such as bounding volume hierarchies or kd-trees. The asymptotic complexity makes ray tracing an attractive choice, especially for rendering of massive models.

The logarithmic growth, however, continues only as long as the system has sufficient main memory to contain the entire model and hierarchical data structures. As models grow much larger, the size of the hierarchical structure also increases linearly and the underlying ray tracer performs its computations in an out-of-core manner, slowing down drastically. A major trend in computing hardware has been the increasing gap between processor speed and memory speed. Moreover, disk I/O accesses are in general more than three orders of magnitude slower than main memory accesses. Because of these gaps, hardware advances are not expected to provide an efficient solution to the problem of ray tracing massive models.

Main Contributions: We present a new algorithm to accelerate ray tracing of massive models using geometric levels-of-detail (LODs). Our approach computes simple and drastic simplifications, called *R-LODs*, of the polygonal model. The R-LODs have a compact representation and are tightly integrated with the kd-tree. We present a simple and efficient LOD error metric to bound the error for primary and secondary rays. Additionally, we use techniques based on ray coherence and cache-oblivious layouts to improve the performance of our LOD based ray tracing algorithm. R-LODs also alleviate the temporal aliasing that can arise during rendering of highly tessellated models.

We have implemented and tested our system on two machines running Windows XP 32-bit and 64-bit with two dual-core Xeon CPUs and have evaluated its performance on different kinds of models with $20 - 128M$ triangles. The performance gain of our LOD based ray tracer is proportional to the reduction in the working set size and the number of intersection tests. The frame rate improvement varies from 2 times on models with small working set size to almost $20 - 50$ times on models with very large working set size.

Our ray tracing algorithm offers the following *benefits*:

1. **Simplicity:** R-LODs are very easy to implement and their representation has small runtime overhead. Our algorithm maintains the simplicity, coherence, and performance of the kd-tree data structure.
2. **Interactivity:** The LOD based ray tracer provides a framework for interactive ray tracing due to the fact that we can trade off image quality for improved frame rate.
3. **Front size:** R-LODs reduce the size of the front traversed in the kd-tree. This results in fewer ray intersection tests and decreases the size of the working set.
4. **Coherence:** R-LODs make memory accesses more coherent and reduce the number of L1/L2 cache misses and page faults. Furthermore, they can also improve the performance of ray coherence techniques.
5. **Generality:** Our algorithm is applicable to a wide variety of polygonal models, including scanned and CAD models.

Organization: The rest of the paper is organized in the following manner: section 2 gives a brief summary of prior work in interactive rendering. We give an overview of our approach in Section 3 and present the R-LOD representation and computation algorithm in Section 4. Section 5 shows acceleration techniques based on cache-coherent layouts and ray coherence. We describe the implementation of our ray tracer and analyze its performance on different models in Section 6. Finally, section 7 compares our algorithm with other approaches.

2 Related work

In this section, we give a short overview of interactive ray tracing and the use of LODs for interactive rendering.

2.1 Interactive Ray Tracing

Ray tracing was introduced by Appel [3] and Whitted [36] and is a very well studied field. In this section, we just briefly survey some recent techniques used to accelerate ray tracing, but a detailed description is available in [27]. At a broad level, we classify prior approaches into four categories:

Exploiting ray coherence: The underlying idea here is not to trace each ray by itself, but to utilize the fact that neighboring rays exhibit spatial coherence. Earlier attempts to exploit this concept were beam tracing [11], pencil tracing [26] and cone tracing [2]. More recently, Wald *et al.* [31] group rays into bundles and use them to accelerate traversal and intersection with primitives for all rays simultaneously by taking advantage of SIMD instructions. Reshetov *et al.* [22] propose an algorithm to integrate beam tracing with the kd-tree spatial structure and were able to further exploit ray coherence.

Hardware acceleration: Another trend has been to use hardware support to accelerate ray tracing. Purcell *et al.* [21] show that ray tracing could be adapted to the streaming model of current programmable GPUs, which are mainly designed for rasterization. Schmittler *et al.* [25] and Woop *et al.* [38] present prototypes for a complete and programmable ray tracing hardware architecture to run at interactive rates.

Parallel computing: Ray tracing is easily parallelizable due to the fact that all rays can be traced independently. Parker *et al.* [19], DeMarle *at al.* [7], and Dietrich *et al.* [8] describe an interactive ray tracer for rendering large scientific or CAD datasets running on shared memory or distributed architectures. Wald *et al.* [34] built a ray tracer to run on clusters of commodity hardware machines and were able to achieve interactive frame rates for large architectural and CAD models. Both of these systems are mainly intended for models that could be kept in the main memory of a shared memory system or of PCs used in the cluster.

Large datasets: Many algorithms have been presented to improve the performance of ray tracing on large datasets [7, 10, 20, 32]. Our approach is complimentary to these methods and can be combined with them to further improve the performance.

2.2 Interactive Rendering using LODs and Out-of-Core Techniques

LODs have been widely used to accelerate rasterization of large polygonal datasets [16]. At a broad level, prior algorithms can be classified into static LODs, view-dependent simplification, image-based representations and hybrid combinations of geometric and image-based representations. Out-of-core algorithms are an active area in computer graphics and visualization with the goal to efficiently handle large datasets [4]. LOD algorithms can be combined with out-of-core techniques to rasterize large polygonal datasets composed of tens or hundreds of millions of polygons at interactive rates on commodity PCs [23, 6, 41, 9].

LOD-based based algorithms can also be applied to accelerate ray tracing. Christensen *et al.* [5] introduce a LOD approach for an offline ray tracer based on ray differentials [12]. Wand and Straßer [35] propose an algorithm for multiresolution ray tracing of point-sampled geometry based on ray-differentials. Another approach is to integrate the LODs into the hierarchical structure [37]. Recently, Stoll *et al.* [28] proposed a novel architecture for dynamic multiresolution ray tracing. They proposed a watertight multiresolution method by interpolating between discrete LODs for each ray. Their discrete LODs are computed from choosing proper tessellation levels for subdivision meshes. Also, efficient algorithms based on depth images can be used to accelerate ray tracing [15, 1].

3 Overview

In this section, we discuss many issues that govern the performance of ray tracing and give an overview of our approach.

3.1 Ray tracing of massive models

In this paper, we restrict ourselves to triangulated models, though our approach can also be extended to other primitives such as point clouds. All efficient ray tracers employ hierarchical data structures to avoid testing each ray with every primitive. We use the kd-tree, which is a special case of the general BSP tree and has recently become a popular choice

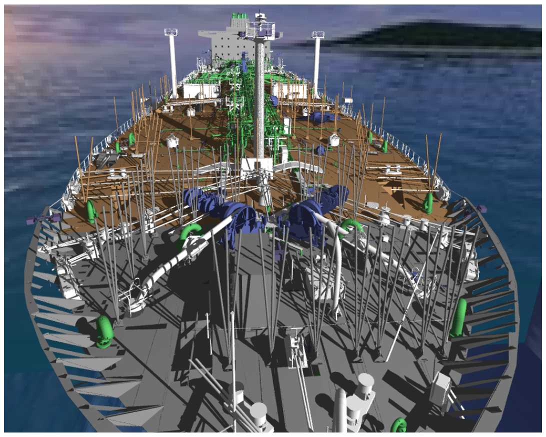


Figure 2. **Double Eagle Tanker**: *The deck of the Double Eagle tanker with shadows is shown using ray tracing. We are able to achieve* 1*-*3*fps at* 512 *by* 512 *image resolution with* $2x2$ *super-sampling and PoE* $= 4$ *on a dual Xeon workstation. In this model, the working set of the ray tracer is low and we are able to achieve up to* 2 *times improvement in the frame rate.*

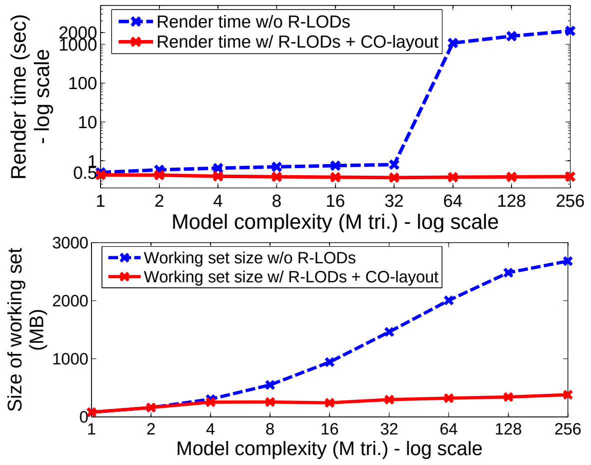


Figure 3. **Performance of Ray Tracing**: *We precompute simplified versions of the St. Matthew model and ray trace each simplification separately from the same viewpoint. We measure the frame time and working set size of the ray-tracer, with and without R-LODs by using different simplified models on a* 64*-bit machine with* 2*GB RAM. Notice the big jump in frame time for ray tracing without R-LODs, as the working set of ray tracing with massive models exceeds the available RAM. On the other hand, our R-LOD based ray tracing combined with cache-oblivious layouts (CO-layout) achieves near-constant performance in terms of frame rate and the working size.*

due to its simplicity and performance [10, 27]. Each node of the kd-tree represents one subdivision of the parent's space and contains information about the axis-aligned plane used for the split as well as pointers to its child nodes. We use the optimized representation proposed by Wald *et al.* [30] and extend it to efficiently handle LODs.

Out-of-core ray tracing: Ray tracers taking advantage of hierarchical data structures should exhibit a logarithmic growth rate as a function of the model complexity [31]. We measured the performance of a coherent ray tracer during rendering different simplification levels of the St. Matthew model, as shown in Fig. 3. Our experiment indicates that ray tracing performance increases as a logarithmic function of model complexity as long as the kd-tree and primitives of a model can fit in the main memory. However, once the model size and the working set size of the kd-tree exceeds the available main memory of the machine, the disk I/O significantly affects the performance of the ray tracer.

