

# Time and Space Efficient Grids for Ray Tracing

*Ares Lagae*      *Philip Dutré*

*Report CW 504, November 2007*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Time and Space Efficient Grids for Ray Tracing

*Ares Lagae*      *Philip Dutré*

*Report CW504, November 2007*

Department of Computer Science, K.U.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 minimum storage method consist 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 storage 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 show that these methods are more efficient in both time and space than traditional methods based on linked lists and dynamic arrays. We also 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 very effective despite its simplicity.

**Keywords :** ray tracing, acceleration structure, grid, row displacement compression, perfect hashing

**CR Subject Classification :** I.3.3

# Time and Space Efficient Grids for Ray Tracing

Ares Lagae & Philip Dutré  
Department of Computer Science  
Katholieke Universiteit Leuven<sup>†</sup>

---

## 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 minimum storage method consist 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 storage 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 show that these methods are more efficient in both time and space than traditional methods based on linked lists and dynamic arrays. We also 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 very effective despite its simplicity.*

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 the 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 ev-

ery frame, and rather than minimizing render time, the time to image, the sum of the build time and the 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.

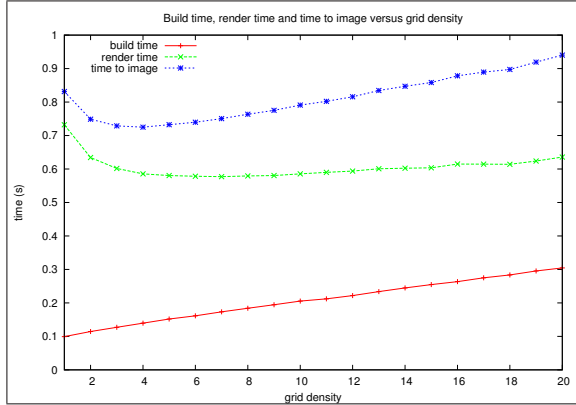
Uniform grids were used in one of the first systems for interactive ray tracing [PMS\*99]. Recent work on grids for ray tracing concentrated on fast traversal [WIK\*06], parallelizing the build process [IRWP06], and choosing the grid size [ISP07]. In this work, we focus on efficient data structures and algorithms for representing and building grids.

The contributions of this work are:

- 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.

---

<sup>†</sup> {ares.lagae, philip.dutre}@cs.kuleuven.ac.be



**Figure 1:** Build time (red), render time (green) and time to image (blue) versus grid density for the Happy Buddha model, rendered at a resolution of  $1024 \times 1024$ . A grid density of 4 minimizes the time to image. Other models have very similar statistics.

- 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.

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.

**Overview** Section 2 motivates design decisions. In section 3 we present our minimum storage grid construction method, and in section 4 we present our hashed storage grid construction method. Section 5 discusses applications. In section 6 we conclude.

## 2. The Grid Acceleration Structure

The grid acceleration structure was introduced by [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 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

$$M = \rho N, \quad (1)$$

where  $\rho$  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

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

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, and then rounded to the closest integer.

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 with 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 faster build times and slower 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 slower build times and faster render times.

Since [IRWP06, 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.

**Traversal** We use a a single ray grid traversal method [CW88]. Since we focus on data structures and algorithms for efficiently building grids, we have not compared different traversal algorithms.

It is worth mentioning however that an efficient ray packet grid traversal method was recently proposed [WIK\*06], and that the grids we build can also be used with that method.

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

**Models** The methods presented in this paper were tested using scanned models from *The Stanford 3D Scanning Repository* (including the *Lucy* model), and models from *The Digital Michelangelo Project* (the *David*, *St. Matthew* and *Atlas* model). These models are composed of compact primitives and work well with grids. However, it is important to realize that the methods presented in this paper do not rely on these model characteristics.

**Implementation and Benchmarks** 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, and the code is single-threaded.

The benchmarks were obtained on a computer with a 2.93 GHz Intel Core 2 Extreme X6800 CPU and 4 Gb of memory. The benchmarks for the 13 Gb *St. Matthew* model and the 18 Gb *Atlas* model were obtained on a computer with a 2.2 GHz AMD Opteron 875 CPU and 32 Gb of memory.

### 3. The Minimum Storage Method

In this section we present the minimum storage 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 minimum storage 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 [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]. 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.

