Eurographics Symposium on Rendering 2008
Steve Marschner and Michael Wimmer
(Guest Editors)

Volume 27 (2008), Number 4

Compact, Fast and Robust Grids for Ray Tracing

Ares Lagae & Philip Dutré†
Department of Computer Science
Katholieke Universiteit Leuven

Abstract

The focus of research in acceleration structures for ray tracing recently shifted from render time to time to image, the sum of build time and render time, and also the memory footprint of acceleration structures now receives more attention. In this paper we revisit the grid acceleration structure in this setting. We present two efficient methods for representing and building a grid. The compact grid method consists of a static data structure for representing a grid with minimal memory requirements, more specifically exactly one index per grid cell and exactly one index per object reference, and an algorithm for building that data structure in linear time. The hashed grid method reduces memory requirements even further, by using perfect hashing based on row displacement compression. We show that these methods are more efficient in both time and space than traditional methods based on linked lists and dynamic arrays. We also present a more robust grid traversal algorithm. We show that, for applications where time to image or memory usage is important, such as interactive ray tracing and rendering large models, the grid acceleration structure is an attractive alternative.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: [ray tracing] [acceleration structure] [grid] [row displacement compression] [perfect hashing]

1 Introduction

Ray tracing is becoming more and more the method of choice for both offline global illumination simulations as well as interactive visualizations. Because intersecting a ray with all objects in a scene is usually very expensive, almost all ray tracers rely on acceleration structures, trading preprocessing time and memory for faster ray-object intersections.

The uniform grid was one of the first proposed acceleration structures [FTI86]. Over time, several other acceleration structures, such as bounding volume hierarchies and kd-trees, have been introduced [Gla89]. For static scenes, kd-trees are by many considered the best acceleration structure [WMG*07]. Uniform grids usually perform worse than kd-trees, mainly because they are not adaptive. For dynamic scenes however, there is no consensus [WMG*07]. The acceleration structure has to be rebuilt every frame, and rather than minimizing render time, the time to image, the sum of build time and render time, has to be minimized. Building a grid can be done in linear time, while other popular acceleration structures require super-linear time. For dynamic scenes, a shorter build time can compensate for a longer render time. Therefore, a grid can result in a shorter time to image than other acceleration structures that are usually considered superior.

Algorithms are typically CPU-bound or memory-bound. The execution time of an algorithm that is CPU-bound mainly depends on the speed of the CPU, while the execution time of an algorithm that is memory-bound mainly depends on the access speed of the memory. Memory-bound algorithms can be made significantly faster just by reducing the memory footprint of the data they work on. Building a grid is memory-bound, while rendering is CPU-bound. Therefore, reducing the memory footprint of a grid can result in shorter build times.

Uniform grids were used in one of the first systems for interactive ray tracing [PMS*99], and are still popular today [GIK*07]. Recent work on grids for ray tracing concentrated on fast traversal [WIK*06], parallelizing the build process [IRWP06], and choosing the grid size [ISP07]. In this paper, we present two efficient methods for representing and building a grid. The *compact grid* method consists of a static data structure for representing a grid with minimal memory requirements, more specifically exactly one in-

† e-mail: {ares.lagae,philip.dutre}@cs.kuleuven.be

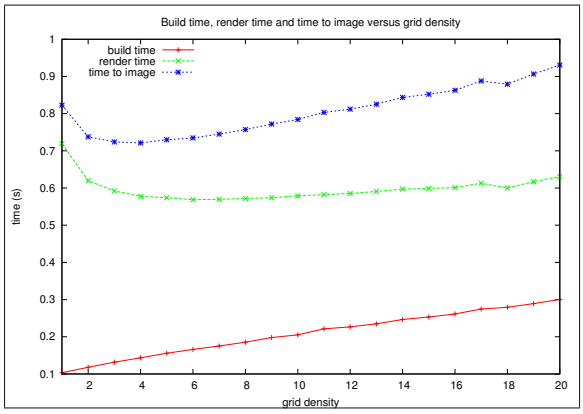


Figure 1: Grid density. Build time (red), render time (green) and time to image (blue) versus grid density for the *Happy Buddha* scene. A grid density of 4 minimizes the time to image.

dex per grid cell and exactly one index per object reference, and an algorithm for building that data structure in linear time. The *hashed grid* method consists of a static data structure for representing a grid that reduces memory requirements even further, by using perfect hashing based on row displacement compression, and a fast algorithm for building that data structure. We believe that these data structures and algorithms are the most space and time efficient methods for representing and building a grid for ray tracing.

2 The Grid Acceleration Structure

The grid acceleration structure was introduced by Fujimoto et al. [FTI86]. A grid uniformly partitions space into cubically shaped cells. Each cell contains references to the objects that overlap the cell. Rather than intersecting a ray with all objects in the scene, only the objects in the cells pierced by the ray have to be intersected, greatly reducing the number of intersection tests. In this section we motivate the design decisions we made for our grid acceleration structure.

Number of Cells The number of cells M should be linear in the number of objects N [Dev88, JW89], or

$$M = \rho N, \tag{1}$$

where ρ is called the grid density. The number of cells M is equal to the product of the resolution of the grid in each dimension and cubically shaped cells work best. The resolution of the grid $M_x \times M_y \times M_z$ is therefore determined as

$$M_i = S_i \sqrt[3]{\frac{\rho N}{V}} \quad (i \in \{x, y, z\}), \tag{2}$$

where S_i is the size of the bounding box of the grid in dimension i and V is the volume of the bounding box.
According to Purcell a grid density of 5 to 10 works best [Hai01], Shirley reports grid densities of 2 to 10 [Shi02], Wald et al. use a grid density of 5 [WIK*06], and Ize et al. empirically determined 8 to be the optimal grid density [ISP07].
In contrast to previous work, we use the time to image rather than the render time to determine the optimal grid density. We use a grid density of 4, as suggested by figure 1. Fortunately, the performance of the grid is not too sensitive to the choice of the parameter.