Ray coherence: Recent approaches that exploit spatial and ray coherence decrease the number of memory accesses and therefore also the number of disk accesses for large models [31, 22]. These algorithms perform traversals and intersections for multiple spatially-coherent rays in a group at the same time. In general, rays in a group exhibit higher coherence at the higher levels of the kd-tree (that usually are in main memory) because each ray in the group is likely to follow same path in the tree as other rays. However, accesses to the nodes deeper in the tree are incoherent and, thus, result in disk cache misses, especially when dealing with massive models, since bounding box of those nodes become smaller compared to width of the ray group. Therefore, in order to accelerate out-of-core ray tracing, we need to reduce the number of accesses made to the nodes deeper in the tree.

3.2 Our Approach

We mainly address the problem of ray tracing massive models. If models have high depth complexity, current traversal algorithms based on kd-trees can efficiently handle such kinds of models. In this case, the working set size is proportional to the number of visible primitives from the primary and secondary rays. Therefore, we primarily deal with the problem of fast ray tracing when the number of visible primitives is high.

We assume that each ray or ray bundle is represented by a pyramidal beam or frustum. As described in the multi-level ray tracing of Reshetov *et al.* [22], during traversal the frustum is checked for intersection with the bounding box of the current kd-tree node by using an inverse frustum culling approach. This results in two interesting cases:

1. Models with large primitives: If the bounding box of the node is larger than the frustum, it is likely that the node intersects with the whole beam, i.e. we can exploit spatial coherence. Typically, architectural models or CAD models result in such cases whenever the model is coarsely tessellated, has large planar primitives or is viewed at close range.

2. Highly tessellated models: In this case, the bounding box is much smaller than the frustum. This implies that the beam needs to be split into smaller sub-beams. However, if the beam represents just one ray, then further subdivision is not possible, even though the sub-tree represented by the node has a high number of descendants and, thus, there is high local geometric complexity. Therefore, ray coherence approaches like multi-level ray tracing and ray packet tracing fall back to normal ray tracing and may not offer much benefit. For example, consider ray tracing a St.Matthew model consisting of 128M triangles at a resolution of 1024^2 primary rays. Assume that every ray hits the model and half of the model's triangles are visible to the eye. In this case, fewer than 1% of the actual triangles are hit by one of the rays. Moreover, each of these triangles is sampled as a representative of several triangles in the subtree. This has two consequences: first, the memory accesses may be *incoherent* because each triangle may lie in a different part of memory. Secondly, *temporal aliasing* can occur between frames since it is likely that a different triangle will be chosen in successive frames.

Our novel LOD-based ray tracing algorithm handles this second case by choosing our precomputed R-LOD representation when traversal determines that a LOD metric is satisfied. This means that traversal can stop before reaching the deep levels of the tree, reducing the number of incoherent accesses and the size of the working set, while maintaining ray coherence so that related techniques still work well. As a result, we obtain significant improvements in rendering speed.

4 LOD Based Ray Tracing

In this section we present the R-LODs that are used to accelerate ray tracing. We first describe our R-LOD representation and the modified traversal algorithm. Then we present our LOD error metric and the R-LOD construction algorithm.

4.1 R-LOD Representation for Ray Tracing

Our goal for interactive ray tracing is to design a LOD representation that retains the benefits of kd-tree based acceleration algorithms, i.e. simplicity, efficiency and low runtime overhead.

A R-LOD consists of a plane with material attributes (e.g. color), which is a drastic simplification of the descendant triangles contained in an inner node of the kd-tree, as shown in Fig. 5. Each R-LOD is also associated with a surface deviation error which is used to quantify the projected screen-space error at runtime.

Let us assume that the original tree has height h, where $h \approx \log_2(n)$, and n is the number of triangles in the original model. The R-LOD associated with a kd-tree node at height k is a simplification into a plane of the 2^k descendant triangles. Our choice to use such a representation is motivated by the following goals:

Simple and efficient LOD representation: Current ray object intersection algorithms based on the kd-tree are highly optimized for interactive ray tracing. We use simple representations for LODs to minimize storage and traversal overhead. Each R-LOD adds 4 bytes to an inner node of the kd-tree. We also use a simple and fast LOD selection algorithm to reduce the traversal overhead.

Drastic model simplification: The computational workload of ray tracing is a logarithmic function of the model complexity. If the model size is reduced by a factor of m, the tree traversal overhead reduces by only $\log(m)$. As a result, m has to be a significant number, say 2^3or 2^4.

High quality rendering: Ray tracing is primarily used to generate high-quality rendering. Since the R-LODs are a drastic simplification of the original model, their use can result in visual artifacts. In order to control the errors caused by R-LODs, we associate a deviation error metric and compute a screen-space projection in terms of *pixels-of-error* (PoE) deviation. Also, we assume that our drastically simplified LOD representations are mainly used given small PoE values (e.g., 1–4 pixels at image resolution 1024×1024) for high-quality rendering.

4.2 Runtime Traversal with R-LODs

Our new traversal algorithm is a modification of the efficient traversal algorithm described in Wald's thesis [30] and [29]. We recursively traverse the kd-tree from the root node or the entry-point that is computed using multi-level ray tracing.

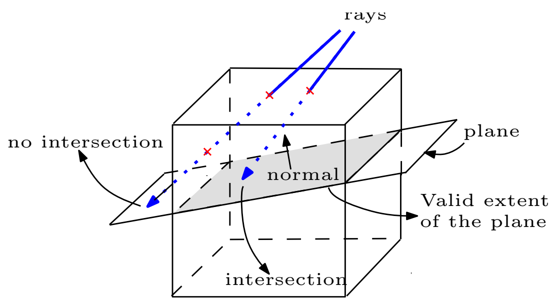


Figure 5. **LOD Representation:** *A R-LOD consists of a plane with material attributes. It serves as a drastic simplification of triangle primitives contained in the bounding box of the subtree of a kd-tree node. Its extent is implicitly given by its containing kd-node. The plane representation makes the intersection between a ray and a R-LOD very efficiently and results in a compact representation.*

When we reach an intermediate node associated with a R-LOD, we check whether we can use the R-LOD based on our LOD error metric. If the current R-LOD satisfies the LOD error metric, we perform an intersection test between a R-LOD and the ray. If there is an intersection, we stop the traversal and return the intersection data of the R-LOD to compute shading and shoot secondary rays, if necessary. If there is no intersection, the algorithm does not traverse the child nodes of the intermediate node associated with the R-LOD. Each R-LOD is bounded by a kd-node and therefore, the extent of the plane of the R-LOD is implicitly bounded by the kd-node during tree traversal. The implicit extent of the plane results in a compact R-LOD representation.

4.3 LOD Error Metric Evaluation

We use a projection-based algorithm integrated with surface deviation error to select appropriate LODs for ray tracing.

Conservative projection algorithm: We use a projection method to efficiently compare the screen-space area of the R-LOD after the perspective projection with the PoE in the screen-space. Conceptually, we position a projection plane at the intersection between the ray and the kd-tree node. The plane is set to be orthogonal to the ray, as shown in Fig. 6. We enclose the R-LOD (and its corresponding simplified geometry) in a sphere. The area of the R-LOD projected onto the projection plane is conservatively measured by computing πR^2, where R is the radius of the sphere. Let R_p be the radius of a sphere inscribed in a rectangular shape pixel of the image screen. In this case, R_p is simply half of the width of the pixel. Then, the projected area of a pixel in the projection plane satisfies the following relationship:

$$\frac{d_{near}}{R_p} = \frac{d_{min}}{\hat{R}_p} \Rightarrow \hat{R}_p = d_{min}\frac{R_p}{d_{near}} = d_{min}C, \qquad (1)$$

where $\hat{R}_p$ is the projected radius of R_p, d_{near} is the distance from the viewer to the image plane, and d_{min} is the distance from the viewer to the intersection point between the ray and the kd-node. Since $\frac{R_p}{d_{near}}$ ($= C$) is a constant, the projected radius, $\hat{R}_p$, is a simple linear function of the distance, d_{min}, along the ray from the eye to the intersecting node. We select an R-LOD if $\hat{R}_p$, is bigger than the radius, R, associated with the R-LOD. Our LOD metric is very efficient as it requires only one multiplication and d_{min} is already known during the

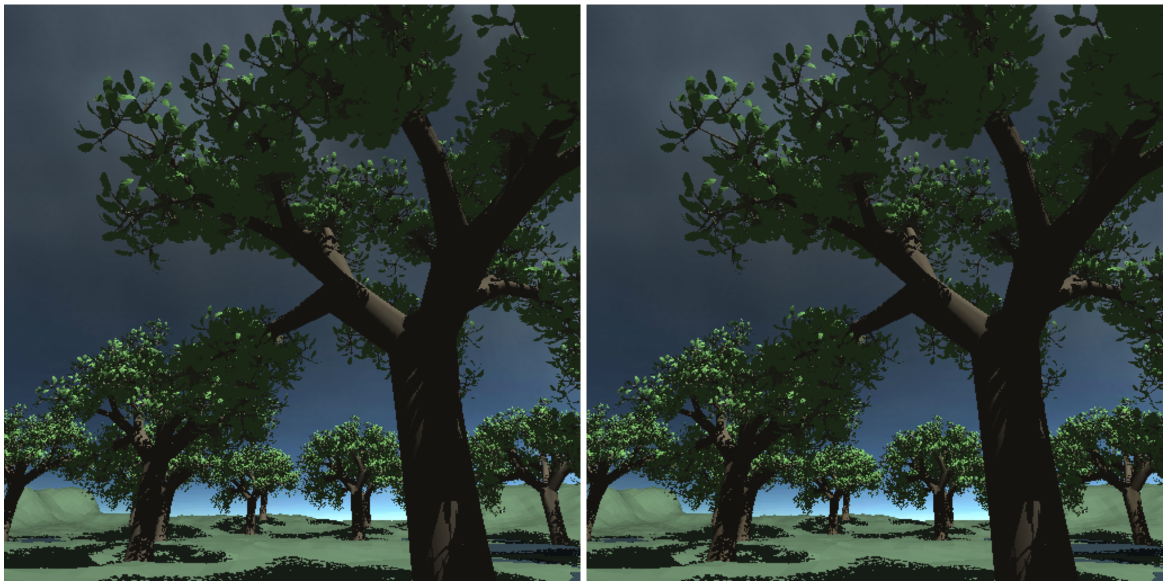

(a) No LOD (b) PoE $= 4$

Figure 4. Forest Model: *We render the forest model consisting of* 32 *million triangles with shadow rays using PoE* $= 0$ *and PoE* $= 4$*. The image resolutions are* 512×512 *without anti-aliasing to highlight image quality differences. We are able to render the model given the viewpoint at the* 1.6 *frames per second and achieve* 5 *times improvement by using R-LODs.*

tree traversal.

