EUROGRAPHICS 2010 / M. Chen and O. Deussen
(Guest Editors)

Volume 30 (2011), Number 2

Two-Level Grids for Ray Tracing on GPUs

Javor Kalojanov[1] and Markus Billeter[2] and Philipp Slusallek[1,3]

[1]Saarland University, [3]DFKI Saarbrücken
[2]Chalmers University of Technology

Abstract

We investigate the use of two-level nested grids as acceleration structure for ray tracing of dynamic scenes. We propose a massively parallel, sort-based construction algorithm and show that the two-level grid is one of the structures that is fastest to construct on modern graphics processors. The structure handles non-uniform primitive distributions more robustly than the uniform grid and its traversal performance is comparable to those of other high quality acceleration structures used for dynamic scenes. We propose a cost model to determine the grid resolution and improve SIMD utilization during ray-triangle intersection by employing a hybrid packetization strategy. The build times and ray traversal acceleration provide overall rendering performance superior to previous approaches for real time rendering of animated scenes on GPUs.

1. Introduction

State of the art ray tracing implementations rely on high quality acceleration structures to speed up the traversal of rays. When rendering dynamic scenes, these structures have to be updated or rebuilt every frame to account for changes in the scene geometry. Hence the rendering performance depends on the tradeoff between the ray traversal acceleration and the time required for updating the structure.

While this problem has been well studied for CPUs as summarized by Wald et al. [WMG*07], only a few recent approaches exist for ray tracing dynamic scenes on GPUs. Zhou et al. [ZHWG08] and Lauterbach et al. [LGS*09] propose fast construction algorithms for kd-trees and bounding volume hierarchies (BVHs) based on the surface area heuristic [MB90]. These methods allow for per-frame rebuild of the structure and thus support arbitrary geometric changes, but the time required to construct the tree is still a bottleneck in terms of rendering performance. Further related work includes the BVH construction algorithm by Wald [Wal10]. Lauterbach et al. [LGS*09] as well as Pantaleoni and Luebke [PL10] propose very fast construction algorithms for linear BVHs (LBVHs) based on placing the primitives in the scene on a Morton Curve and sorting them. Kalojanov and Slusallek [KS09] propose a similar idea for sort-based construction of uniform grids. Alas, uniform grids do not provide good acceleration of the ray traversal, mainly because they suffer from the "teapot in a stadium problem" – the grids fail to eliminate intersection candidates in parts of the scene with high density of geometric primitives.

We describe a modification of the uniform grid that handles non-uniform primitive distributions more robustly. The two-level grid is a fixed depth hierarchy with a relatively sparse top-level grid. Each of the cells in the grid is itself a uniform grid with arbitrary resolution. The structure is a member of a class of hierarchical grids introduced by Jevans and Wyvill [JW89], and to our knowledge its application to ray tracing on GPUs has not been studied before.

We believe that the concepts in this paper are not restricted to CUDA [NBGS08], but we implement the algorithms in this programming language. We therefore use several CUDA-specific notions. A *kernel* is a piece of code executed in parallel by multiple threads. A *thread block* is a group of threads that runs on the same processing unit and shares a fast, on-chip memory called *shared memory*. *Atomic operations* (e.g. addition) on the same memory location are performed sequentially.

2. Data Structure

A data representation of the acceleration structure that enables fast query for primitives located in a given cell is essential for ray tracing. During traversal, we treat the two-level grid as a collection of uniform grids. This is why the data layout of the two-level grid (see Figure 1) is very similar

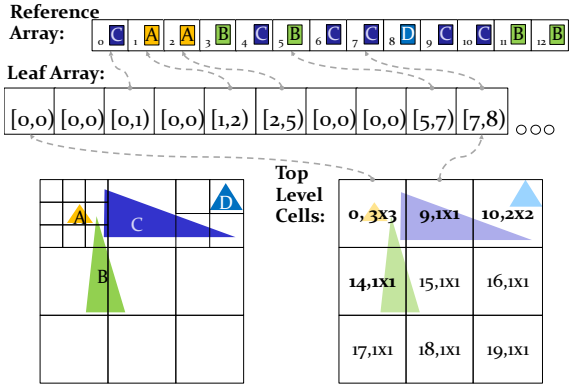

Figure 1: *The two-level grid is represented by the top-level cells (bottom right), the leaf array (second line) and the primitive reference array (top).*

to how uniform grids are typically stored [LD08, WIK*06, KS09].

