High Performance Graphics (2010)
M. Doggett, S. Laine, and W. Hunt (Editors)

HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry

J. Pantaleoni[1] and D. Luebke[1]

[1]NVIDIA Research

Abstract

*We present HLBVH and SAH-optimized HLBVH, two high performance BVH construction algorithms targeting real-time ray tracing of dynamic geometry. HLBVH provides a novel hierarchical formulation of the LBVH algorithm [LGS*09] and SAH-optimized HLBVH uses a new combination of HLBVH and the greedy surface area heuristic algorithm. These algorithms minimize work and memory bandwidth usage by extracting and exploiting coarse-grained spatial coherence already available in the input meshes. As such, they are well-suited for sorting dynamic geometry, in which the mesh to be sorted at a given time step can be defined as a transformation of a mesh that has been already sorted at the previous time step. Our algorithms always perform full resorting, unlike previous approaches based on refitting. As a result they remain efficient even during chaotic and discontinuous transformations, such as fracture or explosion.*

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

1. Introduction

Interactive ray tracing makes wide use of axis-aligned bounding volume hierarchies (BVH) as spatial indexing structures [AL09, WMG*07]. Considerable research has investigated fast construction of acceleration structures for dynamic geometry. Much of that literature deals with serial or moderately parallel computation [WK06, CHCH06, WBS07], but some recent work targets efficient parallel algorithms for multicore [PGSS06, SSK07, Wal07] and many-core architectures [LGS*09, ZHWG08]. We wish to explore massively parallel algorithms suited for real-time ray tracing of complex, fully dynamic datasets such as those in games.

Inspired by the recent work of Garanzha and Loop [GL10], we extend the approach of Lauterbach et al. [LGS*09] by introducing a novel hierarchical algorithm that reduces substantially both the amount of computations and the memory traffic required to build a Linear Bounding Volume Hierarchy (LBVH), while still exposing a data-parallel structure. As our algorithm is based on extracting and exploiting coarse-grained spatial coherence already present in the input meshes, it is particularly efficient for sorting dynamic geometry, where the mesh to be sorted at a given time step is defined as a transformation of a mesh which has been already sorted at the previous time step. Unlike much previous work based on refitting an existing BVH to new vertex positions, our algorithm performs a full sorting operation each frame. This suppports dynamic geometry well, since the algorithm output does not use and is not sensitive to the quality of the BVH constructed at the previous frame. Moreover, it produces efficient hierarchies even for chaotic or discontinuous transformations, as in fracture or explosion.

We make three main contributions. First we introduce a more efficient reformulation of the Morton curve-based primitive sorting step required by the original LBVH building procedure, in which we employ a hierarchical grid decomposition to exploit spatial coherence present in the input mesh. We significantly reduce both work and memory traffic by coupling a compress-sort-decompress (CSD) algorithm [GL10] to sort the Morton codes according to their most significant bits with the use of an in-core block-based algorithm to complete the sorting with respect to the remaining bits. Second, we describe a novel node hierarchy emission procedure that improves on the technique of Lauterbach et al. [LGS*09] in computational and memory efficiency, and solves some memory usage and tree layout issues present in the original algorithm. Third, we propose a new hybrid algo-

rithm that uses the surface area heuristic (SAH) [Hav00] to build the top BVH levels on the coarse clusters produced by our HLBVH scheme, and uses our fast Morton curve-based partitioning within cluster subtrees.

We have implemented our algorithm in CUDA [NBGS08], relying on Thrust (http://code.google.com/p/thrust/) for all standard algorithms such as global memory sorting, reduction and scan operations [GGKM06, SHZO07, SHG09], and benchmarked it on an NVIDIA GeForce 280 GTX GPU. The resulting system can handle fully dynamic geometry containing one million triangles at real-time rates, constructing the BVH in less than 35ms on average using the HLBVH algorithm and in about 100ms using our SAH-based hybrid. We believe this approach will enable future games to use real-time ray tracing on complex dynamic geometry.

2. Background

Researchers have investigated fast construction of bounding volume hierarchies on both multicore CPUs [WK06, Wal07, WBS07] and massively parallel GPUs [LGS*09]. We summarize here the most directly relevant results.