Surface deviation: The error metric described above conservatively measures the projected screen-space area of the R-LOD. We augment the metric to take into account the surface deviation of a R-LOD. For this we first measure the surface deviation between the plane of the R-LOD and all the contained triangles. We combine the surface deviation and the projected screen-space area of the R-LOD in the following geometric formulation. We compute the volume of the surface deviation along the plane and add this volume to the volume of the sphere enclosing the R-LOD. We then treat the summed volume as a volume of an imaginary sphere and use its radius as the error bound of the R-LOD. In this geometric formulation, these two seeming different error bounds can be treated in a uniform manner.

Error quantization: The exact representation of the plane and associated materials takes 32 bytes. Instead of directly associating this information with each node of the kd-tree, we quantize the error bounds associated with the R-LODs and store the quantized error bound as well as an R-LOD index in a 4 byte structure as the part of the kd-node in order to reduce the working set size during traversals. Therefore, only if the error bound of an R-LOD is satisfied within the PoE bound, we load the exact R-LOD representation by using the R-LOD index. When considering a path from a leaf node of the kd-tree to the root node, the error bounds associated with the nodes increase as a geometric series. Therefore, we use a geometric distribution equation to quantize the error values associated with the R-LODs. We found that 5 bits are enough (i.e. 10%–20% quantization error) to conservatively quantize the error bound of the R-LODs in our benchmarks; therefore, each R-LOD index is stored in 27 bits, which are enough to indicate all the R-LODs in our tested models.

Secondary rays: Our LOD metric based on conservative projection also extends to secondary rays. These include *reflection* (in which a ray reflects at an intersection point with a reflective triangle) and *shadow* rays. This is mainly because these secondary rays can be expressed as a linear transformation [11]. In the case of reflection, the radius, R_p, of the sphere inscribed in the pixels of the image space increases linearly based on the sum of the distance from the viewer to an intersecting reflective triangle, and to another intersecting object along the primary or reflective secondary rays. Similarly, our metric also works well for shadow rays and again we use a linear transformation. One issue with using LODs for shadow rays is that they can result in self-shadowing artifacts when different versions of the R-LODs are selected by the primary ray and the shadow ray. We overcome this problem by ignoring the intersections between the shadow ray and the primitives that are within the LOD error bounds associated with the R-LOD selected by the primary ray.

Our projection-based method does not work with *refraction*, since refraction is not a linear transformation [11]. In this case, we use a more general, but expensive method based on ray differentials [12], to decide whether an R-LOD satisfies the PoE bound after refraction.

4.4 R-LOD Construction

Our goal is to compute a plane that approximates the triangles that are contained in the subtree of an intermediate kd-node and also their material properties. If a triangle contained in the subtree is not fully contained in the bounding box of the node, we clip the triangle against the box and do not consider the clipped portion of the triangle. We use principal component analysis (PCA [13]), to compute the plane. PCA computes the eigenvectors that provide a statistical description of input points. We perform PCA computation based on the vertices of the triangles, but also take into account the size of the

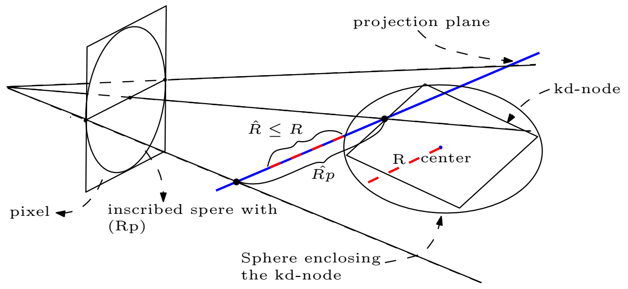


Figure 6. Projection-based LOD Metric: *We place a projection plane at the intersection point between a ray and the kd-node. The plane is orthogonal to the ray. Based on this projection plane, we conservatively check whether the R-LOD satisfi es the error metric.*

triangles by associating the area of the triangle as a weight for each vertex. The plane is computed based on the eigenvector associated with the largest eigenvalue and this eigenvector represents the normal to the plane[1]. We compute material properties that are mean values of the contained triangles and associate them with the R-LOD. The surface deviation of the plane against the geometric primitives is computed based on the smallest eigenvalue, which corresponds to a variance of geometry along the normal of the plane.

Hierarchical R-LOD computation: We can compute the R-LODs associated with each node of the tree in a bottom-up manner. However, a naive algorithm would compute the R-LOD for each node independently and this can result in a $O(n \log n)$ algorithm.

Instead, we present a R-LOD computation algorithm that has linear time complexity and is well suited for out-of-core computation. Each element, σ_{ij}, of (i, j)th component of a covariance matrix for PCA is defined as the following:

$$\sigma_{ij} = \sum_{k=1}^{n} (V_i^k - \mu_i)(V_j^k - \mu_j), \tag{2}$$

where V_i^k is the ith component (e.g. x, y, and z) of kth vertex data, μ_i is the mean of V_i^k, and n is the number of vertices. This equation can be reformulated as:

$$\sigma_{i,j} = \sum_{k=1}^{n} V_i^k V_j^k - \frac{2}{n} \sum_{k=1}^{n} V_i^k \sum_{k=1}^{n} V_j^k + \frac{1}{n^2} \sum_{k=1}^{n} V_i^k \sum_{k=1}^{n} V_j^k, \tag{3}$$

It follows that if we can compute and store the sums of V_i^k, V_j^k, $V_i^k V_j^k$, and n, we can compute the covariance with these sums and n for any intermediate node. In order to compute the covariance matrix of a parent node, we simply add these variables as a weighted sum of the number of vertices contained in each child node. This property is particularly useful to compute the R-LODs of inner nodes in the kd-tree in an out-of-core manner. Our algorithm has linear time complexity and its memory overhead is a function of the height of the tree. In practice, the memory overhead in our benchmarks is less than 1MB.

[1]The direction of the normal is chosen to be closer to the average normal of triangles.

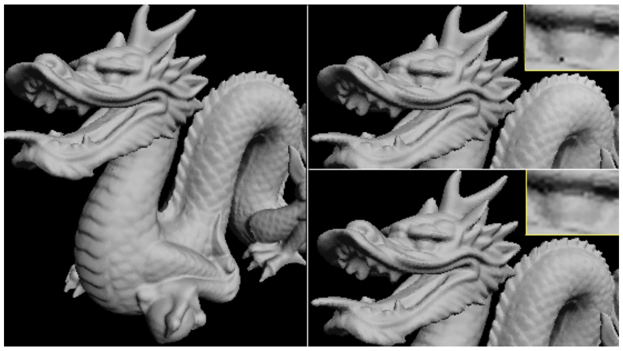


Figure 7. C^0 **Discontinuity**: *The left image shows the Stanford dragon model as rendered by our approach with PoE $= 0$, i.e. using original triangles. The top right image was acquired by setting PoE $= 5$ at 512×512 image resolution with no expansion of R-LODs. As can be seen in the area of the dragon's eye, there is a hole caused by C^0 discontinuity of our LOD representation. By allowing a small amount of expansion of R-LODs, we can avoid having holes in the fi nal image as shown in the bottom right image. Close-ups of the eye are shown in boxes with yellow borders.*

4.5 C^0 Discontinuity between R-LODs

Our LOD computation algorithm computes a drastic simplification. Therefore, if the underlying triangles have high curvature, the PCA-based approximation can have high surface deviation. In this case, it is possible that our algorithm does not maintain C^0 continuity between R-LODs, which can result in some holes in the resulting image (see Fig. 7). This kind of problem has been well-studied in the LOD and point-based rendering literature. Particularly, many techniques in the LOD literature have been proposed to patch these holes using precomputed data structures or runtime algorithms [6, 41]. However, those approaches can increase the storage and runtime overhead of ray tracing algorithms. In our implementation, we do not use any patching techniques.

Instead, we ameliorate this problem through our R-LOD selection algorithm. A very low PoE bound should be used to limit the error introduced by the R-LODs. The low PoE bound also minimizes temporal popping that can arise when we switch between the R-LODs of parent and children nodes during successive frames. Moreover, we assign higher weight to surface deviation computation as part of the error metric computation; therefore, higher resolutions are used in the region with high curvature.

Expansion of R-LODs: In addition to these two heuristics, we also expand the extents of R-LODs to remove holes caused by C^0 discontinuity between R-LODs. Please note that as the surface deviation increases, it is likely that gaps become larger. Therefore, we increase the extent of a R-LOD as a function of the surface deviation associated with the R-LOD. This expansion is efficiently considered during the plane and ray intersection as an additional numerical tolerance. In practice, we found that combining these heuristics work well to remove holes caused by C^0 discontinuity without introducing any noticeable visual artifact given low PoE error bounds (see Fig. 7).

5 Utilizing Coherences

In this section, we describe approaches to improve the performance of our ray tracing algorithm using ray coherence and cache-coherent layouts.

5.1 Ray Coherence

We define ray coherence as the coherence of rays in tree traversal and intersection, i.e. rays may take a similar path in the tree and may hit the same triangles. For primary rays, our ray tracer starts out by assuming there is ray coherence and shoots a beam using the algorithm presented in [22]. We compute a common entry point in the tree for all rays in the beam, at which the beam is split into either sub-beams or ray packets depending on its size. For the latter case, we use the coherent ray tracing algorithm [31] which works on a 2×2 packet of rays in parallel using current processors' SIMD instructions. During all traversal, we check whether we need to use R-LODs that have appropriate resolution based on our LOD metric. If so, we stop traversal of that subtree and intersect with the simplified representation. If the given model is highly tessellated, beam tracing and the use of SIMD instructions may not work well and can even lead to a decrease in performance (as explained in Section 3.2). However, we found that the use of R-LODs alleviates this problem, as we generally do not traverse as deep into the tree and therefore execute less overhead intersections. Secondary rays can be also handled in a similar manner.

