Production Volume Rendering

Fundamentals

SIGGRAPH 2011 Course Notes

Course Organizers

Magnus Wrenninge[1]
Sony Pictures Imageworks

Nafees Bin Zafar[2]
DreamWorks Animation

Updated: 8 okt 2011

1 magnus.wrenninge@gmail.com
2 nafees@nafees.net

Course Description

Computer generated volumetric elements such as clouds, fire, and whitewater, are becoming commonplace in movie production. The goal of this course is to familiarize attendees with the technology behind these effects. The presenters in this course have experience with and have authored proprietary volumetrics systems.

The course begins with a quick introduction to generating and rendering volumes. We then present some of the most fundamental parts of a production usable volumetrics toolkit, focusing on the feature set and why those features are desirable. Specific focus will be given to the approaches taken in tackling efficient data structures, shading approaches, multithreading/parallelization, holdouts, and motion blurring.

Level of difficulty: Intermediate

Intended Audience

This course is intended for artists looking for a deeper understanding of the technology, developers interested in creating volumetrics systems, and researchers looking to understand how volume rendering is used in the visual effects industry.

Prerequisites

Some background in computer graphics, and undergraduate linear algebra.

On the web

http://magnuswrenninge.com/productionvolumerendering

About the presenters

Nafees Bin Zafar is a Senior Production Engineer in the Effects R&D group at DreamWorks Animation where he works on simulation and rendering problems. Previously he was a Senior Software Engineer at Digital Domain for nine years where he authored distributed systems, image processing, volume rendering, and fluid dynamics software. He received a BS in computer science from the College of Charleston. In 2007 he received a Scientific and Engineering Academy Award for his work on fluid simulation tools.

Magnus Wrenninge is a Senior Technical Director at Sony Pictures Imageworks. He started his career in computer graphics as an R&D engineer at Digital Domain where he worked on fluid simulation and terrain rendering software. He is the original author of Imageworks' proprietary volumetrics system Svea and the open source Field3D library, and is also involved with fluid simulation R&D. He has worked as an Effects TD on films such as *Spiderman 3*, *Alice In Wonderland* and *Green Lantern*, and is currently Effects Animation Lead on *Oz: The Great and Powerful*. He holds an M.Sc. in Media Technology from Linköping University.

Presentation schedule

10.45 – 11.00 **Introduction**
11.00 – 11.40 **Basics of volume modeling**
11.40 – 12.10 **Basics of volume rendering**
12.10 – 12.15 **Q&A**

Table of contents

1. Introduction

If you google "volume rendering" or search for it at your favorite book store web site, you will find that most available literature and research regards volume rendering in medical and data visualization contexts. A smaller portion deals with photorealistic rendering of light scattering in participating media, but precious few texts are available that describe how volume rendering is used in practice to create visual effects.

The aim of this course is to give an introduction to volume rendering in visual effects production. Production volume rendering is a fairly isolated subset of volume rendering, and there is little overlap between it and the other volume rendering contexts. We aim to cover only techniques actively used in visual effects production, and while this excludes much of current research into rendering of participating media, we want to highlight the techniques with the most practical applications. We further limit the scope of this course to rendering of true 3D volumes, excluding topics such as sprite- and slice-based methods.

Our goal is to provide enough details about how high-end production volume rendering is accomplished that participants could set about writing their own basic rendering software. Part of the course's purpose is also to discuss the limitations of the techniques used.

1.1. What's new?

The main difference between this and last year's course is that the *Fundamentals* section (which you are currently reading) is separate from the *Systems* section, which is now its own course with separate course notes[3].

Most of the code examples provided in these course notes are also different from last year. They are now excerpts of the upcoming open source renderer *PVR*, which will be released late 2011 in conjunction with the book *Production Volume Rendering – Design and Implementation.*

[3] http://magnuswrenninge.com/productionvolumerendering

1.2. An informal history of volumetric effects

One of the most memorable volumetric effects in cinema history is the "cloud tank" effect from *Close Encounters of the Third Kind*. Developed by Scott Squires, this technique called for filling a tank partially with salt water, then carefully layering on lower density fresh water on top. The clouds were created by injecting paint into the top layer, where it would settle against the barrier between the salt water and the fresh water [Squires, 2009]. Beyond just art direction, this particular cloud effect was a character in it's own way. The goals the special effects crew had during *Encounters* are the same goals we have today. We want to control how the volumetrics look, and how they move.

Cloudtank effect used in Independence Day.

Computer graphics got into the mix shortly thereafter with William Reeves' invention of particle systems. He used particle systems to create the Genesis sequence in *Star Trek II: The Wrath of Khan*. The title of the associated SIGGRAPH publication provides an excellent preview into what we are trying to do: *Particle Systems – A Technique for Modeling a Class of Fuzzy Objects* [Reeves, 1983]. This basic methodology is still prevalent today, and very relevant to this course.

With the advent of digital rotoscoping and compositing it became common practice in live action visual effects to film elements in staged shoots and composite them onto the plate. This allowed the creation of very complex photoreal effects, since the elements were real.

For purely digital effects, particle systems remained the popular choice. Their use in production indicated a certain look barrier to particle based volumetrics. Particles work great when they are used to model media which is well approximated by particles. The problem occurs when one tries to model a media which is meant to be continuous. The use of particles in these cases leads to a very discontinuous sampling. Particles could be combined with some low frequency tricks such as sprites to look good in special cases, but not so in the general case. One could simply choose to use more particles, but that

looses the advantage of the sparse sampling, and comes at an exponential increase in computational cost.

In the late 90's, an alternative approach started taking root in the visual effects industry [Kisacikoglu, 1998; Lokovic and Veach, 2000; Kapler, 2002]. This technique treated space as a discretized volume, where the contents of a given small volume of space is stored in a sample. The kinds of data stored are properties such as density, temperature, and velocity. Each volumetric sample is called a voxel, and the entire collection is referred to as a voxel buffer or voxel grid. Modeling operations are performed on the grid by rasterizing shapes, particles, or noise. Most morphological operations common in the image processing paradigm are also applicable to volumes. This familiarity in workflow also helped artists adapt to this voxel based pipeline.

The Nightcrawler's "Bamf" effect from X2.

One of the first systems successfully used in multiple movies and commercials was the *Storm* software developed by Alan Kapler at Digital Domain. The goal behind *Storm* was to provide a modeling and rendering solution which could be operated efficiently by artists at the resolutions required for feature films. It featured a language where artists could create buffers, model volumetric shapes, perform arithmetic and compositing operations, and control rendering. The modeling commands allowed artists to use different geometric shapes and a rich set of noise algorithms to create high quality effects very quickly. The system was implemented as a plugin to the Houdini animation software which also aided in quick adoption by the artists.

The memory requirements of scenes *Storm* needed to render exceeded 25 gigabytes. A stringent requirement even by today's standards, it was impossible when Digital Domain started working on a CG

avalanche effect for Columbia Pictures' 2002 film *xXx*. Storm utilized in-core data compression techniques, and innovated to use of buffers transformed to fit the camera frustum. These buffers, called "frustum buffers", provided high resolution close to the camera, and low resolution but complete spatial coverage far away from the view point [Kapler, 2003]. For his pioneering efforts in the design and development of *Storm*, Alan Kapler received a Technical Achievement Award from the Academy of Motion Picture Arts and Sciences in 2005.

Digital avalanche in xXx. © 2002 Columbia Pictures. All rights reserved.

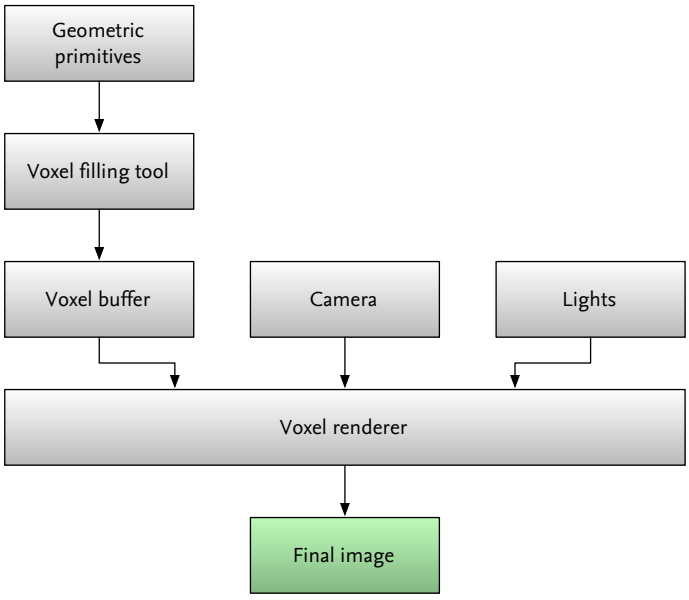
1.3. A simple volumetrics system

A minimal volumetrics system contains three major components. First a data structure for voxel buffers. This means defining both a file format and its in-core representation. A naive implementation is an object which contains a contiguous array, and provides accessor methods to access values with 3D grid indices or positions.

The second component consists of a set of operations which fill the buffer with data. One such operation may simply evaluate noise at each voxel, and store the value. Another operation may take a list of points with radii, and fill the spherical region around the particle with a given value. These modeling commands can involve filtering, distorting, and combining multiple voxel buffers with arithmetic operations. Each operation could be implemented as a separate command line tool, or one tool which processes a sequence of commands and arguments to these operations.

The final component is a renderer to produce an image of the voxel buffer. In addition to the buffer to render, this component also requires specifications for a camera, and lights.

A typical workflow is to model and animate some primitives such as a set of particles, or meshes. Then one creates a voxel buffer around the location of these primitives. The primitives are then rasterized into the voxel buffer. Further volumetric processing operations are performed. Finally the buffer is rendered by the volume rendering component. The next three chapters will expand further upon these components.



A very simple volume modeling and rendering system

2. Volume modeling

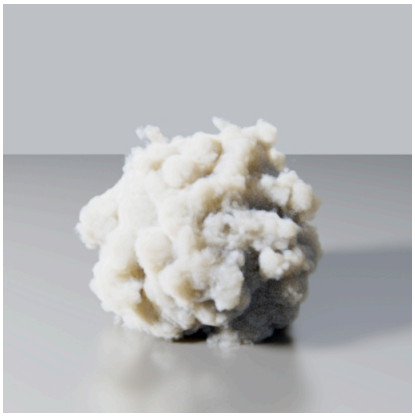
In other forms of volume rendering, such as medical visualization, the data to be rendered is directly available to the system, as in the case of a CT or MRI dataset. When it comes to volume rendering in visual effects, we need to create this data ourselves. The process is called *volume modeling*, and involves turning geometric data into volumetric data, most often in the form of voxel buffers.

A classic example of volume modeling is the use of pyroclastic noise primitives to model billowing smoke, where each primitive is a sphere that can be represented as a position, a radius and various noise parameters. The use of simple geometric primitives combined with noise functions is one of the most fundamental methods of volume modeling.

Volume modeling is an almost endless topic, as there is an infinite number of ways and methods that one can fill a voxel buffer. This chapter will try to describe the basics, from the voxel data structures needed and elementary modeling primitives, to techniques for scaling to high resolution data sets.

Simple sphere

Windowed noise function



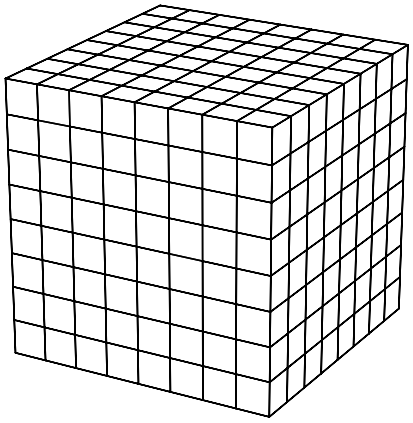
Displacement based on noise

Pyroclastic noise

2.1. Voxel buffers

An ordinary computer image is a two-dimensional orthogonal array that stores either single values (for a grayscale image), or multiple values (for a spectral image, such as RGB). The concept translates directly to three dimensions, where we can imagine a 3D orthogonal array, which stores single or multiple values in each of its cells.

This 3D array goes by many names, such as *voxel grid*, *voxel volume*, *voxel buffer*, etc., and depending on whether it stores scalar- or vector-valued data it is sometimes also referred to as a *scalar field/buffer/grid* or *vector field/buffer/grid*. Throughout these course notes we will refer to the case of discrete voxel arrays used in a program as *voxel buffers*. In the general case of non-voxelized, arbitrary functions in 3-space, we will refer to those as *fields*.



8x8x8 *resolution orthogonal (uniform) voxel grid*

2.1.1. Implementations

There are countless ways to implement voxel buffers. The simplest ones fold the 3D space into a contiguous 1D array, and store the data using `float*` and `malloc()`, or in a `std::vector<float>`. More complex implementations may allocate voxels as-needed, allowing the size and memory use to scale dynamically. Such techniques become important as the resolution of a voxel buffer increases. Densely allocated buffers are manageable up to resolutions of roughly 1000^3. (On very high-memory machines this may stretch to 2000^3 or so.) To reach higher resolutions we need to use different data structures, such as sparsely allocated buffers. We will return to this topic and ways of dealing with it in the section titled *High resolution voxel buffers*.

Though implementing a simple voxel buffer class is straightforward, there are also free, open source libraries. Field3D[4] is one alternative, which has the benefit of being developed and tested in production for volume rendering and fluid simulation. We will use Field3D's data structures in our examples and pseudo-code.

[4] http://github.com/imageworks/Field3D

2.1.2. Voxel indices

Just as with a 2D image, we can access the contents of a voxel by its coordinate. The bottom left corner of the buffer has coordinate [0,0,0] (unless a custom data window is used, see below), and its neighbor in the positive direction along the *x* axis is [1,0,0]. When referring to the index along a given axis, it is common to label the variable *i*, *j* and *k* for the *x*, *y* and *z* axes respectively. In mathematic notation this is often written using subscripts, such that a voxel buffer called *S* has voxels located at $S_{i,j,k}$.

In code, this translates directly to the integer indices given to a voxel buffer class' accessor method, such as:

```
class DenseField
{
  const float& value(int i, int j, int k)
  {
     // ...
  }
  // ...
};

float a = buffer.value(0, 0, 0);
```

2.1.3. Implementation awareness

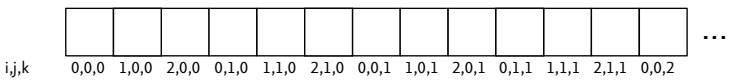
Although it is easy to write code that is agnostic about how voxels are represented in memory, writing efficient code usually means being aware of and taking advantage of the implementation's underlying data structure. For example, a trivial voxel buffer may store its data as a contiguous one-dimensional array, such as

```
std::vector<float> data(xSize * ySize * zSize);
```

Where the mapping of a 3D coordinate to its 1D array index is calculated as

```
int arrayIndexFromCoordinate(int i, int j, int k)
{
  return i + j * xSize + k * xSize * ySize;
}
```

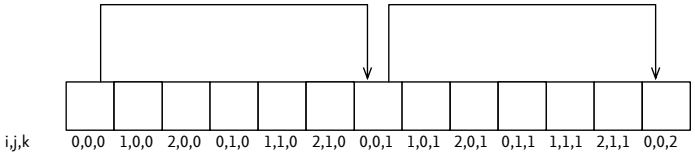
The memory for such a buffer has the following structure:



If we were to loop over all the voxels in the buffer, for example to clear all the values, we might write it as follows:

```
// Naive loop, with x dimension outermost
for (int i = 0; i < xSize; ++i) {
  for (int j = 0; j < ySize; ++j) {
     for (int k = 0; k < zSize; ++k) {
        buffer.lvalue(i, j, k) = 0.0f;
     }
  }
}
```

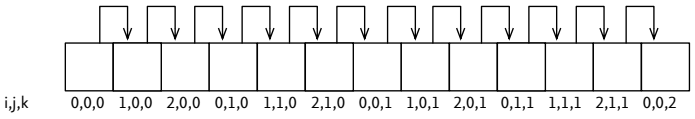
The problem with the code above is that the inner loop steps along the *z* axis, which means the memory access pattern has a stride of `xSize * ySize`. For a buffer of realistic resolution, this will most likely cause a cache miss at each voxel increment and force an entire cache line to be loaded, which cripples performance.