Inserting Objects To insert an object into a grid, all cells that the object overlaps have to be determined. This can be done using the bounding box of the object, or with more accurate object cell overlap tests. Using the bounding box results in shorter build times and longer render times, because the object is also added to cells that overlap the bounding box but not the object. More accurate object cell overlap tests results in longer build times and shorter render times. Since Ize et al. [IRWP06] and Wald et al. [WIK*06] reported that using more accurate object cell overlap tests does not pay off, we insert objects based on their bounding box. However, our construction method does not preclude the use of more accurate object cell overlap tests.

Mailboxing Mailboxing is a technique for avoiding repeated intersection test with the same object. Havran [Hav02] reported that mailboxing does not necessarily pay off, especially for objects that are cheap to intersect. Therefore we do not use mailboxing.

3 The Compact Grid Method

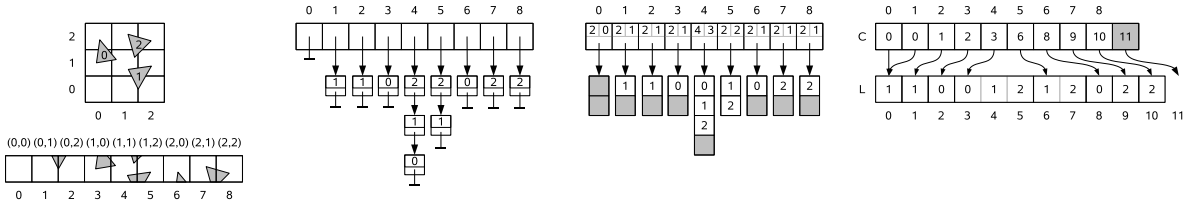
In this section we present the compact grid method. This method consists of a data structure for representing a grid with minimal memory requirements, more specifically exactly one index per grid cell and exactly one index per object reference, and an algorithm for building that data structure in linear time.

3.1 Data Structure

In this subsection, we review the memory requirements of grid representations based on linked lists and dynamic arrays, and we present our compact grid representation.

Linked Lists The most straightforward implementation of a grid uses linked lists [CLRS01]. Figure 2(a) shows a grid and figure 2(b) shows the representation of the grid using linked lists. Figure 2(a) also shows the linearization of the three-dimensional array of cells.
The memory requirements of a grid representation based on linked lists are one machine word per cell (a pointer to a list node) and two or three machine words per object reference (an object index and one or two pointers to list nodes), depending on whether singly linked lists or doubly linked lists are used.

Dynamic Arrays Dynamic arrays or vectors [CLRS01] are often used as an alternative for lists. Dynamic arrays maintain a static array and keep track of its capacity and size. When the size is about to exceed the capacity, a new array with a larger capacity is allocated and the old array is copied and freed. Dynamic arrays typically support faster iteration due to their improved locality of reference [Pha02].



(a) (b) (c) (d)

Figure 2: Data structures for representing a grid. (a) A grid with three triangles and the linearized version of the grid. (b) A traditional grid data structure using linked lists. (c) A traditional grid data structure using dynamic arrays. (c) The compact grid data structure presented in this paper.

Figure 2(a) shows a grid and figure 2(c) shows the representation of the grid using dynamic arrays.
The memory requirements of a grid representation based on dynamic arrays are three machine words per cell (a pointer to the array, the size of the array and the capacity of the array) and anywhere between 1 to 2 machine words per object reference, depending on the number of unused entries in the dynamic array.

Compact Grid Grid representations based on linked lists and dynamic arrays are dynamic data structures. They support insertion of objects and can even support removal of objects. The memory overhead of these data structures is exactly because of this. However, in a setting where the grid is rebuilt from scratch every frame, dynamic data structures are not needed.
The compact grid data structure for representing a grid consists of two static arrays. This is illustrated in figure 2(c). The array L consist of the concatenation of all object lists. The array C stores for each cell the offset of the corresponding object list in L. This data structure is static, objects cannot be inserted nor removed.
The array L is a 1D array. The array C is a 3D array of size $M_x \times M_y \times M_z$, linearized in row major order into a 1D array of size M. The array C supports both 3D indexing $C[z][y][x]$ and 1D indexing $C[i]$

$$C[z][y][x] = C[(((M_y z) + y)M_x) + x]. \tag{3}$$

The size of the object list of the cell with 1D index i is given by $C[i+1] - C[i]$. Note that this expression is invalid for the last object list, since $C[N]$ does not exist. In order to avoid an explicit check for this special case, we extend the array C with one position.
Intersecting all objects in a given cell can be done as follows.

```
/* intersect all objects in cell (x,y,z) */
i = (((M_y * z) + y) * M_x) + x
for (j = C[i]; j < C[i + 1]; ++j) {
  /* intersect object L[j] */
}
```

The memory requirements of the compact grid data structure for representing a grid are exactly one machine word per per cell and exactly one machine word per object reference. If the grid resolution is chosen according to equation 2, this data structure has a space complexity that is linear in the number of objects.

The array L uses 32-bit unsigned integers to index the objects. This is sufficient for indexing over 4 billion objects. The array C uses 32-bit unsigned integers to index the object lists. Note that storing 32-bit unsigned integer indices rather than pointers results in additional memory savings on 64-bit platforms.

3.2 Algorithm

In this subsection, we present the algorithm for building the compact grid representation.