5.2 Cache Coherence

It is highly desirable to maintain cache coherence during runtime tree traversals to help to achieve good performance. We apply the cache-oblivious mesh layout algorithm [39] to compute cache-coherent layouts of the kd-nodes. We interpret the kd-tree as a graph that can represent the expected runtime access patterns. The quality of the layout depends on the structure of the input graph. In order to predict the runtime behavior of tree traversal by the graph, we use a simple method to compute the probability that a node will be accessed given that its parent node has been accessed before, based on their geometric relationship [40]. The ray tracing algorithm traverses the child nodes from the parent node when there is an intersection between a ray and the bounding box of the parent node. Therefore, we estimate that the probability that the child node is accessed increases as its surface area compared to its parent node increases. This property is already well exploited by the kd-tree construction algorithms by using the surface areas of the bounding boxes of the kd-nodes [17]. The layouts can increase the performance of the ray tracer by $10 - 60\%$ on massive models. This is in addition to the speedups obtained by R-LODs.

6 Implementation and Results

In this section, we describe our implementation and highlight the performance of our ray tracer on different benchmarks.

6.1 Implementation

We have implemented our R-LOD construction algorithm and ray tracer on both 32-bit and 64-bit machines that have two dual-core Xeon processors running 32-bit and 64-bit Windows XP, respectively. For runtime ray tracing, we use memory mapped files to efficiently access large files of geometry and kd-tree. However, in the 32-bit OS, we can only map up to 3GB total memory. To deal with larger data, we have implemented explicit out-of-core memory access management.

Model	Vert. (M)	Tri. (M)	Node (M)	Size (GB)	R-LOD Comp. (min)
Forest	19	32	105	4.1	10
Double eagle	77.7	81.7	173	9.1	32
St. Matthew	128	256	378	26	124

Table 1. **Benchmark models**
Model complexity, the number of kd-nodes, the total size of kd-tree, geometry, and R-LODs, and the construction time of R-LODs are shown.

This is not necessary in the 64-bit OS where we just use implicit OS memory mapping functionality.

In order to construct the kd-tree for a model that does not fit into main memory, we first subdivide the model into voxels in an out-of-core manner and then build the kd-tree for each of these voxels individually in core [41]. This step can also be performed in parallel on different voxels for speeding up the construction. Afterwards, the kd-tree for each voxel is merged into the global tree, which is used for ray tracing.

Since we have found that the quality of the kd-tree is the most important factor for fast ray tracing, we build the kd-tree using the surface-area heuristic [17, 10] and some further improvements as presented by [22]. Especially important is to introduce extra splits for empty areas in order to bound the geometry more tightly for our R-LOD representation.

6.2 Results

Benchmarks: We have applied our LOD-based ray tracing algorithm to different benchmarks as shown in Table 1. We computed different paths through these models and measured the performance of the ray tracer with and without LODs using a small PoE metric. We use a resolution of 512×512 pixels for interactive rendering. We also use 2×2 super-sampling per pixel; therefore, we effectively shoot $1K \times 1K$ rays from the eye for each frame. We are able to render most of these models at $5 - 12$ frames a second with primary rays and $1 - 8$ frames a second when we include reflections and shadow rays. These results are shown in the video.

Preprocessing: We only compute R-LODs for a subset of the nodes in the kd-tree to avoid excessive memory overhead. Our current implementation selects every third node on the path from the root node to the leaf node. Our unoptimized R-LOD construction implementation can process 2–3 million triangles per minute; most of the processing time is spent on reading data from the disk. The size of R-LODs associated with each node takes less than 10% of the total storage. However, if we consider the additional 4 bytes for R-LOD index and quantized error bound in the kd-nodes, total storage overhead of our R-LOD nodes is roughly 33% compared to the optimized kd-tree representation[30].

Performance variation as a function of PoE: We vary the PoE metric for the St. Matthew model (256M triangles) and measure its benefit on the rendering time, average number of processed nodes per ray, and size of working set per frame. The working set is measured at a granularity of 4KB. In order to show the relative benefit, we linearly scale each value into $[0, 1]$ by scaling the maximum value of each item to 1. The min and max values of each item are as follows: rendering time (ms)(160, 11914), size of the working set(MB) (2, 1565), and average number of processed nodes per ray (13.6, 22.42). As can be seen in Fig. 9, the performance of the ray tracer increases drastically as we linearly increase the PoE values.

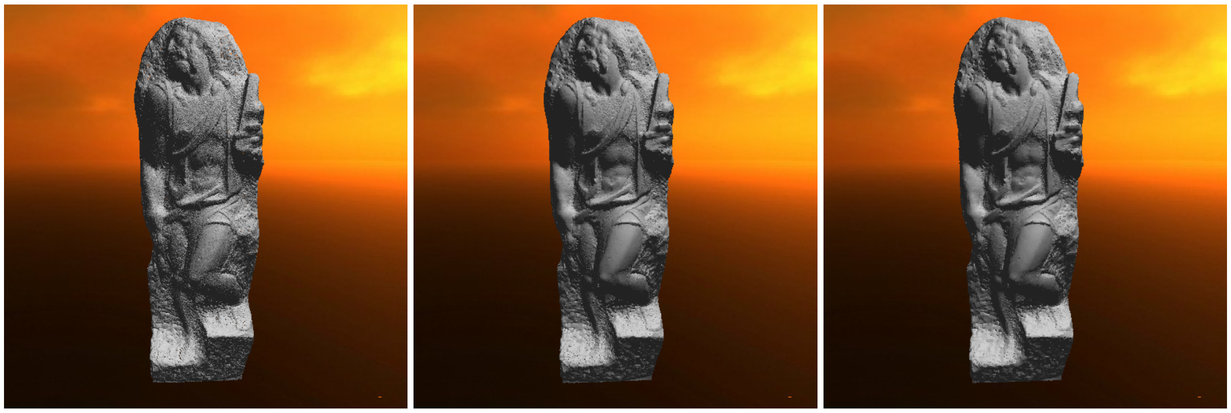

(a) No LOD (b) PoE $= 2.5$ (c) PoE $= 5$

Figure 8. *Images of the St. Matthew model with different PoE values are shown at 512×512 image resolutions. We do not use anti-aliasing to highlight image quality difference. Please note that the original St. Matthew model has many holes. The use of R-LODs can alleviate aliasing artifacts and improve the performance of massive models.*

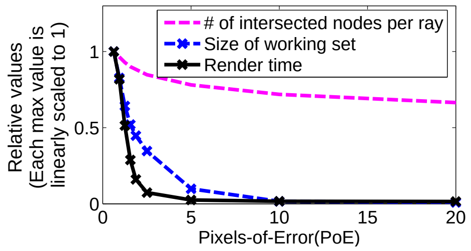


Figure 9. Performance variation as a function of PoE: *We show the relative benefit of R-LODs on different aspects of overall performance of ray tracing St. Matthew model. We measured the rendering time, average number of processed node per ray, and size of working set on a 32-bit Xeon machine with 2GB RAM. All these values are shown in a scale-invariant manner by linearly scaling their maximum values to 1. The performance of our LOD-based ray tracer drastically decreases as we linearly increase the PoE. Moreover, the graph indicates that there is high correlation between the performance of the ray tracer and the size of working set. Image shots generated by tested PoE values can be seen in Fig. 8.*

Runtime performance: The benefit of LODs varies with the reduction in the working set size. For a highly tessellated St. Matthew model with 128M triangles, we achieve more than one order of magnitude reduction in the size of the working set and almost two orders of magnitude improvement in the frame rate. This model has low depth complexity and more than half the primitives are visible from the eye. We show the frame rates obtained during rendering of the St. Matthew model with and without R-LODs and cache-oblivious layouts in Fig. 10. Moreover, we are able to achieve 2.6 frames per second while rendering the model with shadow and reflection with little loss of image quality (see. Fig. 1). For the forest model shown in Fig. 4, we are able to achieve more than five times improvement by using R-LODs.

In the case of the Double Eagle tanker, we get 10%–200% improvement. This model has high depth complexity and is not highly tessellated. As a result, the performance improvement due to LODs is limited. An image of the tanker with shadows is shown in Fig. 2.

7 Analysis and Comparison

In this section, we analyze the performance of our ray tracing algorithm and also compare its performance to prior approaches. We also discuss some limitations of our approach.

7.1 Analysis

We first examine different aspects of our R-LOD representation.

R-LOD overhead: Our algorithm introduces 4 bytes of additional storage for each kd-node. We measure the additional computational overhead of evaluating our LOD metric during traversal by comparing the runtime performance on the Stanford scanned dragon model (870K triangles) of the standard ray tracer using 8 byte sized kd-nodes and of our ray tracer, which uses 12 byte-sized nodes with stored R-LODs. In order to measure the overhead of R-LODs, we set our PoE metric to 0 during LOD tree traversal; consequently, the image quality is the same in both cases. We found that the R-LOD overhead for storage and traversal reduces the performance by 2%–5%, as compared to ray tracing as described in [30].

Performance gains: The use of R-LODs reduces both computational workload and memory requirements. A major benefit of R-LODs is the reduction of the working set size and cache miss ratios of the runtime algorithm. This size decreases almost as a exponential function of the PoE as shown in Fig. 9. As a result, we get fewer L1/L2 cache misses and page faults and our new ray tracing algorithm is more cache coherent.

7.2 Comparison to other approaches

Our algorithm integrates R-LODs with the kd-tree representation for ray tracing. The idea of using an integrated hierarchical representation for traversal, visibility and simplification has been used by other algorithms for interactive rendering. These include the QSplat system [23], which uses a hierarchy of spheres and a screen space PoE metric to stop the tree traversal at a node. However, QSplat is mainly designed for point datasets or dense meshes arising from scanned models. Moreover, our LOD computation and error metric evaluation algorithms are different from QSplat as we take into account primary and secondary rays. The Quick-VDR system [41] uses a two-level multiresolution hierarchy called CHPM for view-dependent simplification and visibility culling of large polygonal models. However, the CHPM representation has a high memory overhead and does not lend itself well to ray tracing.