2.1. LBVH

Lauterbach et al. [LGS*09] introduced a BVH construction algorithm based on sorting the primitives along a space-filling Morton curve running inside the scene bounding box. Space-filling curves have long been used for improving spatial algorithms [Bia69]. The coordinate of a three dimensional point along a Morton curve of order n, also called its *Morton code*, can be computed discretizing its coordinates to n bits and interleaving their binary digits, thus obtaining a $3n$ bit index. As the algorithm transforms the problem of building a hierarchy into the problem of sorting a set of points along a curve, Lauterbach et al. call this the *Linear Bounding Volume Hierarchy* (LBVH) algorithm. We extend their work by solving the same problem - sorting the primitives along a global Morton curve - with a novel, more efficient hierarchical algorithm.

2.2. Refitting

Fast updates to BVHs for deformable models are generally obtained through *refitting*, which keeps the BVH topology while recomputing the extent of the bounding volumes after vertex motion. This operation is very efficient and runs in linear time, but the resulting hierarchies can be poorly suited to high performance ray tracing. However, a pure refitting algorithm can produce arbitrarily bad trees as vertices move from their original positions: large deformations tend to cause significant inflation and overlap of the bounding volumes, significantly reducing ray traversal performance. Researchers have also proposed hybrid algorithms to track which subtrees need a rebuild and which can just be refit after a deformation [Gar08], introducing a tradeoff between hierarchy quality and construction speed; such systems need to reconstruct the entire BVH after significant degradation [LeYM06, WBS07, IWP07]. In contrast, our algorithm always performs a full rebuild, but exploits coarse-grained spatial coherence available in the input ordering of the primitives. This in turn exploits "almost sorted" input: if after a deformation the algorithm processes the primitives in the order induced by a BVH before the deformation occurred (e.g. the BVH from the previous frame), the BVH construction of the deformed mesh will automatically take less time. In other words, by rebuilding from scratch each frame we put a lower bound on the quality of the trees we produce and an upper bound on the generation time.

2.3. Compress-Sort-Decompress

Recently Garanzha and Loop [GL10] introduced an efficient GPU ray sorting method based on hashing rays to integer keys, applying a run-length encoding compression to the keys array, sorting the obtained key-index run-length descriptors, and decompressing the ordered array to obtain the full sorted sequence of hash key-ray index pairs. This *compress-sort-decompress* (CSD) scheme exploits spatial coherence already available in the input sequence by encoding blocks of contiguous elements which can be considered already sorted as single items, and unpacking them after the sorting is done. As future work, the authors suggest the possibility of exploiting this scheme in the context of BVH construction by sorting the primitives along a Morton curve, compressing the obtained array of Morton codes, constructing a BVH and decompressing the result. This scheme forms the basis for some of our contributions.

3. HLBVH

Our contribution starts with the observation that each value assumed by a Morton code identifies a voxel of a regular grid spanning the scene's bounding box (see Figure 2), and that the higher bits represent the parent voxels in a hypothetical hierarchy of such grids: the highest order 3 bits represent the coarsest grid of 8 voxels, the next 3 bits represent a subdivision of each voxel of the coarsest grid in 8 octants, and so on. Each additional bit splits the parent in two. The original LBVH algorithm was implemented using a Morton curve of order $n = 10$, resulting in a 30 bit grid.

Our algorithm splits the original LBVH algorithm into a 2-level hierarchical problem, where the first level of the hierarchy consists in sorting the primitives into a coarse $3m$ bit grid according to an m-bit Morton curve (with $m < n$), and the second level consists in sorting all primitives within each voxel of the coarse grid according to the remaining $3(n-m)$ bits.

If m is relatively small (e.g. 5 or 6), two consecutive primi-

tives in the input mesh will likely fall in the same voxel of the coarse grid. In other words, the higher the spatial coherence of the input mesh, the higher the probability that the most significant $3m$ bits of the Morton codes of two consecutive primitives will be the same. We exploit this fact by using a CSD scheme to perform this *top level* sorting operation and build the corresponding BVH tree. Once the top level tree is built, we proceed with sorting according to the remaining bits and creating the missing levels of the BVH tree. In practice, primitive sorting and hierarchy emission can be decoupled, and we describe them separately.