The algorithm works as follows. First, the bounding box of the objects is computed and the grid resolution is determined using equation 2.
Next, the size of all object lists is computed and stored in the cell array, i.e. $C[i]$ records the size of the object list of the cell with 1D index i. The size of the object lists is needed to compute the offsets of the object lists, and the joint size of the object lists is needed to allocate the object lists array L. The cell array C is allocated and each entry is initialized to zero. The size of all object lists is computed by iterating over all objects and incrementing the object list size of all cells overlapped by an object.
The offsets of the object lists can now be computed by accumulating the size of the object lists. However, inserting the object indices into the object lists is not possible without keeping track of how many objects indices are already inserted during iteration over the objects. Rather than computing for each cell the offset to its object list, the offset to the next object list is computed, i.e. $C[i]$ records the offset of the object list of the cell with 1D index $i+1$. In other words, $C[i]$ points to one past the end of the object list of the cell with 1D index i. This can be accomplished as follows.

```
for (i = 1; i <= M; ++i) {
  C[i] += C[i-1];
}
```

The joint size of the object lists is now given by $C[N-1]$. Finally, the object indices are inserted into the object lists. The object list array L is allocated and the object indices are inserted by reversely iterating over all objects, and for each cell overlapped by an object decrementing the offset of the cell and storing the object index at that offset. This can be done as follows.

```
for (i = N - 1; i >= 0; --i) {
  /* for each cell j overlapped by object i */
  L[--C[j]] = i;
}
```

The object lists are thus filled backwards. After this operation the cell array will contain the correct offsets, since each offset was decremented the appropriate number of times. Note that the indices in each object list are sorted.
This algorithm has a time complexity that is linear in the number of objects and does not require additional memory.

To our knowledge, the compact grid method has never been described in literature. However, some similar methods are used by researchers in the field.
We believe the idea of doing two passes over the triangles can be traced back to a short article of Eric Haines in *Ray Tracing News* [Hai99]. However, the method of Haines uses `NULL` pointers to keep track of available locations in the object lists, making insertion a linear (at best logarithmic) rather than constant time operation. The method of Haines also uses a `NULL` pointer to indicate the end of an object list, resulting in a larger memory footprint.
Personal communication with the authors of [WIK*06] and [ISP07] revealed another similar method, which can probably be traced back to Steven Parker. However, during insertion the method of Parker uses another array to keep track of the next free location of each object list. This almost doubles the memory footprint.
From an algorithmic point of view, the compact grid method is very similar to counting sort, a method for sorting in linear time [CLRS01].

4 The Hashed Grid Method

In this section we present the hashed grid method. This method consists of a data structure for representing a grid that reduces memory requirements even further, by using hashing to avoid the storage of the empty cells in the grid, and an algorithm for building that data structure.

The memory footprint of the cell offsets is significantly larger than the memory footprint of the object lists. Furthermore, the majority of the cells is empty, while the object lists do not contain much redundant information. Experiments with eliminating duplicate lists were not promising. The array with the cell offsets is therefore the best candidate for further reduction in size.
The array with the cell offsets is essentially a sparse array, and the typical solution to avoid the storage of the complete array is to use hashing [CLRS01]. To our knowledge, hashing has never been investigated in detail in the context of grids for ray tracing. Traditional hash tables are dynamic data structures, that support insertion and removal of elements, and handle collisions of the hash function. As in the previous section, we will replace the dynamic data structure by a static one. If the data is static, a hash function that does not result in collisions can be computed. This is called a perfect hash function. For more information about hashing, perfect hashing and perfect spatial hashing, we refer to Cormen et al. [CLRS01], Czech et al. [CHM97] and Lefebvre and Hoppe [LH06].

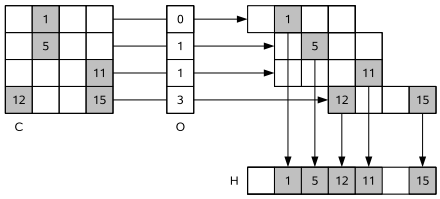


Figure 3: Row displacement compression. The square matrix C is compressed into a hash table H by displacing the rows, and storing the offset of each row in the offset table O.

4.1 Perfect Hashing using Row Displacement Compression

Our method for computing a perfect hash function is based on the row displacement compression algorithm, introduced in 1977 by Aho and Ullman [AU77] as a compaction scheme for transition diagrams of deterministic finite automata. We will explain the method in 2D using sparse matrix compression.

Given a sparse matrix C, the goal is to compute a hash function h and a hash table H such that each non-zero element $C(i,j)$ is hashed to the position $h(i,j)$ in the hash table H. The hash function h should be perfect and close to minimal, i.e. the hash table H should contain as few unused entries as possible.
The algorithm works as follows. The first row of the matrix C is copied to the hash table H at offset 0. Each subsequent row of the matrix C is then copied to the hash table H at the smallest offset, such that non-zero elements do not overlap. Each offset is determined starting from the offset of the previous row. For each row i, this offset is recorded in a 1D offset table O at position i. The algorithm is illustrated in figure 3.
Each non-zero element $C(i,j)$ corresponds to $H[O[i]+j]$, i.e. the hash function h is given by $h(i,j) = O[i]+j$.

Encoding which elements are non-zero can be done using domain bits or using position tags. When using domain bits, a matrix D with the same dimensions as the matrix C records which positions in C are non-zero, using a single bit per element. When using position tags, each entry in the hash table H also stores a tag that identifies the position in the matrix C of the element, i.e. if $H[h(i,j)]$ does not contain the tag associated with position (i,j) then $C(i,j)$ was zero. Note that the hash function h only prevents collisions between non-zero elements. In figure 3 the linearized index is used as position tag.

The worst case time complexity of this algorithm is $O(M^{3/2})$, where M is the number of elements in the matrix C. The perfect hash function is simple and can be evaluated efficiently. The elements in the hash table can be accessed in constant time.

4.2 Data Structure and Algorithm

In this subsection, we present the data structure and algorithm for the hashed grid method.

The data structure consists of four static arrays. The array L consist of the concatenation of all object lists, as in the compact grid method. The array C of the compact grid method, which stores for each cell the offset of the corresponding object list in L, is replaced by a hash table. This hash table is computed with the 3D equivalent of the algorithm presented in the previous subsection. The 3D version of the algorithm processes rows of the array C. Remember that the array C is a 3D array of size $M_x \times M_y \times M_z$, linearized in row major order into a 1D array of size M. The array C therefore consists of $M_y M_z$ rows of size M_x. The offset table O is a 2D array of size $M_y \times M_z$, linearized into a 1D array of size $M_y M_z$. The hash table H is a 1D array, and the domain bits D is an array similar to C but using only one bit per entry.