Each top-level cell provides the data needed to access its sub-cells. Those are the resolution of the grid in this cell as well as the location of its leaf cells. We store all leaf cells in a single *leaf array*.

Each leaf cell has to provide the primitives that it overlaps spatially. We store references to the primitives in a single *reference array*. This array has two important properties. First, each primitive is referenced as many times as the number of cells it overlaps. Second, references to primitives intersecting the same leaf cell are stored next to each other in the array. Hence for each leaf cell in the grid there is an interval in the reference array, and the primitives referenced in this interval are the ones that overlap the cell spatially.

In our implementation we use two 32-bit values for each cell and each leaf. A leaf stores two integral values that indicate where to find the references to the primitives in the leaf (see Figure 1). For each top cell we store the location of the first leaf cell (32 bits) together with the cell resolution compressed in 24 bits. We leave out 8 bits for flags that denote status like "is empty" or "has children". Note that using only 24 bits limits the resolution of each top level cell to 256^3, but this limitation is implementation specific.

3. Sort-Based Construction

We build the two-level grid in a top-down manner. We construct a uniform grid, which we use to initialize the top-level cells. We then construct the reference array for the leaves in parallel by writing pairs of keys and primitive references in a random order and sorting them. Finally, we read out the leaf cells from the sorted array of pairs.

Algorithm 1 Data-Parallel Two-Level Grid Construction. Kernel calls are suffixed by $< >$. Scan and sort may consist of several kernel calls.

$b \leftarrow$ COMPUTE BOUNDS()
$r \leftarrow$ COMPUTE TOP LEVEL RESOLUTION()
$t \leftarrow$ UPLOAD TRIANGLES()
$data \leftarrow$ BUILD UNIFORM GRID(t,b,r)
$A_{tlp} \leftarrow$ GET SORTED TOP-LEVEL PAIRS$(data)$
$n \leftarrow$ GET NUMBER OF TOP-LEVEL REFERENCES$(data)$
$tlc \leftarrow$ GET TOP-LEVEL CELL RANGES$(data)$
▷ COMPUTE LEAF CELL LOCATIONS
$\mathrm{G} \leftarrow (r_y, r_z), \mathrm{B} \leftarrow r_x$
$A_{cc} \leftarrow$ ARRAY OF $r_x * r_y * r_z + 1$ ZEROES
$A_{cc} \leftarrow$ COUNT LEAF CELLS <G,B>(b,r,tlc)
$A_{cc} \leftarrow$ EXCLUSIVE SCAN$(A_{cc}, r_x * r_y * r_z + 1)$
▷ SET LEAF CELL LOCATIONS AND CELL RESOLUTION
$tlc \leftarrow$ INITIALIZE <G,B> (A_{cc}, tlc)
▷ COMPUTE REFERENCE ARRAY SIZE
$\mathrm{G} \leftarrow 128, \mathrm{B} \leftarrow 256$
$A_{rc} \leftarrow$ ARRAY OF $\mathrm{G}+1$ ZEROES
$A_{rc} \leftarrow$ COUNT LEAF REFS<G,B>(t,b,r,n,A_{tlp},tlc)
$A_{rc} \leftarrow$ EXCLUSIVE SCAN$(A_{rc}, \mathrm{G}+1)$
$m \leftarrow A_{rc}[\mathrm{G}]$
$A_p \leftarrow$ ALLOCATE LEAF PAIRS ARRAY(m)
▷ FILL REFERENCE ARRAY
$A_p \leftarrow$ WRITE PAIRS<G,B>$(t,b,r,n,A_{tlp},tlc,A_{rc})$
$A_p \leftarrow$ SORT(A_p)
$leafs \leftarrow$ EXTRACT CELL RANGES< >(A_p, m)

Kalojanov and Slusallek [KS09] reduce the construction of uniform grids to sorting(Figure 2). There, the reference array for the uniform grid is constructed by emitting and sorting pairs of cell index and primitive reference. This algorithm allows for optimal work distribution regardless of the primitive distribution in the scene, but is restricted to constructing a single uniform grid in parallel.

For the leaf cells of the two-level grid we need to construct multiple uniform grids of various resolution, each containing arbitrary number of primitives. Building each of the top-level cells independently would make the entire construction too expensive. The main contribution of this paper is an efficient algorithm for constructing the entire leaf level of the two-level grid with a single sort. Our algorithm is a generalization of the sort-based grid construction proposed by Kalojanov and Slusallek [KS09] based on the following observation: In order to use sorting to construct the entire array of primitive references for the cells at each level of the structure, we have to select a set of keys that form a one-one correspondence with the cells. We take advantage of the fact that we store the leaves in a single array, and choose to use the memory location of the leaf as key.