Several other ray tracing algorithms based on LODs have been proposed. The algorithm that is closest to our approach is the out-of-core ray tracer described in [32]. While we use R-LODs to perform fewer node and triangle intersections, Wald *et al.* use a simplified version only when the data is not in main memory in order to hide the latency incurred by loading data from the disk. This approach works well when the working set is smaller than main memory. Our LOD based algorithm is complimentary to their work and uses a different representation to reduce the size of the working set and perform fewer ray intersections.

Pharr *et al.* [20] describe an algorithm to optimize memory coherence in ray tracing. In their approach, the rays are reordered so that they access the scene data in a coherent manner. Their prime application was accelerating ray tracing for offline rendering. Our LOD based approach is quite complimentary to their algorithm. The LOD-based renderer described by Christensen *et al.*[5] differs from ours in two respects. Firstly, it uses subdivision meshes. Therefore, it is primarily useful for computing appropriate tessellation levels from the coarsest resolution. On the other hand, we compute the R-LODs from the original mesh. Secondly, Christensen *et al.* use ray differentials, which is expensive for real-time ray tracing. In contrast, our LOD metric is very efficient and optimized for interactive rendering.

7.3 Limitations

Our approach has certain limitations. First of all, any LOD-based acceleration technique can result in visual artifacts. We minimize these artifacts by using a low PoE bound and combining the projected screen-space error and surface deviation error of an R-LOD. If we use a high PoE bound, the R-LODs may result in holes on the simplified representation. This visual artifact can be removed by employing implicit surfaces[33, 14] as a LOD and thereby sacrificing some of the efficiency of our LOD representation. Moreover, our current R-LOD representation is a drastic simplification of the underlying geometric primitives and their material properties. As a result, the R-LOD representation may not provide high quality simplification for surfaces that have highly varying BRDF. One possibility is to use a more complex reflectance representation [18] in such cases. Also, our LOD metric does not give guarantees on the errors in the path traced by the secondary rays and the illumination computed at each pixel. However, we indirectly reduce the differences by reducing errors associated with the R-LODs. Finally, our efficient projection-based LOD error metric can currently handle planar reflections and shadow rays, but not refraction nor non-planar reflection.

8 Conclusion and Future Work

We have presented a novel LOD-based ray tracing algorithm to improve the performance of ray tracing massive models. We use the R-LOD representation as a drastic simplification of geometric primitives contained in the subtree of a kd-node and select the LODs based on our projection-based LOD error metric. We have described a hierarchical R-LOD construction algorithm that has linear time complexity and is well suited for out-of-core computation. The use of R-LODs results in fewer intersection tests and can significantly improve the memory coherence of the ray tracing algorithm. We have observed more than an order of magnitude speedup on massive models, and most of these gains are due to improved memory coherence and fewer cache misses.

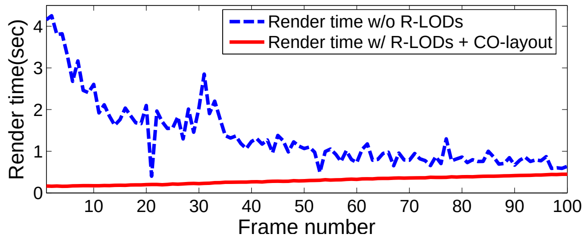


Figure 10. **Frame time with and without R-LODs**: *The graphs shows frame times while rendering the* 128*M St. Matthew model with/without R-LODs and cache-oblivious (CO) layout. We measure frame time when we approach the model starting from the viewpoint shown in Fig. 8. The path is also shown in the video.*

There are many avenues for future work. In addition to addressing current limitations of our approaches, we would like to extend our current R-LOD representation to support smooth implicit surfaces to improve the rendering quality, and still have a compact representation. Moreover, we would like to extend our approach to handle other kinds of input model types such as point clouds [24] and higher order primitives. It might be useful to integrate approximate visibility criteria within our efficient LOD metric to further improve the performance ray tracing on massive models with high depth complexity. Also, we would like to consider visibility issues during construction of R-LODs in order to have better visual quality. Furthermore, we are interested in evaluating our ray tracer on other complex datasets and measuring the performance benefit. LODs could also be potentially useful in the context of designing future hardware for interactive ray tracing.

Acknowledgments

We would like to thank Matt Pharr, Ingo Wald, Eric Haines, and David Kasik for providing helpful feedback on an earlier draft of this paper. The St. Matthew statue are courtesy of the Digital Michelangelo Project at Stanford University. the Double Eagle tanker is courtesy of Newport News Shipbuilding. Simplified St. Matthew models are computed by a streaming simplification tool provided by Peter Lindstrom. This work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, ONR Contract N00014-01-1-0496, DARPA/RDECOM Contract N61339-04-C-0043 and Intel. Some of the work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory.

References

[1] M. Agrawala, R. Ramamoorthi, and A. Moll. Efficient image-based methods for rendering soft shadows. In *ACM SIGGRAPH*, pages 375–384, 2000.

[2] J. Amanatides. Ray tracing with cones. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 129–135, July 1984.

[3] A. Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968.

[4] Y.-J. Chiang, J. El-Sana, P. Lindstrom, R. Pajarola, and C. T. Silva. Out-of-core algorithms for scientifi c visualization and computer graphics. *IEEE Visualization 2003 Course Notes*, 2003.

[5] P. H. Christensen, D. M. Laur, J. Fong, W. L. Wooten, and D. Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum*, 22(3):543–552, Sept. 2003.

[6] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive tetrapuzzles: effi cient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph.*, 23(3):796–803, 2004.

[7] D. E. DeMarle, C. P. Gribble, and S. G. Parker. Memory-savvy distributed interactive ray tracing. In *EGPGV*, pages 93–100, 2004.

[8] A. Dietrich, I. Wald, and P. Slusallek. Large-Scale CAD Model Visualization on a Scalable Shared-Memory Architecture. In *Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization (VMV) 2005*, pages 303–310, 2005.

[9] E. Gobbetti and F. Marton. Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *ACM Trans. Graph.*, 24(3):878–885, 2005.

[10] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[11] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 119–127, New York, NY, USA, 1984. ACM Press.

[12] H. Igehy. Tracing ray differentials. In *ACM SIGGRAPH*, pages 179–186, 1999.

[13] I. Jolliffe. Principle component analysis. In *Springer-Veriag*, 1986.

[14] D. Levin. Mesh-independent surface interpolation. In *Geometric Modeling for Scientifi c Visualization*, pages 37–49, 2003.

[15] D. Lischinski and A. Rappoport. Image-based rendering for non-diffuse synthetic scenes. In *Eurographics Rendering Workshop 98*, pages 301–314, 1998.

[16] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, 2002.

[17] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 1990.

[18] F. Neyret. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 1998.

[19] S. Parker, W. Martin, P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. *Symposium on Interactive 3D Graphics*, 1999.

[20] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proc. of ACM SIGGRAPH*, pages 101–108, 1997.

[21] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. on Graphics (Proc. of SIGGRAPH'02)*, 21(3):703–712, 2002.

[22] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005.

[23] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. *Proc. of ACM SIGGRAPH*, pages 343–352, 2000.

[24] G. Schaufler and H. W. Jensen. Ray tracing point sampled geometry. In *Rendering Techniques*, pages 319–328, 2000.

[25] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek. Realtime ray tracing of dynamic scenes on an FPGA chip. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 95–106, New York, NY, USA, 2004. ACM Press.

[26] M. Shinya, T. Takahashi, and S. Naito. Principles and applications of pencil tracing. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 45–54, July 1987.

[27] P. Shirley, P. Slusallek, B. Mark, G. Stoll, and I. Wald. Introduction to real-time ray tracing. *SIGGRAPH Course Notes*, 2005.

[28] G. Stoll, W. R. Mark, P. Djeu, R. Wang, and I. Elhassan. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Technical Report TR-06-21, Dept of Computer Sciences, University of Texas, 2006.

[29] K. Sung and P. Shirley. Ray tracing with the BSP tree. *Graphics Gems III*, pages 271–274, 1992.

[30] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[31] I. Wald, C. Benthin, M. Wagner, and P. Slusallek. Interactive rendering with coherent ray tracing. In A. Chalmers and T.-M. Rhyne, editors, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)*, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001.

[32] I. Wald, A. Dietrich, and P. Slusallek. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proceedings of the Eurographics Symposium on Rendering*, 2004.

[33] I. Wald and H.-P. Seidel. Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics*, 2005.

[34] I. Wald, P. Slusallek, and C. Benthin. Interactive distributed ray tracing of highly complex models. In S.J.Gortler and K.Myszkowski, editors, *Rendering Techniques 2001 (Proceedings of the 12th EUROGRAPHICS Workshop on Rendering*, pages 277–288. Springer, 2001.

[35] M. Wand and W. Straßer. Multi-resolution point-sampled raytracing. In *Graphics Interface*, 2003.

[36] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

[37] C. Wiley, I. A.T. Campbell, S. Szygenda, D. Fussell, and F. Hudson. Multiresolution BSP trees applied to terrain, transparency, and general objects. In W. Davis, M. Mantei, and V. Klassen, editors, *Graphics Interface*, pages 88–96, May 1997.

[38] S. Woop, J. Schmittler, and P. Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.

[39] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-Oblivious Mesh Layouts. *Proc. of ACM SIGGRAPH*, 2005.

[40] S.-E. Yoon and D. Manocha. Cache-Effi cient Layouts of Bounding Volume Hierarchies. In *Computer Graphics Forum (Eurographics)*, 2006. Conditionally accepted.

[41] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-VDR: Interactive View-dependent Rendering of Massive Models. *IEEE Visualization*, pages 131–138, 2004.

RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs

Christian Lauterbach[1] Sung-Eui Yoon[2] David Tuft[1] Dinesh Manocha[1]

[1] University of North Carolina at Chapel Hill [2] Lawrence Livermore National Laboratory
http://gamma.cs.unc.edu/BVH

UNC Tech. Report: T06-010
March 2006

Abstract
We present an efficient approach for interactive ray tracing of deformable or animated models. Unlike many of the recent approaches for ray tracing static scenes, we use bounding volume hierarchies (BVHs) instead of kd-trees as the underlying acceleration structure. Our algorithm makes no assumptions about the simulation or the motion of objects in the scene and dynamically updates or recomputes the BVHs. We also describe a method to detect BVH quality degradation during the simulation in order to determine when the hierarchy needs to be rebuilt. Furthermore, we show that the ray coherence techniques introduced for kd-trees can be naturally extended to BVHs and yield similar improvements. Our algorithm has been applied to different scenarios composed of tens of thousands to a million triangles arising in animation and simulation. In practice, our algorithm can ray trace these models at 4-20 frames a second on a dual-core Xeon PC.