Access pattern for the naive loop

If we instead reorder the loop so that the *x* axis is the innermost, performance is improved since the access pattern is sequential in memory.

```
// Better loop, with x axis innermost
for (int k = 0; k < zSize; ++k) {
  for (int j = 0; j < ySize; ++j) {
     for (int i = 0; i < xSize; ++i) {
        buffer.lvalue(i, j, k) = 0.0f;
     }
  }
}
```



Access pattern for the improved loop

Of course, we are still doing the multiplication to find the 1D array index once per voxel access, something that could be avoided through the use of *iterators*. Iterators allow code to be written without explicit bounds checks in all dimensions:

```
for (DenseField<float>::iterator i = buffer.begin(); i != buffer.end(); ++i) {
  *i = 0.0f;
}
```

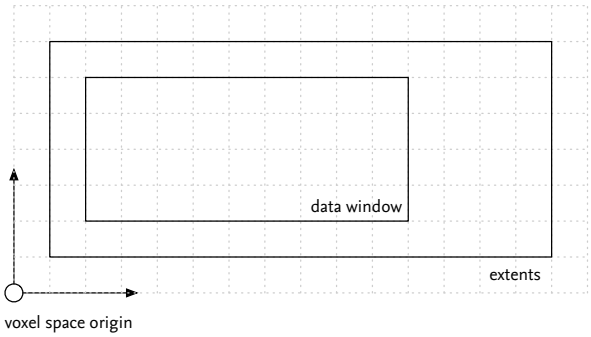
Or even better:

```
std::fill(buffer.begin(), buffer.end(), 0.0f);
```

Iterators have the benefit of both being more efficient and producing cleaner code. We refer the interested reader to the Field3D programmer's guide[5] for a more in-depth look at iterators.

2.1.4. Extents and data windows

As mentioned earlier, voxel indices do not need to start at [0, 0, 0]. As a parallel, images in the OpenEXR file format have a *display window* and *data window* that specify the intended size and the allocated pixels of an image. The same concept translates well to voxel buffers, where we will refer to the intended size of the buffer as *extents* and the region of legal indices as *data window*.



2D example of extents and data window

In the illustration above, the *extents* (which defines the [0, 1] local coordinate space) is greater than the *data window*. It would be the result of the following code:

```
Box3i extents(V3i(1, 1, 0), V3i(15,7,10));
Box3i dataWindow(V3i(2, 2, 0), V3i(11, 6, 10));
buffer.setSize(extents, dataWindow);
```

Using separate extents and data window can be helpful for image processing (a blur filter can run on a padded version of the field so that no boundary conditions need to be handled), interpolation

[5] http://sites.google.com/site/field3d/downloads

(guarantees that a voxel has neighbors to interpolate to, even at the edge of the extents) or for optimizing memory use (only allocates memory for the voxels needed).

2.1.5. Coordinate spaces and mappings

The only coordinate space we've discussed so far is the voxel buffer's native coordinate system. In the future, we will refer to this coordinate space as *voxel space*. In order to place a voxel buffer in space we also need to define how to transform a position from *voxel space* into *world space* (which is the global reference frame of the renderer). Besides *voxel* and *world space*, a third space is useful, similar to RenderMan's NDC space but local to the buffer. This *local space* defines a [0, 1] range over all voxels and is used as a resolution independent way of specifying locations within the voxel buffer. This definition is the same as Field3D uses.

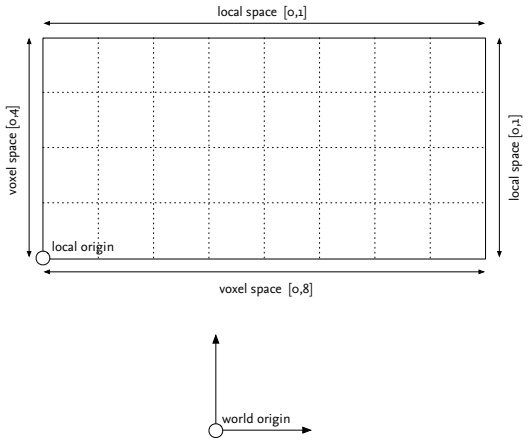


Illustration of coordinate spaces

When constructing a voxel buffer we define a *localToWorld* transform in order to place the buffer in space. This transform is also called *mapping*, and defines the transformation between *local space* and *world space*. Note that the transformation from *local space* to *voxel space* is the same regardless of the buffer's location in space. To sum things up:

- *World space* is the global coordinate system and exists outside of any voxel buffer.
- *Local space* is a resolution-independent coordinate system that maps the full *extents* of the voxel buffer to a [0, 1] space.
- *Voxel space* is used for indexing into the underlying voxels of a field. A field with 100 voxels along the *x* axis maps [100.0, 0.0, 0.0] in *voxel space* to [1.0, 0.0, 0.0] in *local space*.

As a matter of convenience and clarity, we will prefix variables in code and pseudocode with an abbreviated form of the coordinate space. A point `P` in *world space* will be called `wsP`, in *voxel space* `vsP`, and in *local space* `lsP`.

2.1.6. Integer versus floating-point coordinates

Voxel space is different from *local* and *world space* in that it can be accessed in two ways – using integer or floating-point coordinates. Integer access is used for direct access to an individual voxel, and floating-point coordinates are used when interpolating values. It is important to take care when converting between the two. The center of voxel [0, 0, 0] has floating-point coordinates [0.5, 0.5, 0.5]. Thus, the edges of a field with resolution 100 are at 0.0 and 100.0 when using floating-point coordinates but when indexing using integers, only 0 through 99 are valid indices. An excellent overview of this can be found in an article by Paul S. Heckbert – *What Are The Coordinates Of A Pixel?* [Heckbert, 1990]

In practice, it is convenient to define a set of conversion functions to go from `float` to `int`, and `Vec3<float>` to `Vec3<int>`, etc. In this course we will refer to these conversion functions as `discreteToContinuous()` and `continuousToDiscrete()`.

```
int continuousToDiscrete(float contCoord)
{
  return static_cast<int>(std::floor(contCoord));
}

float discreteToContinuous(int discCoord)
{
  return static_cast<float>(discCoord) + 0.5f;
}
```

2.1.7. Boundless fields

Although it is outside the scope of this course, it is possible to design voxel data structures that are *boundless*, i.e. that contain data over an infinitely large domain. In those cases the coordinate spaces would need to be redefined, as the local space concept no longer applies, leaving only a *worldToVoxel* transform.

2.2. Writing to voxel buffers

The fundamental purpose of a voxel buffer is obviously to read and write to it. In this section we will consider a few different ways of writing voxel data, and the methods will serve as the foundation for all subsequent modeling techniques.

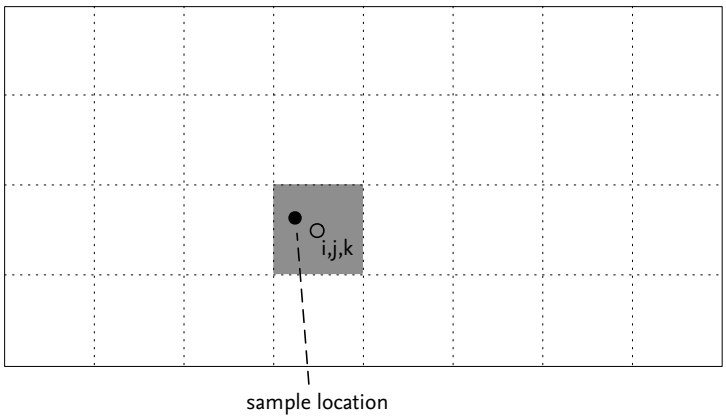
For purposes of illustration, let's consider a simple C++ function for writing a floating-point value to a given voxel:

```
void writeToVoxel(VoxelBuffer::Ptr buffer, int i, int j, int k, float value)
{
  buffer->lvalue(i, j, k) += value;
}
```

Writing a value directly at a voxel location doesn't get us very far in terms of modeling complex voxel buffer however. As it turns out, the most common modeling operation is the writing of a value in-between voxels. In these notes we will refer to this as *splatting*, though it is sometimes also called *stamping* and *baking* a sample.

2.2.1. Nearest neighbor splat

The simplest way to splat a value that lies in-between voxels is to simply round the coordinates to the nearest integers. While this has some obvious aliasing problems, it can sometimes be a reasonable solution, especially when writing large quantities of low-density values which will blend when taken together.



Splatting a sample using the nearest-neighbor strategy

This method can be implemented trivially as:

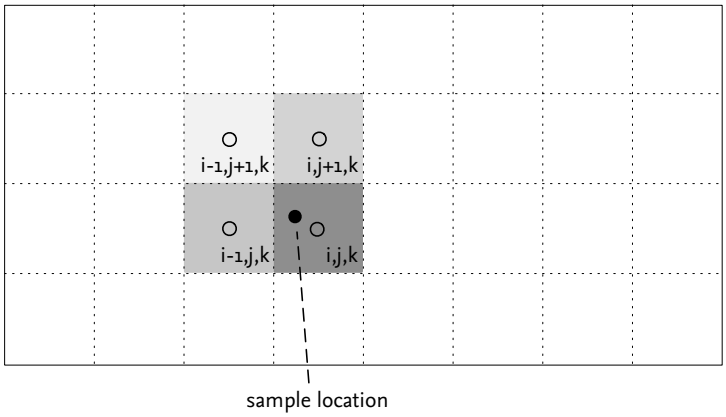
```
inline void writePoint(const Vector &vsP, const Imath::V3f &value,
                       VoxelBuffer::Ptr buffer)
{
  using namespace Field3D;

  int i = contToDisc(vsP.x);
  int j = contToDisc(vsP.y);
  int k = contToDisc(vsP.z);

  if (buffer->isInBounds(i, j, k)) {
    buffer->lvalue(i, j, k) += value;
  }
}
```

2.2.2. Trilinear splat

If antialiasing is important we can use a filter kernel when writing the value. The simplest, and most commonly used form is a triangle filter with a radius of one voxel. This filter will at most have non-zero contribution at eight voxels surrounding the sample location. The value to be written is simply distributed between its neighboring voxels, each weighted by the triangle filter.



Splatting a sample using the trilinear strategy

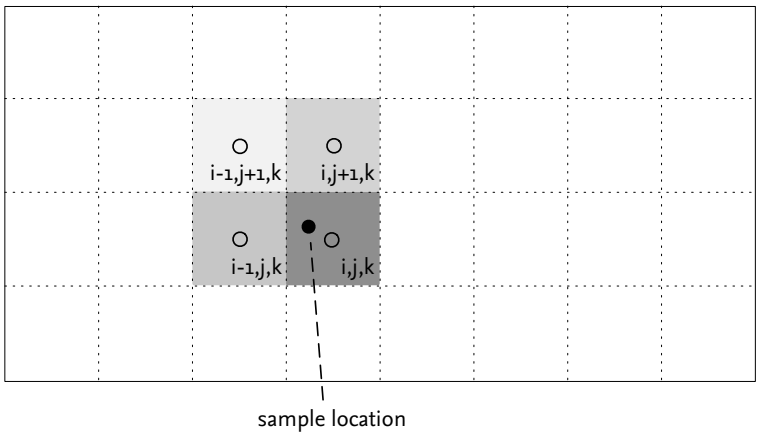
A simple implementation would be:

```
void writeAntialiasedPoint(const Vector &vsP, const Imath::V3f &value,
                           VoxelBuffer::Ptr buffer)
{
  using namespace std;
  using namespace Imath;

  // Offset the voxel-space position relative to voxel centers
  // The rest of the calculations will be done in this space
  Vector p(vsP.x - 0.5, vsP.y - 0.5, vsP.z - 0.5);
  // Find the lower-left corner of the cube of 8 voxels that
  // we need to access
  V3i corner(static_cast<int>(floor(p.x)),
             static_cast<int>(floor(p.y)),
             static_cast<int>(floor(p.z)));
  // Calculate P's fractional distance between voxels
  // We start out with (1.0 - fraction) since each step of the loop
  // will invert the value
  Vector fraction(Vector(1.0f) - (static_cast<Vector>(corner + V3i(1)) - p));
  // Loop over the 8 voxels and distribute the value
  for (int k = 0; k < 2; k++) {
  fraction[2] = 1.0 - fraction[2];
    for (int j = 0; j < 2; j++) {
      fraction[1] = 1.0 - fraction[1];
      for (int i = 0; i < 2; i++) {
        fraction[0] = 1.0 - fraction[0];
        double weight = fraction[0] * fraction[1] * fraction[2];
        if (buffer->isInBounds(corner.x + i, corner.y + j, corner.z + k)) {
          buffer->lvalue(corner.x + i,
                         corner.y + j,
                         corner.z + k) += value * weight;
        }
      }
    }
  }
}
```

2.3. Interpolation

In order to sample an arbitrary location within a voxel buffer we have to use interpolation. The most common scheme is trilinear interpolation which computes a linear combination of the 8 data points around the sampling location. The concept and implementation are very similar to the trilinear splatting described above.



2D illustration of linear interpolation

Depending on the look required, it may be desirable to use higher order interpolation schemes. Such schemes will come at an increased computational cost. Profiling reveals that a significant portion of the runtime of a volume renderer is spent interpolating voxel data. The primary reason is that a naive voxel buffer data structure offers very poor cache coherence. A tiled data storage scheme combined with structured accesses will improve overall performance, but will require a more complicated implementation.

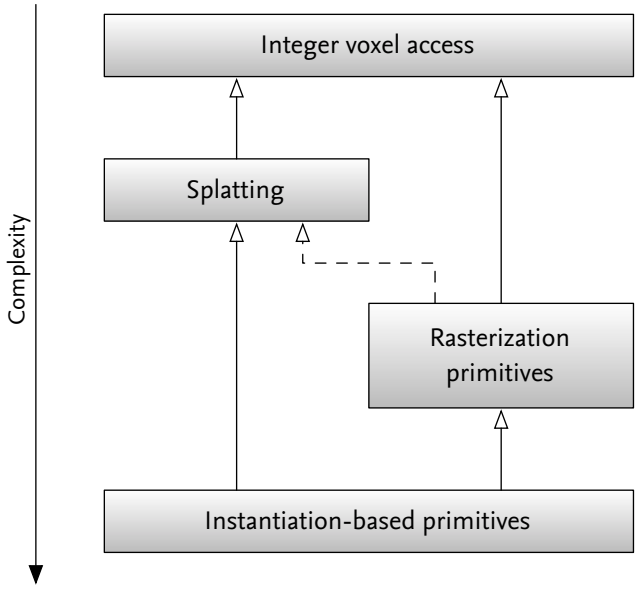
The following is an implementation of trilinear interpolation:

```
float Sampler::trilinearInterpolation(const V3f& vsP)
{
  // Offset the voxel-space position relative to voxel centers
  // The rest of the calculations will be done in this space
  V3f p(vsP.x - 0.5, vsP.y - 0.5, vsP.z - 0.5);
  // Find the lower-left corner of the cube of 8 voxels
  // that we need to access
  V3i lowerLeft(static_cast<int>(floor(p.x)),
                static_cast<int>(floor(p.y)),
                static_cast<int>(floor(p.z)));
  float weight[3];
  float value = 0.0;
  for (int i = 0; i < 2; ++i)
  {
    int cur_x = lowerLeft[0] + i;
    weight[0] = 1.0 - std::abs(p[0] - cur_x);
```

```
    for (int j = 0; j < 2; ++j)
    {
      int cur_y = lowerLeft[1] + j;
      weight[1] = 1.0 - std::abs(p[1] - cur_y);
      for (int k = 0; k <= 1; ++k)
      {
        int cur_z = lowerLeft[2] + k;
        weight[2] = 1.0 - std::abs(p[2] - cur_z);
        value += weight[0] * weight[1] * weight[2] * buffer.value(cur_x, cur_y, cur_z);
      }
    }
  }
  return value;
}
```

2.4. Geometry-based volume modeling

In the previous sections we discussed direct (integer) voxel access, and how to splat filtered samples into a voxel buffer. These can be thought of as the first two layers in the voxel modeling pipeline.



