To appear in the High Performance Graphics 2009 conference proceedings

A Parallel Algorithm for Construction of Uniform Grids

Javor Kalojanov *
Saarland University

Philipp Slusallek †
Saarland University
DFKI Saarbrücken

Abstract

We present a fast, parallel GPU algorithm for construction of uniform grids for ray tracing, which we implement in CUDA. The algorithm performance does not depend on the primitive distribution, because we reduce the problem to sorting pairs of primitives and cell indices. Our implementation is able to take full advantage of the parallel architecture of the GPU, and construction speed is faster than CPU algorithms running on multiple cores. Its scalability and robustness make it superior to alternative approaches, especially for scenes with complex primitive distributions.

Keywords: grid, uniform grid, regular grid, CUDA, ray tracing

1 Introduction

Uniform grids are used in a wide variety of applications including ray tracing of dynamic scenes. Their use as an acceleration structure for ray tracing is motivated by the simplicity of the structure which allows fast, per-frame rebuild, even for very large scenes. While other spatial structures, such as kd-trees and BVHs based on the Surface Area Heuristic, can provide better ray tracing performance, their construction is very time consuming. GPU-algorithms that allow per-frame rebuild of hierarchical data structures in real time were introduced only recently [Lauterbach et al. 2009; Zhou et al. 2008] and build times are still considerably slower than those of grids. We present a fast and parallel grid construction algorithm that can speed up rendering of animated scenes. We demonstrate that in a number of different scenarios, the build time allows us to trace a significant number of rays before the overall performance becomes inferior to concurrent approaches based on hierarchical data structures.

The recent introduction of the CUDA programming model [Nickolls et al. 2008], along with the advancement in GPU hardware design, made GPUs an attractive architecture for implementing parallel algorithms such as ray tracing (see [Popov et al. 2007; Günther et al. 2007; Zhou et al. 2008; Lauterbach et al. 2009]). The use of uniform grids has been investigated both for CPU and GPU ray tracers [Wald et al. 2006; Lagae and Dutré 2008; Purcell et al. 2002; Es and İşler 2007] but we are not aware of other grid construction algorithms on current GPU architectures implemented with CUDA.

The main disadvantage of uniform grids is the inability to adapt to the distribution of the geometry in the scene. They tend to eliminate less intersection candidates compared to other structures. Nevertheless Wald et al. [2006] demonstrate an efficient traversal method for coherent rays on CPUs with performance competitive to approaches based on hierarchical structures. In this paper, we present a novel construction algorithm that exploits the massively parallel architecture of current GPUs. The construction of the grid does not depend on atomic synchronization, which enables us to efficiently handle scenes with arbitrary primitive distributions. We use an atomic addition on shared memory in a non-critical part of our implementation for convenience, but this is not necessary and does not hurt performance.

*e-mail:javor@graphics.cs.uni-sb.de
†e-mail:slusallek@dfki.de

2 Previous Work

While fast grid construction algorithms for CPUs have been investigated [Ize et al. 2006; Lagae and Dutré 2008], to our knowledge, there are no attempts to efficiently implement a *construction* algorithm for a highly parallel architecture such as the GPU. Eisemann and Décoret [2006] propose to use GPU rasterization units for *voxelization* of polygonal scenes in a grid. Their approach is limited to computing a *boundary representation* of the scene, which is only a part of the information required for ray tracing. Patidar and Narayanan [2008] propose a fast construction algorithm for a grid-like acceleration structure, but their algorithm is limited to fixed resolution and while it performs well for scanned models, it relies on synchronization via atomic functions which makes it sensitive to triangle distributions. Ize et al. [2006] describe a number of parallel construction algorithms for multiple CPUs. Their sort-middle approach also does not rely on atomic synchronization, but the the triangle distribution in the scene influences the work distribution and the algorithm performance.

The idea of reducing the construction process to sorting has been used by Lauterbach et al. [2009] for their LBVH structure. In the particle simulation demo in the CUDA SDK [Green 2008] sorting is used for construction of an uniform grid over a set of particles. The approach described here, and concurrently proposed by Ivson et al. [2009], is more general because it handles geometric primitives overlapping any number of cells. This allows for construction of grids that can be used as acceleration structures for ray tracing geometric surfaces.