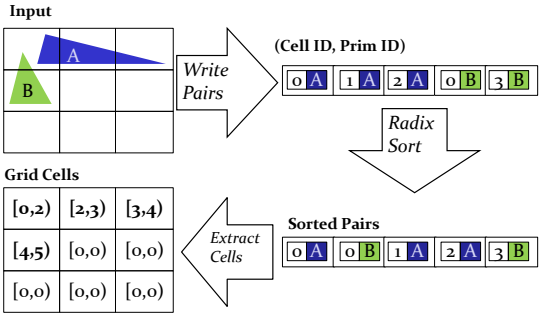

Figure 2: *Sort-based construction of uniform grids. We use this algorithm to compute the top-level of the two-level grid.*

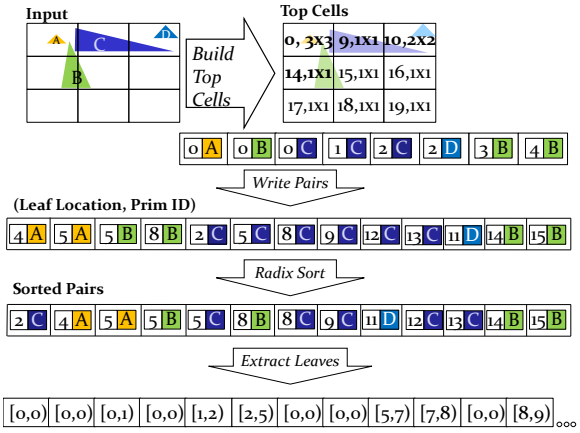

Figure 3: *Two-level grid construction. When we construct the top-level, we use cell indices as keys. The second set of pairs uses the memory location of the leaf cell as key. After being sorted, the leaf pairs yield the reference array (not in the figure) and the leaf cells.*

3.1. Building the Top-Level Cells

To construct the top-level cells, we build a uniform grid using the sorting-based approach by Kalojanov and Slusallek [KS09]. The algorithm amounts to determining all triangle-cell overlaps and storing the information as an array of pairs. As illustrated in Figure 2, this intermediate data is sorted using the cell indices as keys.

A notable modification of the original approach is that we keep the sorted array of pairs (see line 5 of Algorithm 1) and use it as input for the next stages of the construction instead of the original set of input primitives.

The resolution of each top-level cell is computed in constant time from its dimensions and the number of primitives it contains as detailed in Section 5. The resolution also gives the number of leaf cells each top-level cell contains.

We perform a prefix sum over the amount of leaf cells for each top-level cell to compute the location of the first leaf for every top-level cell (lines 9 to 11). After the scan is performed in line 12, the n-th value of the resulting array is the position in the leaf array where the first leaf of the n-th top-level cell is located.

3.2. Writing Unsorted Pairs

Having initialized the data for the top-level cells allows us to start writing the array of pairs that will later be sorted and converted to the final array with primitive references. Because it is not possible to dynamically allocate memory on current GPUs we have to compute the size of the array and allocate it in advance (Lines 16 to 19).

In contrast to the construction of the top-level grid, we cannot use the leaf index relative to the top cell as a key to identify primitive-leaf overlaps, as these leaf indices are not unique. For example, in Figure 1 there are nine leaves with index 0 and two with index 1. We want a single sort to yield the final reference array. To this end we instead use *the location of the cell* in the leaf array with the primitive index. This location is easy to compute from the leaf index and the location of the first leaf for the corresponding top-level cell, which we already have.

We use a fast but conservative triangle-cell intersection test when counting the output pairs and a more precise but slower one when generating them. First we only consider the bounding box of the triangle, and later we also make sure that the plane in which the triangle lies intersects the cell. The invalid intersections are still written with a dummy cell index that is discarded in the sorting stage.

3.3. Sorting the Pairs

The order of the primitive references in the sorted array is the same as in the final reference array. We use an improved version of the radix sort by Billeter et al. [BOA09], which uses a combination of two- and four-way splits. The size of the largest key we need to sort is known, which tells us the number of bits that need to be considered. We sort only on the required bits, up to two bits at a time.