Outline of the volume modeling abstraction hierarchy

The third layer is the rasterization layer. *Rasterization primitives* include types such as pyroclastic points, splines and surfaces, but also includes any primitive that is converted voxel-by-voxel into a volumetric representation. The rasterization process normally accesses voxels directly (i.e. using the integer voxel access layer), although when considering motion blur they may also use the splatting layer.

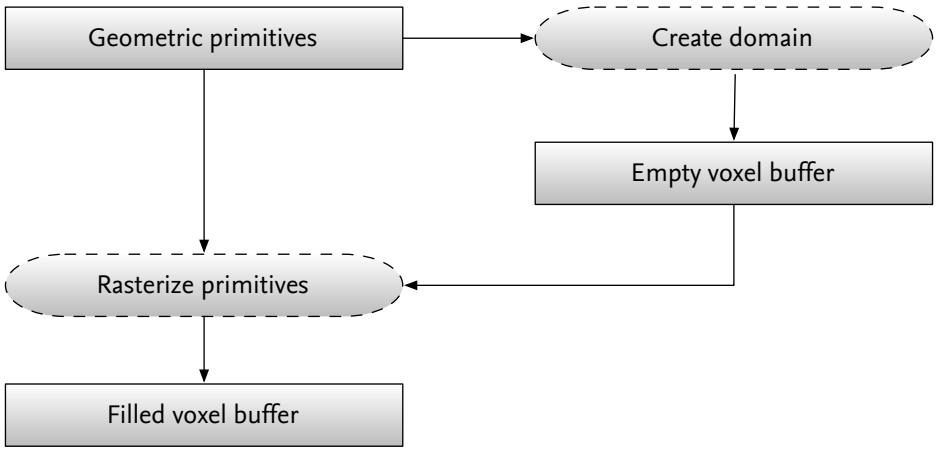
The fourth layer is *instantiation-based primitives*. They are referred to by different names at different facilities, sometimes also called *wisps* or *generators*. These primitives are composed of instances of the lower-level primitives, and either create rasterization primitives, or directly create sample points to be splatted. Instantiation-based primitives may also generate other instances of their own or other instantiation primitive types. Because of this potentially recursive nature, they can be very powerful.

The third and fourth layers can be thought of as two quite different approaches to volume modeling, even though they are often used in conjunction. A useful comparison is that of the difference between a raytracing-based renderer and a micropolygon-based one. Rasterization is similar to raytracing in that it considers each voxel in turn and decides how primitives contribute to it. Instantiation-based primitives (and micropolygon renderers) see primitives as the first-class citizen, and considers which voxels are affected by a given primitive. Rasterization-based modeling *pulls* values into a voxel, and instantiation-based modeling *pushes* values into voxels.

2.4.1. Defining voxel buffer domains

The first step in volume modeling is to determine the *domain* of the voxel buffer that is being created, so that the buffer encloses the space of the primitives that are being rasterized. A very basic implementation might simply compute an axis-aligned or oriented bounding box for the incoming primitives, but a robust solution needs to consider other factors. For example, almost all volumetric primitives extend out past their geometric representation. If the system allows users to create new primitives as plug-ins, it is important to communicate the bounds of a primitive back to the renderer during the domain calculation. This is especially true for primitives that include displacements driven by user input. Although it is possible to let the user dial displacement bounds manually, usability is improved if they are handled automatically.

The motion of a primitive also needs to be considered in order to provide enough room to store the entire length of the sample. Motion blur techniques are discussed further in subsequent sections.

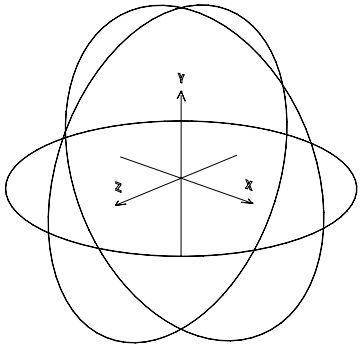


2.4.2. Noise coordinate systems

Volume modeling often, if not always, uses noise functions to add detail to primitives. Noise functions need to be tied to a coordinate system, but almost any geometric primitive that has a reasonable parameterization method for a 3D coordinate system can be used.

For some primitives, such as a sphere, the coordinate system is trivial. Others, such as splines, require a little bit more work. The important thing to consider is that the parameterization should be smooth and reasonably quick to transform in and out of. Therefor we prefer transformations with closed-form solutions, rather than ones that require numerical iteration to find a solution, but as we will see, not all primitives used in production satisfy this preference.

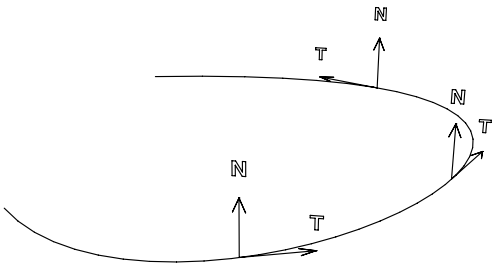
For a sphere, defining a coordinate system is simple. We simply use the object space as the noise coordinate space. Transformations in and out of this coordinate system can be calculated as a simple matrix multiplication.



Coordinate system for a sphere (x, y, z)

Primitive with cartesian coordinate system

For a curve or spline it is most common to use a coordinate system that deforms along with the spline. The coordinate range is [0,1] along the spline and [-1,1] along the two axes that span the radius of the spline. In order to transform in and out of the local/noise space we define two basis vectors. The tangent of the curve itself is used as one basis, and the normal direction of the curve (orthogonal to the tangent) is used as the second. The third basis can be computed as the cross product of the first two. The curve may be a polygonal line or a parametric curve, but regardless of how the vertices are interpolated the transform in and out of this space is much more costly than for a sphere.

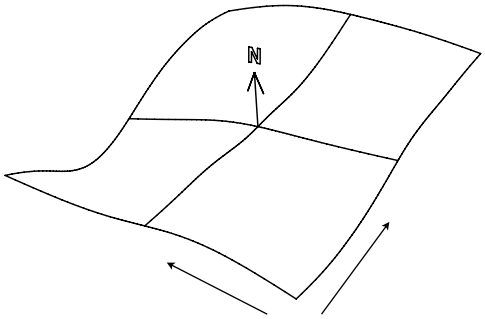


Basis vectors for a curve (N × T, N, T)

Primitive with curve-based coordinate system

The coordinate range for a surface patch is [0,1] along each dimension of the surface and [-1,1] along the normal direction of the surface. This implies that the third dimension extends equally from the back and front of the surface.

A polygon mesh or surface patch can be transformed into and out of using the *dP/du* and *dP/dv* partial derivatives as the first two basis vectors, and the normal direction as the third. Just as with curves, the transformation into this space is costly as the surface primitive may be composed of an arbitrary number of parametric primitives, which need to be searched.



Basis vectors for a surface (dP/du, dP/dv, N)

Primitive with surface-based coordinate system

2.5. Rasterization primitives

Rasterization is the process of building volume data voxel-by-voxel. Fundamentally, there are two approaches to rasterizing. The first is to visit each voxel in the buffer once, the second is to visit each primitive once. Depending on the way the voxel data is stored, one may be more appropriate than the other. For example, some renderers store voxel data as a set of 2D images on disk, each compressed using some form of non-lossy scheme. The overhead of pulling a slice from disk and decompressing it into memory is quite expensive, in which case the better approach is to visit each voxel only once. For these notes, we will assume that the buffer used for rasterization is fully loaded in memory and that there is no penalty for accessing neighboring voxels in any direction (other than potential cache misses) and use the second approach of visiting each primitive once.

2.5.1. Rasterization algorithm

In its most generic form, rasterization involves instancing the primitive representation, bounding it, looping over the voxels that it overlaps, and sampling its density function at each voxel.

```
void Point::execute(Geo::Geometry::CPtr geometry,
                    VoxelBuffer::Ptr buffer) const
{
  AttrVisitor visitor(points, m_params);
  for (AttrVisitor::const_iterator i = visitor.begin(), end = visitor.end();
       i != end; ++i) {
    // Update attributes
    m_attrs.update(i);
    // Transform to voxel space
    Vector vsP;
    buffer->mapping()->worldToVoxel(m_attrs.wsCenter.as<Vector>(), vsP);
    // Calculate rasterization bounds
    BBox vsBounds = vsSphereBounds(mapping, m_attrs.wsCenter.as<Vector>(),
                                   m_attrs.radius);
    // Call Base::rasterize(), which will come back and query getSample() for values
    RasterizationPrim::rasterize(vsBounds, buffer);
  }
}

void RasterizationPrim::rasterize(const BBox &vsBounds,
                                  VoxelBuffer::Ptr buffer) const
{
  FieldMapping::Ptr mapping(buffer->mapping());

  DiscreteBBox dvsBounds = Math::discreteBounds(vsBounds);
  DiscreteBBox bufferBounds = buffer->dataWindow();
  dvsBounds.min -= Imath::V3i(1);
  dvsBounds.max += Imath::V3i(1);
  dvsBounds = Math::clipBounds(dvsBounds, bufferBounds);

  // Iterate over voxels
```

```
  for (VoxelBuffer::iterator i = buffer->begin(dvsBounds),
         end = buffer->end(dvsBounds); i != end; ++i, ++count) {
    RasterizationState rState;
    RasterizationSample rSample;
    // Get sampling derivatives/voxel size
    rState.wsVoxelSize = mapping->wsVoxelSize(i.x, i.y, i.z);
    // Transform voxel position to world space
    Vector vsP = discToCont(V3i(i.x, i.y, i.z));
    mapping->voxelToWorld(vsP, rState.wsP);
    // Sample the primitive
    this->getSample(rState, rSample);
    // Write to buffer
    if (Math::max(rSample.value) > 0.0f) {
      *i += rSample.value;
    }
  }
}

void Point::getSample(const RasterizationState &state,
                      RasterizationSample &sample) const
{
  if ((state.wsP - m_attrs.wsCenter).length < m_attrs.radius) {
    sample.value = m_attrs.density.value();
  } else {
    sample.value = 0.0;
  }
}
```

The `m_attrs.update()` call is responsible for finding the current primitive's point attributes. The second step, `vsSphereBounds()`, returns the voxel-space bounding box of the primitive. Once the primitive is prepared and the region of voxels to traverse is known, the primitive is evaluated at each voxel location and the density is recorded in the voxel buffer.

2.5.2. Rasterizing primitives

Sphere-based primitives always carry two fundamental attributes: their position (center), and their radius. On top of these an arbitrary number of attributes are used to define the various noise parameters that control its look.

For a sphere-shaped primitive the bounding box is a fairly tight fit, but for curves and surfaces many voxels will be calculated that lie far away from the primitive's region of influence. This has the downside of causing lots of unnecessary voxels to be computed. The rasterization loop can be improved in those cases, for example by determining the distance to a primitive before calculating the density function. However, even with that optimization, the world-to-local transform is quite expensive for curves and surfaces, and in practice point-instantiation techniques are used for those types of primitives. The next chapter will describe that approach in more detail.

2.5.3. Solid noise primitives

One of the most straight-forward sphere-based primitives is the solid noise primitive. It uses the location of the sphere and its radius to "window" a noise function, so the density function is simply the sum of the windowing function and a fractal function.

Solid noise point

The function can be written as:

$noiseDensity(P) = fbm(P) + (1 - |P/radius|)$

where P is in the local space of the primitive. We notice that because of the fractal function, the density function may be positive outside the radius of the sphere. If the maximal amplitude of the fractal function fbm is A, it follows that the function has non-negative values at most A units away from the radius, as $A + (1 - |1+A|) = 0$.

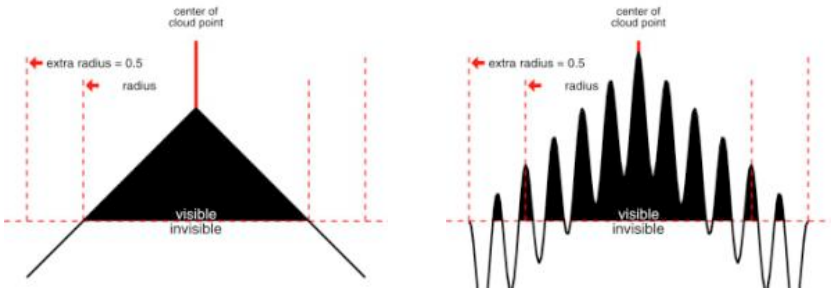


Illustration of density function and the required bounds padding

Because of this added distance, solid noise points are an example of a primitive that requires padding of its bounds calculations (as mentioned in *Defining voxel buffer domains*).

The `getSample()` call of a solid noise primitive would be:

```
void SolidNoisePoint::getSample(const RasterizationState &state,
                                RasterizationSample &sample) const
{
  // Transform to the point's local coordinate system
  float radius = m_attrs.radius;
  Vector lsP = (state.wsP - m_attrs.wsCenter.as<Vector>()) / radius;
  Vector nsP = lsP;
  // Compute fractal function
  double fractalFunc = m_attrs.fractal->eval(nsP);
  fractalFunc = Math::gamma(fractalFunc, m_attrs.gamma.value());
  fractalFunc *= m_attrs.amplitude;
  // Compute final value
  double distanceFunc = 1.0 - lsP.length();
  sample.value = m_attrs.density.value() *
                 std::max(0.0, distanceFunc + fractalFunc);
}
```

2.5.4. Pyroclastic sphere primitives

Pyroclastic primitives have been mentioned several times so far, so let's see how one can be implemented. A pyroclastic noise function uses a distance function to determine its location in the scene, and adds a procedural noise value (usually a fractal function, for example *fractal brownian motion*[6]) to the distance function. By thresholding the final value we create the pyroclastic look, although for antialiasing purposes it's better to use a smoothstep function, so that the transition in density is gradual.

pyroclasticDensity(P) = max(radius - |P/radius| + abs(fbm(P)), 0)
density = smoothStep(pyroclasticDensity, 0, 0.05)

A single pyroclastic noise primitive

[6] For a complete description of the fbm function, see *Ebert et al – Texturing & Modeling (Morgan Kaufmann publ.)*

The pyroclastic look comes from using the noise function as a displacement, rather than by directly rendering it, and because the displacement is done per-voxel on the distance function itself, it is possible to produce overhangs, where parts of the density disconnects from the main body. If this is undesirable, the noise function lookup point can be projected onto the sphere primitive, effectively making the displacement amount constant for all points along the same normal vector. This is illustrated in the figure *3D vs. 2D displacement* below.