**Minimum Storage** 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 minimum storage data structure for representing a grid consists of two static arrays. This is illustrated in figure 2(d). 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  $C[i]$

$$C[z][y][x] = C[\left(\left(\left(M_y z\right) + y\right) M_x\right) + x]. \quad (3)$$

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 minimum storage data structure for representing a grid are exactly one machine word 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.

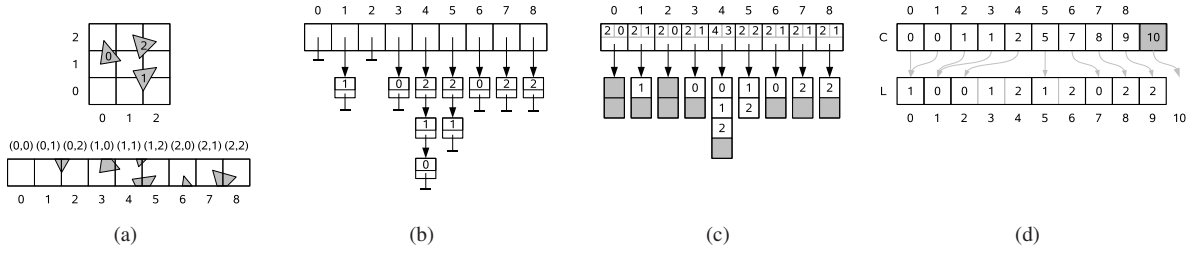
The array  $L$  uses 32-bit unsigned integers to index the objects. This is sufficient for indexing over 4 billion objects. The array  $C$  uses 32-bit unsigned integers to index the object lists. Note that storing 32-bit unsigned integer indices rather than pointers results in additional memory savings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

#### 3.2. Algorithm

In this subsection, we present the algorithm for building the minimum storage 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



**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. (d) The minimum storage grid data structure presented in this paper.

cells overlapped by the 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 the object decrementing the offset of the cell and storing the object index at that offset. This can be accomplished 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.

### 3.3. Results and Discussion

Table 1 shows several results for the *Bunny*, *Armadillo*, *Dragon*, *Happy Buddha*, *Asian Dragon*, *Thai Statue* and *Lucy* model.

For each model, the number of triangles and the size of the model is given. The size of the model is computed using a representation of 36 bytes per triangle (three single precision floating point numbers for each coordinate).

The most important figures are the time to image and the memory used by the grid. The models are rendered at a resolution of  $1024 \times 1024$  with one ray per pixel and diffuse shading.

For example, the *Lucy* model consists of 28,055,742 triangles and has a memory footprint of 963.22 Mb. The time to build a grid for the model is 3.1445 s, and the grid has a memory footprint of 605.98 Mb. The time to render the model is 1.7564 s, and the total time to image is therefore 4.9009 s.

Several other statistics are also provided. The memory footprint of the grid is roughly about three quarter of the memory footprint of the model. The table also mentions the memory footprint broken down into the memory needed for the cells (the  $C$  array) and the memory needed for the object lists (the  $L$  array). The memory needed for the cells is significantly larger than the memory needed for the object lists. Note that the majority of the cells is empty, and that each triangle is only in a few cells. Intersecting rays visit on the average roughly two cells. The average number of intersection tests per intersecting ray is higher than for other acceleration structures.

Figure 3 shows a comparison of memory footprint and build time between the minimum storage method and straightforward implementations using linked lists and dynamic arrays. We used `std::list` and `std::vector` from the Standard C++ Library. The memory footprint (figure 3(a)) of the minimum storage method is about a factor 4 smaller than the memory footprint of the implementations using linked lists and dynamic arrays. This verifies the theoretical analysis of section 3.1. The build time (figure 3(a)) of the minimum storage method is about a factor 3 smaller than the build time of the implementations using linked lists and dynamic arrays. Although the minimum storage method requires an extra pass over the triangles, the method has a better locality of reference and does not need to maintain dynamic data structures. The render time (not shown) of the minimum storage method and the implementations using linked lists and dynamic arrays are about the same. The render time is the largest for the implementation using linked