3 GPU Grid Construction

In the following section we describe our parallel grid construction algorithm and its implementation in CUDA.

3.1 Data Structure

Like the compact grid representation by Lagae and Dutré in [Lagae and Dutré 2008], we store the structure in two parts. An indirection array contains triangle references. The grid cells are stored separately. Each grid cell stores the beginning and the end of a range inside the array such that the triangles referenced in this interval are exactly those, contained in the cell. In Lagae's representation, a single index per cell is stored. This index is both the beginning of the interval of the current cell, and the end of the interval for the previous. We store independent cell intervals which doubles the memory consumption but simplifies the parallel building process, by allowing us to initialize all cells as empty prior to the build and then only touch the non-empty cells.

Algorithm 1 Data-Parallel Grid Construction. Kernel calls are suffixed by $<<<>>>$.

```
    b ← COMPUTE BOUNDS()
 2: r ← COMPUTE RESOLUTION()
    t ← UPLOAD TRIANGLES()
 4: G ← 128, B ← 256, i ← ARRAY OF G + 1 ZEROES
    i ← COUNT REFERENCES<<<G,B>>>(t, b, r)
 6: i ← EXCLUSIVE SCAN<<<1,G + 1>>>(i)
    n ← i[G]                          ▷ NUMBER OF REFERENCES
 8: a ← ALLOCATE REFERENCES ARRAY(n)
    a ← WRITE REFERENCES<<<G,B>>>(t, b, r, i)
10: a ← SORT(a)
    cells ← EXTRACT CELL RANGES<<<>>>(a)
```

3.2 Algorithm

Constructing a grid over a scene consisting of triangles (or any other type of primitive), amounts to determining the bounding box of the scene, the resolution of the grid in each dimension and, for each cell of the grid, which triangles overlap it. Our construction algorithm (Algorithm 1) consists of several steps. First we compute an unsorted array that contains all primitive-cell pairs. This array is sorted and the grid data is extracted from it in a final step. The same idea is used in the particle simulation demo in the CUDA SDK [Green 2008]. The only difference is that each of the primitives handled by our algorithm can overlap arbitrary number of cells.

3.2.1 Initialization

Once the bounding box of the scene and the grid resolution is determined on the host, we upload the primitives to the GPU. Since computing the bounding box involves iteration over the scene primitives, we perform this operation while reorganizing the data for upload.

3.2.2 Counting Triangle Copies

Because it is not possible to dynamically allocate memory on GPUs, we have to know the size of the array that stores the primitive references in advance. To compute it, we first run a kernel that loads the scene primitives in parallel and for each determines the number of cells it overlaps (Line 5). Each thread writes the counts into an individual shared memory cell and then a reduction is performed to count the total number of primitive-cell pairs computed by each thread block. Next we perform an exclusive scan over the resulting counts to determine the total number of primitive-cell pairs and allocate an output array on the device. The scan additionally gives us the number of pairs each block would output.

3.2.3 Writing Unsorted Pairs

Having the required memory for storage, we run a second kernel (Line 9). Each thread loads a primitive, computes again how many cells it overlaps and for each overlapped cell writes a pair consisting of the cell and primitive indices. The output of the exclusive scan is used to determine a segmentation of the array between the thread blocks. We have to avoid write conflicts inside a block since each thread has to write a different amount of pairs. We can use the shared memory to write the pair counts and perform a prefix sum to determine output locations. In our implementation each thread atomically increments a single per-block counter in shared memory to reserve the right amount of space.

3.2.4 Sorting the Pairs

After being written, the primitive-cell pairs are sorted by the cell index via radix sort. We used the radix sort implementation from the CUDA SDK examples for this step of the algorithm.

3.2.5 Extracting the Grid Cells

From the sorted array of pairs it is trivial to compute the reference array as well as the triangles referenced in each cell. We do this by invoking a kernel that loads chunks of the sorted pairs array into shared memory. We check in parallel (one thread per pair) if two neighboring pairs have different cell indexes. This indicates cell range boundary. If such exists, the corresponding thread updates the range indexes in both cells. Note that in this stage of the algorithm only non-empty cells are written to. After this operation is completed the kernel writes an array that stores only the primitive references to global memory. In this part of the implementation we read the data from shared memory and the writes to global memory are coalesced, so the introduced overhead is very small. It is also possible to directly use the sorted pairs to query primitives during rendering. However getting rid of the cell indices frees space, and accesses to the compacted data during rendering are more likely to get coalesced.