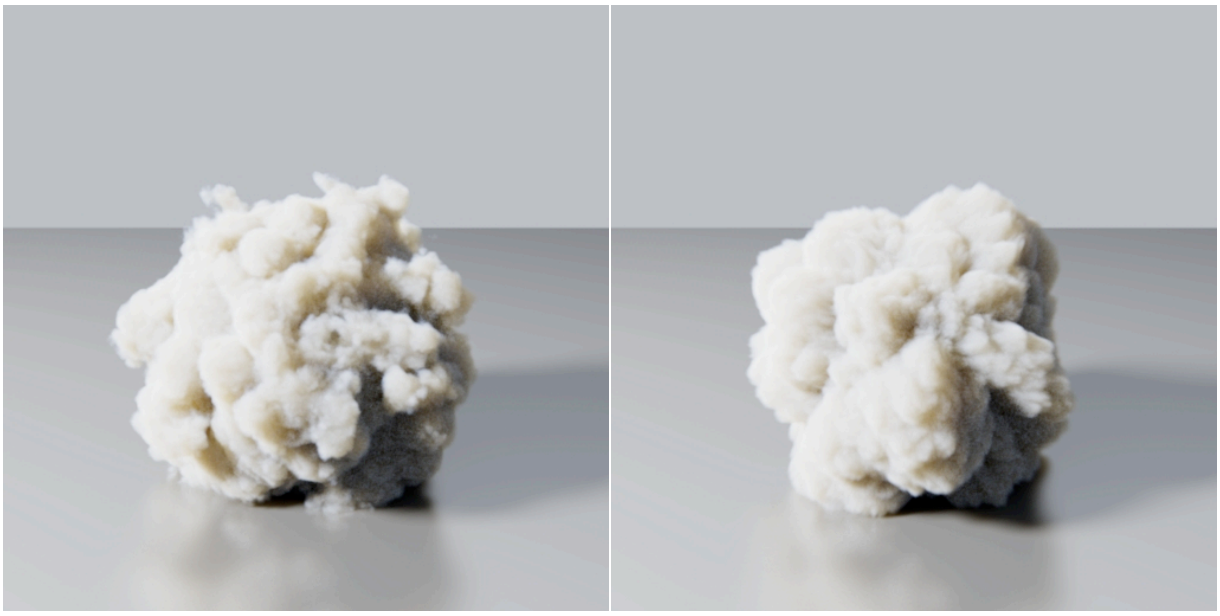
In its simplest form, the code would be:

```
void PyroclasticPoint::getSample(const RasterizationState &state,
                                 RasterizationSample &sample) const

{
  // Transform to the point's local coordinate system
  float radius = m_attrs.radius;
  Vector lsP = (state.wsP - m_attrs.wsCenter.as<Vector>()) / radius;
  Vector nsP = m_attrs.displace2D ? lsP.normalized() : lsP;
  // Compute fractal function
  double fractalFunc = m_attrs.fractal->eval(nsP);
  fractalFunc = Math::gamma(fractalFunc, m_attrs.gamma.value());
  fractalFunc *= m_attrs.amplitude;
  // Compute final value
  double sphereFunc = lsP.length() - 1.0;
  float filterWidth = state.wsVoxelSize.length();
  double thresholdWidth = filterWidth * 0.5 / radius;
  double pyroValue;
  pyroValue = Math::fit(sphereFunc - fractalFunc,
                        -thresholdWidth, thresholdWidth, 1.0, 0.0);
  sample.value = m_attrs.density.value() * pyroValue;
}
```

By varying the noise parameters (amplitude, scale, gain), and animating the noise offset, it is possible to create a wide range of looks even from such a simple primitive. And yet more variations can be had by using a vector-valued noise function to displace the sample point used by the pyroclastic noise function.

Varying noise amplitude



3D displacement vs. 2D displacement

2.5.5. Sampling and antialiasing

Rasterization is prone to aliasing artifacts and sampling problems in the same way that surface rendering is. Where the projected pixel size, or the spot size, is used in surface rendering to antialias shader functions, we can use the voxel size to do the same in volume rasterization. The sampling frequency is simply the inverse of the voxel size. Once the sampling frequency is knows, we can apply the same frequency clamping and other antialiasing techniques as used in surface shading. Larry Gritz' section in the 1998 Advanced RenderMan SIGGRAPH course notes [Gritz, 1998] is a good starter.

Similarly to how the sample positions in surface rendering may be randomized, we can add a small offset to each voxel's sample position using some function with a nice poisson distribution that prevents sample locations from bunching up.

2.6. Instantiation-based primitives

Rasterization primitives work well when modeling clouds, fog and other phenomena that are inherently continuous in nature and where the primitives fill in a major portion of the voxels in the buffer. In cases where primitives contain a lot of negative space the overhead of traversing and calculating the density function for all voxels can be quite substantial, and the more sparse the primitives become, the worse the performance. The problem is inherent to the pull-nature of the rasterization algorithm, and it is difficult to optimize away the wasteful sampling without incurring too much overhead in bookkeeping. Instantiation-based primitives avoid this problem as their push-nature means that calculations only take place for parts of the primitive that actually contribute density. This means that calculation costs are proportional to the amount of voxel data that is actually visible, instead of proportional to the coverage of the base primitive, as in the case of rasterization primitives.

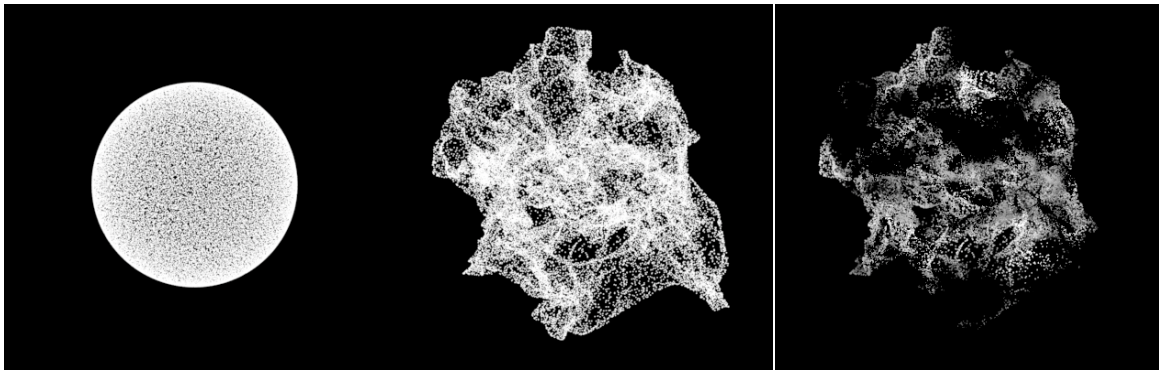
Although instantiation-based primitives theoretically can instantiate any other primitive, the most common case is the class which instantiates points (usually in very large numbers) to fill in a volume. We will refer to those as point instantiation primitives.

Point instantiation primitives have other advantages beside their inherent efficiency. They also support the full range from smooth-looking to granular primitives, simply by varying the number of instances used to fill the primitive.

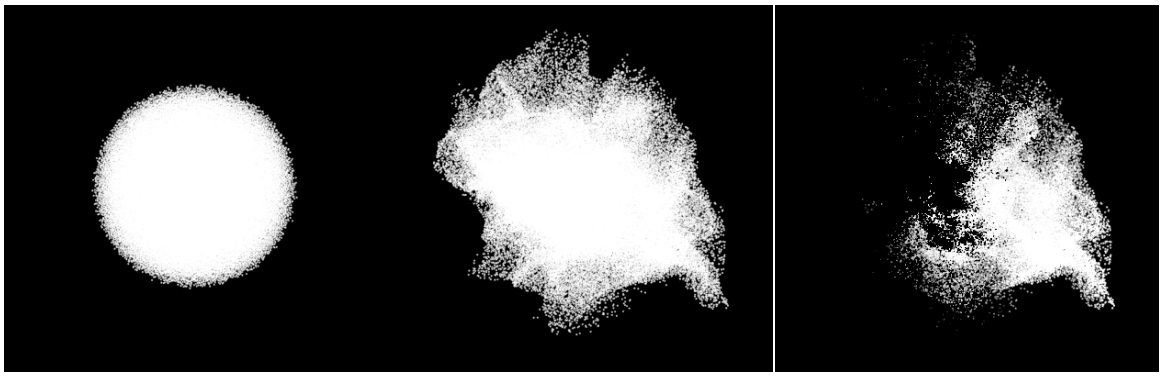
Examples of primitives using point instantiation

2.6.1. Point instantiation

The simplest point instantiation primitive uses a sphere as its base primitive. The first step in its generation is the scattering of points. Depending on the desired look, the scattering may be done only on the surface of the sphere, or inside the entire volume of the sphere. The images below show the different results of the two techniques.



left) Points scattered at sphere radius
middle) Points displaced by noise function
right) Point color modulated by noise function



left) Points scattered to fill inside of sphere
middle) Points displaced by noise function
right) Point color modulated by noise function

Once the points are scattered, any number of noise- or texture-based modulations and displacements may occur. In our simple example the points are displaced by a vector-valued fractal noise function, and then a scalar-valued noise function is used to modulate their color. From this simple foundation, point instantiation primitives used in production typically add large numbers of control parameters and noise functions, each responsible for manipulating the final appearance in a different way.

The following code implements a simple sphere-based instancing algorithm

```
void Sphere::execute(const Geo::Geometry::CPtr geo) const
{
  size_t numPoints = numOutputPoints(geo);
  Particles::Ptr particles = Particles::create();
  particles->add(numPoints);

  AttrTable &points = particles->pointAttrs();
  AttrRef radius    = points.addFloatAttr("radius", 1, vector<float>(1, 1.0));
  AttrRef density   = points.addVectorAttr("density", Vector(1.0));

  // Loop over input points
  size_t idx = 0;
  AttrVisitor visitor(geo->particles()->pointAttrs(), m_params);
  for (AttrVisitor::const_iterator i = visitor.begin(), end = visitor.end();
       i != end; ++i) {
    // Update attributes
    m_attrs.update(i);
    // Seed random number generator
    Imath::Rand48 rng(m_attrs.seed);
    // For each instance
    for (int i = 0; i < m_attrs.numPoints; ++i, ++idx) {
      // Randomize local space position
      Vector lsP(0.0);
      if (m_attrs.doFill) {
        lsP = solidSphereRand<V3f>(rng);
      } else {
        lsP = hollowSphereRand<V3f>(rng);
      }
      // Define noise space
      V3f nsP = lsP;
      // Set instance position
      V3f instanceWsP = m_attrs.wsCenter;
      instanceWsP += lsP * m_attrs.radius;
      // Apply displacement noise
      if (m_attrs.doDispNoise) {
        V3f noise = m_attrs.dispFractal->evalVec(nsP);
        instanceWsP += noise * m_attrs.dispAmplitude * m_attrs.radius;
      }
      // Set instance density
      V3f instanceDensity = m_attrs.density;
      // Apply density noise
      if (m_attrs.doDensNoise) {
        float noise = m_attrs.densFractal->eval(nsP);
        instanceDensity *= noise;
      }
      // Set instance attributes
      particles->setPosition(idx, instanceWsP);
      points.setVectorAttr(wsV, idx, Vector(0.0));
      points.setVectorAttr(density, idx, instanceDensity);
      points.setFloatAttr(radius, idx, 0, m_attrs.instanceRadius);
    }
  }
}
```

Note that in this example we accumulate all points in a collection which gets sent to the rasterizer once instantiation is complete. It would also be possible to directly rasterize each point as it is instanced.

2.6.2. Curve-based point instantiation

Sphere-based primitives are particularly convenient because of their simple parameterization, and because of how easy it is to define a coordinate space that travels with the primitive. As we will see, curves and surfaces can also be used, but their coordinate spaces are a little more involved to define.

The following images show various noise techniques applied to a curve primitive. Each primitive uses roughly 40 million points.

Constant-radius curve primitive

Varying the radius of the curve

Absolute-valued perlin noise modulating density

Displacing points along normal using absolute perlin noise (pyroclastic)

Points displaced by vector-valued perlin noise

Density first modulated by scalar-valued perlin noise, then displaced using vector-valued perlin noise

In the example above the point distribution is completely random, which can lead to bunching up of point locations and lead to a grainy look in the final result. Depending on the desired look, this may or may not be a good thing. If a smooth look is the goal, it may be better to use a blue noise/poisson distribution of points.

The following code shows a simple curve-based instancing algorithm.

```
void Line::execute(const Geo::Geometry::CPtr geo) const
{
  size_t numInstancePoints = numOutputPoints(geo);
  Particles::Ptr particles = Particles::create();
  particles->add(numInstancePoints);

  AttrTable &points = particles->pointAttrs();
  AttrRef radiusRef = points.addFloatAttr("radius", 1, oneDefault);
  AttrRef densityRef = points.addVectorAttr("density", Vector(1.0));

  // Loop over input polygons
  size_t         idx = 0;
  Polygons::CPtr polys = geo->polygons();
  AttrVisitor    polyVisitor(polys->polyAttrs(), m_params);
  AttrVisitor    pointVisitor(polys->pointAttrs(), m_params);

  for (AttrIter iPoly = polyVisitor.begin(), endPoly = polyVisitor.end();
       iPoly != endPoly; ++iPoly) {
    // Update poly attributes
    updatePolyAttrs(iPoly);
    // Update point attributes
    size_t first = polys->pointForVertex(iPoly.index(), 0);
    size_t numPoints = polys->numVertices(iPoly.index());
    updatePointAttrs(pointVisitor.begin(first), numPoints);
    // Seed random number generator
```

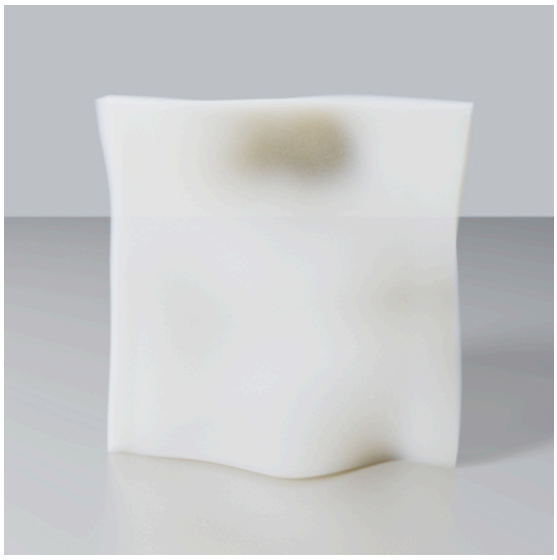
```
        Imath::Rand48 rng(m_polyAttrs.seed);
        // For each instance
        for (int i = 0; i < m_polyAttrs.numPoints; ++i, ++idx) {
          // Randomize local space position
          V2f disk;
          if (m_polyAttrs.doFill) {
            disk = solidSphereRand<V2f>(rng);
          } else {
            disk = hollowSphereRand<V2f>(rng);
          }
          V3f lsP(disk.x, disk.y, rng.nextf());
          // Let t be floating-point index
          float t = lsP.z * (m_pointAttrs.size() - 1);
          // Interpolate instance attributes
          V3f instanceDensity = LINE_INST_INTERP(density, t);
          V3f instanceWsP     = LINE_INST_INTERP(wsP, t);
          V3f instanceWsV     = LINE_INST_INTERP(wsVelocity, t);
          V3f wsN             = LINE_INST_INTERP(wsNormal, t).normalized();
          V3f wsT             = LINE_INST_INTERP(wsTangent, t).normalized();
          float radius        = LINE_INST_INTERP(radius, t);
          // Compute first basis vector from N and T
          V3f wsNxT = wsN.cross(wsT);
          // Offset instance position based on local space coordinate
          instanceWsP += lsP.x * wsNxT * radius;
          instanceWsP += lsP.y * wsN * radius;
          // Apply noises
          V3f nsP = lsP;
          if (m_polyAttrs.doDensNoise) {
            V3f nsLookupP = nsP / m_polyAttrs.densScale.value();
            instanceDensity *= m_polyAttrs.densFractal->eval(nsLookupP);
          }
          if (m_polyAttrs.doDispNoise) {
            V3f nsLookupP = nsP / m_polyAttrs.dispScale.value();
            V3f disp = m_polyAttrs.dispFractal->evalVec(nsLookupP);
            instanceWsP += disp.x * wsNxT * radius *
              m_polyAttrs.dispAmplitude.value();
            instanceWsP += disp.y * wsN * radius *
              m_polyAttrs.dispAmplitude.value();
            instanceWsP += disp.z * wsT * radius *
              m_polyAttrs.dispAmplitude.value();

          }
          // Set instance attributes
          particles->setPosition(idx, instanceWsP);
          points.setVectorAttr(densityRef, idx, instanceDensity);
          points.setFloatAttr(radiusRef, idx, 0, m_polyAttrs.instanceRadius);
        }
      }
    }
```

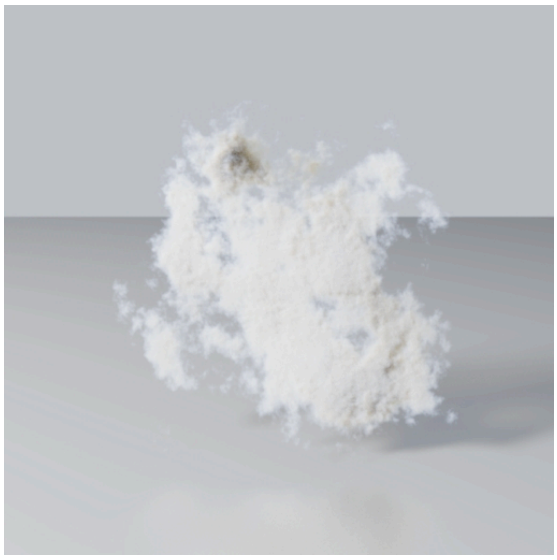
`LINE_INST_INTERP()` macros are used to calculate the values in-between the control vertices of the curve, and they may use any method for this. If curves are finely tessellated then a piecewise linear function may be enough, although it is more common to use a spline function.