3.1. Primitive Sorting

Figure 1 summarizes the top level primitive sorting algorithm. Initially, we compute and store the 30-bit Morton codes of each of the N input primitives with a simple *foreach* data-parallel construct. The top level primitive sorting step starts run-length encoding the codes by their high $3m$ bits, applying a compaction kernel to extract a list of indices pointing to the beginning of each run, obtaining an array with $M \leq N$ items called *run_heads*. We then construct two more arrays using one more *foreach* loop, where the first contains the M unique $3m$-bit values corresponding to each run and the second assigns them an increasing id from 0 to $M-1$. Next, we invoke a key-value pair $3m$-bit radix sorting algorithm using the run values as keys and the run ids as values. The first stage is then completed by decoding the sorted runs: an operation which involves computing an exclusive scan on the sorted run lengths to determine the new offsets for each run, and another *foreach* loop to expand the runs.

At this point, the Morton codes are sorted by their coarse-grid index: the array can be thought of as if it was composed by sorted segments, where each segment corresponds to all points contained in a given $3m$-bit voxel. The least significant $3(n-m)$ bits in each segment are yet to be sorted. In order to complete the sorting, we proceed by grouping consecutive segments in small blocks, containing up to 2000 elements each, and we launch a single CUDA kernel performing an individual sorting operation per block, where each block is assigned to a single concurrent thread array (CTA), and the entire sorting is performed within shared memory using an odd-even sorting algorithm [Knu68]. The maximum block size is determined by the amount of shared memory available per processor. Here, we assume that each voxel contains less than 2000 elements, which is reasonable if the coarse grid is fine enough (in practice, on all meshes we tested containing up to 1M triangles we have found this to be the case with $m = 6$, while for 10M triangles we needed to raise the coarse grid resolution to $m = 7$).

Compared to a global radix sort, the advantages of our approach are three-fold:

- **Reduced work and global memory traffic:** due to the use of an $O(M)$-complexity CSD for the most significant bits, and fast shared memory sorting for the least significant ones;
- **Reduced global synchronization points:** by using a $3m$ bit radix sort in the first phase, rather than a $3n$ bit one, we reduce the number of global synchronization points. The second phase of our algorithm is performed entirely within a single kernel call;
- **Greater efficiency:** our intra-CTA odd-even sorting algorithm is up to $8\times$ more efficient than the state-of-the-art global radix sorting procedure.

```
top_level_sorting()
 1  // compute n bit Morton codes
 2  foreach i in [0,N)
 3    codes[i] = morton_code( prim[i], n );
 4
 5  int d = 3 * (n - m);
 6
 7  // run-length encode the Morton codes
 8  // based on their high order 3m bits
 9  run_heads = compact i such that (c_r ≠ c_l), with:
10        c_r = codes[i] >> d;
11        c_l = codes[i-1] >> d;
12
13  int M = run_heads.size();
14
15  foreach i in [0,M)
16    run_codes[i] = codes[run_heads[i]] >> d;
17    run_indices[i] = i;
18
19  // sort the compressed Morton code runs
20  radix_sort( run_codes, run_indices, 3 * m );
21
22  // compute offsets for the sorted run-lengths
23  offsets = ex_scan( run_length(i) ) with:
24      run_length(i) =
25        run_heads[run_indices[i]+1] -
26        run_heads[run_indices[i]];
27
28  // decode the sorted indices
29  foreach i in [0,M)
30    foreach j in [0,run_length(i))
31      out_indices[offset[i] + j] =
32        run_heads[run_indices[i]] + j;
33
34  // decode the sorted Morton codes
35  foreach i in [0,N)
36    out_codes[i] = codes[out_indices[i]];
```

Figure 1: *Pseudocode for the top level primitive sorting step. Keywords in bold represent data-parallel constructs.*