	Bunny 	Armadillo 	Dragon 	Happy Buddha 	Asian Dragon 	Thai Statue 	Lucy 
#tri's	69,451	345,944	871,414	1,087,716	7,219,045	10,000,000	28,055,742
memory	2.38 Mb	11.88 Mb	29.92 Mb	37.34 Mb	247.85 Mb	343.32 Mb	963.22 Mb
grid res	71 × 71 × 55	109 × 129 × 99	223 × 157 × 100	121 × 295 × 121	428 × 237 × 284	302 × 508 × 261	485 × 278 × 832
build time	0.0090 s	0.0484 s	0.1123 s	0.1401 s	0.7957 s	1.1454 s	3.1445 s
memory	1.90 Mb	8.32 Mb	21.86 Mb	27.50 Mb	155.27 Mb	222.53 Mb	605.98 Mb
# cells	277,255	1,392,039	3,501,100	4,319,095	28,807,824	40,041,576	112,178,560
% empty cells	92.32 %	96.53 %	95.44 %	94.86 %	98.99 %	98.44 %	99.00 %
avg # tri's / non-empty cell	10.34	16.35	13.96	13.02	40.79	29.25	41.50
avg # cells / tri	3.17	2.28	2.56	2.66	1.65	1.83	1.66
mem cells	1.06 Mb	5.31 Mb	13.36 Mb	16.48 Mb	109.89 Mb	152.75 Mb	427.93 Mb
mem object lists	0.84 Mb	3.01 Mb	8.51 Mb	11.03 Mb	45.37 Mb	69.78 Mb	178.06 Mb
render time	0.7031 s	0.9338 s	0.8255 s	0.5917 s	1.4294 s	1.5506 s	1.7564 s
avg # non-empty cells / isect ray	2.01	2.17	2.08	2.25	1.99	2.16	1.93
avg # isect tests / isect ray	23.38	38.35	31.77	34.48	91.66	66.93	92.15
time to image	0.7121 s	0.9823 s	0.9378 s	0.7318 s	2.2251 s	2.6960 s	4.9009 s

Table 1: Results for the minimum storage method. See subsection 3.3 for a detailed discussion.








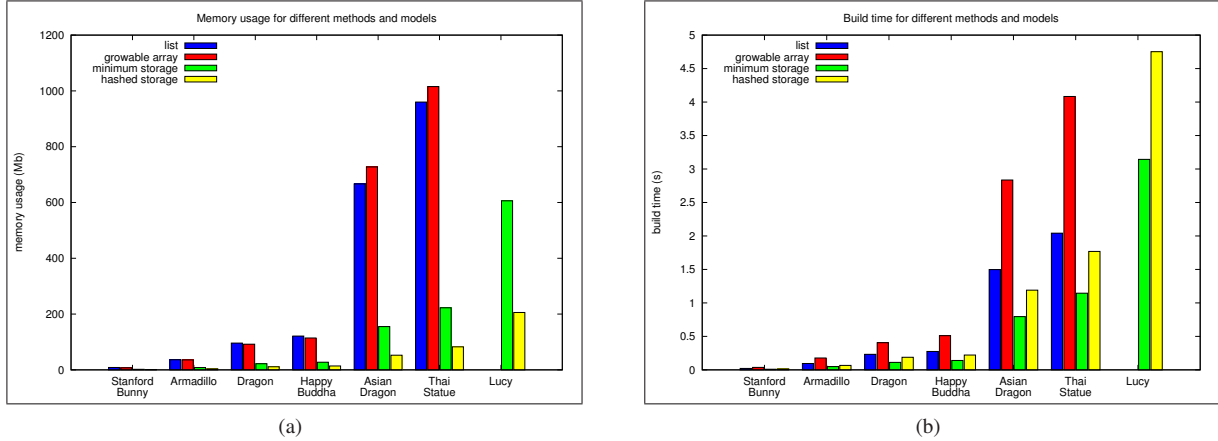
	Bunny 	Armadillo 	Dragon 	Happy Buddha 	Asian Dragon 	Thai Statue 	Lucy 
#tri's	69,451	345,944	871,414	1,087,716	7,219,045	10,000,000	28,055,742
memory	2.38 Mb	11.88 Mb	29.92 Mb	37.34 Mb	247.85 Mb	343.32 Mb	963.22 Mb
grid res	71 × 71 × 55	109 × 129 × 99	223 × 157 × 100	121 × 295 × 121	428 × 237 × 284	302 × 508 × 261	485 × 278 × 832
build time	0.0143 s	0.0655 s	0.1826 s	0.2187 s	1.1542 s	1.6696 s	4.5293 s
memory	0.99 Mb	3.49 Mb	10.04 Mb	12.91 Mb	50.84 Mb	78.75 Mb	199.04 Mb
# cells	277,255	1,392,039	3,501,100	4,319,095	28,807,824	40,041,576	112,178,560
# non-empty cells	21,289	48,261	159,669	221,965	291,586	625,374	1,124,791
data density	7.68 %	3.47 %	4.56 %	5.14 %	1.01 %	1.56 %	1.00 %
hash table size	27,871	69,525	277,365	322,694	466,915	967,592	1,763,271
hash table load factor	76.38 %	69.42 %	57.57 %	68.78 %	62.45 %	64.63 %	63.79 %
mem domain bits	0.03 Mb	0.17 Mb	0.42 Mb	0.51 Mb	3.43 Mb	4.77 Mb	13.37 Mb
mem offset table	0.01 Mb	0.05 Mb	0.06 Mb	0.14 Mb	0.26 Mb	0.51 Mb	0.88 Mb
mem hash table	0.11 Mb	0.27 Mb	1.06 Mb	1.23 Mb	1.78 Mb	3.69 Mb	6.73 Mb
compression ratio	6.86	11.07	8.70	8.75	20.08	17.03	20.39
mem object lists	0.84 Mb	3.01 Mb	8.51 Mb	11.03 Mb	45.37 Mb	69.78 Mb	178.06 Mb
render time	0.6951 s	0.9111 s	0.8185 s	0.5793 s	1.2763 s	1.4453 s	1.4792 s
time to image	0.7094 s	0.9766 s	1.0011 s	0.7980 s	2.4306 s	3.1149 s	6.0085 s