The algorithm works as follows. First, the bounding box of the objects is computed, and the grid resolution is determined using equation 2.
Next, the domain bits are computed. The array D is allocated and each bit is initialized to zero. The domain bits are computed by iterating over all objects, and setting the bit corresponding to each cell overlapped by an object to one. The number of non-empty cells is also computed.
Then, the hash function and the size of the hash table are computed. The array O is allocated and filled in with the 3D equivalent of the algorithm presented in the previous subsection. This is done using the domain bits and a temporary hash table. For the temporary hash table we use a dynamic array with an initial size equal to twice the number of non-empty cells. A static array of size M_x would also suffice, because only the last M_x entries of the hash table are relevant, but this results in a slightly slower algorithm.
Finally, the offsets of the object lists are computed and the object indices are inserted into the object lists. The hash table H is allocated and each entry is initialized to zero. Then the algorithm proceeds in exactly the same way as the compact grid method, but using $H[h(x,y,z)]$ rather than $C[z][y][x]$, where $h(x,y,z) = O[(M_y z) + y] + x$. The cells in H are in a different order than in C, because the hash function is not order preserving, but the size of the object list of the cell at location (x,y,z) is still given by $H[h(x,y,z)+1] - H[h(x,y,z)]$.

This algorithm has a time complexity that is linear in the number of objects, except for the computation of the hash function. The worst case time complexity of that part of the algorithm is $O(M^{4/3})$, where M is the number of cells in the grid.

5 Robust Grid Traversal

Grid traversal methods usually reject and discard intersections outside of the cell being traversed. This is also the case for the popular single ray grid traversal method of Cleary and Wyvill [CW88] we use in this paper. However, this can easily lead to numerical robustness errors.

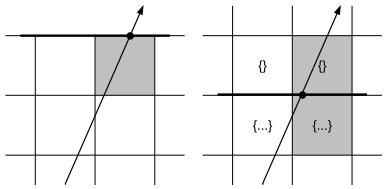


Figure 4: Numerical robustness errors. An intersection with a triangle (a) on the bounding box of the grid and (b) on a plane implied by the grid is missed due to numerical robustness errors.

Figure 4 shows two examples. In figure 4(a), the intersection of the triangle on the bounding box of the grid and the ray, computed while visiting the gray cell, is rejected because it is outside of the gray cell due to numerical robustness errors. In figure 4(b), the intersection of the triangle on a plane implied by the grid and the ray, computed while visiting the first gray cell is rejected, and is not detected while visiting the second gray cell, because the triangle was not added to that cell. These errors can be alleviated using ε-tests during building and/or traversal of the grid. However, choosing the correct ε is hard and scene-dependant. Instead we use a small modification to existing grid traversal methods to solve this problem. Instead of rejecting and discarding intersections outside of the cell being traversed, we keep the ray parameter of the closest intersection during grid traversal, even if outside of the cell being traversed, and terminate ray traversal when the maximum ray parameter for the cell being traversed is larger than the ray parameter of the closest intersection. We call this method robust grid traversal.

Although robust grid traversal is used by some researchers in the field [PH04], several important sources still discard intersections [Shi00, Suf07]. This is why we stress on robust ray traversal in this paper.

6 Results and Discussion

In this section we present a thourough evaluation of the performance of the compact grid method and the hashed grid method. We present results, we compare to traditional grid representations and to other acceleration structures, we discuss parallelization and we present several applications.

6.1 Methodology

The methods presented in this paper were tested using scanned models from *The Stanford 3D Scanning Repository* and *The Digital Michelangelo Project*, and scenes from the recent *bwfirt* benchmark [RGH*07], including scenes from *Radiance*. This set of scenes includes both scenes that work well with grids (e.g. the scanned models), and scenes that work less well with grids (e.g. the *Nature* and *Radiance* scenes).
The methods presented in this paper were implemented in high-level C++ using templates and STL. Low-level optimizations such as SIMD were not used. The methods were

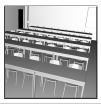
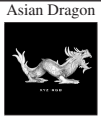
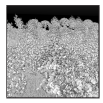
	Ulm Box	Classroom	Office	Bunny	Soda	Cabin	Armadillo	Atrium
								
scene statistics								
#tri's	492	9.25 K	34.00 K	69.45 K	141.64 K	219.44 K	345.94 K	559.99 K
memory	17.30 Kb	325.30 Kb	1.17 MB	2.38 MB	4.86 MB	7.53 MB	11.88 MB	19.23 MB
grid statistics								
grid res	13x13x13	36x58x18	61x36x61	71x71x55	84x40x168	112x88x90	109x129x99	88x256x99
# cells	2.20 K	37.58 K	133.96 K	277.25 K	564.48 K	887.04 K	1.39 M	2.23 M
% empty cells	56.94 %	75.11 %	85.35 %	92.32 %	87.69 %	47.70 %	96.53 %	89.44 %
avg # tri's / n-emp cell	3.87	6.27	6.85	10.34	11.42	4.82	16.35	7.38
avg # cells / tri	7.44	6.34	3.95	3.17	5.60	10.19	2.28	3.11
render statistics								
avg # n-emp cells / isect ray	3.68	4.08	4.12	2.15	4.08	24.17	2.33	4.34
avg # isect tests / isect ray	11.82	38.71	75.41	24.74	93.10	106.31	40.99	55.78
	Dragon	Conference	Buddha	Cruiser	Asian Dragon	Thai Statue	Lucy	Nature
								
scene statistics								
#tri's	871.41 K	987.52 K	1.09 M	3.64 M	7.22 M	10.00 M	28.06 M	41.35 M
memory	29.92 MB	33.90 MB	37.34 MB	124.84 MB	247.85 MB	343.32 MB	963.22 MB	1.39 GB
grid statistics								
grid res	223x157x100	296x188x71	121x295x121	523x176x158	428x237x284	302x508x261	485x278x832	906x202x902
# cells	3.50 M	3.95 M	4.32 M	14.54 M	28.81 M	40.04 M	112.18 M	165.08 M
% empty cells	95.44 %	92.24 %	94.86 %	95.54 %	98.99 %	98.44 %	99.00 %	92.04 %
avg # tri's / n-emp cell	13.96	9.43	13.02	11.67	40.79	29.25	41.50	28.52
avg # cells / tri	2.56	2.93	2.66	2.08	1.65	1.83	1.66	9.06
render statistics								
avg # n-emp cells / isect ray	2.24	4.83	2.52	8.69	2.14	2.38	2.06	12.12
avg # isect tests / isect ray	33.78	28.14	37.27	116.69	96.93	73.91	99.28	302.23