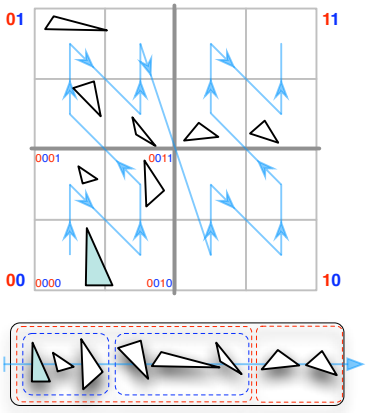

Figure 2: *Morton curve ordering and the corresponding Morton codes.*

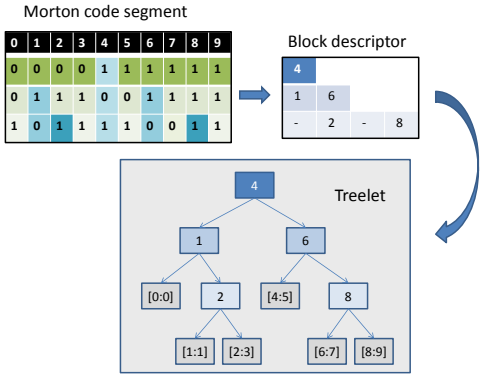

Figure 3: *Treelet emission process. A small block descriptor is extracted from an example segment of Morton codes (with the bit planes shown in different tints of green and the splits highlighted in blue), and the block descriptor is used to generate the corresponding treelet. The internal nodes of the treelet report their split index, while leaves report their spanned range of primitive indices.*

3.2. Hierarchy Emission

The original hierarchy emission algorithm employed by LBVH proceeded as follows: given the sorted list of Morton codes, a kernel proceeded extracting the most important bit, or *level*, at which each key differed from its neighbour. This level represents the highest level at which the key and its direct neighbour are separated by a split in the final node hierarchy. Each such split was then used to generate $3n - l$ more splits, i.e. one for each level below the first split. In order to do so, an exclusive scan counted the number of splits which needed to be generated, and a separate kernel produced two paired arrays of split levels and split positions. A stable radix sort was then used to sort this list of key-value pairs by split level, so as to collect first all splits happening at level 0, then all splits happening at level 1, and so on, ordered by split position within each level. Finally, these arrays were used to generate the node hierarchy, which required to perform an additional sorting pass to order the ranges of primitives spanned by each node by their end index, so as to easily determine parent-child relantionships between all nodes (when this ordering is used, a node is a leaf when its ending Morton code differs from the one of its right neighbour, whereas otherwise its neighbour is its rightmost child).

While all these passes could be performed with data-parallel algorithms, it is important to notice that the two additional sorting operations were performed on arrays significantly larger than N, as each Morton code would often contribute several splits. Moreover, the algorithm had the negative side-effect of producing several *singletons*, i.e. nodes with a single child, and thus required to operate on a very large memory arena, potentially $3n$ times larger than N. Those singletons were later removed skipping all pointers to individual children, but the memory layout of the tree was left unmodified, potentially leaving many holes and harming cache performance during traversal.

We propose a novel algorithm which proceeds in several passes, where each pass analyzes p consecutive bit planes of the sorted Morton codes from most to least significant bit, similarly to an MSD radix sort, and emits all the corresponding depth p treelets. The overall idea is shown in Figure 3 and can be summarized as follows: at the beginning of each pass the primitive array is conceptually split in segments corresponding to each leaf in the currently emitted hierarchy. Each of these segments will generate a small treelet of depth p. These treelets can be fully described by the list of split planes in each segment, i.e. the list of indices where each Morton code differs from its predecessor. As we are analyzing only p bits at a time, there are only $2^p - 1$ possible split planes in each segment, which can hence be organized in small *block descriptors*: short integer arrays with $2^p - 1$ entries, organized as follows (assuming $p = 3$):

- entry 0 represents the position of the highest level split plane (corresponding to the most important bit) in the segment, i.e. the first element whose current p bit planes are of the form $1xx$.
- entries 1 and 2 represent the position of the 2 splits corresponding to the second bit, i.e. the position of the first element whose current p bit planes start by $01x$ or $11x$.
- entries 3 to 6 represent the position of the 4 splits corresponding to the third bit, i.e. the position of the first element whose current p bit planes start by 001, 011, 101 or 111 respectively.

