Vis Comput
DOI 10.1007/s00371-011-0593-8

ORIGINAL ARTICLE

Grid-based SAH BVH construction on a GPU

Kirill Garanzha · Simon Premože · Alexander Bely · Vladimir Galaktionov

Abstract We present an efficient algorithm for building an adaptive bounding volume hierarchy (BVH) in linear time on commodity graphics hardware using CUDA. BVHs are widely used as an acceleration data structure to quickly ray trace animated polygonal scenes. We accelerate the construction process with auxiliary grids that help us build high quality BVHs with SAH in $O(k*n)$. We partition scene triangles and build a temporary grid structure only once. We also handle non-uniformly tessellated and long/thin triangles that we split into several triangle references with tight bounding box approximations. We make no assumptions on the type of geometry or animation motion. However, our algorithm takes advantage of coherent geometry layout and coherent frame-by-frame motion. We demonstrate the performance and quality of resulting BVHs that are built quickly with good spatial partitioning.

Keywords GPU · Ray tracing · Acceleration structure · Triangle subdivision · SAH · Surface area heuristic · BVH · Bounding volume hierarchy

K. Garanzha (✉) · V. Galaktionov
Keldysh Institute of Applied Mathematics, Moscow, Russia
e-mail: kirill@garanzha.com

V. Galaktionov
e-mail: vlgal@gin.keldysh.ru

K. Garanzha
NVIDIA, Moscow, Russia

S. Premože
Santa Monica, CA, USA
e-mail: simon.premoze@gmail.com

A. Bely
Capital Research and FUGU, Moscow, Russia
e-mail: a.bely@capital-research.ru

1 Introduction

Ray tracing has become ubiquitous for providing global illumination, soft shadows, glossy reflections, motion blur and depth of field. Efficient ray tracing makes use of spatial indexing data structures such as grids, KD-trees and bounding volume hierarchies (BVHs) to accelerate intersection between rays and scene geometry. Many other applications such as collision detection and visibility culling also benefit from efficient spatial partitioning. An efficient acceleration structure is fast to build, requires little extra memory and allows fast searching and traversal. While there is no consensus which acceleration method is the best, we focus on a Bounding Volume Hierarchy (BVH) as it provides a good balance between building time, traversal efficiency and ability to handle dynamic geometry. One advantage of the BVH is predictable memory consumption as each scene primitive is referenced only once in the tree.

As modern computer architectures become massively parallel, building and traversal of spatial acceleration data structures must also be done in parallel if they are to scale with the number of available processors. Until recently, most of the research work has been done on serial or only moderately parallel building algorithms and architectures. Our goal is to explore massively parallel real-time construction algorithms for fully dynamic geometry while making few or no assumptions about underlying geometry or motion. Recent groundbreaking work by Lauterbach et al. [10] made fast BVH building entirely on GPUs possible. Pantaleoni and Luebke [12] made many improvements to this algorithm by exploiting coarse-grain coherency in the input geometry. Their work inspired us to improve building of high quality BVHs for interactive applications and further reduce building times while maintaining bounded memory usage.

BVHs adapt very poorly to non-uniformly tessellated scenes. Many scenes fall into this category as architectural

Published online: 05 May 2011

scenes modeled by various Computer-Aided Design (CAD) packages contain many long and wide (or thin) triangles. The choice of partitioning method significantly influences the quality of the tree. Surface Area Heuristic (SAH) considers the density of geometry in 3D space and efficiently culls empty regions. Unfortunately, building a high quality tree is expensive. This becomes even more obvious for dynamic and animated scenes where BVH trees have to be rebuilt for every frame.

Our BVH building algorithm splits long and wide triangles into several triangle references with a tight AABB approximation. We compute a 3D density grid and its mip-map for triangle references. The grid helps computing the SAH cost of each node split. The cost of computing SAH split is bounded for all BVH nodes. We make several contributions. First, we utilize spatial splitting that produces high quality BVHs even for non-uniformly tessellated scenes without memory overflow problems. Second, we describe a fast Surface Area Heuristic (SAH)-based builder that exhibits linear asymptotic behavior. We explicitly bound the computational (and memory fetch) work to produce a single node split. The cost of building a single top level node and a bottom level (leaf) node is the same. Third, we reduce the amount of sorting needed. Fourth, the BVH building algorithm and described concepts are well suited for massively parallel architectures and are relatively simple to implement.

We implement our algorithm in CUDA [11] and benchmark it on NVIDIA GeForce 480 GTX (Fermi) GPUs. The implementation of the algorithm can handle fully dynamic geometry. We compare our algorithm with the results from some recent papers in Sect. 4.

2 Background

Spatial acceleration data structures have been used in ray tracing and collision detection for many years. With the advent of GPUs and multicore architectures, research community has renewed interest in building fast and efficient acceleration structures. We only mention some recent developments in the field. We direct reader to Havran's dissertation [6] for an excellent and comprehensive overview acceleration structures and in-depth discussion of problems.

Until recently, most of the research has focused on serial algorithms and memory optimizations and improvements [17]. With the paradigm shift in computer architecture from a single high performing processor towards multicore and highly parallel architectures, there have been many new innovative algorithms. Foley and Sugerman [3] looked into building and traversing KD-trees on modern GPUs. There have been numerous other improvements, most notably described in the works of Popov et al. [13], Shevtsov et al. [15], Zhou et al. [20]. An efficient implementation of KD-tree builder on multicore platforms was presented by Choi et al. [2]. Wald gives a great recipe for building fast BVHs with Surface Area Heuristic [18]. Some acceleration structures such as KD-trees produce high quality trees, but they are not applicable for dynamic geometry or animated scenes due to long building times. Wald et al. [19] have extended BVHs building to be more amenable for deforming and animated geometry. Ize et al. [7] give yet another extension of building BVH on moderately parallel systems. On the other hand, Kalojanov and Slusallek [8] tackle the problem of efficient acceleration using grids and hierarchical grids that can be build entirely on a GPU. Kalojanov and Slusallek [9] further improved grid traversal efficiency using a hierarchical grid. In our work, we use grid only to help with efficient BVH building.