Table 1: Scene statistics, grid statistics and render statistics for various scenes.

also implemented in the recent *bwfirt* benchmark [RGH*07]. All timings were obtained on a computer with two four-core 3 GHz Intel Xeon X5365 CPU's and 16 Gb of memory. Only a single core was used, unless noted otherwise. All images were rendered at a resolution of 1024×1024 with one ray per pixel and diffuse shading, unless noted otherwise.

6.2 Compact Grid Method and Hashed Grid Method

Table 1 shows scene statistics, grid statistics and render statistics for the test scenes. These statistics are the same for both methods, since they produce the same grids. The size of the scenes is computed using a representation of 36 bytes per triangle (three single precision floating point numbers for each coordinate). The grid resolution is determined according to equation 2. The average number of triangles per non-empty cell and the average number of cells per triangle are both relatively low. The majority of the cells is empty, and each triangle is only in a few cells. The average number of non-empty cells and intersection tests per intersecting ray are both relatively low, indicating relatively good ray tracing performance, also for scenes that work less well with grids.

Table 2 shows statistics of the compact grid method and hashed grid method. The most important figures are the build time, the time to image and the memory.

The build time of the compact grid method is relatively short, and is roughly linearly in the number of triangles. This results in a short time to image. The memory footprint of the compact grid method is roughly about three quarter of the memory footprint of the scene. The memory footprint is broken down into the memory needed for the cells and the memory needed for the object lists. The memory needed for the cells is significantly larger than the memory needed for the object lists.

The memory footprint of the hashed grid method is significantly smaller than the memory footprint of the compact grid method. The memory footprint of the cells is replaced by the sum of the memory footprints of the domain bits, the offset table and the hash table. The memory footprint of the object lists remains the same. The build time moderately increases due to computation of the hash function. The render time slightly decreases due to an improved locality of reference. The perfect hashing algorithm using row displacement compression works surprisingly well. It achieves compression ratios of up to 20:1, and the load factor of the hash table is within a factor two of the optimal solution.

Perfect spatial hashing was recently studied by Lefebvre and Hoppe [LH06] in the context of the GPU. Their method produces hash tables with a higher load factor but the algorithm is significantly more complex and slower. In the context of grids for ray tracing, running time is more important because even with moderate load factors, the memory requirements for the grid are already well below the memory requirements

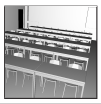
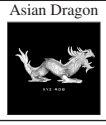
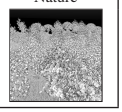
	Ulm Box	Classroom	Office	Bunny	Soda	Cabin	Armadillo	Atrium
								
compact grid statistics								
mem cells	8.58 Kb	146.81 Kb	523.27 Kb	1.06 MB	2.15 MB	3.38 MB	5.31 MB	8.51 MB
mem tri lists	14.30 Kb	229.16 Kb	524.87 Kb	859.69 Kb	3.03 MB	8.53 MB	3.01 MB	6.64 MB
build time	0.00 s	0.00 s	0.00 s	0.01 s	0.02 s	0.04 s	0.05 s	0.08 s
render time	0.48 s	1.13 s	1.68 s	0.68 s	2.23 s	2.56 s	0.90 s	1.93 s
time to image	0.48 s	1.14 s	1.68 s	0.69 s	2.25 s	2.60 s	0.95 s	2.01 s
memory	22.88 Kb	375.97 Kb	1.02 MB	1.90 MB	5.18 MB	11.91 MB	8.32 MB	15.14 MB
hashed grid statistics								
data density	43.06 %	24.89 %	14.65 %	7.68 %	12.31 %	52.30 %	3.47 %	10.56 %
hash table size	1.07 K	10.74 K	23.49 K	27.87 K	96.89 K	492.23 K	69.53 K	303.55 K
hash table load factor	88.58 %	87.10 %	83.52 %	76.38 %	71.73 %	94.24 %	69.41 %	77.61 %
mem domain bits	275.00 b	4.59 Kb	16.35 Kb	33.84 Kb	68.91 Kb	108.28 Kb	169.93 Kb	272.25 Kb
mem offset table	676.00 b	4.08 Kb	8.58 Kb	15.25 Kb	26.25 Kb	30.94 Kb	49.89 Kb	99.00 Kb
mem hash table	4.17 Kb	41.95 Kb	91.75 Kb	108.88 Kb	378.48 Kb	1.88 MB	271.59 Kb	1.16 MB
mem cells	5.10 Kb	50.61 Kb	116.68 Kb	157.97 Kb	473.64 Kb	2.01 MB	491.40 Kb	1.52 MB
compression ratio	168.26 %	290.08 %	448.44 %	685.58 %	465.55 %	168.04 %	1110 %	559.54 %
mem tri lists	14.30 Kb	229.16 Kb	524.87 Kb	859.69 Kb	3.03 MB	8.53 MB	3.01 MB	6.64 MB
build time	0.00 s	0.00 s	0.01 s	0.01 s	0.03 s	0.07 s	0.07 s	0.10 s
render time	0.48 s	1.12 s	1.69 s	0.67 s	2.26 s	2.57 s	0.89 s	1.86 s
time to image	0.48 s	1.12 s	1.70 s	0.69 s	2.29 s	2.64 s	0.96 s	1.97 s
memory	19.40 Kb	279.77 Kb	641.55 Kb	1017.66 Kb	3.49 MB	10.54 MB	3.49 MB	8.16 MB
	Dragon	Conference	Buddha	Cruiser	Asian Dragon	Thai Statue	Lucy	Nature
								