3.3 Triangle Insertion

In our algorithm we have to compute which cells are overlapped by each input triangle twice. When counting the total number of references (Line 5) we conservatively count the number of cells overlapped by the bounding box of the triangle. Afterwards, when we want to write triangle-cell pairs (Line 9), we do a more precise (but still efficient) test. We check if each cell overlapped by the triangle bounding box is intersected by the plane in which the triangle lies. An exact triangle-box overlap test [Akenine-Möller 2001] did not pay off for any of the tested scenes.

We test our implementation only with scenes consisting of triangles, but the same approach can be used for various geometric primitives. The only requirement is that one can (efficiently) determine the cells of the grid that each primitive overlaps.

3.4 Analysis

Both the complexity of the algorithm and the performance of the implementation are dominated by the sorting of the primitive-cell pairs (Figure 1). Under the assumption that the number of cells overlapped by each triangle can be bounded by a constant, all parts of the algorithm have linear work complexity. We chose to use radix sort because it is well suited for the data we have, it also has linear work complexity, and there exists a fast and scalable GPU implementation [Sengupta et al. 2007]. Sorting on the device alleviates the need for expensive data transfers. The only information we must communicate to the CPU during construction is the size of the references array so that we can allocate it.

An important advantage of our algorithm is that there are no write conflicts, and hence, no atomic synchronization is required throughout the build. This implies that the performance of the construction algorithm depends only on the number of primitive references that are inserted in the grid, and not on the primitive distribution in the scene. In fact, as discussed in Section 3.2.3, we use an atomic operation on shared memory when we write output pairs. This however is neither necessary (can be done efficiently via prefix sum), nor performance critical since we require a single atomic operation per primitive and not per primitive insertion in a cell.

Scene	Thai Statue	Soda Hall	Conference	Dragon	Sponza	Ruins
Default	$325 \times 547 \times 280$	$262 \times 274 \times 150$	$210 \times 133 \times 50$	$104 \times 147 \times 65$	$116 \times 55 \times 52$	$53 \times 69 \times 52$
Cost-Based	$313 \times 487 \times 281$	$256 \times 268 \times 164$	$164 \times 110 \times 50$	$137 \times 105 \times 71$	$115 \times 67 \times 63$	$60 \times 69 \times 59$

Table 1: *Grid resolutions computed via heuristic (Default) and cost-based approach (Cost-Based). Instead of trying to make the grid cells as close to a cube as possible, one can try to find a resolution that minimizes the expected cost for tracing a random ray trough the grid.*

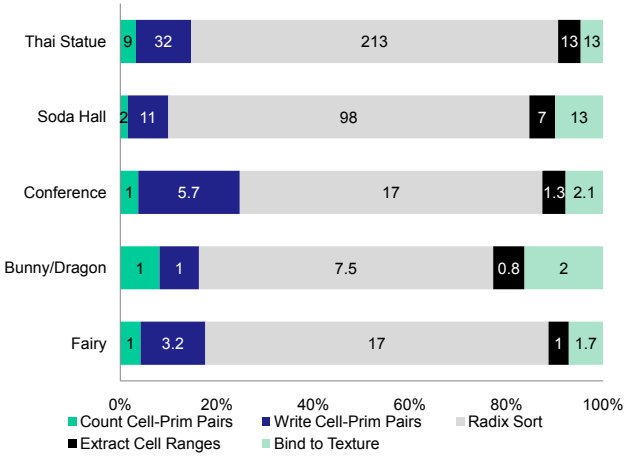

Figure 1: *Times for the different stages of the build algorithm in milliseconds. We also include the time needed to bind the grid cells to a 3D texture.*

The memory requirements for the grid and its construction can become a concern when dealing with very large models. Our method requires additional (but temporary) memory for storing primitive-cell pairs instead of only primitives. We additionally need a second array of pairs during sorting. After the sort stage we extract the primitive references from the sorted array and free the additional space. The main memory bottleneck is the space for storing the grid cells. Each of them is 8 bytes large and we also store empty cells. Despite the relatively large memory footprint, we were able to construct grids for all models that we tested, including the Thai Statue which has 10 million triangles. We discuss a memory issue that we had with this model in the results section.