We have also already mentioned the problem of building structures suitable for non-uniform tessellation of geometry. Stich et al. [16] present an easily directable BVH algorithm for such geometric configurations utilizing SAH for primitive splitting. Unfortunately, the algorithm is not directly amenable to efficient GPU implementation.

2.1 LBVH and HLBVH

Lauterbach et al. [10] described a BVH construction algorithm based on space-filling Morton curve and sorting primitives in the scene along the curve. Each point in space can be discretized to n bits. By interleaving the coordinates of a 3D point, we get a $3n$-bit index, called *Morton code,* that enumerates where this discretized point lies on the Morton curve of order n. If all geometric primitives are enumerated with Morton codes and then sorted, a *Linear Bounding Volume Hierarchy* (LBVH) [10] is constructed. LBVH building is extremely fast, however, it produces trees that are less than optimal for fast traversal and it does not exploit any existing coherency in the scene geometry.

Pantaleoni and Luebke [12] introduced *Hierarchical Linear Bounding Volume Hierachy* (HLBVH) that significantly reduces memory traffic and the amount of sorting work done. HLBVH takes advantage of any spatial coherency in the input mesh by doing hierarchical grid decomposition. *Surface Area Heuristic* (SAH) is used to construct top levels of the BVH and Morton curve-based partitioning for bottom levels. The resulting algorithm produces high quality trees that are still very efficient for traversals while having impressive building times.

2.2 Massively parallel computing model

Modern desktop GPU is a powerful device that is best suited for streaming data-parallel algorithms with a coherent execution and memory access pattern. The GPUs are composed of several independent cores [11]. A host (CPU) initiates parallel kernels on a device (GPU). A *kernel* executes a program across many (thousands) threads. GPU threads are organized into blocks (each block is executed on the single GPU core). Each block of threads is organized into several *warps* (bundles of 32 threads) which execute a single kernel instruction for the entire warp of threads. An algorithm that can be efficiently implemented on massively-parallel platforms like GPUs must: (a) decompose work into suitably sized chunks that will be mapped onto thread blocks, (b) have enough fine grained parallelism and (c) be conscious of memory access and memory utilization.

3 BVH construction algorithm

Overview We assume building a binary BVH of Axis Aligne Bounding Boxes (AABBs). Wald [18] and Lauterbach et al. [10] proposed CPU/GPU BVH builders that are based on binning technique and recursive triangle list partitioning at each tree level. In these algorithms, a transient grid structure is created in the process of each BVH node creation (binning). Triangle references are distributed in the grid and the SAH cost of the best partition split is evaluated using the grid.

We also accelerate the BVH build process with the help of auxiliary grids that help us build high quality SAH BVH in $O(k * n)$. In contrast to previous methods, we partition scene triangles and build the grid only once. We construct several mip levels of the density grids from the input set of triangles. Each grid cell encodes a range of primitives inside. The root node of the BVH is constructed from the coarse mip level (8^3): we evaluate 21 possible split planes (all planes between grid cells for each axis), select the best one and partition the input set of primitives. Then we find split positions for resulting two subsets of primitives and recursively continue. The number of primitives and approximate bounding boxes for each partition are computed quickly from the density grid. We can select the best split plane with a fewer accumulation/comparison operations than the binning approach by Wald [18] where all primitives are processed to evaluate the best split plane. Our algorithm takes advantage of the compression–sorting–decompression (CSD) technique introduced by Garanzha and Loop [4].

$Grid_0$ setup First, we build the highest resolution grid (labeled $Grid_0$) for the entire scene. Scene AABB extents determine the grid dimensions. We use $1024 \times 1024 \times 1024$ resolution for $Grid_0$ (each *cell ID* can be encoded with a 30-bit key). Storing 1024^3 cells is impractical since most grid cells are empty. We decompose $Grid_0$ into a two-level hierarchy of $TopGrid_0$ and a number of bottom grids created within each non-empty cell of $TopGrid_0$. In practice, we use $128 \times 128 \times 128$ resolution for $TopGrid_0$ and $8 \times 8 \times 8$ for each $BottomGrid_0$.

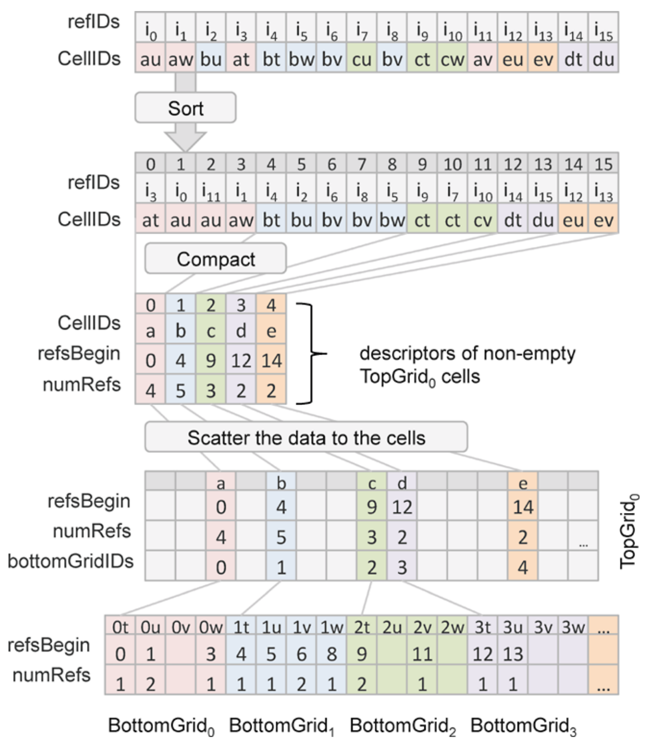
refIDs
CellIDs
i0 i1 i2 i3 i4 i5 i6 i7 i8 i9 i10 i11 i12 i13 i14 i15
au aw bu at bt bw bv cu bv ct cw av eu ev dt du
Sort
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
i3 i0 i11 i1 i4 i2 i6 i8 i5 i9 i7 i10 i14 i15 i12 i13
at au au aw bt bu bv bv bw ct ct cv dt du eu ev
Compact
refsBegin
numRefs
0 1 2 3 4
a b c d e
0 4 9 12 14
4 5 3 2 2
descriptors of non-empty TopGrid0 cells
Scatter the data to the cells
bottomGridIDs
a b c d e
0 4 9 12 14
4 5 3 2 2
0 1 2 3 4
...
TopGrid0
0t 0u 0v 0w 1t 1u 1v 1w 2t 2u 2v 2w 3t 3u 3v 3w ...
0 1 3 4 5 6 8 9 11 12 13
1 2 1 1 1 2 1 2 1 1 1 ...
BottomGrid0 BottomGrid1 BottomGrid2 BottomGrid3