1. Introduction

Ray tracing is a classic problem in computer graphics and has been studied in the literature for more than three decades. Most of the earlier ray tracing algorithms were used to generate high quality images for offline rendering. Over the last few years, there has been renewed interest in real-time ray tracing. At a broad level, most of the work in real-time ray tracing algorithms can be classified into three main categories: improved techniques to compute acceleration structures, exploiting ray coherence, and parallel algorithms on shared memory or distributed memory systems.

Most current interactive ray tracing algorithms use kd-trees as an acceleration data structure [RSH05, Wal04]. In practice, kd-trees are simple to implement, can be stored in a compact manner, and are used for efficient tree traversal during ray intersections. However, one of the the main disadvantages of kd-trees is the high construction time; current algorithms can take many seconds even on models composed of tens of thousands of triangles [Hav00, WH06]. Furthermore, no simple and fast algorithms are known for incrementally updating the kd-tree hierarchy, even when the primitives undergo a simple deformation. As a result, current algorithms for interactive ray tracing are mainly limited to static scenes.

Main results: In this paper, we present a simple and efficient algorithm for interactive ray tracing of dynamic scenes. We analyze many issues with respect to computation and incremental updates of hierarchies. Our algorithm uses bounding volume hierarchies (BVHs) of axis-aligned bounding boxes (AABBs), for which we describe efficient techniques to recompute or update these hierarchies during each frame. In practice, rebuilding of BVHs can be expensive, so we minimize these computations by measuring BVH quality degradation between successive frames. We also apply the ray coherence techniques developed for kd-trees to BVHs and obtain similar speedups. Finally, we describe techniques to parallelize these computations on multi-core architectures and improve the cache efficiency of the resulting algorithms. We have implemented our algorithm and highlight its performance on several dynamic scenes. Our system can render these datasets with secondary and shadow rays at $4-20$ frames per second on a dual-core 2.8GHz Xeon PC with 2GB of memory.

Overall, our approach offers the following advantages:

1. **Simplicity:** Our algorithm is very simple and easy to implement.
2. **Interactivity:** We are able to handle dynamic scenes with up to a million triangles at interactive rates on current desktop PCs.
3. **Generality:** Our algorithms make no assumptions about the motion of the objects or the underlying simulation or animation.

The rest of the paper is organized in the following manner: We give a brief overview of previous methods in Section 2. We present our BVH hierarchy computation algorithm and

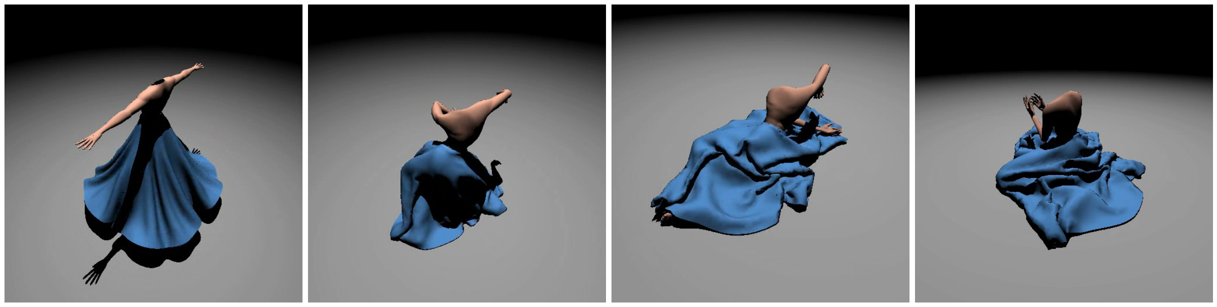


Figure 1: ***Princess Model**: Four different images of a* 220 *step sequence from a dynamic cloth simulation consisting of* 40*K triangles. By computing and updating the AABB hierarchy of the deforming model, we are able to achieve* 16 *frames per second on dual Xeon processors.*

evaluate its features with other approaches in Section 3. Section 4 describes our ray tracing algorithm for dynamic scenes based on BVHs and addresses the issue of utilizing multi-core architectures. Finally, we show the results obtained by our approach on several benchmarks in section 5.

2. Previous Work

In this section, we give a brief overview of prior work in interactive ray tracing and dynamic scenes.

Interactive ray tracing: Since its early introduction in [App68, Whi80], the ray tracing algorithm has been very well studied in computer graphics due to its generality and high rendering quality. Recently, several systems have been presented that are capable of generating ray traced images at interactive speeds. A recent survey is given in [SSM*05]. Parker *et al.* [PMS*99] present a real-time ray tracing algorithm on a shared-memory supercomputer. Several approaches use ray coherence to improve performance and achieve interactive performance on commodity desktop systems for large static datasets, such as coherent rat tracing [WBWS01, Wal04]. More recently, MLRT [RSH05] integrates kd-tree traversal with beam tracing to further improve performance.

Dynamic Scenes: There is relatively less work on ray tracing dynamic scenes. Reinhard et al. [RSH00] use a grid structure that can be updated efficiently for any type of animation. Lext *et al.* [LAAM01] present a general purpose framework and benchmarks for ray tracing animated scenes. They also propose an algorithm that uses oriented bounding boxes along with regular grids [LAM01b]. Wald et al. [WBS03] describe a distributed system for dynamic scenes that differentiates between transformations and unstructured movement in the scene. Recently, Ingo *et al.* [WIK*06] proposed coherent grid traversal algorithm to handle dynamic models.

Bounding volume hierarchies: BVHs have been widely used to accelerate the performance of ray tracing algorithms [RW80, Smi98]. In the case of static scenes, algorithms based on kd-trees and nested grids seem to outperform BVH-based algorithms [Hav00]. Larsson and Akenine-Möller [LAM01a] present a lazy evaluation and hybrid update method to efficiently update BVHs in collision detection. They also use the algorithm to ray trace models composed of tens of thousands of polygons [LAM03]. BVHs have also been used to accelerate the performance of collision detection algorithms for deformable models [vdB97, TKH*05].

3. BVHs for dynamic scenes

In this section, we analyze the problem of ray tracing using BVHs. We show that BVHs can offer better performance than kd-trees on dynamic environments and present optimizations to speed up rendering.

3.1. Choice of Hierarchies

A BVH is a tree of bounding volumes. Each inner node of the tree corresponds to a bounding volume (BV) containing its children and each leaf node consists of one or more primitives. Common choices for BVs include spheres, AABBs, oriented bounding boxes (OBBs) or k-DOPs (discretely oriented polytopes). Many efficient algorithms have been proposed to compute sphere-trees [Hub93], OBB-trees [GLM96], and k-DOP-trees [KHM*98]. However, we use AABBs as the BV as they provide a good balance between the tightness of fit and computation cost. We also use efficient algorithms for ray-box intersection [SM03, WBMS05].

3.2. AABB hierarchies vs. kd-trees

In this section, we evaluate some features of BVHs based on AABBs and compare them with kd-trees for ray tracing.

Recently, many efficient and optimized ray tracing systems have been proposed based on kd-trees [Wal04]. As far as static scenes are concerned, analysis has shown that optimized algorithms based on kd-trees will outperform BVH-based algorithms [Hav00]. There are multiple reasons to explain this behavior: First, even the most optimized ray-AABB intersection test (e.g. from [WBMS05]) is more expensive than split plane intersection for kd-trees. This is due to the fact that in the worst case (i.e. no early rejection) up to 6 ray-plane intersections need to be computed for AABB trees, as opposed to just one for a kd-tree node. Another important aspect is that a BVH does not provide real front-to-back ordering during traversal. As a result, when if a primitive intersects the ray, the algorithm cannot terminate (as is the case for a kd-tree), but needs to continue the traversal to find all other intersections. Furthermore, kd-tree nodes can be stored more efficiently (8 bytes per node [WDS04]) than an AABB possibly could. On the other hand, we found that BVHs often need fewer nodes overall to represent the scene as compared to a kd-tree (please see Table 1). This is mainly due to the fact that primitives are referenced only once in the hierarchy, whereas kd-trees usually have multiple references because no better split plane could be found. In addition, AABBs have the advantage of providing a tighter fit to the geometric primitives with fewer levels in the tree, e.g. kd-trees need multiple subdivisions in order to discard empty space. Most importantly, the major benefit of BVHs is that the trees can be easily updated in linear time using incremental techniques. No similar algorithms are known for updating kd-trees.

3.3. BVH Construction

We construct an AABB hierarchy in a top-down manner by recursively dividing an input set of primitive into two subsets until each subset has the predetermined number of primitives. We have found that subdividing until each leaf just contains one primitive yields the best results at the cost of a deeper hierarchy, as – similar to kd-trees – node intersection is comparably cheaper to primitive intersection. During hierarchy construction, the most important operation is to find a divider for the two subsets that will optimize the performance of runtime ray hierarchy traversal. One of the best known heuristics for tree construction for ray tracing is the *surface-area heuristic* (SAH) [GS87], which has been shown to yield higher ray tracing performance. However, it also has a much higher construction cost, which can take a significant fraction of a frame time for dynamic environments. Because of this, we use the midpoint of one of the dimensions and sort the primitives into the child nodes depending on their location with respect to the midpoint. We observe that the midpoint heuristic provides good rendering performance and is very fast to compute. Note that even though we just split along one dimension, the bounding box will still be tight along all the three dimensions. As this method will often distribute a similar number of primitives to both children, the resulting tree will likely be nearly balanced. As we are storing just one primitive per leaf, it is also easy to see that the total number of nodes in the tree for n primitives will always be $2n-1$, which allows us to allocate the space needed for any subtree during construction.