Table 2: Results for the hashed storage method. See subsection 4.3 for a detailed discussion.



**Figure 3:** A comparison of (a) memory consumption and (b) build times between our methods and straightforward implementations using linked lists and dynamic arrays, for different models. Our minimum storage method consistently uses less memory and performs faster than the straightforward implementations. Our hashed storage method uses even less memory and is only a bit slower than the minimum storage method. Missing timings for the Lucy models are due to memory exhaustion.

lists and the smallest for the minimum storage method. This is again due to locality of reference.

To our knowledge, the minimum storage 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 minimum storage method is very similar to counting sort, a method for sorting in linear time [CLRS01].

#### 4. The Hashed Storage Method

In this section we present the hashed storage 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.

Table 1 shows that 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 (each triangle is referenced in roughly two object lists). The array with the cell offsets is therefore the best candidate for further reduction in size.

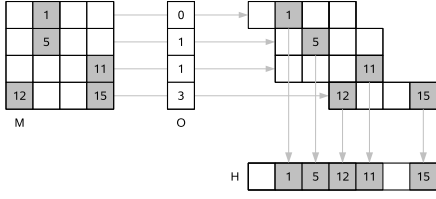
The array with the cell offsets is essentially a sparse array, and the typical solution to avoid the storage of the complete array is to use hashing [CLRS01]. Traditional hash tables are dynamic data structures, that support insertion and removal of elements, and handle collisions of the hash function. 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 [CLRS01], [CHM97] and [LH06].

##### 4.1. Perfect Hashing using Row Displacement Compression

Our method for computing a perfect hash function is based on the row displacement compression algorithm. Row displacement compression was introduced in 1977 by [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



**Figure 4:** Row displacement compression. The square matrix  $M$  is compressed into a hash table  $H$  by displacing the rows, and storing the offset of each row in the offset table  $O$ .

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 4.

Each non-zero element  $C(i, j)$  corresponds with  $H[O[i] + j]$ , 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 4 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. It is important to realize that, although the position of the elements cannot be changed, their value can be changed.

#### 4.2. Data Structure and Algorithm

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

The data structure consists of four static arrays. The array  $L$  consist of the concatenation of all object lists, as in the minimum storage method. The array  $C$  of the minimum storage 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 the 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 minimum storage 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.

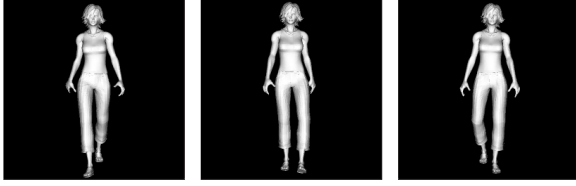
#### 4.3. Results and Discussion

Figure 3 and table 2 shows several results for the *Bunny*, *Armadillo*, *Dragon*, *Happy Buddha*, *Asian Dragon*, *Thai Statue* and *Lucy* model.