Even if there is not enough memory for the grid cells, one can modify the algorithm to construct the acceleration structure incrementally. We have not implemented this since the size of the grids and the added memory transfers to the CPU and back will most likely result in build times of more than a second.

3.5 Grid Resolution

An important part of the building process is the choice of the grid resolution. This is the only factor one can vary in order to influence the quality of the structure for ray tracing. Sparser grids cannot eliminate as many intersection candidates but a higher resolution results in bigger cost for traversal. The resolution is typically chosen as:

$$R_x = d_x \sqrt[3]{\frac{\lambda N}{V}}, R_y = d_y \sqrt[3]{\frac{\lambda N}{V}}, R_z = d_z \sqrt[3]{\frac{\lambda N}{V}} \qquad (1)$$

where $\vec{d}$ is the size of the diagonal and V is the volume of the scene's bounding box. N is the number of primitives, and λ is a user-defined constant called *grid density* [Devillers 1988; Jevans and Wyvill 1989]. Like Wald et al. [2006], we set the density to 5 in our tests. A more extensive study on good choices of grid density is done by Ize et al. [2007]. In the following we describe another approach to choosing resolution of uniform grids.

MacDonald and Booth [1990] introduced the Surface Area Metric for measuring the expected cost of a spatial structure for ray tracing. Given the cost for traversing a node of the structure C_t and the cost for testing a primitive for intersection C_i, the expected cost for tracing a ray is

$$C_e = C_t \sum_{n \in Nodes} Pr(n) + C_i \sum_{l \in Leaves} Pr(l) Prim(l) \qquad (2)$$

$Pr(n)$ and $Pr(l)$ are the probabilities with which a random ray will intersect the given node, $Prim(l)$ is the number of primitive stored in the leaf l. In the case of grids, since all cells are regarded as leaves and have the same surface area (i.e. same intersection probability), Equation 2 simplifies to

$$C_e(G) = C_t N_c + C_i \frac{SA(c)}{SA(G)} N_{pr} \qquad (3)$$

where $SA(c)$ and $SA(G)$ are the surface areas of a cell and the grid's bounding box, N_{pr} is the number of primitive references that exist in the grid, and N_c is the expected number of grid cells intersected by a random ray. The surface areas of a cell and the grid can be computed in constant time, and N_c can be bounded by the sum of the number of cells in each dimension. The only non-trivial part for computing the expected cost is counting the number of primitive references in the grid. Since we were able to estimate this number relatively fast (Algorithm 1, Line 5), we tried to find the best grid resolution for several test scenes. We empirically found that $C_t = 1.5$ and $C_i = 8.5$ work well for our rendering algorithm and used Equation 1 to have an initial estimate $\vec{r}$. We exhaustively tested all possible grid resolutions in the range $\left(\frac{3}{4}\vec{r}; \frac{5}{4}\vec{r}\right)$.

While the resulting grids (Table 1) improved rendering performance for our ray tracer, the differences were very small both for primary rays and for path tracing of diffuse surfaces. For example the average number of intersection tests per ray for Sponza and Ruins was reduced by up to 2 which is around 10%. Also the times for cost estimation allowed computing the cost-based resolutions only in a preprocessing stage of the algorithm. Unless noted, we used the default grid resolutions for all tests.

4 Results

We implemented our construction algorithm as a part of a GPU ray tracer in the CUDA programming language. All tests were performed on a machine with an NVIDIA Geforce 280 GTX with 1 GB memory and a Core 2 Quad processor running at 2.66 GHz. The only computationally demanding tasks for which we use the CPU are key frame interpolation for dynamic scenes and data transfers.