3.4. Extracting the Leaf Cells

In the final stage of the construction algorithm, we compute the leaf cells directly from the sorted array of pairs. We check in parallel for range borders, i.e. neighboring pairs with different keys (e.g. the second and third sorted pairs in Figure 3). To be able to access the primitives inside each cell in constant time during rendering, we store the start and

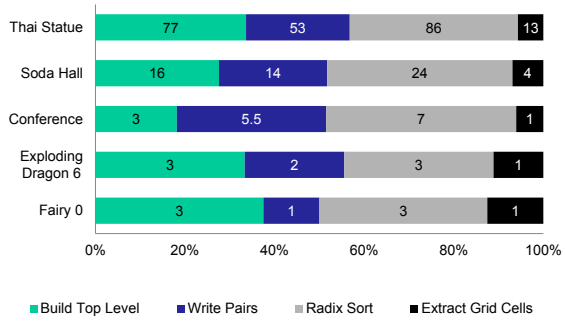

Figure 4: *Times for the different stages of the build algorithm in milliseconds. The complete runtime is the sum of the measured values. Exploding Dragon 6 and Fairy 0 means that the seventh and first frame from the animations were measured. Times are measured on a GTX 285.*

end of each range inside the corresponding leaf cell. This is straight-forward to do, since we can read the location of the cell from the key we used for sorting the pairs. After we output the leaf array, we extract the reference array out of the pairs array to compact the data and free space.

4. Analysis

Apart from being fast, the most important feature of the construction algorithm is that its work complexity is linear. This is one of the reasons to prefer radix sort, another one is the availability of a fast GPU implementation by Billeter et al. [BOA09]. Note that the data we have is well suited for bucket sort algorithms like radix sort because the range of different key values is small compared to the input size. The complexity would not be linear if we were not able to determine the cell resolution in constant time, knowing the number of overlapped primitives and the extent of the bounding box of each top-level cell.

4.1. Runtime

The runtime is more or less equally distributed between the different stages of the construction (see Figure 4). The most time-consuming stages of the algorithm – writing the pairs and sorting them – are memory bandwidth limited. We do not require any synchronization inside the kernels. Atomic synchronization is not necessary, but we use atomic increment on shared memory for convenience when writing pairs in the unsorted array.

The algorithm is able to exploit the high parallelism of modern GPUs because the work is evenly distributed among as many threads as the hardware is able to manage. Of course this holds only if there are enough input primitives, which is almost always the case. This is possible because in each step threads are mapped to tasks independently. In addition to the high parallelism, the runtime is independent of the primitive distribution in the scene exactly like the sort-based uniform grid construction [KS09].

4.2. Memory Footprint

In terms of memory footprint, the two-level grids consume less memory mainly because of the reduced amount of cells compared to a single-level uniform grid. The main memory bottleneck here is the space required to store the references. Because the two-level grids are more memory friendly than uniform grids, we were able to construct the acceleration structure for models such as the Thai Statue, which consists of 10 million triangles, on a graphics card with 1GB of memory.

5. Grid Resolution

Our choice of grid resolution is based on the heuristic for uniform grids introduced by Cleary et al. [CWVB83] and also used by Wald et al. [WIK*06]:

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

where $\vec{d}$ is the extent of the box in all dimensions 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*. Thus the number of cells is a multiple of the input primitives. Ize et al. [ISP07] investigate optimal grid densities for uniform and two-level grids. The best densities for CPU ray tracing suggested by the authors proved sub-optimal for our GPU ray tracer. Ize et al. base their theoretical analysis on several assumptions including fast atomic operations and compute limited traversal, which did not hold for the hardware we used. Additionally Popov et al. [PGDS09] demonstrate that for other acceleration structures like BVHs treating the geometric primitives as points can severely limit the quality.

To find optimal grid densities and work around the issues mentioned above we employ the Surface Area Metric, introduced by MacDonald and Booth [MB90], to compute the expected cost (C_e) for traversing a random ray through the acceleration structure:

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

where C_t and C_i are constant costs for traversing a node of the structure and intersecting a geometric primitive, $Pr(\cdot)$ is the probability of a random line intersecting a node or leaf, and $Prim(\cdot)$ is the number of geometric primitives in a leaf node. In our case the nodes of the structure are the grid cells and computing the cost for a particular grid density amounts to constructing the top level of the grid and counting the

	Soda	Venice	Conf.	Dragon	Fairy	Sponza
$C_t = 1\frac{1}{2}$	2.4	2.1	1.4	1.9	1.4	2.1
$C_t = 2$	1.8	1.8	1.3	1.4	0.9	1.6
$C_t = 3$	1.2	1.2	1.3	1.0	0.9	1.3

Table 1: *Optimal densities λ for the leaf level of the grid computed according to the cost model based on the Surface Area Metric. Cost for intersection was 1 for all listed traversal costs (C_t). The first level ot the grid had $\lambda = \frac{1}{16}$*

primitives referenced in the leaves, which is easily done after slight modification of the initial stage of our construction algorithm – the kernel that counts the fragments in the leaf level of the grid.