Pseudocode for the body of the loop is given in Figure 4, and the whole algorithm is described here in more detail. Initially, there is a single leaf node containing all primitives. Throughout the outermost loop on all groups of p bit planes,

we will need a way to compute the mapping between any arbitrary primitive index i and the leaf that contains it. We break down this problem in that of mantaining three separate tables:

- an array *head_to_node* of N integers is kept to indicate the mapping between the beginning of each leaf interval, or *segment*, and the corresponding node index: if index i is the beginning of a leaf, *head_to_node[i]* will point to the corresponding node, otherwise it will be flagged with an invalid number, e.g. -1. Initially, $head\_to\_node[0]$ is set to 0 (the index of the root node), and all other indices are set to -1.
- a list of segment heads, where $segment\_head[segment]$ identifies the index of the first primitive in a segment.
- a *segment_id* array of the form $\{1,1,1,1,2,2,2,3,...\}$, mapping each primitive index i to the segment it belongs to. This is obtained performing a scan on a vector containing a 1 for each index i such that $head\_to\_node[i] \neq -1$, and 0 otherwise.

Given these three arrays, we can trivially compute the mapping between any arbitrary primitive index i and its leaf by dereferencing *head_to_node[segment_id[i]-1]*. Similarly, we get the first primitive of the leaf containing primitive i as *segment_head[segment_id[i]-1]*.

The algorithm then proceeds as follows: A *foreach* loop through all the Morton codes determines the highest level split generated by each primitive, writing it in the corresponding position of the corresponding block descriptor. Not all split planes in a block might actually be present: a given bit plane of a node might contain only zeros (or ones), in which case all objects would be left (or right) of the hypotetical split. Again, we handle these situations by marking the corresponding slots with special ids. The number of nodes emitted for each block will be equal to twice the number of actual split planes in each block descriptor, as each split will generate two nodes. We can thus use an exclusive scan to compute the offset at which we will output each treelet. Finally, another *foreach* loop through all the blocks emits the corresponding treelet based on the block descriptor and updates the auxiliary *head_to_node* and *segment_head* vectors.

This algorithm, easily written as a series of data-parallel operations, has two main advantages over the original LBVH hierarchy emission procedure:

- **reduced work and global memory traffic:** by not requiring any additional sorting operation on larger than input arrays, it is a lower complexity algorithm.
- **better memory layout:** by not producing any singletons which must later be eliminated, it outputs a 2-4x smaller node arena without any holes.

3.3. SAH-Optimized HLBVH

While our HLBVH algorithm is capable of producing LBVH hierarchies in a fraction of the time and with a superior mem-

```
hierarchy_emission(codes, N_prims, n_bits)
 1  int segment_heads[]
 2  int head_to_node[ N_prims ] = {-1}
 3  head_to_node[0] = segment_heads[0] = 0
 4
 5  for (level = 0; level < n_bits; level += p)
 6    // compute segment ids
 7    segment_id[i] = scan (head_to_node[i] ≠ −1)
 8
 9    // get the number of segments
10    int N_segments = segment_id[N_prims-1]
11
12    int P = (1 << p) - 1
13
14    // compute block descriptors
15    int block_splits[ N_segments * P ] = {-1}
16    foreach i in [0,N_prims)
17      emit_block_splits(
18        i, [in] primitive index to process
19        codes, [in] primitive Morton codes
20        [level, level + p), [in] bit planes to process
21        segment_id, [in] segment ids
22        head_to_node, [in] head to node map
23        segment_heads, [in] segment heads
24        block_splits ) [out] block descriptors
25
26    // compute the block offsets summing
27    // the number of splits in each block
28    int block_offsets[ N_segments + 1 ]
29    block_offsets[s] = ex_scan (count_splits(s))
30    int N_splits = block_offsets[N_segments]
31
32    // emit treelets and update
33    // segment_heads and head_to_node
34    foreach segment in [0,N_segments)
35      emit_treelets(
36        segment, [in] block to process
37        block_splits, [in] block descriptors
38        block_offsets, [in] block offsets
39        segment_id, [in] segment ids
40        head_to_node, [in/out] head to node map
41        segment_heads ) [in/out] segment heads
42
43    node_count += N_splits * 2
```