Fig. 1 The process of triangles distribution in a $Grid_0$

Distribution of triangle references in $Grid_0$ (See Fig. 1.) Each triangle is represented by a triangle reference (triangle ID, AABB over a triangle or a triangle part). An array of reference IDs, *refIDs*, is initially filled with sequential IDs. Using $Grid_0$ resolution, we compute a 30-bit cell ID of each triangle centroid. The most significant 21 bits represent a cell ID within $TopGrid_0$ (Fig. 1(a)–(e) sub-keys); the least significant 9 bits represent a cell ID within $BottomGrid_0$ (Fig. 1(t)–(w) sub-keys).

We then sort the *refIDs* array using cell IDs as keys in radix sort. This step is accelerated by the CSD technique [4]. Adjacent elements of *CellIDs* array may be equal and can be compressed into representative chunks (cell ID, base, length). A shorter array of chunks is sorted faster than the original *CellIDs* array. Sorted array of chunks is then decompressed into a final sorted *CellIDs* array. We then further compact it (using the most significant 21-bits, sub-key, of CellID) into an array of non-empty $TopGrid_0$ cell descriptors. Entries with equal sub-keys will fall into the same descriptor. Each cell descriptor represents a segment of sorted triangles in the grid cell. The most significant 21-bit CellID of each descriptor is used as a cell address inside $TopGrid_0$ where we write triangle segment information.

Similarly, we then compact CellIDs array (using all 30 bits) into an array of non-empty $BottomGrid_0$ cell descriptors. These descriptors are used to construct *BottomGrid* by concatenating all $8 \times 8 \times 8$ refinements of non-empty cells of $TopGrid_0$.

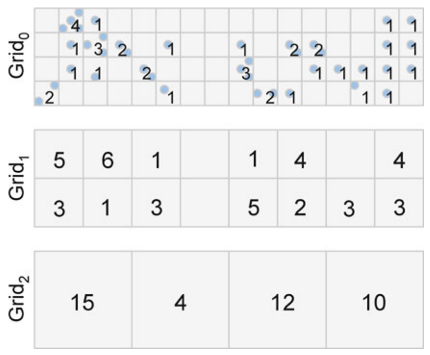

Fig. 2 Mipmap-style computing of coarser-level grids (the number of triangles per cell is highlighted). 2D-projection

More implementation details with pseudocode are given in the Appendix and the distribution process is illustrated in Fig. 1.

Coarse $Grid_i$ creation (See Fig. 2.) Any coarse $Grid_i$ resolution is twice smaller than previous $Grid_{i-1}$. We create n grids in total; the largest resolution of any dimension of $Grid_{n-1}$ is 8. For each $Grid_i$ we maintain a single array *numRefs* (e.g., $Grid_i$*.numRefs* that keeps the number of triangle references per cell). Each element of this array is equal to the sum of underlying 8 elements in the finer $Grid_{i-1}$.

We choose $Grid_{n-1}$ resolution bound of 8 to efficiently map the hierarchy emission algorithm to CUDA implementation. We take into account the warp width (group of threads that execute together) and the amount of computation. The maximum resolution of 16 was also tested and resulted in slower BVH emission but faster ray tracing afterwards.

Hierarchy emission At this stage, we build a hierarchy of links between BVH nodes without computing actual bounding boxes. We build an adaptive BVH using approximate SAH heuristic guided by $Grid_i$ $(i = n-1, n-2, \ldots, 0)$. We build the tree in the breadth-first order where each tree level is generated in parallel on the GPU (see Algorithm 2). We do not repartition triangle references.

Similar to Lauterbach et al. [10] we maintain a queue of nodes that are subject to split operation. The maximum size of each split queue is twice the number of triangle references. Each node split operation may result in 0, 1 or 2 new split queries for the next level generation. These split queries are written into the output split queue sparsely. A hierarchy of BVH links (*linkArray*) is accumulated sequentially level by level with nodes from compacted output queue. Namely, the BVH nodes are stored breadth-first, and we can extract node ranges for each tree level (levelOffset array, see Algorithm 2). The output queue is considered as an input split queue for the next level generation. The work is stopped when there are no new split queries.