2.6.3. Surface-based point instantiation

The following images show various noise techniques applied to a surface primitive. Each primitive uses roughly 400 million instanced points in order to achieve a smooth result at 1024x1024 pixels. A frustum-shaped voxel buffer of resolution ~1200x1200x250 was used. It should be noted that primitives that modulate their density using noise are slightly "wasteful", because we need to instantiate a point, calculate its noise space position and evaluate the full fractal noise function before knowing whether to cull it due to zero density.



Constant-thickness surface primitive



Perlin noise modulating density

Absolute-valued perlin noise modulating density

Displacing points along normal using absolute perlin noise (pyroclastic)

Points displaced by vector-valued perlin noise

Density first modulated by scalar-valued absolute perlin noise, then displaced using vector-valued perlin noise

The following code shows a simple surface-based instancing algorithm.

```
void Surface::execute(const Geo::Geometry::CPtr geo) const
{
  size_t numInstancePoints = numOutputPoints(geo);
  Particles::Ptr particles = Particles::create();
  particles->add(numInstancePoints);

  AttrTable &points = particles->pointAttrs();
  AttrRef radiusRef = points.addFloatAttr("radius", 1, oneDefault);
  AttrRef densityRef = points.addVectorAttr("density", Vector(1.0));

  // Loop over input points
  size_t       idx = 0;
  Meshes::CPtr meshes = geo->meshes();
  AttrVisitor  meshVisitor(meshes->meshAttrs(), m_params);
  AttrVisitor  pointVisitor(meshes->pointAttrs(), m_params);

  for (AttrIter iMesh = meshVisitor.begin(), endMesh = meshVisitor.end();
       iMesh != endMesh; ++iMesh) {
    // Update mesh attributes
    updateSurfAttrs(iMesh);
    // Update point attribute
    size_t first = meshes->startPoint(iMesh.index());
    size_t numCols = meshes->numCols(iMesh.index());
    size_t numRows = meshes->numRows(iMesh.index());
    size_t numPoints = numCols * numRows;
    updatePointAttrs(pointVisitor.begin(first), numPoints);
    // Seed random number generator
    Imath::Rand48 rng(m_surfAttrs.seed);
    // For each instance
```

```
      for (int i = 0; i < m_surfAttrs.numPoints; ++i, ++idx) {
        // Randomize local space position
        V3f lsP;
        lsP.x = rng.nextf();
        lsP.y = rng.nextf();
        lsP.z = rng.nextf();
        // Let s,t be floating-point index
        float s = lsP.x * (numCols - 1);
        float t = lsP.y * (numRows - 1);
        // Interpolate instance attributes
        V3f instanceDensity = SURFACE_INST_INTERP(density, s, t);
        V3f instanceWsP     = SURFACE_INST_INTERP(wsP, s, t);
        V3f instanceWsV     = SURFACE_INST_INTERP(wsVelocity, s, t);
        V3f wsN             = SURFACE_INST_INTERP(wsNormal, s, t).normalized();
        V3f wsDPds          = dPds(s, t);
        V3f wsDPdt          = dPdt(s, t);
        float thickness     = SURFACE_INST_INTERP(thickness, s, t);
        // Offset along normal
        instanceWsP += Math::fit01(lsP.z, -1.0f, 1.0f) * wsN * thickness;
        // Apply noises
        V3f nsP = lsP;
        if (m_surfAttrs.doDensNoise) {
          V3f nsLookupP = nsP / m_surfAttrs.densScale.value();
          float noise = m_surfAttrs.densFractal->eval(nsLookupP);
          float fade = edgeFade(lsP.x, lsP.y, lsP.z);
          instanceDensity *= noise + fade;
        }
        if (m_surfAttrs.doDispNoise) {
          V3f nsLookupP = nsP / m_surfAttrs.dispScale.value();
          V3f disp = m_surfAttrs.dispFractal->evalVec(nsLookupP);
          instanceWsP += disp.x * wsDPds * thickness *
            m_surfAttrs.dispAmplitude.value();
          instanceWsP += disp.y * wsDPdt * thickness *
            m_surfAttrs.dispAmplitude.value();
          instanceWsP += disp.z * wsN * thickness *
            m_surfAttrs.dispAmplitude.value();
        }
        // Set instance attributes
        particles->setPosition(idx, instanceWsP);
        points.setVectorAttr(densityRef, idx, instanceDensity);
        points.setFloatAttr(radiusRef, idx, 0, m_surfAttrs.instanceRadius);
      }
    }
  }
```

Just as curves can be any number of segments, each surface primitive can be made up of an arbitrary number of pieces. Traditionally, each piece is a quad-connected set of vertices, which can be used to create either a regular polygon mesh or a parametric surface. Either way, the `SURFACE_INST_INTERP()` macro needs to evaluate quickly for any coordinate in the patch based on its *st* coordinate.

2.7. Modeling with level sets

Level sets are a technique for tracking interfaces. From its introduction to the computer graphics community in the late 1990s the level set method has quickly become one of the workhorses of the industry. Level sets are useful for collision detection, fluid simulation, and rendering. They are also featured in popular third party applications, such as Houdini and Real Flow.

Typically interfaces in graphics, such as the model of a character, are represented explicitly with polygon meshes or NURBS, for example. There is a rich history of tools and techniques for dealing with such explicit representations. However it is very difficult to implement operations like unions or differences with explicit representations. Additionally, topological changes due to animation need to be handled in special ways which are not robust. The level set method works by representing an orientable manifold surface as a function which tracks the signed distance to the nearest point on the interface from any point in space. In the general case the level set method defines the evolution of the level curve in the normal direction at a certain speed. Most of the time we are interested in the Euclidean distance, which leads to a special case of level sets called signed distance fields (SDF).

Level sets are typically stored in the same volumetric data structures we have been discussing. Each voxel stores the level set value, ϕ, at that location. This is the distance to the nearest point on the interface. As the name signed distance field suggests these values are oriented based on whether the location is inside or outside of an object. For our discussion we assume that level set values outside the object are positive, $\phi > 0$, and negative, $\phi < 0$, inside the object. The zero level, $\phi = 0$, represents the exact interface.

2.7.1. Constructive Solid Geometry Operations

We can extend our voxel buffer machinery with level set specific methods to obtain some really powerful features. The most trivial ones to implement are CSG operations. This pseudocode for a union operation demonstrates one of the reasons behind the viral popularity of level sets: they are extremely trivial to implement.

```
/*! Performs a CSG union between level sets A and B, and
stores the results in A. Assumes A and B have the same transform .*/
void union(VoxelBuffer& A, const VoxelBuffer& B) {
  BBox dims = lightBuffer .dims();
  for (int k = dims.min.z; k <= dims.max.z; ++k) {
     for (int j = dims.min.y; j <= dims.max.y; ++j) {
        for (int i = dims.min.x; i <= dims.max.x; ++i) {
           A.value(i,j,k) = std::min(A.value(i,j,k), B.value(i,j,k));
        }
     }
  }
}
```

The difference between two buffers A and B can be calculated by computing the maximum value at each voxel between value in A and the negated value in B. Intersections between two buffers are computed by taking the maximum value at each voxel. In user interface terms the intersection corresponds to a copy operation, the union is a paste operation, and the difference is a cut operation.

Operation	**Implementation**
Union	min(A, B)
Intersection	max(A, B)
Difference	max(A, -B)

2.7.2. Rendering Level sets

Level sets can be rendered as a solid object, or as a volumetric element. The simplest volumetric treatment assigns a constant density value to each inside voxel, $\phi \leq 0$. In order to avoid aliasing artifacts a roll-off can be applied to the voxels in a band near the surface.

```
phi = levelSet.value( i,j,k );
if ((phi <= 0) && (phi >= -bandwidth))
  density = defaultDensity * smoothstep(-phi, 0, bandwidth);
```

Surface rendering of level sets can be performed directly, or it can be converted back to an explicit mesh. Ray tracing level sets is very efficient because the level set values can be used to accelerate the ray intersection tests. We evaluate the level set value at the start position of the ray. This value tells us how far along the ray we have to advance before we are at the surface. We then evaluate the level set at this new location, and iterate a fixed number of times to find an accurate intersection point if one exists. We can convert level sets to polygon meshes using the popular marching cubes algorithm.

In order to be representable as a level set, an object must have a clearly defined inside and outside. Museth et al. provide discussion of conversion techniques in their paper *Algorithms for Interactive Editing of Level Set Models*. We recommend the excellent text *Level Set Methods and Dynamic Implicit Surfaces* by Stanley Osher and Ronald Fedkiw for more details on level sets and the useful things you can do with them.

2.8. Motion blur

So far we have only considered primitives that are stationary. Of course, to create a production-quality volume renderer we need to consider primitives in motion as well. When it comes to surface rendering, micropolygon-based renderers record the motion per-fragment and assigns a time to each pixel sample. A raytracing-based renderer also assigns a time to each ray, and displaces the contents of the scene so that the ray sees the appropriate state.

In volume rendering, true motion blur is often too expensive to calculate. In some cases, such as eulerian motion blur of procedural fields and simulation data, the motion blur calculation can be done correctly. However, when considering thousands or millions of volumetric primitives we simply cannot produce correct motion blur – in fact, the use of rasterization into voxel buffers prevents it.

The most common solution to producing *almost correct* motion blur in voxel buffers is to smear each sample along its motion vector. Smearing has the following properties: It distributes the value evenly across all the voxels it touches, and the sum of all values written to those voxels is equal to the original value.

2.8.1. Line drawing

The first approach we can use to smearing the sample is to employ standard line-drawing in 3D. In order for the motion blur to look smooth we need to antialias the line. Fortunately, algorithms for drawing an antialiased line are commonplace in computer graphics, and we refer the reader to the standard literature for implementation details.

2.8.2. Splat-based smearing

The second approach is to draw multiple trilinear splats to make up the line. This has the benefit of being easier to implement, and as we'll see below it also introduces the opportunity to control the quality of the smear.

```
void writeLine(const Vector &vsStart, const Vector &vsEnd,
               const Imath::V3f &value, VoxelBuffer::Ptr buffer)
{
  using namespace std;
  using namespace Imath;
  using namespace Field3D;
  Vector vsLine = (vsEnd - vsStart);
  size_t numSamples = static_cast<size_t>(std::ceil(vsLine.length()));
  numSamples = max(static_cast<size_t>(2), numSamples);
  for (size_t i = 0; i < numSamples; ++i) {
    double fraction = static_cast<double>(i) /
      static_cast<double>(numSamples - 1);
    Imath::V3f sampleValue = value / static_cast<double>(numSamples);
    Vector vsP = Imath::lerp(vsStart, vsEnd, fraction);
    writeAntialiasedPoint(vsP, sampleValue, buffer);
  }
}
```

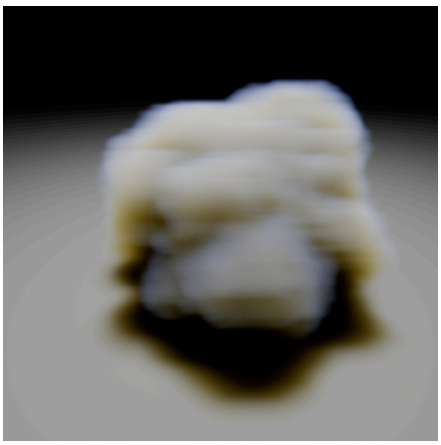
By calculating the fraction of the distance travelled instead of incrementing the position at each step of the loop we avoid accumulation of errors in the splat positions.

Using splats has another interesting possibility – undersampling. Though we know how many samples we *should* use, we could potentially use fewer to speed up the rasterization. Just as with nearest-neighbor splatting, when a lot of primitives are involved, their random distribution tends to hide undersampling and noise artifacts. Thus, we may add a scaling factor to the function, which lets the user control how many samples should be used to draw each line.

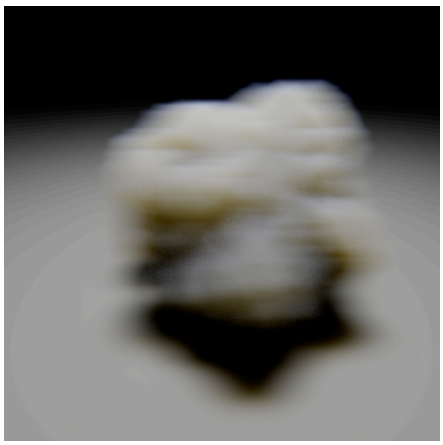
```
size_t numSamples = static_cast<size_t>(samplingFactor * std::ceil(vsLine.length()));
numSamples = max(static_cast<size_t>(2), numSamples);
```

2.8.3. Smearing problems

Of course, smearing the samples before lighting is computed is technically incorrect. In an abstract sense, we are folding the temporal domain into the spatial domain, and in doing so we lose all information about when a given primitives occupies a given position in space. This problem becomes apparent in the loss of lighting detail during subsequent rendering. A sharp feature that is smeared will no longer shade the same as when stationary, and the result tends to look artificially soft. However, this downside is usually acceptable when considering the alternative of calculating full deformation blur during rendering.



Smeared primitive produces incorrect lighting



Correct result

We find another problem if the camera is moving at the same speed as the primitives being rasterized, any motion should be cancelled out and the result look sharp. But because the motion blur is baked into the voxel buffer this is not possible.

2.8.4. Post-rasterization smearing

An alternative to smearing each individual sample is to use a separate buffer to accumulate a velocity vector for each voxel. Once all rasterization and/or splatting is done, the velocity is used to smear the entire buffer in a single step. This is often faster than smearing each sample as it is written to the buffer, but it suffers from a potentially large problem, depending on the input geometry. The problem occurs when the input primitives overlap and have drastically different motion vectors. In this case it becomes impossible to calculate a valid direction to smear in. It is possible to resort to keeping track of the average motion vector in each voxel that has overlapping primitives, but this can cause visual artifacts in the final render.

A variant of this method is to simply retain both the density buffer and the velocity buffer and calculate the motion blur during rendering. This still suffers from the problem with overlapping primitives but does avoid the problem of reduced shading detail in motion blurred areas. Microvoxel-based volume renderers lend themselves well to this approach.

A third approach is possible if the lit volume itself is voxelized. In this case, motion blur can be applied after. This solves the problem of lighting a voxel buffer with motion blur baked in, but can still show artifacts where overlapping primitives have different velocity vectors.