Figure 4: *Pseudocode for our hierarchy emission loop.*

Scene	# of Triangles	LBVH	HLBVH		HLBVH + SAH	
			max	min	max	min
Armadillo	345k	61 ms	27 ms	18 ms	72 ms	65 ms
Stanford Dragon	871k	98 ms	36 ms	28 ms	111 ms	95 ms
Happy Buddha	1.08M	117 ms	43 ms	32 ms	150 ms	137 ms
Turbine Blade	1.76M	167 ms	54 ms	42 ms	162 ms	158 ms
Hair Ball	2.88M	241 ms	95 ms	83 ms	460 ms	456 ms

Table 1: *Build time statistics for various scenes. For the first five static scenes, the HLBVH building times are reported both for sorting the original model from scratch, and for resorting it once it has already been sorted.*

ory layout, the topology of the generated trees is equivalent to that obtained with the original algorithm. As such, in some situations HLBVH trees can present highly suboptimal traversal quality, with considerable spatial overlap between different subtrees.

In this section we show how to produce high quality SAH-optimized trees at a modest cost. The idea is to consider the bins, or *clusters*, produced by the coarse Morton curve sorting step in the HLBVH algorithm and feed them to a standard SAH sweep builder [Hav00, WBS07] to construct the top level tree. The only slight but important modification to the original SAH sweep builder comes from the observation that rather than assuming that all primitives have the same intersection cost, we can use the actual SAH cost [GS87] of the cluster subtrees (see Section 4.2 for a more detailed explanation). The rest of the algorithm remains unchanged: we still perform the same exact primitive sorting procedure and we build the bottom level hierarchy (that is to say the hierarchy corresponding to the least significant $3(n-m)$ bits of the Morton codes) using the data-parallel algorithm described in the previous section. While SAH-optimized construction is generally many orders of magnitude slower than our fast HLBVH algorithm, by applying it to the already formed clusters in our coarse grid we keep its overhead at a minimum: typically, a mesh containing one million triangles will be subdivided in about 15k - 30k clusters when a Morton curve of order 6 is used (equivalent to a regular grid subdivision with 2^{18} bins). This procedure can be seen as another instance of the CSD scheme: the primitives are first compressed to a small set of clusters defined as the bins of a coarse grid, sorted by a SAH-based builder in their compressed representation, and finally decompressed.

Note that this approach is the exact opposite of the hybrid algorithms proposed by Lauterbach et al. [LGS*09] and Wald et al. [Wal07], which essentially built the top level hierarchy by a regular grid subdivision (respectively through Morton curve sorting and direct binning) and its subtrees by a SAH-optimized procedure. We believe our choice to be largely superior in the context of ray tracing, as the most important part of the hierarchy where spatial overlap needs to be minimal is the top of the tree, which is touched by almost all rays.

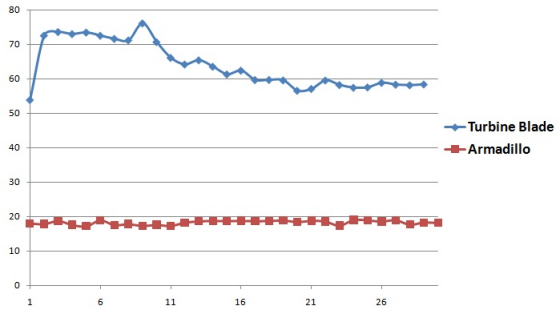

Figure 5: *Rebuilding times for the Bending Armadillo and the Exploding Turbine Blade (milliseconds).*

4. Results

We have implemented all our algorithms using Thrust, a high level library of parallel primitives based on CUDA. The resulting source code is very terse and concise and will be freely available at http://code.google.com/p/hlbvh/.

We have run our algorithms on a variety of typical ray tracing scenes with various complexity: the Armadillo, the Stanford Dragon, the Happy Buddha, the Turbine Blade and the Hair Ball (Figure 6). In order to test dynamic geometry we have built two small animations, the Bending Armadillo and the Exploding Turbine Blade (Figure 7). Our benchmark system uses a GeForce 280 GTX GPU with 1GB of GPU memory, and an Intel Core i7 860 @ 2.8GHz CPU with 4GB of main memory.