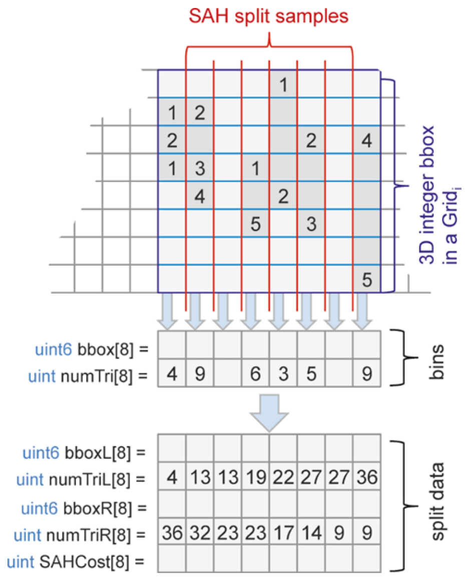

Fig. 3 Warp-wise grid-based evaluation of the SAH cost function for inner node creation. 2D-projection

The *linkArray* array represents relations between inner nodes of the BVH while leaf nodes represent blocks of triangle references. Using *levelOffset* array and the number of tree levels produced we refit AABBs of all nodes in the bottom-up order. At each tree level iteration, the bounding boxes of nodes at this level are computed in parallel using boxes from the previous level.

Approximate SAH evaluation Each element in the split queue represents a task to evaluate a SAH cost function and produces two new BVH nodes. For each task, we load the address of $Grid_i$ and *Integer Bounding Box* (IBB) (see Fig. 2). The integer bounding box represents the grid subset we are working on. For initial split task (the root node) we load the address of $Grid_{n-1}$ and the $IBB = (0, 0, 0, 8, 8, 8)$. If all extents of the current *IBB* are less than 5 then we advance to the finer level $Grid_{i-1}$ and multiply all *IBB* components by 2. Thus, we always work in a grid subset with resolution up to $8 \times 8 \times 8$ (see blue border in Fig. 3). Each cell of the grid subset contains the number of triangles references (see Fig. 3). For each axis, x, y and z, we evaluate up to 7 SAH split candidates (see red lines in Fig. 3) using binning approach (up to 8 bins). For each bin, we track the number of triangle references and the integer bounding box (bounding boxes are shown with grey cells on Fig. 3). In practice, the task processing is assigned to 8 parallel threads (e.g., each CUDA 32-thread warp evaluates 4 split tasks in parallel). The split data is generated using warp-wise prefix sums.

The optimal best split candidate can be computed by the *SAH* cost [5]:

$$SAH(P) = C_T + C_I\left(\frac{SA_L}{SA}N_L + \frac{SA_R}{SA}N_R\right), \tag{1}$$

where P is the splitting plane candidate, SA is the surface area of the current node, SA_L, SA_R are surface areas of the current node to the left and to the right of P, N_L and N_R are the numbers of the node's triangles to the left and to the right of P, and C_T and C_I are relative costs of plane intersection and node traversal.

Our best split candidate is computed using the following *SAH* cost metric:

$$SAH(P) = SA(bboxL)N_L + SA(bboxR_R)N_R. \tag{2}$$

Using this split plane, the inner-node is produced and two open splits are added to the working queue (the first one is associated with the current $Grid_i$ and $IBB = bboxL$, the second one is associated with $Grid_i$ and $IBB = bboxR$). The leaf node is created only if the current *IBB* bounds only one cell of $Grid_0$.

Object-median split We may have multiple triangles references in a small region even when we use 30-bit triangle/grid cell mapping. In a post-process, we find BVH leaves that contain more than 4 triangle references. We subdivide these nodes into small subtrees using object median split until each leaf contains no more than 4 triangles.

BVH refitting Finally, we compute actual AABBs of BVH nodes using bottom-up refitting (we exploit the references generated in the hierarchy construction stage). This process is also implemented on the GPU: we execute refitting kernels for each level of the BVH starting from the lowest level.

Triangle split stage The SAH evaluation step of our builder uses a density grid of triangle centroids. Actual bounding boxes of triangles are not taken into account during the SAH evaluation as it is done in the Walds binning approach [18]. If triangles are large and their distribution is non-uniform, there may be a significant overlap between AABBs that bound these triangles. Bounding box overlaps may result in inefficient raytracing. Stitch et al. [16] described a solution to this problem for the CPU-side BVH builder that supports spatial splits within the BVH (as in kd-trees) if they improve the SAH cost. The spatial splits, if applied, may split a long triangle into multiple references that are bounded with tighter AABBs. As a result, any triangle can be approximated with a number of tight AABBs that better cull the empty space. The use of spatial splits improves ray tracing performance by 20–60% for the scenes with long and wide/thin triangles reducing the overlaps between the AABBs and increasing the number of triangle references. This process is relatively slow. Furthermore, it can cause a memory overflow, especially for CAD scenes (composed of long/wide triangles), and cannot allocate large arrays for triangle references and BVH boxes.

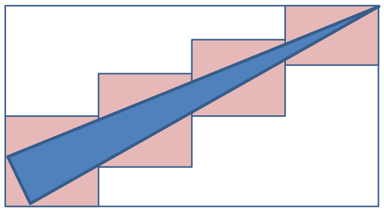

Fig. 4 A triangle reference subdivision. Large triangles can be subdivided into many new references

We implement the triangle split stage that tries to split all long triangles into several parts that are represented with AABBs (see Fig. 4) prior to the BVH build stage. Each triangle can be represented with several AABBs to better approximate its shape. The number of approximating AABBs computed for all scene triangles is bounded by some fixed number of references (e.g., no more than 10% new references of existing triangles). We opt to have triangle reference dimensions not larger than

$$\text{sceneCellWidth} = \frac{\text{sceneExt}}{\text{NoOverlapResolution}}, \tag{3}$$

where sceneExt is the maximum scene extent and NoOverlapResolution is the resolution of a virtual scene grid G where the overlaps are not allowed. The split algorithm tries to split triangles that are longer than the cell width of this virtual grid. This virtual grid is never constructed, only its dimensions are used in our computation.