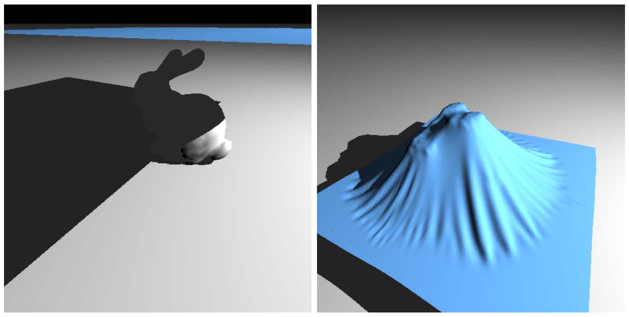


Figure 2: ***Cloth on Bunny Simulation:*** *Two shots of a* 315 *step dynamic simulation of cloth dropping on the Stanford bunny. We achieve* 19 *frames per second on average during ray tracing of this deforming model.*

Regardless of the heuristic for finding a split, the time complexity, $T(n)$, of the recursive AABB hierarchy construction algorithm, given an input model consisting of n primitives, satisfies $T(n) = kT(\frac{n}{k}) + O(n)$ due to its recursive formulation, where k is the number of children of each node. Therefore, the time complexity is $O(n\log_k(n))$.

3.4. Updating the hierarchy

The main advantage of using BVHs for ray tracing is that animated or deforming primitives can be handled by updating the BVs associated with each node in the tree. Our algorithm makes no assumptions about the underlying motion or simulation. In order to efficiently update the hierarchy, we recursively update the BVHs by using a postorder traversal. We initially traverse down to leaves from the root nodes. As we encounter a leaf node, we efficiently compute a new BV that has the tightest fit to the underlying deformed geometry. As we traverse from the leaf node in a bottom-up manner, we initialize the BV of an intermediate node with a BV of the leftmost node and expand it with the BVs of the rest of the sibling nodes.

The time complexity of this approach is $O(n)$, which is lower than the construction method. This is reflected by fast update times (see Table 1), which can be one order of magnitude lower than rebuilding the tree for models with hundreds of thousands of polygons. Therefore, we rely on hierarchy update operations to maintain interactive performance for dynamic environments.

3.5. BVHs for deformable scenes

We initially build an AABB tree of a given scene. As the model deforms or some objects in the scene undergo motion, the BVH needs to be updated or rebuilt. Updating the BVH is to recompute the bounds of each BV node, and rebuilding the BVH is to recompute the entire BVH from scratch and re-clustering the primitives. At runtime, we traverse the BVH to compute the intersections between the rays and the primitives.

If the algorithm only updates the BVH between successive frames, the runtime performance of BVHs can degrade over the animation sequence because the grouping of the primitives and structure of the hierarchy does not change. As a result, the BVs may not provide a tight fit to the underlying geometric primitives. This is often characterized by growing and increasingly overlapping BVs, which subsequently deteriorate the quality of the BVH for fast runtime BVH traversal and by adding more intersections between the ray and AABBs. In such cases, rebuilding the AABB tree or parts of it is desirable.

We found that updating the BVH works well with relatively small changes to the scene or structured movement to groups of primitives. When primitives move independently, however, for example in different directions, changes to the actual tree structure may be necessary to reflect the new positions of the deforming geometry. Still, rebuilding the BVH can be considerably more expensive than updating the BVH. As a result, we want to minimize the number of times rebuilding is performed. Therefore, we need to efficiently decide when updating the BVH is sufficient or rebuilding the BVH is required. This is non-trivial because the actual degradation of a BVH depends on many factors, such as the speed with which primitives move and the general characteristics of the motion of objects in the scene. Simple approaches such as rebuilding the tree every t frames have the disadvantage of not being adaptable to different characteristics over the animation and need to be chosen a priori. Conservatively choosing t means adding a lot of rebuilding overhead, which is especially unwanted in an interactive context. In order to efficiently detect when updating tree or rebuilding tree is required, we use a simple heuristic that is described in the next section.

3.6. Rebuilding criterion

We assume that BVH quality degradation is marked by bounding box growth that is not caused by actual primitive size, but by distribution of primitives or subtrees in the box. For example, consider two primitives moving in opposite directions. The parent node containing them will have to grow to accommodate for the movement, resulting in a bounding box that is relatively large, but mostly empty. Since the probability that a box will be intersected by a ray rises with its surface area, we want to rebuild a subtree to find a more advantageous tree topology. To find these cases and prevent them from impacting performance, we need to measure BVH degradation during each frame by using a simple and inexpensive heuristic.

Our heuristic is based on the idea that we can find nodes that are large relative to their children by comparing their surface area. In order to have a relative metric independent of scale, we measure the ratio of each parent node's surface area to the sum of the area of its two children. The larger the ratio becomes, the more imbalance exists in the sizes. We first compute the ratio during tree construction and store it in a field of the optimized AABB data structure (see next section). Whenever the tree is updated, the changed surface areas are automatically computed as and each inner node can easily calculate its new ratio. Since we assume that the ratio stored from the construction is as good as we can do, we find the difference between the new and old ratio and add them to a global accumulation value. Once the bottom-up update reaches the root, we have computed the sum of all the differences. To assure that this value can be tested independently of the tree size, we normalize it by dividing by the number of nodes that contribute to the sum, i.e. the sum of inner nodes, which is always $n-1$. This yields a relative value describing the overhead incurred by updating the BVH instead of rebuilding it. This value is then simply compared to a predefined threshold value and the tree is rebuilt if the threshold is exceeded.

This approach has several advantages: it will detect a good time to rebuild regardless of the actual frame rate and without any scene-specific settings. Furthermore, in scenes where there is little to no degradation, the heuristic will never need to initiate a rebuild. It is also possible to use the method to just rebuild subtrees, but we found that this cannot fully replace a complete rebuild since degradations in the upper levels of the hierarchy typically have the highest impact on the performance of ray tracing.

4. Ray Tracing with BVHs

In this section we describe our runtime BVH traversal algorithm. Also, we present techniques to extend the algorithm to multi-core architectures.

4.1. Traversal and Intersection with BVHs

We use a simple algorithm to compute the intersection of a ray and the scene primitives using the BVH. The ray is checked for intersections with the children of the current node starting at the root of the tree. If it intersects the child BV, the algorithm is applied recursively to that child, otherwise that child is discarded. Whenever a leaf node is reached, the ray is intersected with the primitives contained in that node. For most rays, the goal is to find the first hit point on the ray, so even if a ray-primitive intersection is found, the

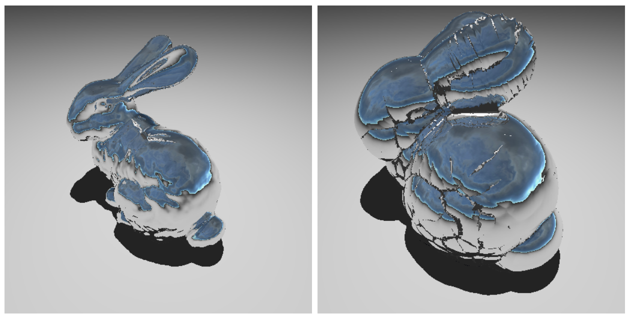


Figure 3: ***Bunny blowing up :*** *Two images show frames from a* 113 *step animation of a deforming Stanford bunny. We achieve* 8 *frames per second on average during ray tracing this deforming model with shadow and reflection rays.*

algorithm has to search the other sub-trees for potential intersections. An exception to this are shadow rays, where (at least for directional or point lights) any single hit is considered sufficient and traversal can stop.

BVH traversal optimizations: Experience with kd-trees has shown that front-to-back ordering is a major advantage for ray tracing. Although BVHs do not provide a strict ordering, we found that storing the axis of maximum distance between children for each AABB and using that information during traversal together with the ray direction to determine a 'near' and 'far' child improves the traversal speed, especially for scenes with a high depth complexity. Another issue is cache coherence during traversal: similar to the compact kd-tree representations [WDS04], we can optimize the AABB representation to fit within 32 bytes of data, which also includes the information that is needed to rebuild the tree. Our profiling shows that BVH traversal using our AABBs has the same cache efficiency as the kd-tree traversal.

Use of ray coherence techniques: One of the main techniques used in current real-time ray tracers is to exploit ray coherence to reduce the number of traversal steps and primitive intersections per ray. Those algorithms were originally designed for the kd-tree acceleration structure. It is relatively straightforward to extend them to work with BVHs as well. In order to use coherent ray tracing [WBWS01] the BVH traversal has to be changed so that a node is traversed if *any* of the rays in the packet hits it and skipped if *all* of the rays miss it. A hit mask is maintained throughout the traversal to keep track of which rays have already hit an object and their distance. However, the traversal does no longer require that the rays have the same direction signs because unlike kd-trees the traversal order does not determine the correctness for a BVH. We have implemented ray packet traversal for 2x2 ray bundles and found that it yields a speedup of about 2 to 3, which is even above the improvement obtained for kd-trees. Adapting the MLRT algorithm [RSH05] to BVHs is also straightforward.

4.2. Multi-Core Architectures

One of major features of current computing trends is that there are multi-cores and hyper-threading functionality available on commodity architectures. Therefore, it is desirable to design our hierarchy construction, update, and runtime traversal such that they take advantage of available parallelism.

Hierarchy construction: There are no good and optimal algorithms that can easily parallelize hierarchy construction. Since ray tracing scales well with multiple processors, it is desirable to speed up construction by distributing the work over several threads and cores. To achieve this, we first divide a set of triangles and vertices up to four sets by using one thread. Then, we assign each thread to construct a sub-BVH for each divided set. In general, this may not achieve high load balancing. However, we found that this simple method works well with our benchmarks since BVHs of our benchmarks are well balanced.

Update: Our update method takes advantage of multi-core processors by using a bottom-up update method. Given the number of available threads, *n*, we decompose an input BVH into *n* sub-BVHs. For this, we simply compute *n* different children by traversing the tree from the root in the breadth-first manner. Then, each thread performs a bottom-up update from one of the computed nodes in parallel. After all the threads are done, we then sequentially update the upper portion of the *n* nodes. We particularly choose the bottom-up approach since it is well suited to parallel processing. For example, we do not require any expensive synchronization for each thread since data that are accessed by threads are mutually exclusive to each other. Table 1 shows the timings for our results. Since our current BVHs are relatively well balanced, this simple scheme provides reasonably good load balancing in practice.