We set the top level density to $\lambda = \frac{1}{16}$ to make better use of our hybrid intersector, which is explained in detail later in Section 7 and benefits from cells that contain at least 16 primitives. Then we tested our traverser and intersector performance separately and together on an artificially created grid with various number of primitives per cell. We took the ratios of empty to full leaf cells from our test scenes. Our ray tracing implementation performed best on relatively sparse grids when we did $1\frac{1}{2}$ intersection tests per traversal step. Thus we tried $C_t = 1\frac{1}{2}, 2, 3$ and $C_i = 1$ looking for close to optimal rendering performance and sparse grid resolutions that would allow faster construction and the best time to image for dynamic scenes. Given the costs for traversal and intersections we tested all grid densities starting with $\frac{1}{16}$, and adding $\frac{1}{16}$ until we reached $\lambda = 8$. The densities with optimal costs for some of our test scenes are listed in Table 1.

While the optimal density for the two-level grid is scene dependent, the rendering performance for all scenes did not vary noticeably for values of λ between $\frac{1}{2}$ and 2. In the remainder of the paper the density of each top-level cell is set to $\lambda = 1.2$. Note that we use a global value for λ across all top-level cells. Choosing the different density or even different resolution for different top level cells can provide higher quality grids, but would make the construction too slow as already observed by Kalojanov and Slusallek [KS09]. Also note that we used the Surface Area Cost Model to determine a single grid density that works well with our traversal implementation for all test scenes.

6. Ray Traversal

Similar to the uniform grids, the traversal of two-level grids is very inexpensive. We modified the DDA traversal algorithm by Amanatides and Woo [AW87]. We step through the top level of the structure as we would do for a uniform grid. We treat each top cell as a uniform grid itself and use the same algorithm to advance through the leaf cells. One can reuse some of the decision variables for the top-level traversal to spare computations when initializing the decision variables for the leaf level.

The traversal algorithm has properties similar to the uniform grid traversal. It is SIMD (or SIMT) friendly since there is no code divergence, except when the rays disagree on which level of structure they need to traverse. The algorithm favors coherent rays. In the case of uniform grids, one can easily show that for rays starting in the same grid cell will traverse every cell of the grid at exactly the same step of the DDA traversal algorithm:

Statement:Let R_o be a set of rays, all of which have origin (not necessarily the same) in a given grid cell o. If a cell c is intersected by any ray $r \in R_o$ and is the n-th cell along r counted from the origin, then c will be n-th cell along any ray $r_2 \in R_o$ that intersects c.

Proof: Let o be the 0-th cell along the rays in the set. For each cell c holds that it is at distance (x, y, z) from o, where x, y and z are integers and each denotes the distance (in number of cells) along the coordinate axis with the same name. During traversal, the rays advance with exactly one neighbor cell at a time and the two cells share a side. This means that c will be traversed at $x + y + z = n$-th step of the traversal algorithm for any ray with origin in o if at all.

This means that a cell along the traversal path, if loaded, will be loaded exactly once from memory. This property transfers to the two-level grid structure when traversing the top level and when rays agree on the first leaf cell when they traverse the leaves.

7. Triangle Intersection

During the ray-triangle intersection stage of the ray traversal the coherence in the computations executed by the different threads is lost as soon as some of the rays traverse a full cell, while others pass through an empty one. This is the case because we parallelize by assigning different rays to each thread, which ties SIMD efficiency to ray coherence. The two-level grids we use for rendering are sparse compared to other acceleration structures and contain more primitives per leaf. We exploit this property and distribute work for a single ray to many threads to speed up traversal by improving SIMD utilization.

We employ a hybrid packetization strategy during ray-triangle intersection. In each *warp* (SIMD unit consisting of 32 threads), we check if all rays have intersection candidates and if this is the case we proceed with the standard ray-triangle intersection. If this is not the case, we check if there are rays with *high workload* - at least 16 intersection candidates. If these are more than 20 we proceed with the standard intersection routine. This part of the algorithm maps better to Fermi GPUs, because they support more Boolean operations on warp level in hardware.