For each AABB of the ith triangle reference, we compute the number of requested triangle splits:

$$\text{reqSplits}(\text{refAABB}_i) = \text{int}\left(\frac{\text{refMaxExt}_i}{\text{sceneCellWidth}}\right), \tag{4}$$

where refMaxExt_i is the maximum extent of the reference AABB. The value of reqSplits is rounded to the nearest integer and represents the number of new references that we need to satisfy the size condition. This value is scaled by $\text{scaled\_reqSplits}_i$:

$$\text{scaled\_reqSplits}_i = \frac{\text{reqSplits}(\text{refAABB}_i)}{\sum_{j\in\text{all refs}}\text{reqSplits}(\text{refAABB}_i)}\text{availableMemory},$$

where availableMemory is the number of entries that can be used for new references storage. Each ith triangle reference from the input queue is divided into the $\text{scaled\_reqSplits}_i + 1$ new AABBs uniformly distributed along the longest extent of the parent AABB (see Fig. 4). Resulting triangle references bounded with tighter AABBs are written to the references output queue.

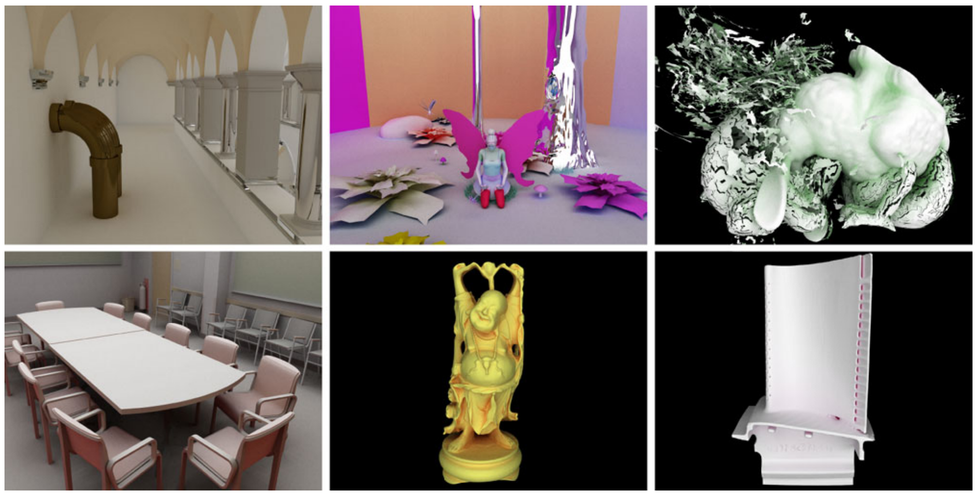
Fig. 5 The screenshots from our test scenes rendered (Sponza, Fairy Forest, Exploding Dragon, Conference, Happy Buddha, and Turbine Blade). The BVHs for these scenes were built using our approach (see Table 1 for builder stats). The absolute path tracing timings (5-bounce diffuse inter-reflection): 55 ms (45 Mrays/s), 46 ms (65 Mrays/s), 35 ms (63 Mrays/s), 36 ms (65 Mrays/s), 16 ms (105 Mrays/s), 19 ms (98 Mrays/s)

We perform two passes of triangle subdivision. In the first pass, the initial triangles are divided along their longest dimensions (e.g., x-dimension). In the second pass, the references from the first pass are also divided along their longest dimension (for a wide triangle, it is a different dimension than in the first pass). This process is implemented in a GPU-friendly fashion. Small triangles can be divided into 2 or 3 new references (or not divided at all). Very long triangles can be divided into 1000s of new triangle references. The overall amount of output triangle references will be bounded by 110% of initial number of triangles. The irregularity of subdivision is controlled efficiently using explicit work queue organization using GPU scan and segmented-Scan procedures [14].

4 Results and comparisons

Implementation setup We implemented our BVH building algorithm using CUDA 3.0. All measurements were done with NVIDIA GTX 480 with 1.5 GB of GPU memory and Core 2 Duo 2.13 GHz with 2 GB of main memory. We use Utah Fairy Forest, UNC Exploding Dragon, Conference, Sponza, Stanford Dragon, Happy Buddha, Turbine Blade scenes for our tests (see Fig. 5). Conference and Sponza scenes contain long and wide/thin triangles and provide a good test case for our split triangles approach. Other scenes are finely tessellated, and Fairy Forest and Exploding Dragon are animated scenes that demonstrate dynamic BVH building.

For all scenes, we split triangles using 128^3 virtual grid resolution. We limit the number of new split triangle references to be no more than twice the original number of triangles. We build BVHs using a 30-bit cell id (21 bits encode the TopGrid cell ID, 9 bits encode the BottomGrid cell ID). For each node the SAH is evaluated using up to $8 \times 8 \times 8$ sub-grid range. We continue hierarchy emission until each tree node contains no more than 4 triangles. We analyze ray tracing performance BVHs using a path tracing test with 5 diffuse ray bounces at 1024×768 resolution. The path tracer is implemented using the depth-first ray traversal kernel improved by persistent threads where each ray is mapped to one thread [1]. The path tracer is additionally accelerated using a fast ray sorting stage [4]. The path tracing kernel takes the stream of sorted rays for the coherent execution on the GPU.

Build stats In Table 1, we show absolute building times for our BVH construction algorithm. The absolute building times include all stages from “Split triangles” stage to “AABB refit” stage. For larger scenes, the building time increases relatively slowly. The reason is the bounded computational work for SAH evaluation of each node. While we use the same resolution of TopGrid in all scenes, the number of non-empty TopGrid cells varies from scene to scene. This determines the number of bottom grids and influences building times. The number of non-empty cells within TopGrid varies from 0.1% to 2%.

Tree quality We evaluate tree quality using path tracing performance and the SAH cost [5] of produced hierarchies. This SAH metric represents the probabilistic estimate of the number of operations required to traverse a tree with a random ray. The metric is computed recursively starting at the root node:

$$C(N) = 2 + C(NL)\frac{\mathrm{area}(NL)}{\mathrm{area}(N)} + C(NR)\frac{\mathrm{area}(NL)}{\mathrm{area}(N)},$$
$$C(\mathrm{leaf}) = \mathrm{numTriangles(leaf)},$$

where NL and NR are the children of the node N, area is the surface area of the bounding box of the node. Each leaf node returns the number of triangles. This traversal cost metric is independent of traversal algorithm used.