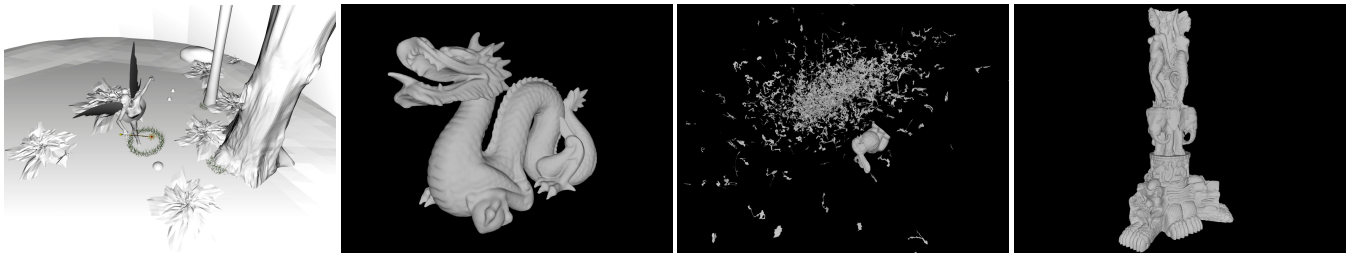

Figure 2: *Some of our test scenes, from left to right - Fairy Forest, Dragon, Bunny/Dragon, and Thai Statue. We can render them at 3, 14, 7 and 5 fps with simple shading in a* 1024×1024 *window and can construct grids in 24, 16, 13 and 280 milliseconds on a GTX280. Frame rate for Fairy Forest and Bunny/Dragon includes rebuild of the grid.*

Scene	Tris	Refs	Resolution	Time
Thai Statue	10M	19M	$325 \times 547 \times 280$	417
Thai Statue	10M	14.8M	$192 \times 324 \times 168$	280
Soda Hall	2.2M	6.7M	$262 \times 274 \times 150$	130
Conference	284K	1.1M	$210 \times 133 \times 50$	27
Dragon	180K	0.7M	$104 \times 147 \times 65$	16
Fairy Forest	174K	1.1M	$150 \times 38 \times 150$	24
Sponza	60K	490K	$116 \times 55 \times 52$	13
Ruins	53K	310K	$53 \times 69 \times 52$	10

Table 2: *Build statistics for test scenes of different sizes. "Refs" means the number of triangle references stored in the grid after construction. Times are in milliseconds and are measured on a GTX280.*

Model (Triangles)	Grid Dual Xeon	LBVH GTX280	H BVH GTX280	Grid GTX280
Fairy (174K)	68ms 3.9 fps	10.3ms 1.8 fps	124ms 11.6 fps	24ms 3.5 fps
Bunny/Dragon (252K)	- -	17ms 7.3 fps	66ms 7.6 fps	13ms 7.7 fps
Conference (284K)	89ms 4.0 fps	19ms 6.7 fps	105ms 22.9 fps	27ms 7.0 fps
Soda Hall (2.2M)	- 8.0 fps	66ms 3.0 fps	445ms 20.7 fps	130ms 6.3 fps

Table 3: *Build times and frame rate (excluding build time) for primary rays and simple shading for a* 1024×1024 *image. We compare performance of Wald's CPU implementation [2006] running on a dual 3.2 GHz Intel Xeon, Günther's packet algorithm [2007] with LBVH and Hybrid BVH (H BVH) as implemented by Lauterbach et al. [2009] to our implementation. See Table 2 for grid resolutions. The grid resolution of the Bunny/Dragon varies with the scene bounding box.*

4.1 Construction

From the results in Table 2 one sees that the performance of the construction algorithm scales with the number of references in the grid. The reported times are for the runtime of the construction algorithm and include all overheads for memory allocation and deallocation, the computation of the grid resolution, and a texture bind to a 3D texture, but do not include the time for the initial upload of the scene to the GPU. When rendering dynamic scenes, we use a separate CUDA stream for uploading geometry for the next frame while rendering the current one. We were able to completely hide the data transfer by the computation for the previous frame for all tested animations. Note that we also update the bounding box of the scene together with the data upload.

The time to build the full resolution grid for the Thai Statue ($325 \times 547 \times 280$) does not include 198 milliseconds for copying the grid cells to the CPU and then back to a GPU-texture. We were not able to perform the texture bind directly, because this involves duplication of the grid cells (nearly 400 MB), for which the GPU-memory was not sufficient. Note that the copy and the texture bind are only necessary if the rendering must be done on the device and the grid cells must be stored in a three dimensional texture. We include the build time for the full resolution and the sparser grid in Table 2 for comparison. This resolution allows us to achieve reasonable rendering performance - between 3 and 5 frames per second. Ize et al. [2006] report build times of 136 and 21 milliseconds for the Thai Statue ($192 \times 324 \times 168$) and the Conference on 8 Dual Core Opterons running at 2.4 GHz with bandwidth of 6.4 GB/s each.