2.9. High resolution voxel buffers

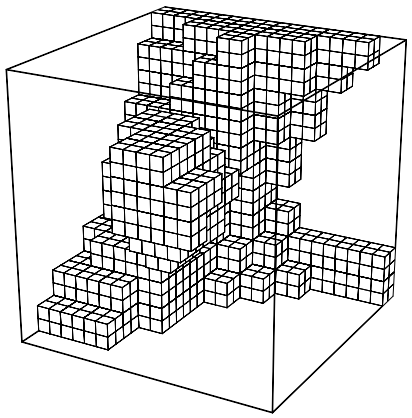
Up until now we've expected that voxels exist everywhere within the domain of the voxel buffer. And in this domain each voxel is the same size. This is fine when the volumetric element that is being modeled is small in screen-space, but if we need to get close to the element, or if the element extends across the entire visible frame, the resolution required in order to provide sharp details will likely range in the thousands along each axis.

Two approaches are most common in visual effects production when trying to solve this problem. The first addresses the problem of unused voxels occupying memory, and the second amounts to adapting the voxel size so that voxels close to camera are small and those far away are large.

2.9.1. Sparse data structures

Any time a dense voxel buffer stores a zero density it is effectively wasting memory. This happens because dense buffers blindly allocate storage for every voxel in its domain without considering what areas will be populated. Since most volumetric elements tend to have some sort of connectedness and generally don't occupy the entire domain of the voxel buffer, finding a data structure that allocates memory more intelligently would help improve memory use.

One of the simplest such structures is the *block-based sparse* buffer. (What is referred to here as *blocks* is sometimes also called *tiles*.) It can be thought of as a two-level-deep hierarchical data structure where the domain of the buffer is subdivided into coarse *blocks*, and where a *block* can contain either a single value (usually zero, though for storage of level sets it can be useful to assign a different value), or an N^3 array of voxels, representing the actual voxel data in the block.



A sparsely allocated voxel buffer (with unallocated blocks hidden)

The illustration above shows a sparsely allocated voxel buffer with blocks of size 2^3 (for purposes of clarity). A more common block size would be between 8^3 and 32^3.

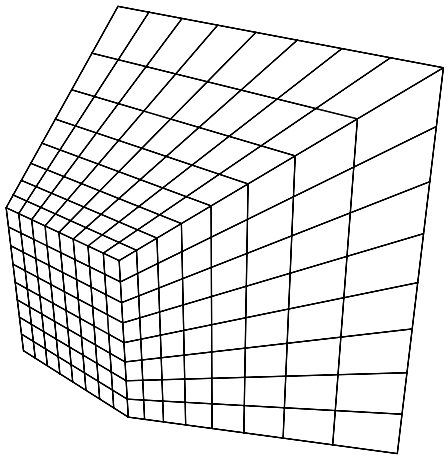
A block remains unallocated (storing just a single value) until the first write-access to one of its voxels. Once the first write happens, all of the block's voxels are allocated. Each block is thus effectively its own, small, dense buffer once allocated.

Using a fixed-depth hierarchy means that voxel read and write access is *O(k)*, or constant time, though with a larger *k* than an ordinary dense buffer. The allocation that happens on the first write access is amortized over all subsequent accesses.

Field3D provides an implementation of this type of data structure in its `SparseField` class.

2.9.2. Frustum-shaped voxel buffers

The second approach is to adapt the voxel size to account for the fact that objects far away from camera require less detail than those close up. The most common way of accomplishing this is to use frustum-shaped voxel buffers, usually referred to simply as *frustum buffers*. Frustum buffers are tied to a camera (usually the main shot camera), and follow any animation applied to the camera. When seen from the side (below), each voxel looks stretched and sheared, but when viewed from the camera, each row of voxels lines up perfectly with the projection of a pixel into the scene.



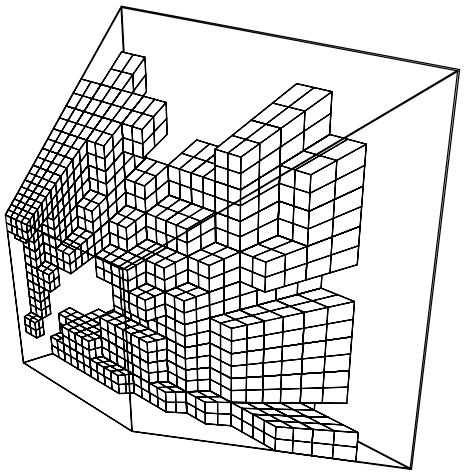
A frustum-shaped voxel buffer

The resolution of the above buffer is 8x8 in the *XY* plane, with 8 *slices* along the *Z* axis. The *XY* resolution is normally locked to the resolution of the camera, times some multiplier, and the number of *Z* slices is left as a user parameter. The *Z* resolution is normally much lower than either of the *XY* axes, usually on the order of a few hundred.

The transform from from world space to voxel space can be implemented as:

```
void FrustumMapping::worldToVoxel(const V3f &wsP, V3f &vsP)
{
  // The camera's 0..1 NDC space matches the local space of the voxels
  V3f nsP;
  m_camera->worldToNdc(wsP, nsP);
  localToVoxel(nsP, vsP);
}
```

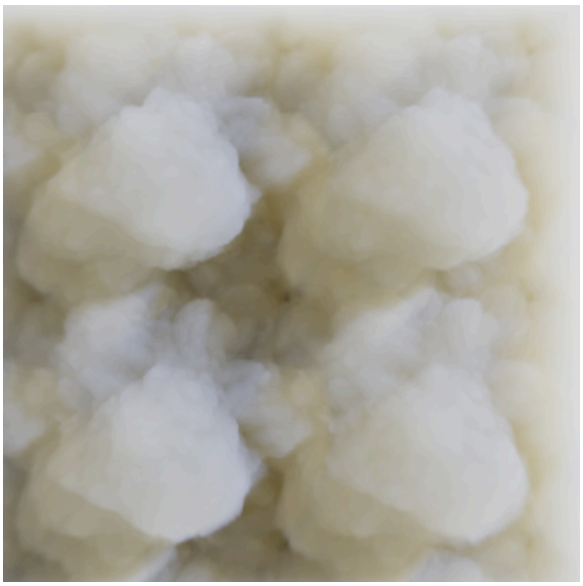
To further optimize memory use we can combine sparse buffers and frustum buffers. Since the coordinate transform is independent of the data structure used for voxel storage, this is straightforward. This approach combines the benefit of finer detail close to camera with the empty space optimization of the sparse buffer.



A sparsely allocated frustum-shaped voxel buffer

2.9.3. Problems with frustum buffers

Frustum buffers are not without drawbacks. *Light leaks* may be visible along the edges of the frame during rendering, since no density is available outside the view of the camera to stop light from penetrating into the buffer. This can usually be addressed by padding the bounds of the frustum buffer, so that it extends outside the view of the camera, though in extreme cases the amount of padding needed negates the performance gains and reduced memory usage offered by the frustum buffer.



Light leaks along right edge of frustum

Another problem occurs when primitives in the scene are widely distributed along the camera's depth axis. This forces each *Z* slice to become excessively deep, which manifests itself as aliasing artifacts, called *slicing artifacts*. These are visible as a posterization-like look, with poor lighting detail. The artifacts can be reduced by careful antialiasing of the volumetric primitives during the rasterization phase, but to completely avoid them an increase in *Z* resolution is required.

Sufficient detail along z axis (150 slices)

Slicing artifacts due to low z resolution (25 slices)

Frustum buffers are also more prone to aliasing due to the noise functions used and if an insufficient number of *z* slices is used the effect can be very visible. The example below shows a render using the same 25 slices as above, but with noise antialiasing disabled:

Aliasing artifacts due to excessive high-frequency detail in noise function

It can also be difficult to correctly capture moving primitives at the edge of the frustum. Primitives that motion blur into the buffer need to be considered even if only a few samples of their smeared contribution fall into the buffer, otherwise leaks will occur due to the loss of density, similar to the light leaks discussed earlier. In certain cases it may also be difficult and/or expensive to determine if a primitive should be included, for example if a primitives both enters and exits the frustum buffer in a single time frame.

3. Volume rendering

Now that we have the foundations in place for creating the data for volumetric effects we move on to looking at how to render the data. Volume rendering is about the mathematics of how light behaves in a *participating medium*, where the medium may be anything from smoke or water vapor to clouds and atmospheric haze. The equations that govern volume rendering are applicable across a large range of media, and we can use the same approach to render almost any kind of volumetric effect.

Volume rendering for visual effects production also extends past the physical in order to achieve certain desired looks, and to better integrate with the rest of the production pipeline. It often becomes necessary to bend the rules for what *should* happen, and the constant challenge is to find methods for doing so that are controllable yet plausible.

This chapter will describe the fundamental components required for creating a production-grade volume renderer, and will cover basic scattering theory and the raymarching approach to solving the scattering problem. It will also cover efficiency and optimization, integration issues and the challenges associated with motion blur. Most topics will be familiar to those familiar with volume rendering in general, but in this course we aim to describe which specific techniques are used in day-to-day production environments, and how those techniques are integrated to create a practical and functioning system. There are of course other approaches to volume rendering than those described here, and several different ones are actively used in the visual effects community as well, but in these course notes we will focus on the raymarching-based approach.

3.1. Lighting theory

When designing a volumetric effects we want to describe both its shape and motion (the *modeling* part of volume rendering), as well as its appearance (the real *rendering* part of volume rendering). When describing the appearance it is useful to break it down into some fundamental *characteristics*, which can then be combined in various relationships to achieve a wide variety of looks.

As it turns out, there are only three fundamental characteristics needed to describe any given volumetric element. Each describes a different physical process.

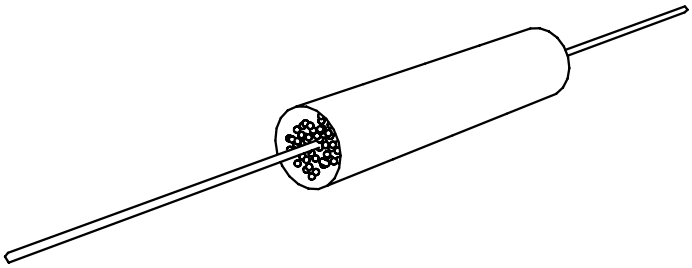
- *Absorption* is the loss of radiant energy along a ray of light due to energy being converted into some other form, such as heat. Black smoke is a good example of an *absorbing* medium.
- *Emission* adds radiant energy and happens where the medium itself is luminous. Flames and fire are examples of *emissive* media.
- *Scattering* describes how likely a medium is to cause a ray of light to collide with a particle and change its direction. As an example, water vapor is an almost completely *scattering* medium. There are two types of scattering to consider. First, light traveling from a distant object towards the camera has a probability of being reflected off to another direction, which is called *out-scattering*. Another possible outcome is that a ray of light traveling in some random direction gets reflected into the view ray of the camera, which is called *in-scattering*. Each of these probabilities is equally likely to occur, since the light being reflected has no idea of the position and orientation of any observers.

A purely scattering volume

A purely absorbing volume

A purely emissive volume

Each of these characteristics can be isolated and discussed in terms of how they affect a ray of light traveling through space. When deriving the mathematical model for how light is affected by participating media it is useful to consider a differential cylinder: a cylinder, infinitely thin, of unit length, through which a ray of light passes.



A differential cylinder filled with a participating medium

We will use the following notation in the equations that follow:

- p – the position of the cylinder, and the interaction location
- $\boldsymbol{\omega}$ – the direction of the ray
- L – the radiance quantity
- L_i – the incoming radiance, before any interaction with the medium
- L_o – the outgoing radiance, after the interaction with the medium

3.1.1. Modeling absorption

In discussing the light scattering properties of volumes we will draw some parallels to surfaces and their BRDFs. A dark surface is dark because of its low reflectivity. The laws of physics dictate that whatever incident radiant energy that is not reflected away from the surface must be absorbed – energy does not disappear, it only changes form, in this case into heat. Volumes share this property, but instead of describing the fraction of light absorbed after interaction with a surface the absorption coefficient determines how likely it is that a ray of light is absorbed as it travels through a volume.

The unit of the absorption coefficient σ_a is a reciprocal distance (i.e. m^{-1}), which essentially describes the probability that a ray of light is absorbed as it travels through one length unit of the medium. Being a reciprocal means that it can assume any positive value – it can be infinitely large.

Mathematically formulated, the absorption interaction can be described as:

$$L_o = L_i + dL$$
$$dL = -\sigma_a L_i \, dt$$

3.1.2. Modeling emission

When the medium emits light, for example as the result of some chemical reaction, or due to the thermal properties of the medium, it adds to the radiance of a ray passing through it.

The emissive term $L_e(p, \omega)$ is a measurement of radiance, which describes the amount emitted in direction ω along a one unit long section of a ray.

The mathematic formulation for emission is:

$$L_o = L_i + L_e$$
$$L_e = \sigma_e \, dt$$

3.1.3. Modeling scattering

The scattering property describes the likelihood that a ray of light traveling through the medium will bounce and reflect to another direction. As mentioned before, this interaction accounts for both in-scattering and out-scattering, although when calculating lighting effects in a volume, the effect of out-scattering is usually folded into the absorption calculation, since the net result is identical. We refer to the *extinction term* when considering absorption and out-scattering together.

The unit of the scattering coefficient σ_s is (just as absorption) a reciprocal distance, which describes the probability that a ray of light is scattered as it travels through one length unit of the medium. This means that a ray traveling through a medium with $\sigma_s = 0.1$ will travel on average a distance of $0.1^{-1} = 10$ units before a scattering event occurs. This distance can also be referred to as the *scattering length*.

Given a light source S, whose function $S(p, \omega')$ describes the quantity of light arriving at point p from direction ω', we can formulate the scattering interaction as:

$$Lo = L_i + dL_{in} + dL_{out}$$
$$dL_{in} = \sigma_s \, p(\omega, \omega') \, S(p, \omega')$$
$$dL_{out} = -\sigma_s \, L_i \, dt$$

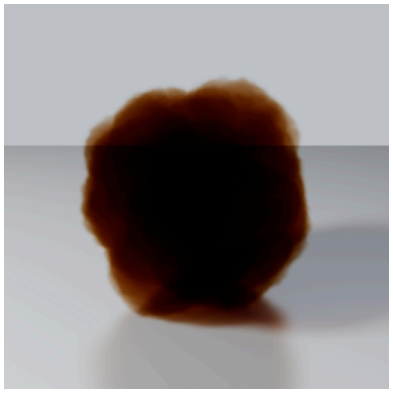
The function $p(\omega, \omega')$ is called the phase function, and the next section will detail what it is and how it affects the scattering interaction.

3.1.4. Wavelength-dependency

Each of the physical properties of a volume may be wavelength dependent, where the amount of scattering/absorption/emission varies across the color spectrum. Scattering in particular produces interesting results as the color of light shifts to the complementary as it penetrates deeper into a volume. The effect occurs naturally, especially in mixed/polluted media such as smoke, but also in the atmosphere itself[7].

Color balance can shift as light penetrates through smoke

Scattering coefficient: 0.5,0.7,1.0



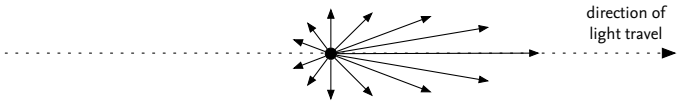
Absorption coefficient: 0.4,0.5,1.0

3.1.5. Phase functions

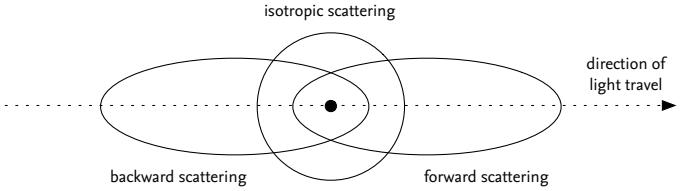
The property of a volume that relates closest to a surface BRDF function is the *phase function*. A BRDF defines how much of light hitting a surface while traveling in direction $\boldsymbol{\omega}$ will scatter to direction $\boldsymbol{\omega}'$, and similarly the phase function determines how much light traveling through a medium in direction $\boldsymbol{\omega}$ will,

[7] The sky is blue due to the wavelength-dependent scattering behavior of the atmosphere.

upon scattering, reflect to direction ω', i.e. *probability* = $p(\omega, \omega')$. Phase functions (at least the ones relevant to our purposes) have a few important properties. First, they are isotropic, meaning that the function is rotationally invariant, and only the relative angles between ω and ω' need to be considered, thus we can write $p(\omega, \omega') = p(\theta)$. Second, they are reciprocal, so $p(\omega, \omega') = p(\omega', \omega)$. Third, they are normalized such that integrating across all angles for ω while holding ω' constant gives exactly 1.