Scene	Tris	Refs	Top	Leaf	MB	Time
Thai	10M	27.2M	226K	11.8M	194	257
Soda	2.2M	6.6M	48K	1.4M	52	67
Venice	1.2M	3.7M	27K	1.5M	26	32
Conference	284K	2.3M	6K	358K	10	17
Dragon	252K	905K	5K	307K	5	10
Fairy	174K	514K	3K	206K	4	8

Table 2: *Build statistics with density of each top-level cell* $\lambda = 1.2$. *Refs is the number of primitive references in the final structure. Top and Leaf are amount of top-level and leaf cells. MB is the amount of memory (in MB) required to store all cells and the references. Times (in ms) are measured on a GTX 285. We used the second frame of the Exploding Dragon animation, and the sixth frame of the Fairy Forest animation.*

Model	LBVH	H BVH	Grid	2lvl Grid
Fairy (174K)	10ms 1.8 fps	124ms 11.6 fps	24ms 3.5 fps	8ms 9.2 fps
Conference (284K)	19ms 6.7 fps	105ms 22.9 fps	27ms 7.0 fps	17ms 12.0 fps
Expl. Dragon (252K)	17ms 7.3 fps	66ms 7.6 fps	13ms 7.7 fps	10ms 10.3 fps
Soda Hall (2.2M)	66ms 3.0 fps	445ms 20.7 fps	130ms 6.3 fps	67ms 12.6 fps

Table 3: *Build times and frame rates (excluding build time) for primary rays and dot-normal shading for a* 1024×1024 *image. We compare our non-optimized implementation to the results of Lauterbach et al. [LGS*09] and Kalojanov and Slusallek [KS09]. The Hybrid BVH (H BVH) [LGS*09] is a BVH with close to optimal expected cost for traversal. All times are measured on a GTX 280.*

If we have determined that standard packetization would be inefficient and there are rays with high workload, we insert those into a buffer in shared memory. This is done with an atomic increment on a shared counter. We only need to store the index of the ray and the first intersection candidate, because the indices of the remaining primitives are consecutive. We iterate on the entries of this buffer taking two at a time and perform 32 intersection tests for a pair of rays in parallel. We combine the results by performing reduction on the arrays with the intersection distances, which are also shared between threads. Finally, the thread responsible for tracing the ray records the closest intersection (if there is one) in thread local memory. When all rays with high workload have been processed we perform the remaining intersection tests one by one for all rays as usual.

We chose to define 16 intersection candidates as high workload, since the actual SIMD width of 32 was too big and in our structure there were not many cells with enough primitives. Having to process more than 2 rays at a time on the other hand turned out to introduce too much overhead. To derive the last parameter – the number (20) of active rays above which horizontal parallelization does not pay off – we benchmarked the speed of our implementation with all possible numbers of active rays.

8. Results

We implemented the two-level grid builder together with a GPU ray tracer in CUDA. We tested performance on a machine with an NVIDIA GeForce GTX 285, GTX 280, or a GTX 470 and an Intel Core 2 Quad processor running at 2.66 GHz. The CPU performance is almost irrelevant for the results since we only use it to schedule work and transfer data to the graphics card.

8.1. Construction

We give the performance of our implementation in Table 2. We include all overheads except the initial upload of the scene primitives to the GPU. The runtime grows with the number of pairs (i.e. number of primitive references), which is scene dependent. The measured frames from the Exploding Dragon and Fairy Forest animations are the worst case examples for the build performance during the animations.

Note that the number of primitive references required for rendering is smaller than the total number of references. This is the case because we use the sorting stage to discard cell-triangle pairs that were generated using a more conservative triangle insertion test.

Pantaleoni and Luebke [PL10] improve on the LBVH construction algorithm proposed by Lauterbach et al. [LGS*09] and achieve superior build performance and scalability for their HLBVHs. To our knowledge the HLBVH is the structure that is fastest to construct on modern GPUs. The test results in Table 2 suggest that our algorithm also scales very well with scene size. The build times for the two-level grids depend on the chosen density and the primitive distribution – we need around 10 ms per 1 million references. For densities around 1, which proved to be optimal for our renderer, construction times were close to and sometimes faster than those of Pantaleoni and Luebke.