compact grid statistics								
mem cells	13.36 MB	15.07 MB	16.48 MB	55.48 MB	109.89 MB	152.75 MB	427.93 MB	629.72 MB
mem tri lists	8.51 MB	11.03 MB	11.03 MB	28.84 MB	45.37 MB	69.78 MB	178.06 MB	1.40 GB
build time	0.11 s	0.12 s	0.14 s	0.39 s	0.80 s	1.17 s	3.15 s	9.12 s
render time	0.80 s	2.28 s	0.58 s	2.49 s	1.43 s	1.55 s	1.90 s	10.75 s
time to image	0.91 s	2.40 s	0.72 s	2.89 s	2.23 s	2.72 s	5.05 s	19.87 s
memory	21.86 MB	26.10 MB	27.50 MB	84.32 MB	155.27 MB	222.53 MB	605.98 MB	2.01 GB
hashed grid statistics								
data density	4.56 %	7.76 %	5.14 %	4.46 %	1.01 %	1.56 %	1.00 %	7.96 %
hash table size	277.37 K	454.81 K	322.69 K	1.14 M	466.89 K	967.47 K	1.76 M	28.82 M
hash table load factor	57.57 %	67.41 %	68.78 %	56.71 %	62.45 %	64.64 %	63.76 %	45.57 %
mem domain bits	427.38 Kb	482.30 Kb	527.23 Kb	1.73 MB	3.43 MB	4.77 MB	13.37 MB	19.68 MB
mem offset table	61.33 Kb	52.14 Kb	139.43 Kb	108.62 Kb	262.92 Kb	517.92 Kb	903.50 Kb	711.73 Kb
mem hash table	1.06 MB	1.73 MB	1.23 MB	4.36 MB	1.78 MB	3.69 MB	6.73 MB	109.94 MB
mem cells	1.54 MB	2.26 MB	1.88 MB	6.20 MB	5.47 MB	8.97 MB	20.98 MB	130.31 MB
compression ratio	869.89 %	667.82 %	875.44 %	895.01 %	2010 %	1700 %	2040 %	483.24 %
mem tri lists	8.51 MB	11.03 MB	11.03 MB	28.84 MB	45.37 MB	69.78 MB	178.06 MB	1.40 GB
build time	0.19 s	0.19 s	0.22 s	0.72 s	1.22 s	1.76 s	4.77 s	21.23 s
render time	0.79 s	2.22 s	0.57 s	2.52 s	1.25 s	1.43 s	1.53 s	10.07 s
time to image	0.98 s	2.41 s	0.79 s	3.25 s	2.47 s	3.18 s	6.30 s	31.30 s
memory	10.04 MB	13.29 MB	12.91 MB	35.04 MB	50.84 MB	78.75 MB	199.04 MB	1.52 GB

Table 2: Compact grid statistics and hashed grid statistics for various scenes.

of the object lists. This means that more complex hashing methods will most likely not pay off.

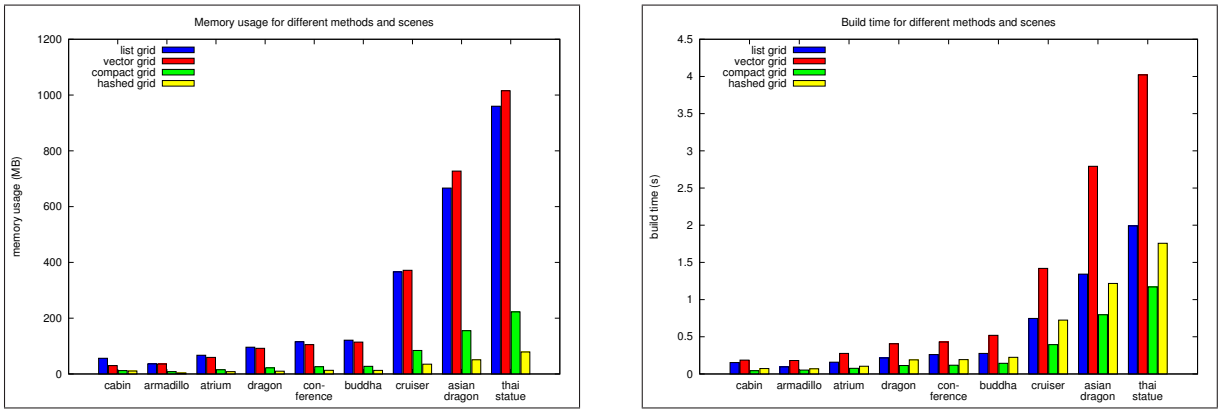
6.3 Comparison with Traditional Grid Representations

Figure 5 shows a comparison of memory footprint and build time between the compact grid method and the hashed grid method and straightforward representations using linked lists and dynamic arrays (vectors).

We used `std::list` and `std::vector` from the STL to implement these traditional grid representations. Replacing these STL components with custom list and dynamic array implementations could improve the performance of the traditional grid representations. However, a straightforward grid implementation is likely to use standard components,and the added complexity of custom list and dynamic array implementations is probably larger than that of the methods presented in this paper.

The memory footprint of the compact grid method is about a factor 4 smaller than the memory footprint of the representations using linked lists and dynamic arrays. This verifies the theoretical analysis of section 3.1. The memory footprint of the hashed grid method is even smaller.