In Table 1 we report absolute build times for HLBVH and our SAH-optimized HLBVH algorithm, as well as those of a reference LBVH implementation. For the static scenes, we report both the time needed to sort the original model and that to do a resorting step once the model is already perfectly sorted. For the two animated scenes we report the worst and average build times. Table 2 provides a more detailed breakdown of the timings of the individual components of our builders on some of the same scenes.

We have also run preliminary benchmarks on an early prerelease sample of the upcoming NVIDIA GF100 GPU, obtaining roughly 1.5-2x speedups. By optimizing our algo-

Scene	HLBVH (LBVH)		HLBVH + SAH		SAH	
	nodes	cost	nodes	cost	nodes	cost
Armadillo	586 k (2.02 M)	69.6 (125%)	586 k	60.8 (109%)	691 k	55.7 (100%)
Stanford Dragon	1.66 M (4.69 M)	119.0 (126%)	1.66 M	106.5 (112%)	1.74 M	95.1 (100%)
Happy Buddha	2.04 M (5.73 M)	143.6 (133%)	2.04 M	123.8 (114%)	2.17 M	108.0 (100%)
Turbine Blade	2.40 M (7.39 M)	149.0 (120%)	2.40 M	135.9 (109%)	3.53 M	124.4 (100%)
Hair Ball	4.20 M (11.3 M)	923.9 (104%)	4.20 M	902.7 (101%)	5.75 M	888.6 (100%)

Table 3: *Tree quality statistics for the various builders. The first item in each column is the number of emitted nodes while the second is the SAH cost of the tree. The number in brackets in the first column represents the size of the node arena generated by the original LBVH building procedure. The percentages represent the ratio to the best SAH cost resulting from a full SAH sweep build.*

Scene	HLBVH	HLBVH + SAH
Morton code setup	0.04 ms	same
run-length encoding	2.1 ms	same
top level sorting	1.3 ms	same
run-length decoding	0.7 ms	same
voxel sorting	2.5 ms	same
top level hierarchy	4.5 ms	70.0 ms
voxel hierarchy	9.3 ms	same
bbox computation	0.9 ms	same
other costs	6.4 ms	7.5 ms

Table 2: *Timing breakdown for the Stanford Dragon. The entry called* other costs *refers to overhead due to data layout transformations needed between the various passes of the algorithm.*

Scene	LBVH	HLBVH	reduction
Armadillo	7.8 GB	390 MB	20x
Stanford Dragon	10.1 GB	728 MB	13.9x
Happy Buddha	13.7 GB	918 MB	14.9x
Turbine Blade	14.0 GB	1.2 GB	11.7x
Hair Ball	19.7 GB	1.8 GB	10.9x

Table 4: *Bandwidth usage statistics for LBVH and our novel HLBVH algorithm. In both cases, over 90% of the bandwidth is consumed by hierarchy emission.*

rithms for the richer features of this architecture (e.g. larger shared memory, new cache), we believe we could obtain still higher performance.

4.1. Bandwidth Usage

Table 4 reports the amout of external memory bandwidth consumed by LBVH and HLBVH on all static test scenes. In both cases, up to 90% of the bandwidth is actually consumed by the hierarchy emission procedure. As can be noticed, HLBVH reduces bandwidth by a factor of 10-20x. In both cases, the tests have been carried out on the *unsorted* models: in the case of resorting the bandwidth reduction achieved by HLBVH is improved even further.

4.2. Tree Quality

Besides measuring building performance, we measured the quality of the trees produced by all our algorithms. Table 3 shows the SAH cost [GS87] of the hierarchies produced by LBVH and HLBVH (generating equivalent trees), the SAH-optimized HLBVH, and a reference SAH sweep build implementation as described in [WBS07]. This metric represents a probabilistic measure of the number of operations required to traverse the tree to find all intersections with random rays, and can be computed as the cost of its root node according to the recursive formulas:

$$\begin{aligned} C(node) &= T_1 \cdot N_{children} + \sum_{i}^{N_{children}} C(child_i) \cdot P_{hit}(child_i) \\ C(leaf) &= T_2 \cdot N_{children} \\ P_{hit}(node) &= \frac{area(node)}{area(parent)} \end{aligned} \tag{1}$$