8.2. Rendering

We tested the build and render times for the two-level grid in Table 3. We clearly see that the new structure handles a teapot in the stadium scene like the Fairy Forest much better than uniform grids. Here we did not include persistent threads - an optimization proposed by Aila and Laine [AL09], that deals with a drawback of the GT200-class NVIDIA GPUs related to work distribution. We did this with the purpose to allow a fair comparison, at least to

	Ogre (50K)	Fairy (174K)	Conf. (284K)	Venice (2.2M)
GTX 285				
Dot-Normal	31 fps	15 fps	22 fps	18 fps
Direct Illumination	7.8 fps	2.7 fps	3.0 fps	3.3 fps
Path Tracing	3.4 fps	2.1 fps	1.6 fps	1.4 fps
GTX 470	Hybrid			
Dot-Normal	44 fps	22 fps	27 fps	26 fps
Direct Illumination	10 fps	4.6 fps	3.7 fps	4.7 fps
Path Tracing	4.7 fps	2.8 fps	2.4 fps	2.3 fps
GTX 470	Simple			
Dot-Normal	46 fps	21 fps	26 fps	24 fps
Direct Illumination	10 fps	3.7 fps	3.3 fps	4.5 fps
Path Tracing	4.7 fps	2.6 fps	2.4 fps	2.1 fps

Table 4: *Frame rate (in frames per second) for rendering a* 1024×1024 *image on a GTX 285, GTX 470 with our hybrid intersector and without it. We measure the time required to generate an image with dot-normal shading (1 primary ray per pixel), diffuse shading and direct illumination from an area light source (1 primary and 4 shadow rays), and path tracing with 3 rays per pixel on average. This implementation includes persistent threads. The frame rates for the Ogre and Fairy scenes include rebuild of the grid for each frame.*

some extent, as neither Lauterbach et al. nor Kalojanov and Slusallek used this optimization. In Table 4, we give frame rates of the same implementation after we added persistent threads.

Since primary rays are very coherent and maybe the fastest to traverse, we also evaluated the traversal performance of other types of rays as well. In Table 4 we test performance on two dynamic and two static scenes. We compare the performance for dot-normal shading with an implementation of diffuse direct illumination from an area light source. For the latter, we trace 4 shadow rays starting at each hit point and ending in a uniformly sampled random point on the light source.

The last scenario we tested in Table 4 is path tracing diffuse surfaces. We terminate the paths based on Russian Roulette with probability $\frac{1}{2}$ and shoot a single shadow ray at the end of each path to compute the direct contribution from the area light source previously used for the direct illumination test. We do not shoot other shadow rays except the one at the end of the path and generate secondary ray directions with cosine distribution around the surface normal. Note that this benchmark produces very incoherent rays and the SIMD units are under-utilized because of ray termination. For each sample per pixel the average number of rays is 3, the longest path has length 15, and the warps stay active for 6 bounces on average for the test scenes.

On GT200 class GPUs, the hybrid ray-triangle intersection (Section 7) improved the overall performance for incoherent rays by up to 25%, but only if we did not use persistent threads [AL09]. Thus we did not use it when testing on the GTX285. Fermi GPUs (e.g. the GTX 470) have hardware support for population count and prefix sum on warp level which allows to implement the intersector more efficiently. We used the hybrid intersector together with persistent threads on the newer Fermi GPUs, which improved the overall performance of our path tracer. The improvement increased when rendering smaller resolution windows, which reduces ray coherency, but the overall performance gain due to the hybrid intersector did not exceed 10%. Although the gain is not big, it is interesting to see that such hybrid parallelization scheme can work in practice. This indicates that control flow divergence can limit the performance on wide SIMD machines and adding some computational overhead to reduce it can pay off on future hardware.

Overall, the traversal performance provided by two-level grids is not as high as state-of-the-art approaches for *static scenes*. We implemented the test setup described by Aila and Laine [AL09]. We were able to trace 24 and 22 million primary rays per second for the Conference and the Fairy scene compared to 142 and 75 million as reported by Aila and Laine. We used the viewports in Figure 5. In terms of traversal performance, the two-level grids are comparable to other acceleration structures that are fast to construct like the LBVH or the Hybrid BVH, but our approach offers inferior performance to algorithms optimized for static geometry.

9. Discussion and Future Work

There are various possible continuations of this work. Since the main drawback of our approach is the ray traversal performance, it will be interesting to investigate whether or not different traversal algorithms can map better to the hardware architecture and provide faster rendering times.

The sort-based construction algorithm can be extended to build hierarchies with arbitrary depth. The tree has to be constructed top-down and breadth-first. At each level d one needs to identify a unique index of every node with depth $d+1$. The index can be paired with the indices of the overlapping primitives. After the pairs are sorted, the tree nodes of depth $d+1$ can be read out from the sorted array. Hierarchical grids with depth larger than two might offer better acceleration for ray tracing on future hardware. Extending our traversal implementation to three-level grids did not make sense, because of the increased amount of registers required to run the code.