The length of each vector illustrates the probability of scattering in that direction



Isotropic and anisotropic phase functions

Phase functions come in two flavors, *isotropic* and *anisotropic*. An isotropic (not to be confused with isotropic in the rotationally invariant sense, as in the previous paragraph) phase function scatters lights equally in all directions. Anisotropic phase functions are biased either forward or backward, as seen from the direction of light travel before the scattering event.

Isotropic phase functions are perfectly sufficient when rendering low-albedo media, such as ash clouds, dust etc., but for media such as clouds and atmospheres, anisotropy is an important element to include in lighting calculations. Anisotropic behavior in participating media can be thought of as the parallel to specular BRDFs, and isotropic to diffuse/lambertian BRDFs. In everyday life, anisotropy is responsible for the *silver lining* in clouds, where the edge of a cloud becomes increasingly bright as the sun reaches a grazing angle.

Phase functions are well researched, and the two most common ones are the *Rayleigh* model (which describes atmospheric scattering, the interaction of light with particles the size of molecules), and the *Mie* model (which is more general and can handle much larger particle sizes, for example water vapor and droplets suspended in the atmosphere). In production rendering we often use a few other, simpler models, since Rayleigh and particularly Mie are expensive to evaluate. *Henyey-Greenstein* is a simple model that can handle both isotropic and anisotropic media, and a similar, but cheaper one is the *Schlick* phase function. These functions can all be found in the standard literature, but one especially good overview of phase functions in the context of volume rendering can be found in the book *Physically Based Rendering*[8].

[8] *Matt Pharr & Greg Humphreys – Physically Based Rendering (Morgan Kaufmann publ.)*

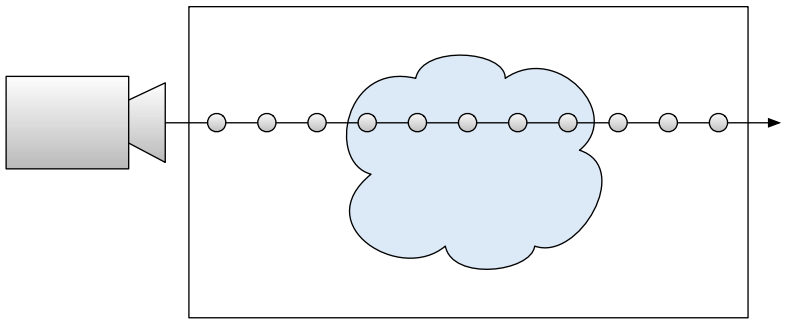
3.2. Raymarching

Transmittance is the fraction of light which passes through a volumetric sample after being impeded by the material in the sample. It is the ratio of the outgoing light to the incoming light. Beer-Lambert's Law relates the absorption capacity of a material to the transmittance. We write this equation as: $T = e^{-\sigma\rho l}$.

- T is transmittance
- σ is the coefficient of absorption
- ρ is amount of absorbing material in the sample
- l is the length of the path through the sample.

Opacity is the fraction of light which is absorbed by the sample. $\alpha = 1 - T$. With respect to rendering, opacity is accumulated to compute the alpha channel.

As in any other kind of rendering we want to figure out the light that gets to the camera. Real participating media attenuates the light, modulates the frequency, and alters the path of light. In other words a lot of complicated physical processes which we are going to simplify. This simplification was introduced by Kajiya and Von Herzen in 1984 in their seminal SIGGRAPH paper *Ray Tracing Volume Densities*. The process is to trace a ray from the cam- era through the volume, and compute the illumination in small segments along the ray. The illumination at each segment must be attenuated by the density between the camera and the segment. The ray is traced until the ray exits the volume, or it can see no further into the volume. The illumination calculation at each segment requires determining how much light is reaching that segment. This implies that we must perform another raymarch from each light to the segment. We must also account for material properties such as albedo, and the light absorption capacity of the material. This is a very simplified process, and does not account for some common effects such as volumes that emit light, such as fire, or scattering effects. But this is a good starting point.



In order to compute the pixel value for a ray, $\boldsymbol{x}$, we initialize color, C and opacity, α, to zero. Transmittance, T, is initialized to 1. We then intersect the ray against the voxel buffer to determine where we need to perform the integration. Finally we integrate this interval by sampling along $\boldsymbol{x}$ in steps of length $\triangle x$. The contribution of a sample i is:

$$
\begin{aligned}
T_i &= e^{-\sigma\rho\triangle x} \\
T_{total} &= T_{old} * T_i \\
C_i &= T_{total}\, L(\boldsymbol{x}_i) c(\boldsymbol{x}_i) \rho(\boldsymbol{x}_i) \triangle x \\
C_{total} &= C_{old} + C_i \\
\alpha_{total} &= \alpha_{old} + (1 - T_i) * (1 - \alpha_{old})
\end{aligned}
$$

where

T_i : *Transmittance at* $\boldsymbol{x}_i$

T_{total} : *Total transmittance from the start of the ray to* $\boldsymbol{x}_i$

$L(\boldsymbol{x}_i)$: *Incident lighting at sample location*

$c(\boldsymbol{x}_i)$: *Color of the material at sample location*

$\rho(\boldsymbol{x}_i)$: *Density at sample location*

$\triangle x$: *Ray step length*

α : *Opacity*

In order to compute the incident lighting we need to compute the transmittance from the light to the sample position. This requires us to perform another ray march between the sample position and the light. Employing the same mechanism used above:

$$
\begin{aligned}
T_i &= e^{-\sigma\rho\triangle x} \\
T_{total} &= T_{old} * T_i \\
L &= T_{total} * C_{light} * P(\theta)
\end{aligned}
$$

where

C_{light} : *Light color*

$P(\theta)$: *Phase function*

The following method is a simple integration module. Note that no lighting calculations are performed.

```
float Raymarcher::integrate(const V3f& pos, const V3f& dir, const float absorption) const
{
    // Determine intersection with the buffer
    float t0, t1;
    if (false ==  m_buffer->intersect(pos, dir, t0, t1))
        return 1.0f;

    // Calculate number of integration steps
    const int numsteps = int(ceil(t1 - t0) / m_stepsize);

    // Calculate step size
    const float ds = (t1 - t0) / numsteps;
    V3d stepdir(dir);
    stepdir *= ds;

    V3d raypos(pos);
    raypos += stepdir;

    const float rhomult = -absorption * ds;

    // Transmittance
    float T = 1.f;

    for (int step = 0; step < numsteps; ++step) {
        float rho = m_buffer.trilinearInterpolation(raypos);
        T *= std::exp(rhomult * rho);
        if (T < 1e-8)
            break;
        raypos += stepdir;
    }

    return T;
}
```

In order to accelerate the lighting calculation we precompute the transmittance values from each light through the volume. During rendering we interpolate this data set to determine lighting at the sample point. This is discussed in further detail in the section on pre-computed lighting.

It is often necessary to model volumetric materials which are emitting light, such as fire. Such materials are like diffuse densities, except that we can consider them to have intrinsic lighting. The rendering procedure involves augmenting the lighting function, $L(\boldsymbol{x}_i)$, with an additional source for the emissive light.

3.2.1. Artistic controls

This simple illumination model offers a lot of room for art direction. The constants which appear in the expressions above are the simplest of these controls. While these parameters are rooted in physical accuracy we need not keep to this constraint. The absorption coefficient, σ, can be different between the

lighting and rendering calculations. It can also be varied for each color channel for a complex lighting effect.

We have found it useful to provide a multiplier for the density, ρ, parameter. This provides the user a simple way to manipulate the source data at render time. The step length, $\triangle x$, parameter is the simplest way to trade off between quality and rendering speed. Large step sizes means the ray integration has to consider a lot fewer samples, and is quicker to compute. Consequently it also fails capture all of the detail available in the voxel buffer. The concept of frustum buffers is based on the observation that we may not require high quality integration far away from the source of the ray. Step length is typically expressed in world space units.

3.2.2. Implementation

The modular nature of the ray marching algorithm hints at how we can implement a generic volumetric shading architecture. In a simple scheme we can have a main renderer, material shader plugins, and light shader plugins. The main renderer is responsible for rendering a given voxel buffer. The user specifies which material shader to apply to that buffer, as well as the lights. The renderer invokes the ray marching, and performs the integration. At each sample along the ray the material shader is called with the shading position, voxel buffer, and information about the lights. The material shader is then responsible for returning a shaded color and transmittance for a single sample. This requires that the shader loop over all the lights and invoke the light shaders. In the case of using precomputed lighting the method to obtain lighting from the light shaders can simply return the appropriate value for the given sample position.

3.3. Pre-computed lighting

Lighting calculations are one of the computationally heaviest steps of the raymarching algorithm. Since at each step we need to sample incoming light, the number of samples needed quickly approaches the billions. A completely unbiased approach to lighting would, for each raymarch step (of which there are at least 100 per pixel) along a primary ray, perform another raymarch toward each light in the scene to determine the amount of occlusion (multiplying the number of samples by another 100 or so). A brute force implementation of this scheme is incredibly slow, and as such it is mostly useful as a way of verifying the result of other techniques. Production renderers usually use other approaches to speed up the calculation, most of which rely on pre-computation.

3.3.1. Voxelized lighting

One method that is easy to implement and allows for very fast lookup times is to simply sample and voxelize the incoming light across the full domain of the scene. Since this is decoupled from the sample density of the camera rays, it is possible to sample less frequently than the final raymarch, and then interpolate values from the voxel representation. While this approach allows for fast lookups during final rendering, the pre-computation can be expensive, as for each voxel a full raymarch toward each light is performed.

```
void voxelizeLights(const Scene &scene, const std::vector<Light> &lights,
                    VoxelBuffer &lightBuffer)
{
  BBox dims = lightBuffer.dims();
  for (int k = dims.min.z; k <= dims.max.z; ++k) {
     for (int j = dims.min.y; j <= dims.max.y; ++j) {
        for (int i = dims.min.x; i <= dims.max.x; ++i) {
           V3f vsP = discreteToContinuous(i, j, k);
           V3f wsP;
           lightBuffer.mapping().voxelToWorld(vsP, wsP);
           Color incomingLight = 0.0f;
           for (int light = 0; light < lights.size(); ++light) {
              float intensity = lights[light].intensity(wsP);
              // Raymarch toward light to compute occlusion. This is a costly operation.
              float occlusion = computeOcclusion(lights[light].wsPosition(), wsP);
              incomingLight += intensity * (1.0 - occlusion);
           }
           lightBuffer.lvalue(i, j, k) = incomingLight;
        }
     }
  }
}
```

The biggest downsides to voxelized lighting is the cost of pre-computation. Because each voxel is calculated separately, we suffer similar performance problems as brute-force calculations, although we gain control over the sampling density (i.e. the resolution of the voxelized lighting buffer), which can be used to tune the quality/speed trade-off.

3.3.2. Deep shadows

One important aspect of the lighting calculation that isn't taken advantage of in the voxelized lighting approach is that light travels linearly from each light source. If the incoming light at voxel V1 in the illustration below has been computed, then the incoming light calculation at voxel V2, V3, etc. could potentially use the occlusion value at V1 to speed up their calculations. In practice however, figuring out which values can be re-used quickly becomes difficult and incurs its own performance overhead from storing book keeping data.

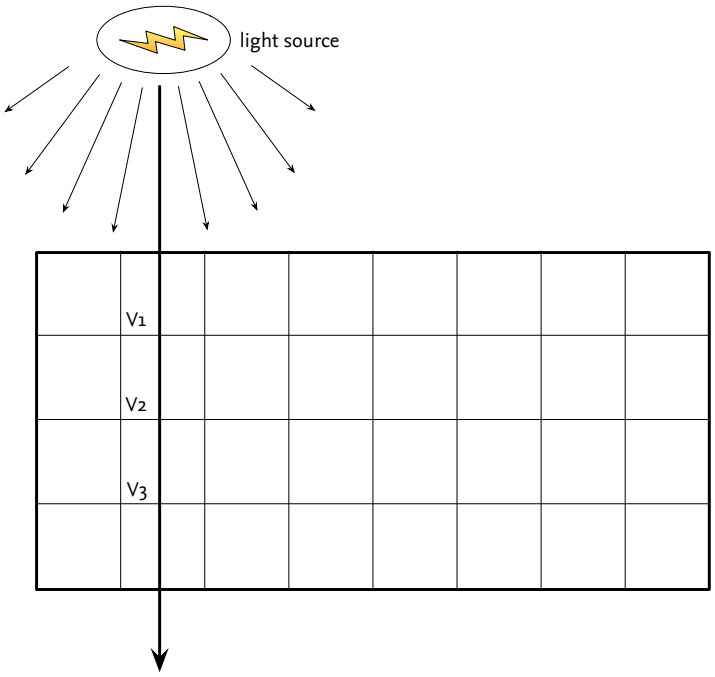
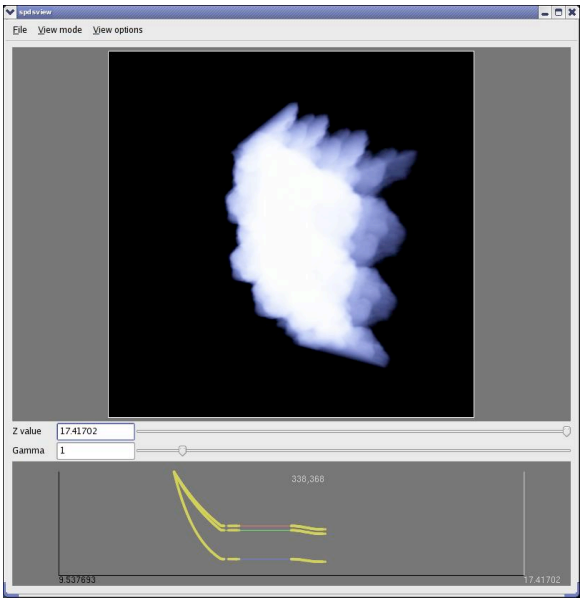


Illustration of light propagation

A better approach is to simply calculate occlusion as seen from the light. This is equivalent to how *shadow maps* work, and it requires us to choose a resolution controlling how finely the scene is to be sampled. For each pixel in this map, we then need to calculate the transmittance function.



Visualization of a deep shadow map, with spectral transmittance function for a pixel displayed below

This turns out to be trivial, as it is the exact same calculation that we perform when raymarching a volume from the camera and only accumulating transmittance/opacity. The only difference is that we need to record what the accumulated transmittance is at each raymarch step, so that this can later be queried for any point in space that is visible from the light source. The simplest way of storing the data is as a sequence of depth/transmittance pairs, ordered away from the light.

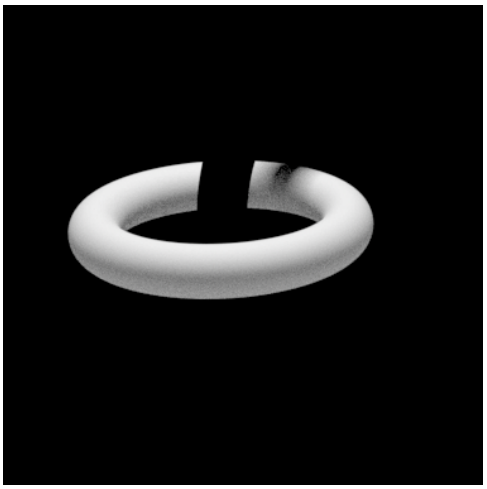
Technically, we could voxelize this data set and use it in beauty renders the same way as in the previous section, but a more efficient way to store this data is to leave it in its native form, i.e. a monotonically decreasing function per pixel seen from the light source. When storing it in this way, it is equivalent to the *deep shadow maps* technique described by Tom Lokovic and Eric Veach in their paper *Deep Shadow Maps* [Lokovic, 2000].