Table 1 Build stats for our grid-assisted BVH builder. These build stats use parameters described in Implementation Setup section. The second column is the number of triangles in each scene. The third column is the number of AABBs that approximate triangles relative to the number of triangles. The forth column is the total memory consumption of the TopGrid, the number of BottomGrids, triangle ranges for all the cells and grid mip-map levels. The fifth column represents the number of BVH nodes produced by our build. The sixth column is the absolute building time, including triangles split, grid construction, hierarchy emission and AABB-tree refitting

	Num triangles	Num Refs	Grid memory	Num nodes	Build time
Plane Light Source	8	200%	20 Mb	15	4ms
Sponza	67K	186%	116 Mb	83K	5.1ms
Fairy Forest	174K	127%	118 Mb	140K	7.5ms
Exploding Dragon	252K	100%	27 Mb	166K	8.1ms
Conference	282K	148%	115 Mb	278K	8.2ms
Stanford Dragon	870K	100%	67 Mb	497K	11.1ms
Happy Buddha	1080K	100%	73 Mb	590K	11.4ms
Turbine Blade	1760K	100%	80 Mb	990K	16.1ms

Stage Breakdown

Stage	Time
Split triangles	2.7 ms
Compute Cell IDs	0.19 ms
Radix Sort	1.64 ms
Build TopGrid	0.5 ms
Build BottomGrid	0.4 ms
Emit Hierarchy	4.67 ms
Refit AABB Tree	1.0 ms

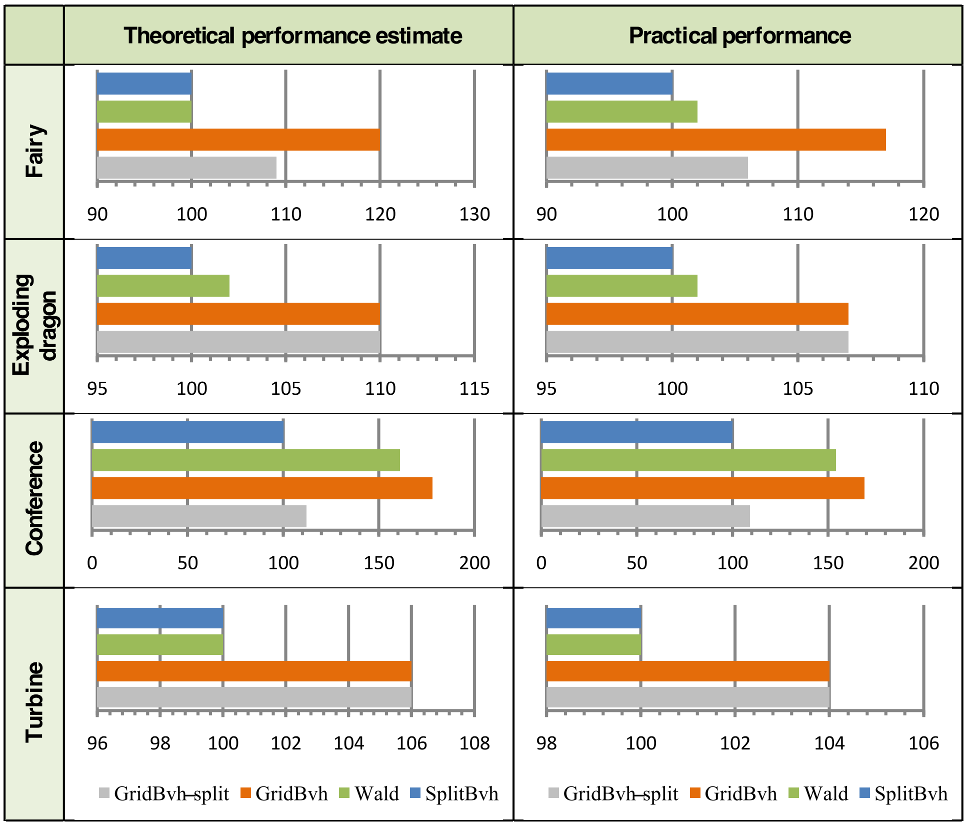

Table 2 Performance evaluation of our BVH builder algorithm (GridBvh with and without triangle split stage) compared to our implementation of some recent algorithms (SplitBvh by Stitch et al. [16] and Wald [18]) for the four scenes. The left column shows theoretical BVH performance estimate that is computed using SAH cost metric over the tree. The right column shows practical performance of resulting BVHs with measured timings of a 5-bounce path tracing (see Fig. 5). All values in the chart are normalized to 100% where 100% is performance of SplitBvh by Stitch et al. [16]. Smaller values mean better performance. SplitBvh approach results in the fastest ray tracing; however, the BVH construction time is longer. In contrast, GridBvh-split has comparable ray tracing performance, but can be constructed in a few milliseconds

In Table 2, we show quality measurements of BVHs produced by our builder with and without triangle split stage, high-quality Wald's CPU builder [18] and SplitBVH CPU builder [16]. Wald's builder uses binning approach to accelerate building, and Stitch et al. [16] use full SAH sweep [6] building and spatial splits that reduce the bounding box overlap problem. We have implemented these builders (stopping tree construction when the node contains no more than 4 triangles) on the CPU and transfer the generated BVHs to the GPU where they are used in the same traversal algorithm. Table 2 shows that SplitBVH [16] produces trees that result in the fastest path tracing. However, this builder is relatively slow (a few seconds or minutes). Our GridBVH without triangle-split stage can result in a slower ray tracing for scenes with non-uniformly sized triangles such as Conference and Sponza. However, the introduction of a cheap GPU-based triangle split stage makes our builder competitive with other high-quality builders.