Compared to the minimum storage method, the memory footprint of the grid is significantly reduced, the build time moderately increases and the render time slightly decreases. The decrease in render time is due to an improved locality of reference.

For example, the *Lucy* model consists of 28,055,742 triangles and has a memory footprint of 963.22 Mb. The time to build a grid for the model increased from 3.1445 s to 4.5293, and the memory footprint of the grid decreased from 605.98 Mb to 199.04 Mb. The time to render the model decreased from 1.7564 s to 1.4792, and the total time to image increased from 4.9009 s to 6.0085 s.

The hashed storage method is slightly slower than the min-



**Figure 5:** Interactive ray tracing. Three frames of an animation of the Jessi model walking. The animated Jessi model consists of 260,724 dynamic triangles. The animation is rendered at a resolution of  $512 \times 512$  at 8.12 FPS.

imum storage methods, but still faster than straightforward implementations using linked lists and dynamic arrays. 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 [LH06] in the context of the GPU. Their method produces hash tables with a higher load factor but the algorithm is 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 of the object lists.

## 5. Applications

In this section we discuss applications of our methods in interactive ray tracing and ray tracing large models, and we provide a short comparison with previous techniques.

### 5.1. Interactive Ray Tracing



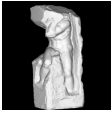
Due to the short time to image, the grid acceleration structure is well suited for interactive ray tracing of dynamic scenes. Although the time to image given in tables 1 and 2 can simply be extrapolated to frames per second, we have also tested our methods using an animation.

Figure 5 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 260,724 dynamic triangles and has a memory footprint of 8.95 Mb. The animation is rendered at a resolution of  $512 \times 512$ , at 8.12 FPS using 6.96 Mb of memory with the minimum storage method, and at 7.21 FPS using 3.88 Mb of memory with the hashed storage method.

The quarter million triangles animated Jessi model can be rendered at an interactive framerate.

### 5.2. Ray Tracing Large Models

Due to the low memory consumption and the fast build algorithms, the grid acceleration structure is well suited for

	David	St. Matthew	Atlas
			
#tri's	56,230,343	372,767,445	507,512,682
memory	1.9 Gb	13 Gb	18 Gb
<b>minimum storage method</b>			
build time	5.7598 s	74.3785 s	116.242 s
memory	1.17 Gb	3.63 Gb	6.28 Gb
render time	1.5205 s	6.3858 s	12.3975 s
time to image	7.2803 s	80.7643 s	128.64 s
<b>hashed storage method</b>			
build time	8.4717 s	105.6 s	142.093 s
memory	379.93 Mb	2.31 Mb	3.01 Gb
render time	1.1905 s	4.5713 s	7.12935 s
time to image	9.6622 s	110.171 s	149.222 s

**Table 3:** Ray tracing large models. Results for the minimum storage method and the hashed storage method for large models, rendered at a resolution of  $1024 \times 1024$ . See subsection 5.2 for a detailed discussion.

ray tracing large models. Tables 1 and 2 include results for the 28 million triangles *Lucy* model, but we have tested our methods on even larger models.

Table 3 shows results for the minimum storage method and the hashed storage method for the *David*, *St. Matthew* and *Atlas* models, rendered at a resolution of  $1024 \times 1024$ . The results for the *David* model were obtained on a computer with a 2.93 GHz Intel Core 2 Extreme X6800 CPU and 4 Gb of memory. The results for the 13 Gb *St. Matthew* model and the 18 Gb *Atlas* model were obtained on a computer with a 2.2 GHz AMD Opteron 875 CPU and 32 Gb of memory. The 56 million triangles *David* model can be visualized on a commodity PC with only 4 Gb of memory in less than ten seconds. The time to image for the half a billion triangles *Atlas* model, probably the largest model available, is only two minutes.

### 5.3. Comparison with Previous Techniques

Although a detailed comparison of the time to image and memory usage for different acceleration structures is beyond the scope of this work, we would like to mention some numbers.

Table 4 shows a comparison between our methods and several state of the art acceleration structures. These results are difficult to compare, but they at least indicate that our method is competitive, especially for larger models.