A variant of the algorithm can be used to construct multiple uniform grids over different objects in parallel. This requires almost no modifications of the current implementation. The construction of the leaf level of the two-level grid in fact performs this task, where the grids are actually cells of the top level of the structure.

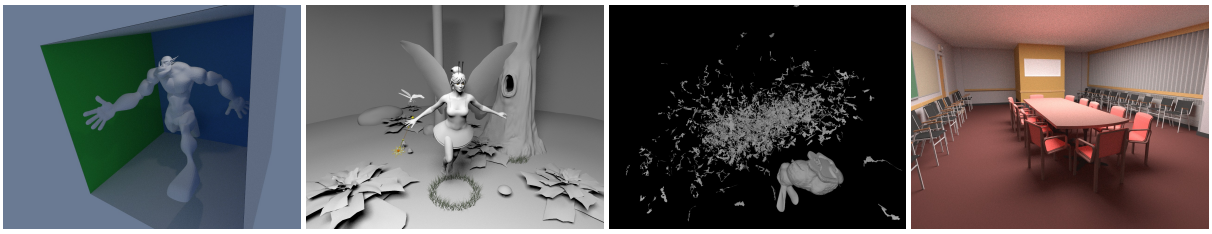

Figure 5: *Some of our test scenes, from left to right - Ogre, Fairy Forest, Exploding Dragon and Conference. The images were rendered using path tracing, diffuse direct illumination, dot-normal shading and path tracing respectively.*

10. Conclusion

The acceleration structure discussed in this paper is a simple extension of the uniform grid that copes better with non-uniform triangle distributions. Our goal was to both improve the quality of the acceleration structure and maintain the fast build times. For our tests scenes, the rendering performance increased overall, and especially when rendering parts of the scene with more complex geometry. The build times of the two-level grids are comparable and in general lower than those of uniform grids when using resolutions optimal for rendering in both cases. To our knowledge, there is no acceleration structure that is significantly faster to construct on modern GPUs. We believe that this and the reasonable rendering performance makes the two-level grid a choice worth considering for ray tracing dynamic scenes. We also hope that the general ideas for the parallel construction and the hybrid packetization can be adapted and used in different scenarios, e.g. with different spatial index structures.

Acknowledgements

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 Vincent Pegoraro for the suggestions which helped improving the quality of this paper.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *HPG '09: Proceedings of the 1st ACM conference on High Performance Graphics* (2009), ACM, pp. 145–149. 6, 7

[AW87] AMANATIDES J., WOO A.: A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*. Elsevier Science Publishers, 1987, pp. 3–10. 5

[BOA09] BILLETER M., OLSSON O., ASSARSSON U.: Efficient stream compaction on wide SIMD many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (2009), ACM, pp. 159–166. 3, 4

[CWVB83] CLEARY J. G., WYVILL B. M., VATTI R., BIRTWISTLE G. M.: Design and analysis of a parallel ray tracing computer. In *Graphics Interface '83* (1983), pp. 33–38. 4

[ISP07] IZE T., SHIRLEY P., PARKER S.: Grid creation strategies for efficient ray tracing. *Symposium on Interactive Ray Tracing* (Sept. 2007), 27–32. 4

[JW89] JEVANS D., WYVILL B.: Adaptive voxel subdivision for ray tracing. In *Proceedings of Graphics Interface '89* (June 1989), Canadian Information Processing Society, pp. 164–72. 1

[KS09] KALOJANOV J., SLUSALLEK P.: A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the 1st ACM conference on High Performance Graphics* (2009), ACM, pp. 23–28. 1, 2, 3, 4, 5, 6

[LD08] LAGAE A., DUTRÉ P.: Compact, fast and robust grids for ray tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering) 27*, 4 (June 2008), 1235–1244. 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. In *Proceedings of Eurographics* (2009). 1, 6

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6 (1990), 153–65. 1, 4

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKARDON K.: Scalable parallel programming with CUDA. In *Queue 6*. ACM Press, 2 2008, pp. 40–53. 1

[PGDS09] POPOV S., GEORGIEV I., DIMOV R., SLUSALLEK P.: Object partitioning considered harmful: space subdivision for bvhs. In *HPG '09: Proceedings of the 1st ACM conference on High Performance Graphics* (2009), ACM, pp. 15–22. 4

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing. In *High Performance Graphics* (2010). 1, 6

[Wal10] WALD I.: Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics* (2010). (to appear). 1

[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. 2, 4

[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). 1

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers* (2008), ACM, pp. 1–11. 1