Table 3 Comparison times for acceleration data structure builders presented in recent papers. In brackets we show hypothetical $2\times$ faster times assuming these algorithms would run on GTX 480 that is used for our measurements. Preliminary evaluation of the HLBVH algorithm [12] on GTX 480 showed 1.5–2 times faster running time compared to GTX 280

	Fairy	Exploding Dragon	Buddha	Turbine
GridBVH ~110% SAH Cost **(good tree quality)** BVH, GTX 480	8ms	8ms	11ms	16ms
Pantaleoni and Luebke ~120% SAH Cost **(slightly worse tree quality)** HLBVH, GTX 280			32ms (16ms)	42ms (21ms)
Pantaleoni and Luebke ~110% SAH Cost **(good tree quality)** HLBVH + SAH, GTX 280			137ms (68ms)	158ms (79ms)
Lauterbach et al. LBVH, GTX 280	124ms (62ms)	66ms (33ms)		
Kalojanov and Slusallek Uniform grid, GTX 280	24ms (12ms)	16ms (8ms)		

Algorithm 1 GPUBVHBuild(BVH bvh, **Mesh** mesh)

```
int numTriangles = mesh.numTriangles;
int refArraySize;
// TriTefernce is a (triangleID, AABB) pair that encloses
// a triangle or a part of a triangle
TriReference refArray[2*numTriangles]
// Split wide triangles into many approximating
// tight AABBs
refArray = SplitBVH(mesh.triangleArray);

int CellIDs[refArraySize]
// 30-bit cell id where first 21 bits encode topGrid cell
// ID and remaining 9 bits encode bottomGrid ID.
CellIDs = ComputeCellIDs(refArray);

int refIDs[refArraySize] = {0, 1, 2, 3, ..., refArraySize-1};
// Reorder refIDs array with keys represented by CellIDs.
RadixSort(CellIDs, refIDs);

// topGrid represents a mip-map of numRefs values.
// (resx, resy, resz) is the grid size.
// Top mip level has resolution 8 × 8 × 8.
Grid topGrid = AllocateGrid(resx, resy, resz, 1);
// Uses the most significant 21 bits of cellID. Every
// non-empty cell contains a range [refBegin..numRefs)
// of refIDs that fall into the cell.
int numNonEmptyTopGridCells =
                    BuildTopGrid(topGrid, CellIDs);

Grid bottomGrid =
       AllocateGrid(8, 8, 8, numNonEmptyTopGridCells);
// Uses last 9 bits of the CellID and distributes triangles,
// every bottom grid is referenced in non-empty cell
// of topGrid for transition during hierarchy emission.
// Every non-empty bottomGrid cell contains a range
// of refIDs that fall into the cell. bottomGrid can also
// represent the mip-map of numRefs values.
bottomGrid = BuildBottomGrid(topGrid, CellIDs);

// Each linkArray[i] references two child nodes or the
// range of primitives. Results in a breadth-first layout.
EmitHierarchy(bvh.linkArray, topGrid);

// Bottom up refit of aabbArray considering the
// linkArray hierarchy. The nodes from the same BVH
// level are updated in parallel using LevelOffset array
// (Algorithm 2)
RefitAABB(bvh.aabbArray, bvh.linkArray);
```

Varying build parameters We have described build parameters for our BVH construction algorithm. Varying these parameters can result in different tree qualities and build times. For example, $16 \times 16 \times 16$ sub-grid range for SAH evaluation results in $\approx 30\%$ slower building time and ≈ 3–5% faster ray tracing. Cell ID representations with 24- or 27-bits encoding triangle centroids result in significantly lower memory consumption, slightly faster BVH build time (≈ 2–3%) and slightly slower ray tracing ($\approx 5\%$). Lower virtual grid resolution for triangle split stage results in extended AABBs and slower ray tracing. On the other hand, higher virtual grid resolution results in more triangle references across the tree that can be tested for intersection with the same ray multiple times.

Comparison to the other GPU builders Thus far we have analyzed the quality of BVHs produced by our algorithm. Here we compare our builder with recent results of other acceleration data structure builders (see Table 3).

We use Nvidia GTX 480 for our measurements; all the other results from Table 3 use GTX 280 hardware. In brackets we show $2\times$ shorter times (we assume that the algorithms listed would run up to $2\times$ faster on GTX 480). Building times from Pantaleoni and Luebke approach [12] with $\approx 120\%$ SAH Cost trees (pure HLBVH) are equal to our building times for trees with $\approx 110\%$ SAH Cost, although their algorithm does not address the AABB overlap issue in BVHs for scenes with long and wide/thin triangles. This issue was solved in our real-time triangle split stage (also in Stitch et al. [16] for offline CPU builder). If we compare our $\approx 110\%$ SAH Cost trees with Pantaleoni's $\approx 110\%$ SAH Cost trees, then our build times are $4\times$ faster. Kalojanov and Slusallek [8] grid construction time is fast, but provides much slower ray tracing performance.