The performance of our method can be further improved by using the coherent grid traversal method recently introduced by [WIK\*06], by parallelizing the render process, and by parallelizing the build process as in [IRWP06].

	Bunny	Armadillo	Dragon	Happy Buddha	Asian Dragon	Thai Statue	Lucy
<b>kd-tree of [WH06]</b> (build algorithm with time complexity of $O(N \log N)$ , the theoretical lower bound)							
build time	3.2 s	5 s	16 s	21 s		430 s	
<b>InView (timings from [WK06])</b> (very simple shader, $2 \times 2$ SSE accelerated ray bundles, $640 \times 480$ pixels, 2.8 GHz Intel Pentium 4HT, max $10^6$ triangles)							
time to image	9.283 s		44.500 s	53.819 s			
memory	5.90 Mb		24.99 Mb	31.01 Mb			
<b>kd-tree of [WH06] (timings from [WK06])</b> (very simple shader, $2 \times 2$ SSE accelerated ray bundles, $640 \times 480$ pixels, 2.6 GHz AMD Opteron)							
time to image	4.800 s		23.900 s	32.200 s			
<b>kd-tree of [WK06]</b> (very simple shader, $2 \times 2$ SSE accelerated ray bundles, $640 \times 480$ pixels, 2.8 GHz Intel Pentium 4HT)							
time to image	0.445 s		3.106 s	3.695 s			
memory	4.15 Mb		22.90 Mb	29.16 Mb			
<b>bounding interval hierarchy [WK06]</b> (very simple shader, $2 \times 2$ SSE accelerated ray bundles, $640 \times 480$ pixels, 2.8 GHz Intel Pentium 4HT)							
time to image	0.176 s		1.557 s	1.837 s			
memory	0.93 Mb		12.84 Mb	16.54 Mb			
<b>kd-tree of [SSK07]</b> ( $4 \times 4$ SIMD ray packet traversal, $640 \times 480$ pixels, 2-way 3.0 GHz Intel Core 2 Duo, 1 core)							
time to image	0.104 s		0.751 s	0.696 s			
<b>kd-tree of [SSK07]</b> (lighting and shadows (1 point light source), $4 \times 4$ SIMD ray packet traversal, $1024 \times 1024$ pixels, 2-way 3.0 GHz Intel Core 2 Duo, 4 threads on 4 cores)							
construction time				0.45 s	1.7 s	2.46 s	
rendering performance				15.4 FPS	2.9 FPS	3.14 FPS	
peak size of memory footprint					1.24 Gb	1.42 Gb	
<b>minimum storage method</b> (diffuse shading, single ray traversal, $1024 \times 1024$ pixels, 2.93 GHz Intel Core 2 Extreme X6800, 1 core)							
time to image	0.7121 s	0.9823 s	0.9378 s	0.7318 s	2.2251 s	2.6960 s	4.9009 s
memory	2.38 Mb	11.88 Mb	29.92 Mb	37.34 Mb	247.85 Mb	343.32 Mb	963.22 Mb
<b>hashed storage method</b> (diffuse shading, single ray traversal, $1024 \times 1024$ pixels, 2.93 GHz Intel Core 2 Extreme X6800, 1 core)							
time to image	0.7094 s	0.9766 s	1.0011 s	0.7980 s	2.4306 s	3.1149 s	6.0085 s
memory	0.99 Mb	3.49 Mb	10.04 Mb	12.91 Mb	50.84 Mb	78.75 Mb	199.04 Mb

**Table 4:** A comparison between our methods and several state of the art acceleration structures. (When comparing statistics take into account the conditions under which they were produced.)

## 6. Conclusion

We have presented two methods for representing and building grids for ray tracing. The minimum storage method has minimal memory requirements and builds grids faster than traditional implementations based on linked lists and dynamic arrays. The hashed storage method reduces these memory requirements even further at the cost of a small increase in build time.

We have shown 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 should not be dismissed as an inferior acceleration structure. For visualizing massive models, such as the *David*, *St. Matthew* or *Atlas* model, the grid acceleration structure is probably even superior to many other acceleration structures.

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

We acknowledge *The Stanford 3D Scanning Repository* for the *Bunny*, *Armadillo*, *Dragon*, *Happy Buddha*, *Asian Dragon*, *Thai Statue* and *Lucy* model, and *The Digital Michelangelo Project* for the *David*, *St. Matthew* and *Atlas* model. Ares Lagae is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

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