Deep shadow-style maps have several advantages. The transmittance function is mostly smooth and can be compressed efficiently to reduce the storage required both in memory and on disk. Also, since the number of samples in the transmittance function can vary from pixel to pixel, there is little cost associated with storing empty pixels. The downside is that the cost of lookups into the transmittance function are higher than for a voxelized representation, because each lookup requires a search in the transmittance function vector in order to find the appropriate depth sample.

3.4. Holdouts

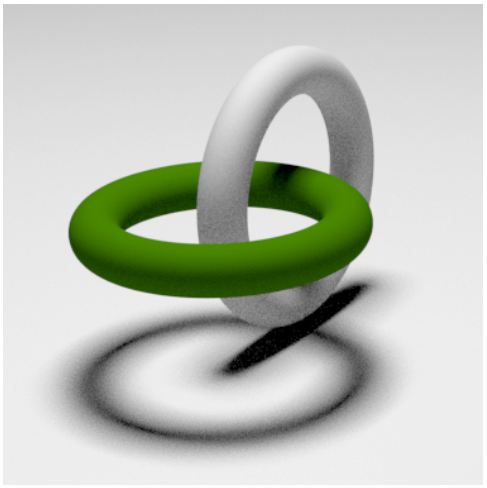
Volumetric elements rarely exist in isolation, and in almost any shot there are other elements which the volumetric must integrate with in the final composite. *Holdouts* are a common way of dealing with integration of multiple elements, both in surface and volume rendering. A holdout is an object in the scene that occludes and shadows other objects but does not itself show up in the final frame. *Matte objects, phantom objects* and *holdout objects* are all different names for the same thing.

Ground and object A visible, object B as holdout



Object B visible, ground and object A as holdouts

Composited image (additive)



Color correction applied to object B's image

Holdouts allow a render to be broken into multiple images that can be manipulated individually, and that composite easily into a final frame. They also allow the breaking down of complex scenes, as object occlusion is handled per-pixel, without the need to know the correct depth sort order. In the example

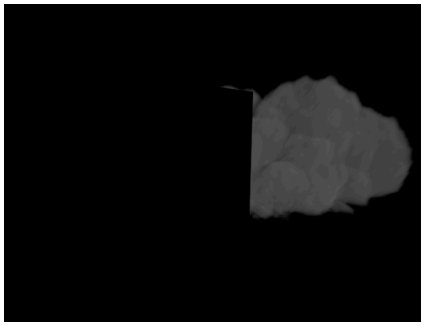
above, separating objects A and B without the use of holdouts would be tedious, as they both occlude each other in different parts of the image.

Holdouts are fairly trivial to implement in a surface shader, but in volume rendering, they can be implemented in a number of different ways, depending on the rendering approach and the type of holdout types that are supported.

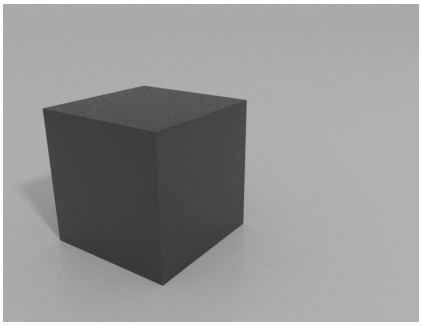
3.4.1. Holdouts in volume rendering

In the microvoxel-based renders that handles both surfaces and volumes (such as PRMan or Mantra), geometry-based holdouts are a convenient way of defining holdout objects. For a raymarch-based volume renderer however, geometric holdouts are somewhat cumbersome to deal with. In raymarching, holdout objects need to be able to answer a visibility query for any point in space along a ray, as seen from the camera's position, which means a ray must be traced against the geometry each time the information is required. To make things even more expensive, the holdout value should represent the pixel coverage or transparency at a given depth, so the holdout value usually needs to be supersampled and jittered across the pixel, because a simple binary visible/hidden answer will cause aliasing artifacts at geometry edges.

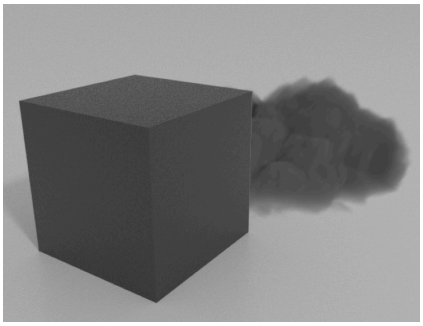
We can take advantage of the fact that this holdout query is the same form of query discussed in the precomputed lighting section, where a *transmittance function* determined how much light could travel between a light source and a given point in space. Holdout functions answer the inverse question: how much light could travel from a given point in space all the way to the camera?



Volumetric element with surfaces held out



Rendered image of surfaces



Composite using over operator

3.4.2. Implementing holdouts

If we consider the calculation of transmittance in the raymarch loop

```
T *= exp(-tau * stepLength);
```

We can rearrange this expression in the following way

```
T = T * exp(-tau * stepLength)
T = 0 * (1 - exp(-tau * stepLength)) + T * exp(-tau * stepLength)
T = lerp(0, T, exp(-tau * stepLength))
```

Which is to say that our incremental multiplication of T may be seen as a linear interpolation (or *lerp*) between zero and T by the factor $exp(-\tau * stepLength)$. While this is rather useless in itself, it becomes a much more convenient formulation once we introduce holdouts.

A holdout object in volume rendering only needs to answer one question: at a given point in the scene, how visible is that point to the camera? That is to say

```
Color Tholdout = holdout.transmittance(wsP);
```

Logically, this function should start at 1.0 and be monotonically decreasing for any line pointing away from the camera, since once an obstacle is found its reduction of transmittance cannot be undone. If we consider the zero value in our `lerp()` expression above, that corresponds to the expected result of a holdout object that occludes nothing, i.e.

```
Tholdout = 1;
opacity = 1 - Tholdout;
opacity = 0.
```

As it turns out, accounting for the holdout object is as easy as replacing the zero in the lerp expression with `1 - Tholdout`. So the complete formulation of the raymarcher's transmittance calculation becomes:

```
Tholdout = holdout.transmittance(wsP);
T = lerp(1.0 - Tholdout, T, exp(-tau * stepLength));
```

If we were to create a procedural holdout function that emulated a semi-transparent glass pane right in front of the camera lens, we might expect it to always return 0.5, for example. Logically, this would be reflected in our final alpha and color values by making them half as large.

3.4.3. Holdout maps

In the previous chapter we saw that deep shadow maps are a convenient way of storing transmittance functions, in fact the information that is required of the holdout function is exactly the same information used in lighting calculation. One solution to supporting geometric holdouts is therefor to create a transmittance function for each pixel in the final output by raytracing any holdout geometry as a pre-process to the volume integration step. Also, users of some common surface renderers (for example Pixar's PRMan and SideFX's Mantra) can output a deep shadow or deep shadow-like representation of

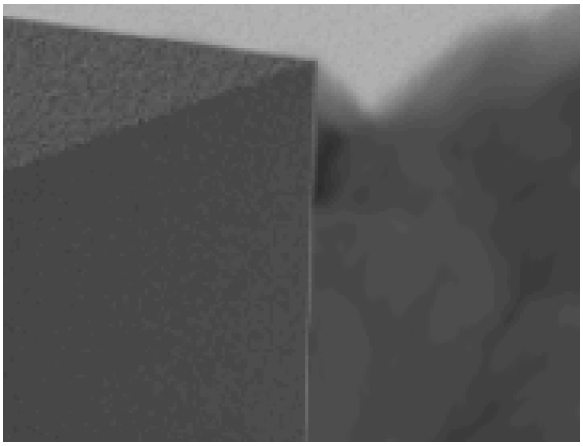
depth-varying pixel coverage or transmittance, which can then be used directly in a stand-alone volume renderer.

3.4.4. Problems with holdouts

In our first holdout example we saw how two interlocking rings could be held out against each other and then composited correctly using an additive operation. This type of render is a *two-sided holdout*, where each element is held out in all other images being rendered. When using a dedicated volume renderer to produce elements that need to composite against images from a surface renderer this becomes a catch-22, because the surface renderer would need to be aware of how to render the volume data as a holdout object, though of course the whole point of writing a standalone volume renderer was that we couldn't (or didn't want to) do it inside the surface renderer.

Because of this we often have to resort to *one-sided holdouts*, where the surfaces are rendered in their entirety and only the volume rendered image has objects held out. One-sided holdouts are technically inaccurate, but can be composited with reasonable results using an *over* operation instead of an add. The composite is correct for all pixels that have either no holdout or full holdout effect, but breaks down where partial occlusion happens.

To illustrate this we look closer at the previous example:



Light pixels bleeding through holdout edge

As we can see, the high intensity pixel values of the ground are creeping into pixels that should really only contain values from the dark box and the smoky volume. This is often referred to as a *matte line*, and shows the breakdown of one-sided holdouts. They occur because the high intensity values are already mixed with the foreground elements into the antialiased edge pixels, and there is no way to tell what portion of the pixel's value was contributed by values at any given depth; that information has been lost.

One way to work around one-sided holdouts is to output a *deep image* from both the volume renderer and the surface renderer, and do a *deep composite* of the two in a 2D application.

3.5. Motion blur

In the modeling section we discussed techniques for motion blur of volumetric primitives and how that is commonly treated at the rasterization step rather than at render time. Here we will introduce motion blur in a few new contexts, all of which need to be handled by a production grade volume renderer.

Calculating perfectly accurate motion blur is expensive (this is why we voxelized our volumetric primitives and smeared them to begin with), and implementation techniques tend to vary from facility to facility. The techniques range from brute-force solutions to clever tricks that fake the effect, though in production rendering each one can be a perfectly valid solution. Because the solutions vary from facility to facility, we will introduce the topic in this section, but the solutions will be discussed in part two of the course.

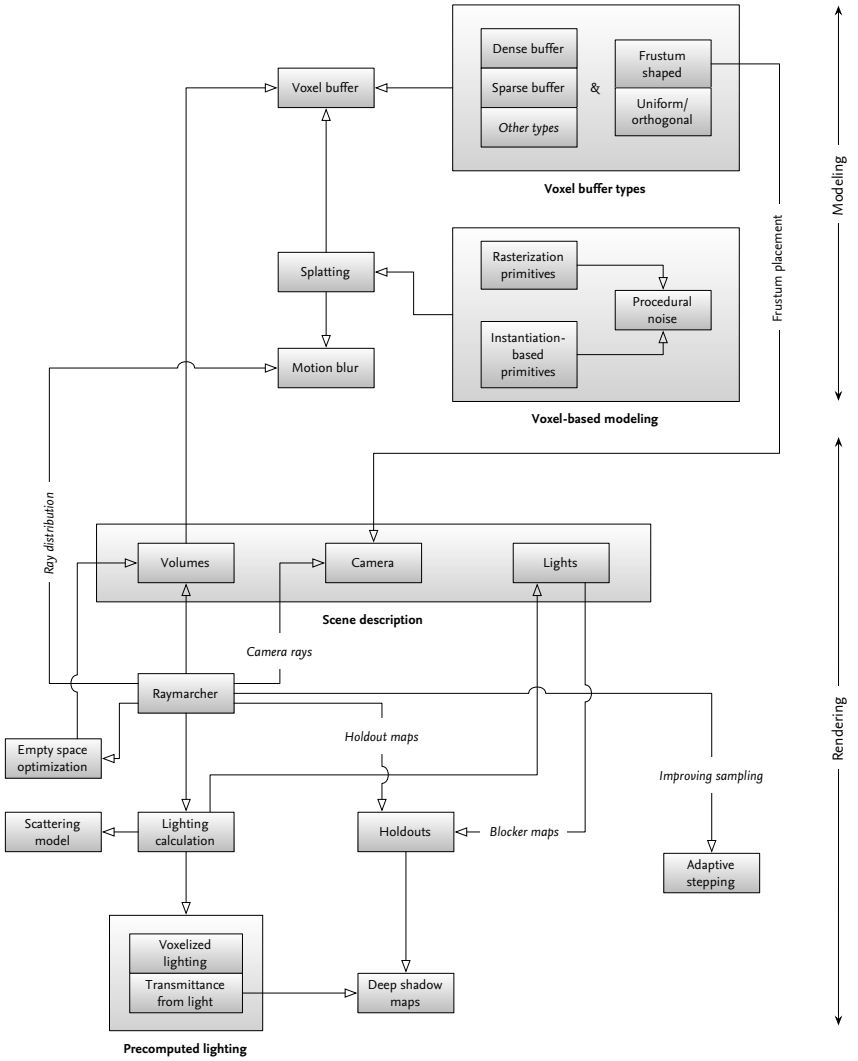
There are three main types of motion blur that need to be handled:

- **Object motion blur** is caused by motion of a volume or voxel buffer as a whole which often occurs when attaching a fluid simulation to a moving object in the scene. To account for the motion of the volume we need two or more *motion samples*. Each motion sample describes the local transform of the volume at a given time, usually the shutter open and shutter close times.
- **Deformation motion blur** occurs when the velocity throughout a particular volume varies. For example, a fluid simulation can be motion blurred at render time by looking not only at the density field but also at the velocity field.
- **Camera motion blur** occurs whenever the render camera itself is moving.

4. Putting it all together

The material covered so far should provide the basis needed to implement a simple volume modeling and rendering application. With those basics in mind, the next few chapters will dive into detail about how some actual battle-proven implementations work, and will show that the same problem often gets solved quite differently from facility to facility. Our hope is that, taken together, they will give a good overview of the state-of-the-art in production volume rendering.

The following diagram attempts to illustrate how each of the pieces described so far are connected to one another, and will serve the reader as an guide to how the advanced topics that follow fit into the volume rendering pipeline.



5. References & Further reading

References

Gritz, L. 1998. Basic Antialiasing in Shading Language, Advanced RenderMan SIGGRAPH course, In *course notes pp. 62-80.*

Heckbert, P. S. 1990. What Are The Coordinates Of A Pixel? In *Graphics Gems I, pp. 246-248. Andrew Glassner (editor), Academic Press.*

Kajiya J. T., Herzen B. P. V. 1984. Ray tracing volume densities. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques (New York, NY, USA, 1984), ACM Press, pp. 165–174.*

Kapler, A. 2002. Evolution of a vfx voxel tool. In *SIGGRAPH '02: ACM SIGGRAPH 2002 conference abstracts and applications, pages 179–179, New York, NY, USA. ACM.*

Kapler, A. 2003. Avalanche! Snowy fx for xXx. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications, pages 1–1, New York, NY, USA. ACM.*

Kisacikoglu, G. 1998. The making of black-hole and nebula clouds for the motion picture "sphere" with volumetric rendering and the f-rep of solids. In *SIGGRAPH '98: ACM SIGGRAPH 98 Conference abstracts and applications, page 289, New York, NY, USA. ACM.*

Lokovic, T. and Veach, E. 2000. Deep shadow maps. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pages 385–392, New York, NY, USA. ACM Press/ Addison-Wesley Publishing Co.*

Reeves, W. T. 1983. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph., 2(2):91–108.*

Squires, S. 2009. Cloud Tank effect. http://effectscorner.blogspot.com/2009/02/cloud-tank-effect.html.

Books

Pharr & Humphries – Physically Based Rendering. Morgan Kaufmann publ.
Ebert et al. – Texturing & Modeling. Morgan Kaufmann publ.
Jules Bloomenthal (edited by) – Introduction to Implicit Surfaces. Morgan Kaufmann publ.

Websites

http://flam3.com/