The build time of the compact grid method is about a factor 3 smaller than the build time of the representations using linked lists and dynamic arrays. Although the compact grid



(a) (b)

Figure 5: Comparison with traditional grid methods. The (a) memory consumption and (b) build times for the different grid representations. The compact grid uses less memory and builds faster than grids based on lists and vectors. The hashed grid uses even less memory and still builds faster than grids based on lists and vectors.

method requires an extra pass over the triangles, the method is faster because it uses less memory, has a better locality of reference, and does not need to maintain dynamic data structures. This nicely illustrates that memory-bound algorithms can be made significantly faster just by reducing the memory footprint of the data they are working on. The build time of the hashed grid method is longer than the build time of the compact grid, but shorter than the build time of the traditional methods.

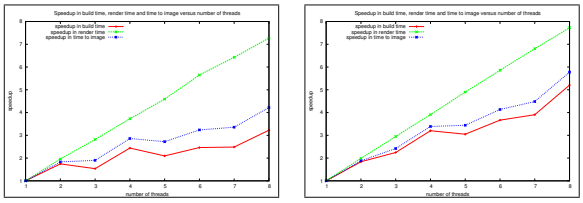
The render time of the compact grid method and the representations using linked lists and dynamic arrays are about the same. The render time is the largest for the implementation using linked lists and the smallest for the compact grid method. This is due to locality of reference.

The render time of the compact grid method and hashed grid method can most likely be further decreased by using packet grid traversal or coherent grid traversal [WIK*06]. However, these optimizations are mostly orthogonal to the ones presented in this work, since the time to build the grid is the major improvement over traditional grid representations.

6.4 Comparison with other Acceleration Structures

The timings included in recent work [WH06,WK06,SSK07] at least indicate that the methods presented in this paper are competitive with state of the art methods. For example, Wald et al. [WIK*06] reported a build time, render time and time to image of respectively 1.65 *s*, 0.55 *s* and 2.20 *s* for the *Thai Statue* scene, using a packet size of 4 and a computer with a dual 3.2 GHz Intel Xeon CPU. However, a direct comparison is difficult without integrating these methods into a single framework. In order to give a rough idea of how our method compare to other acceleration structures we have integrated the methods presented in this paper into the recent *bwfirt* benchmark [RGH*07].

The *bwfirt* benchmark is an extensible framework that compares the performance of ray tracing kernels using a renderer based on path tracing in terms of build time, render



(a) (b)

Figure 6: Parallel building and renderering. Speedup in build time (red), render time (green) and time to image (blue) versus number of threads for (a) the *Asian Dragon* scene and (b) the *Nature*.

time and memory usage. The *bwfirt* includes ray tracing kernels based on kd-trees and bounding volume hierarchies, described as *decent performance but not state of the art*. We have integrated the compact grid method into *bwfirt*, and we have added a simple renderer to *bwfirt* in order to support scenes without light sources. The reports generated by *bwfirt* show that the compact grid method is competitive with the ray tracing kernels of *bwfirt*.

The memory usage of the compact grid method is in almost all cases less than that of the kd-tree (up to a factor of 10) and the bounding volume hierarchy (up to a factor of 2). The build time of the compact grid method is only a fraction of the build time of the kd-tree and the bounding volume hierarchy. The render time of the compact grid method is in most cases between the render time of the kd-tree and the bounding volume hierarchy (with differences up to a factor of 2). The time to image of the compact grid method is usually less than the time to image of the kd-tree and the bounding volume hierarchy (up to a factor of 10).

The integration into *bwfirt* also shows that the compact grid method is very simple to implement. The implementation of the algorithm for building the grid is only about 130 lines of code. This can be an important advantage over other acceleration structures.

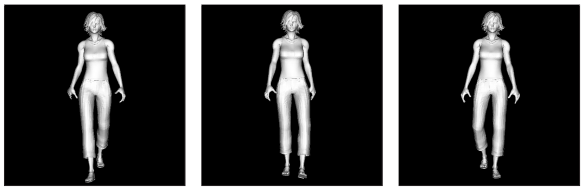


Figure 7: Interactive ray tracing. Three frames of an animation of the *Jessi* model walking. The animated *Jessi* model consists of 260.72 K dynamic triangles. The animation is rendered at a resolution of 512×512 at 8.38 FPS.

6.5 Parallelization

Modern hardware architectures are becoming increasingly multi-core. Therefore, we have implemented a parallel version of the compact grid method.

We have combined our compact grid representation with the scalable sort-middle grid build method of Ize et al. [IRWP06]. Figure 6 shows the speedup in build time, render time and time to image versus the number of threads for the *Asian Dragon* and *Nature* scene. Rendering time scales almost perfectly with the number of threads. However, build time does not scale as well, with a speedup of only 3 and 5 for 8 threads. This indicates that the build algorithm is indeed memory-bound. The build method of Ize et al. is designed for a NUMA architecture, in which each node has its own memory controller. This is essential for speeding up memory-bound algorithms. In contrast to the dual-core AMD Opteron CPU's used by Ize et al., the quad-core Intel Xeon X5365 CPU's in our benchmark computer do not have an integrated memory controller. Therefore, building does not scale very well on this architecture.

The hashed grid representation can be combined with the scalable sort-middle grid build method of Ize et al. by parallelizing the computation of the domain bits, and then parallelizing the computation of the hash function by distributing the rows of the grid over the threads.

Although multi-threaded grid building is a viable option, the build times are small compared to those of other acceleration structures. Therefore, single-threaded building and multi-threaded rendering probably offers most bang for buck.