4.2 Rendering

Our ray tracing implementation does not make explicit use of packets, frusta, or mailboxing like Wald's [Wald et al. 2006]. We used the traversal algorithm proposed by Amanatides and Woo [1987] without any significant modifications, and the ray-triangle intersection algorithm by Möller and Trumbore [1997]. We store the grid cells in a 3D texture in order to make better use of the texture cache during traversal. We represent a triangle as three indices to a global vertex array. We also store the triangles and the vertex array in textures since this improved the rendering time. In our tests we measure performance for primary rays and simple shading (Figure 2 and 3 and Table 3).

As illustrated in Table 3, the performance we achieve for primary rays does not compare well to the CPU algorithm by Wald et al. [2006] that uses SIMD packets and frusta. Their ray tracer can exploit the high coherency of primary rays, which makes it superior to our naïve GPU implementation. Since the grids constructed by our algorithm have at least the same quality[1] as those used by Wald et al. [2006], the presented construction algorithm can be used to speed up their ray tracer implementation. For a hybrid GPU-CPU implementation the texture bind should be replaced by two memory copies to the host - one for the gird cells and one for the array of triangle references. Together, these were never more than twice slower than the texture bind itself.

[1]The grid quality can be better, because of the more precise test for triangle-cell overlap that we perform.

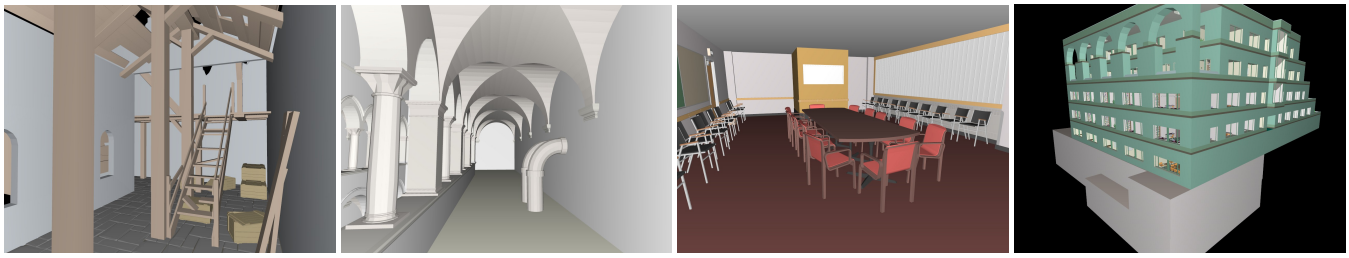

Figure 3: *Some of our test scenes, from left to right - Ruins, Sponza, Conference, and Soda Hall. We can render them at 21, 13, 7 and 6 fps with simple shading in a* 1024×1024 *window and can construct grids in 10, 13, 27 and 130 milliseconds on a GTX280.*

Both our building and rendering performance are comparable to the results of Lauterbach et al. in [Lauterbach et al. 2009] for their LBVH. The LBVH has an advantage in terms construction time, but the rendering performance suggests that it does not offer better acceleration for ray tracing than grids. The quality disadvantage is bigger for the relatively large Soda Hall model.

The Hybrid BVH [Lauterbach et al. 2009] offers fast construction times and between three and four times better rendering performance than our implementation of grids. Nevertheless the better construction time allows us to rebuild the structure and trace a significant amount of rays before the overall performance becomes worse.

Please note that the disadvantage in terms of rendering times that our implementation has is partly due to the fact that the alternative approaches make explicit use of ray-packets. On the other hand, our approach is less sensitive to ray coherency.

Despite their disadvantages grids can provide an alternative to hierarchical acceleration structures if the primitive distribution is to some extent even, or in real-time applications in which the number of rays that have to be traced is not very large. While high-quality SAH-based acceleration structures enable faster ray tracing, their construction time is a performance bottleneck in dynamic applications. The fast build times of grids are almost negligible and shift the computational demand entirely toward tracing rays, a task which is easier to parallelize.