Runtime traversal: We employ image-space partitioning to allocate coherent regions to each thread. Also, in order to achieve reasonably good load balancing, we first decompose image-space into small tiles (e.g., 16×16) and, then, allocate each tile to each thread. After a thread finishes its computation, it continues to process another tile. We found that this approach works well with our benchmarks.

5. Implementation and Results

In this section, we describe our implementation and highlight the results of our ray tracer on different benchmarks.

5.1. Implementation

We have implemented our interactive ray tracer for deformable models using BVHs in a dual Intel Xeon machine at 2.8 GHz. To compare the performance of BVHs with previous interactive ray tracing work for rendering static scenes, we also implemented kd-tree rendering(without animation capability). Both acceleration structures support ray packet traversal using the SSE SIMD instruction set on Intel processors. For efficiency reasons, we only support triangles as primitives. To speed up rendering, we employ multithreaded rendering and hierarchy updates using OpenMP.

5.2. Results

We have tested our system on four animated scenes of varying complexity as well as one more complex static model to measure performance of our approach (see Table 1). In general, building a BVH tree using the naive midpoint method is much faster than the optimized surface-area heuristic kd-tree construction. In most cases, both structures have a similar memory footprint, but kd-trees need more nodes because primitives can be located in multiple nodes.

Benchmarks: We show five different test cases (Refer Table 1): Scene 1 (shown in Fig. 2) and Scene 3 (shown in Fig. 1) in the respective rows of the table demonstrate performance on a typical animation including simulated cloth at different complexity, both rendered including shadow rays. Even though most of the mesh is moving, BVH updates turn out to be sufficient to maintain the quality of the structure. Scene 4 (shown in Fig. 3) applies a non-rigid deformation to the Stanford bunny model with reflection and shadow rays. To maintain BVH quality, some parts of the tree have to be rebuilt. Scene 2 (shown in Fig. 4) is a part of the BART animated ray tracing benchmark [LAAM01] and shows a set of triangles with mostly unstructured, random movement. Since it has high depth complexity and overlapping primitives, this scene is one of the worst cases for BVH rendering as well as hierarchy update. For the former, we have found that the ordering approach for BVHs ameliorates the effects of depth complexity. Additionally, the independent movement of each triangle leads to extreme degradation in BVH quality, so that our heuristic rebuilds parts of the tree quite often. Finally, we demonstrate a more complex static scene of 1M Buddha (Scene 5) to show that BVH ray tracing can compete with kd-trees even for larger models. Unfortunately, the update time grows linearly with model size, so a more efficient update scheme would be needed to be able to render this or any larger model at high frame rates.

We tested our heuristic for tree rebuilding on the test models and found that in all cases except the BART model, just hierarchy updates can be efficient enough for rendering. The unstructured, random movement of triangles in the BART scene makes several tree rebuilds necessary, however. Without doing that, we found that frame rates will decrease by over an order of magnitude in just a few frames. To test how well the rebuild times are chosen, we benchmarked the animation while rebuilding only via heuristic (with the threshold set to 0.4) as well as rebuilding the hierarchy every frame. We found that even when looking just at pure rendering time without counting rebuilding and updating, the animation rendered with new hierarchy in each frame was only 20% faster than rendering using our heuristic. The latter needed only a few rebuilds, so the total overhead incurred by updates and rebuilds was only 2s over the whole sequence, as compared to 15s for rebuilding.

6. Future Work and Conclusion

We have proposed an algorithm for interactive ray tracing of deformable, animated models. We used BVH hierarchies as an acceleration data structure of the deformable models and showed optimizations that will result in performance competitive or even exceeding rendering using kd-trees. We were also able to integrate efficient ray coherence techniques for kd-trees to our BVHs. We do not make any assumptions about the possible deformation or motion of objects and dynamically update or rebuild the hierarchy depending on our simple heuristic.

There are many interesting directions for future work. Our current algorithm is mainly designed for small to intermediate model complexity. We would like to extend our algorithm to handle larger deforming models, which would require more efficient or localized update methods. Another interesting problem is the better use of multiprocessor architectures in the context of hierarchy construction and updates. We plan to extend our current methods to be more general and flexible for these applications. We found that there is concurrent work on ray tracing of dynamic models based BVHs [WBS06] with ours. We would like to perform a detailed comparison of our algorithm with theirs.

References

[App68] APPEL A.: Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.* (1968), vol. 32, pp. 37–45.

[GLM96] GOTTSCHALK S., LIN M., MANOCHA D.: OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96* (1996), 171–180.

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7*, 5 (1987), 14–20.

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[Hub93] HUBBARD P. M.: Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality* (October 1993).

Scene	Triangles	**BVH:**nodes	memory	build time	update time	fps
1) Cloth on bunny	16K	31923	997 KB	98 ms	4ms	19
2) BART model	16K	32767	1024 KB	96 ms	5ms	12
3) Cloth model	40K	80059	2501 KB	224 ms	8ms	16
4) Bunny	69K	138901	4340 KB	395 ms	11ms	8
5) Buddha	1M	2175431	67982 KB	7593 ms	167ms	4

Scene	Triangles	**kd-tree:**nodes	memory	build time
1) Cloth on bunny	16K	64137	859 KB	1487ms
2) BART model	16K	11075	1426 KB	1902ms
3) Cloth model	40K	218845	2778 KB	5s
4) Bunny	69K	442347	5072 KB	10s
5) Buddha	1M	2989439	33225 KB	80s

Table 1: ***Benchmarks and Timings:*** *Results for BVH ray tracing of several scenes. The benchmark configuration for each of the scenes is described in section 5. The top table shows the performance for a BVH. The bottom table shows the tree computation time and memory overhead for a kd-tree of the same model (for comparison). All benchmarks were performed at* 512^2 *resolution on a dual Xeon machine at 2.8 GHz using 2x2 ray packet traversal. Both hierarchies were built using a single thread only.*

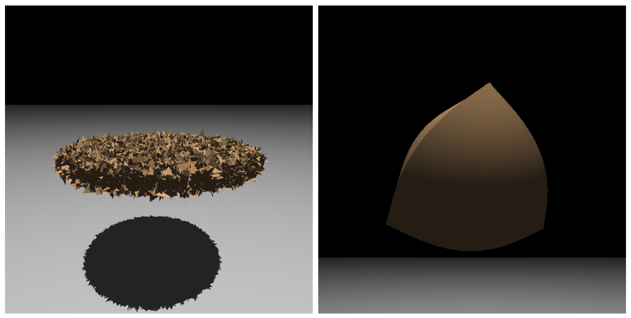


Figure 4: ***BART Randomly Moving Triangles:*** *Two image shots from* 170 *steps of a randomly deforming model from the BART deforming data benchmark. We are able to achieve* 12 *frames per second on average during ray tracing this model with shadow rays.*

[KHM*98] KLOSOWSKI J., HELD M., MITCHELL J., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Trans. on Visualization and Computer Graphics 4*, 1 (1998), 21–37.

[LAAM01] LEXT J., ASSARSSON U., AKENINE-MÖLLER T.: A benchmark for animated ray tracing. In *IEEE Computer Graphics and Applications* (2001).

[LAM01a] LARSSON T., AKENINE-MÖLLER T.: Collision detection for continuously deforming bodies. In *Eurographics* (2001), pp. 325–333.

[LAM01b] LEXT J., AKENINE-MÖLLER T.: Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001, short presentation* (2001).

[LAM03] LARSSON T., AKENINE-MÖLLER T.: *Strategies for Bounding Volume Hierarchy Updates for Ray Tracing of Deformable Models*. Tech. rep., 2003.

[PMS*99] PARKER S. G., MARTIN W., SLOAN P.-P. J., SHIRLEY P., SMITS B. E., HANSEN C. D.: Interactive ray tracing. In *SI3D* (1999), pp. 119–126.

[RSH00] REINHARD E., SMITS B., HANSEN C.: Dynamic acceleration structures for interactive ray tracing. In *Proceedings Eurographics Workshop on Rendering* (June 2000), pp. 299–306.

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM Trans. Graph. 24*, 3 (2005), 1176–1185.

[RW80] RUBIN S. M., WHITTED T.: A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics 14*, 3 (July 1980), 110–116.

[SM03] SHIRELY P., MORLEY R. K.: *Realistic Ray Tracing*, second ed. AK Peters Limited, 2003.

[Smi98] SMITS B.: Efficiency issues for ray tracing. *Journal of Graphics Tools: JGT 3*, 2 (1998), 1–14.

[SSM*05] SHIRLEY P., SLUSALLEK P., MARK B., STOLL G., WALD I.: Introduction to real-time ray tracing. *SIGGRAPH Course Notes* (2005).

[TKH*05] TESCHNER M., KIMMERLE S., HEIDELBERGER B., ZACHMANN G., RAGHUPATHI L., FUHRMANN A., CANI M.-P., FAURE F., MAGNENAT-THALMANN N., STRASSER W., VOLINO P.: Collision detection for deformable objects. *Computer Graphics Forum 19*, 1 (2005), 61–81.

[vdB97] VAN DEN BERGEN G.: Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools 2*, 4 (1997), 1–14.

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[WBMS05] WILLIAMS A., BARRUS S., MORLEY R. K., SHIRLEY P.: An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools: JGT 10*, 1 (2005), 49–54.

[WBS03] WALD I., BENTHIN C., SLUSALLEK P.: Distributed Interactive Ray Tracing of Dynamic Scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)* (2003).

[WBS06] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *Technical Report, SCI Institute, University of Utah, No UUSCI-2005-014 (conditionally accepted at ACM Transactions on Graphics)* (2006).

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001)* (2001), Chalmers A., Rhyne T.-M., (Eds.), vol. 20, Blackwell Publishers, Oxford, pp. 153–164.

[WDS04] WALD I., DIETRICH A., SLUSALLEK P.: An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proceedings of the Eurographics Symposium on Rendering* (2004). (to appear).

[WH06] WALD I., HAVRAN V.: *On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)*. SCI Institute Technical Report UUSCI-2006-009, University of Utah, 2006.

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Commun. ACM 23*, 6 (1980), 343–349.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *Technical Report, SCI Institute, University of Utah, No UUSCI-2005-014 (conditionally accepted at ACM SIGGRAPH 2006)* (2006).