6.6 Applications

The methods presented in this paper are well suited for interactive ray tracing of dynamic scenes, and might be a viable alternative for recent methods for interactive ray tracing of dynamic scenes such as the dynamic bounding volume hierarchy methods by Günther et al. [GPSS07] and Wald [Wal07]. Although the time to image given in table 2 can simply be extrapolated to frames per second, we have also tested our methods using an animation. Figure 7 and the accompanying video show an animation of the *Jessi* model walking. The character was modeled and animated using *Poser 7*. The animated *Jessi* model consists of 260.72*K* dynamic triangles. The quarter million triangles animated *Jessi* model is rendered at a resolution of 512×512, at 8.38 *FPS*

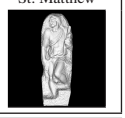
	David	St. Matthew
		
scene statistics		
#tri's	56.23 M	372.77 M
memory	1.89 GB	12.50 GB
compact grid statistics		
build time	5.81 s	N/A
render time	1.74 s	N/A
time to image	7.55 s	N/A
memory	1.17 GB	N/A
hashed grid statistics		
build time	8.92 s	57.82 s
render time	1.29 s	2.934 s
time to image	10.21 s	60.75 s
memory	379.94 MB	2.36 GB

Table 3: Ray tracing large scenes. Scene statistics, compact grid statistics and hashed grid statistics for the *David* and *St. Matthew* model. (Compact grid statistics for the *St. Matthew* model are missing due to memory exhaustion.)

using 6.96 *Mb* of memory with the compact grid method, and at 7.34 *FPS* using 3.88 *Mb* of memory with the hashed grid method.

The methods presented in this paper are also well suited for ray tracing very large models, due to their low memory consumption and fast build algorithms. Table 3 shows results for the compact grid method and the hashed grid method for the *David* and *St. Matthew* models, rendered at a resolution of 1024×1024. The 56.23 million triangles *David* model can be visualized in less than eight seconds. The time to image for the 372.77 million triangles *St. Matthew* model, one of the largest models available, is only about 60 seconds.

7 Conclusion

We have presented two methods for representing and building grids for ray tracing. The compact grid method uses less memory and builds grids faster than traditional representations based on lists and dynamic arrays. The hashed grid method uses even less memory and still builds grids faster than traditional representations. We have shown that the compact grid method and the hashed grid methods have several important advantages over alternative acceleration structures, such as a short build time, a short time to image, robust ray traversal, and easy implementation.

In future work, we would like to extend these methods to hierarchical grids, and we would like to apply the idea of replacing dynamic data structures with static data structures to other acceleration structures.

Acknowledgments

Ares Lagae is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO). We are grateful to the anonymous reviewers for their valuable comments. We acknowledge *The Stanford 3D Scanning Repository*, *The Digital Michelangelo Project*, the *bwfirt* benchmark, Matthias Rolf, Bernhard Finkbeiner and Greg Ward for the scenes.

References

[AU77] AHO A. V., ULLMAN J. D.: *Principles of Compiler Design*. Addison-Wesley Longman Publishing Co., Inc., 1977.

[CHM97] CZECH Z. J., HAVAS G., MAJEWSKI B. S.: Perfect hashing. *Theoretical Computer Science 182*, 1-2 (1997), 1–143.

[CLRS01] CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C.: Introduction to algorithms.

[CW88] CLEARY J. G., WYVILL G.: Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer 6*, 2 (1988), 65–83.

[Dev88] DEVILLERS O.: *Methodes d'optimisation du tracé de rayons*. PhD thesis, Université de Paris-sud, 1988.

[FTI86] FUJIMOTO A., TANAKA T., IWATA K.: Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications 6*, 4 (April 1986), 16–26.

[GIK*07] GRIBBLE C. P., IZE T., KENSLER A., WALD I., PARKER S. G.: A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007).

[Gla89] GLASSNER A. S. (Ed.): *An introduction to ray tracing*. Academic Press Ltd., 1989.

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (2007), pp. 113–118.

[Hai99] HAINES E.: Quicker grid generation via memory allocation. *Ray Tracing News 12*, 1 (1999).

[Hai01] HAINES E.: Siggraph 2001 ray tracing roundtable report. *Ray Tracing News 14*, 1 (2001).

[Hav02] HAVRAN V.: Mailboxing, yea or nay? *Ray Tracing News 15*, 1 (2002).

[IRWP06] IZE T., ROBERTSON C., WALD I., PARKER S. G.: An evaluation of parallel grid construction for ray tracing dynamic scenes. In *Proceedings of the IEEE 2006 Symposium on Interactive Ray Tracing* (2006), pp. 47–55.

[ISP07] IZE T., SHIRLEY P., PARKER S.: Grid creation strategies for efficient ray tracing. In *Proceedings of the IEEE 2007 Symposium on Interactive Ray Tracing* (2007).

[JW89] JEVANS D., WYVILL B.: Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89* (1989), pp. 164–172.

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. *ACM Transaction on Graphics 25*, 3 (2006), 579–588.

[PH04] PHARR M., HUMPHREYS G.: *Physically Based Rendering*. Morgan Kaufmann Publishers, Inc., San Fransisco, CA, USA, 2004.

[Pha02] PHARR M.: Array good, linked list bad. *Ray Tracing News 15*, 1 (2002).

[PMS*99] PARKER S., MARTIN W., SLOAN P.-P. J., SHIRLEY P., SMITS B., HANSEN C.: Interactive ray tracing. In *Symposium on Interactive 3D Graphics* (1999), pp. 119–126.

[RGH*07] RAAB M., GRÜNSCHLOSSL., HANIKAZ J., FINCKHX M., KELLER A.: Benchmarking ray tracing for realistic light transport algorithms, 2007.

[Shi00] SHIRLEY P.: *Realistic ray tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2000.

[Shi02] SHIRLEY P.: Objects per grid cell. *Ray Tracing News 15*, 1 (2002).

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum 26*, 3 (2007).

[Suf07] SUFFERN K.: *Ray Tracing from the Ground Up*. A. K. Peters, Ltd., Natick, MA, USA, 2007.

[Wal07] WALD I.: On fast construction of SAH based bounding volume hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (2007).

[WH06] WALD I., HAVRAN V.: On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *Proceedings of the IEEE 2006 Symposium on Interactive Ray Tracing* (2006), pp. 61–69.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics 25*, 3 (2006), 485–493.

[WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006 (Proceedings of the 17th Eurographics Symposium on Rendering)* (2006), pp. 139–149.

[WMG*07] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the art in ray tracing animated scenes. In *Eurographics 2007 State of the Art Reports* (2007).