5 Conclusion and Future Work

We presented a robust and scalable parallel algorithm for constructing grids over a scene of geometric primitives. Because we reduce the problem to the sorting of primitive-cell pairs, the performance does not depend on the triangle distribution in the scene. When used for ray tracing dynamic scenes, the fast construction times allow us to shift the computational effort almost entirely toward the rendering process. We also showed a method for choosing the resolution of the grid in a way that minimizes the expected cost for tracing a ray. Unfortunately this could not solve the problems that grids have with triangle distributions.

An interesting continuation of this work would be to investigate the use of the construction algorithm for different primitive types. We concentrated on the use of grids for ray tracing, but there are other applications that can benefit from the construction algorithm. We want to further research acceleration structures for ray tracing that are fast to build, but maintain high quality even for scenes with non-uniform triangle distributions. For example, it might be efficient to use the data produced after the sorting stage for constructing an octree. To this end only the last step of the algorithm should be replaced by a kernel that builds the tree on top of the sorted cells. It should also be possible to reduce the construction of regular acceleration structures such as hierarchical or multilevel grids to a sorting problem.

Acknowledgements

The authors would like to thank Felix Klein for modelling the Ruins model. The models of the Dragon and the Thai Statue are from *The Stanford 3D Scanning Repository*, the Bunny/Dragon and the Fairy Forest Animation are from *The Utah 3D Animation Repository*. We would like to thank the anonymous reviewers and Mike Phillips for the suggestions which helped to improve the quality of this paper.

References

AKENINE-MÖLLER, T. 2001. Fast 3d triangle-box overlap testing. *Journal of Graphics Tools 6*, 29–33.

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*. Elsevier Science Publishers, Amsterdam, North-Holland, 3–10.

DEVILLERS, O. 1988. *Methodes doptimisation du tracé de rayons.* PhD thesis, Université de Paris-Sud.

EISEMANN, E., AND DÉCORET, X. 2006. Fast scene voxelization and applications. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 71–78.

ES, A., AND İŞLER, V. 2007. Accelerated regular grid traversals using extended anisotropic chessboard distance fields on a parallel stream processor. *J. Parallel Distrib. Comput. 67*, 11, 1201–1217.

GREEN, S., 2008. Particles demo. NVIDIA CUDA SDK v2.2 http://www.nvidia.com/object/cuda_sdks.html.

GÜNTHER, J., POPOV, S., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Realtime ray tracing on GPU with BVH-based packet traversal. *Symposium on Interactive Ray Tracing*, 113–118.

IVSON, P., DUARTE, L., AND CELES, W. 2009. Gpu-accelerated uniform grid construction for ray tracing dynamic scenes. Master's Thesis Results 14/09, PUC-Rio, June.

IZE, T., WALD, I., ROBERTSON, C., AND PARKER, S. 2006. An evaluation of parallel grid construction for ray tracing dynamic scenes. *Symposium on Interactive Ray Tracing*, 47–55.

IZE, T., SHIRLEY, P., AND PARKER, S. 2007. Grid creation strategies for efficient ray tracing. *Symposium on Interactive Ray Tracing* (Sept.), 27–32.

JEVANS, D., AND WYVILL, B. 1989. Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89*, Canadian Information Processing Society, Toronto, Ontario, 164–72.

LAGAE, A., AND DUTRÉ, P. 2008. Compact, fast and robust grids for ray tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering) 27*, 4 (June), 1235–1244.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH construction on GPUs. In *Proceedings of Eurographics*.

MACDONALD, J. D., AND BOOTH, K. S. 1990. Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6, 153–65.

MÖLLER, T., AND TRUMBORE, B. 1997. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools 2*, 1, 21–28.

NICKOLLS, J., BUCK, I., GARLAND, M., AND SKARDON, K. 2008. Scalable parallel programming with CUDA. In *Queue 6*. ACM Press, 2, 40–53.

PATIDAR, S., AND NARAYANAN, P. 2008. Ray casting deformable models on the GPU. *Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, 481–488.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (Sept.).

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July), 703–712. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 97–106.

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics 25*, 3, 485–493.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, ACM, New York, NY, USA, 1–11.