5 Conclusions

In this paper, we have presented an efficient algorithm for constructing and adaptive bounding volume hierarchy in linear time on a GPU. The constructed BVH is used

Algorithm 2 EmitHierarchy(**int2*** linkArray, **int*** sortedRefIDs, **Grid** topGrid, **int** numRefs)

```
BVHQueue qSplit[2];

// levelOffset[] array and numLevels are
// used for bottom-up BVH refitting.
int numLevels = 0;
int levelOffset[60];

int numNodesTotal = 0;
int numQElems = 1;
int qin = 0;

levelOffset[numLevels++] = 0;
IAABB box = make_iaabb(0, 0, 0,
            topGrid.resx, topGrid.resy, topGrid.resz);
qSplit[qin] = SplitQueueInit(numRefs, box);
numNodesTotal++;

// Build the hierarchy of links
while numQElems > 0 do
    LevelOffset[numLevels++] = numNodesTotal;

    // Evaluates SAH, split each node into two and write
    // them into qSplit[1−qin] that can be 2× larger
    // than qSplit[qin].
    SplitQueueProcess(qSplit[qin], qSplit[1−qin],
                      numQElems);

    // Compute prefix sum over non-empty qSplit[1−qin]
    // positions and compact them.
    int newNumQElems =
        CompactQueue(qSplit[1-qin], 2*numQElems);

    // for each i = [0.. newNumQElems)
    //    linkArray[numNodesTotal + i] =
    //         qSplit[1−qin].NodeInfo[i];
    linkArray = AccumulateLinks(numNodesTotal,
                      qSplit[1−qin], newNumQElems);

    numQElems = newNumQElems;
    numNodesTotal + = newNumQElems;
    qin = 1 − qin;
end while
```

as an acceleration data structure for ray tracing animated polygonal scenes. We accelerate the construction process of BVHs using several techniques: compression–sorting–decompression, an approximated SAH evaluation using uniform grids, and limited work to produce a single BVH node. We do not make any assumptions about the type of geometry or underlying motion. We also address the problem of long and non-uniformly tessellated triangles that cause many overlapping bounding boxes. We split long triangles into several triangle references with tight bounding box approximations. We demonstrate the performance of resulting BVHs using path tracing and show that high-quality BVHs can be constructed quickly and result in fast ray triangle intersections.

In the future, we plan to add support for very large models and other geometry primitives. It would also be interesting to test our algorithm on other platforms and adjust implementation to exploit new architectures.

Appendix

The pseudocode for the building procedure is summarized in Algorithm 1, and the pseudocode for the actual hierarchy emission is in Algorithm 2.

References

1. Aila, T., Laine, S.: Understanding the efficiency of ray traversal on GPUs. In: Proc. High-Performance Graphics 2009, pp. 145–149 (2009)
2. Choi, B., Komuravelli, R., Lu, V., Sung, H., Bocchino, R.L., Adve, S.V., Hart, J.C.: Parallel SAH k-D tree construction. In: Proceedings of High Performance Graphics (2010)
3. Foley, T., Sugerman, J.: KD-tree acceleration structures for a GPU raytracer. In: Graphics Hardware 2005, pp. 15–22 (2005)
4. Garanzha, K., Loop, C.: Fast ray sorting and breadth-first packet traversal for GPU ray tracing. Comput. Graph. Forum **29**(2) (2010)
5. Goldsmith, J., Salmon, J.: Automatic creation of object hierarchies for ray tracing. IEEE Comput. Graph. Appl. **7**(5), 14–20 (1987)
6. Havran, V.: Heuristic ray shooting algorithms. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague (2000)
7. Ize, T., Wald, I., Parker, S.G.: Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures. In: Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization (2007)
8. 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, pp. 23–28 (2009)
9. Kalojanov, J., Billeter, M., Slusallek, P.: Two-level grids for ray tracing on gpus. In: Eurographics 2011. Comput. Graph. Forum **30**(2) (2011)
10. Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., Manocha, D.: Fast BVH construction on GPUs. Comput. Graph. Forum **28**(2), 375–384 (2009)
11. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. ACM Queue **6**(2), 40–53 (2008)
12. Pantaleoni, J., Luebke, D.: HLBVH: Hierarchical LBVH construction for real-time ray tracing. In: High Performance Graphics (2010)
13. Popov, S., Günther, J., Seidel, H.P., Slusallek, P.: Stackless KD-tree traversal for high performance GPU ray tracing. Comput. Graph. Forum **26**(3), 415–424 (2007)
14. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (2009)
15. Shevtsov, M., Soupikov, A., Kapustin, A.: Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. Comput. Graph. Forum **26**(3) (2007)

16. Stich, M., Friedrich, H., Dietrich, A.: Spatial splits in bounding volume hierarchies. In: Proc. High-Performance Graphics (2009)
17. Wachter, C., Keller, A.: Instant ray tracing: The bounding interval hierarchy. In: Proceedings of the 17th Eurographics Symposium on Rendering, pp. 139–149 (2006)
18. Wald, I.: On fast Construction of SAH based bounding volume hierarchies. In: Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing (2007)
19. Wald, I., Boulos, S., Shirley, P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. ACM Trans. Graph. **26**(1) (2007)
20. Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time KD-tree construction on graphics hardware. ACM Trans. Graph. **27**(5), 126:1–126:11 (2008)

Kirill Garanzha received his M.S. in Computer Science from Bauman Moscow State Technical University, Russia, in 2009. Since then he has been a Ph.D. student at the Keldysh Institute of Applied Mathematics, Russian Academy of Sciences. Kirill's primary interest area is Computer Graphics, in particular, global illumination rendering. His Ph.D. dissertation thesis title is "Out-of-core global illumination rendering on memory limited architectures". Kirill has also been a member of OptiX group at NVIDIA since October 2010.

Simon Premože received his B.S. in Computer Science from University of Colorado at Bouler, USA, in 1996. He received his Ph.D. in University of Utah, Salt Lake City, USA, in 2003. Simon's primary interest area is photorealistic rendering, in particular, global illumination rendering. His Ph.D. dissertation thesis title is "Approximate Methods For Illumination and Light Transport in Natural Environments".

Alexander Bely is currently a master-student in RosNOU University, Moscow Russia. His interests include Computer Graphics and business activities. He is the co-founder of the companies such as FUGU and Capital Research, Russia.

Vladimir Galaktionov has received Ph.D. in Physics and Mathematics in 1982 (Moscow Institute of Physics and Techniques), Doctor of Science degree in Physics and Mathematics in 2006 (Keldysh Institute of Applied Mathematics). Since 2003 is the head of computer graphics department in KIAM. More than 50 publications in the area of computer graphics and computational optics.