where T_1 and T_2 are control parameters representing the cost of intersecting a bounding box and that of intersecting a primitive respectively (both set to 1 in our implementation). We believe this metric to be a fairly representative measure of the general ray tracing performance of a given bounding volume hierarchy: in fact, while it is only a heuristic, it provides a very good indicator of the amount of spatial overlap between all subtrees in a hierarchy, and at the same time it is not unnecessarily coupled with any of the many specific ray tracing algorithms with different performance characteristics available today (e.g. while-while or if-if GPU traversal [AL09], CPU or GPU packet tracing algorithms [WBS07, GPSS07], ray sorting methods [GL10] just to name a few). In all our experiments the SAH cost of LBVH and HLBVH trees is at most 135% higher than that obtained by a full SAH sweep build. Furthermore, our SAH-optimized HLBVH algorithm reduces the cost by an additional 10-20%.

5. Summary and Discussion

We have presented two novel BVH building algorithms. The first is HLBVH, a new hierarchical LBVH formulation run-

Figure 6: *Our static test scenes. From left to right: Armadillo, Stanford Dragon, Happy Buddha, Turbine Blade and Hair Ball.*

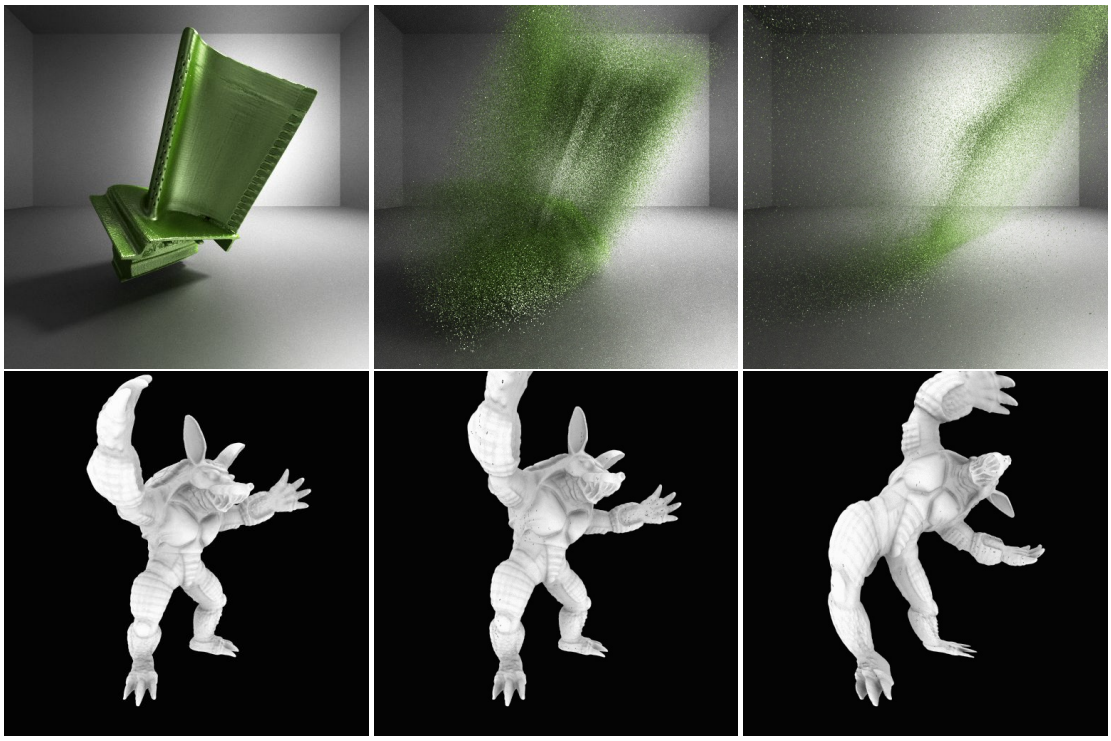

Figure 7: *Exploding Turbine Blade and Bending Armadillo. Frames 0, 3 and 15. The Armadillo is subject to a simple deformation, while the Turbine Blade is blasted into 1.7 million particles (its constituent triangles) moving with random velocities.*

ning at 2-3x the speed of the original algorithm, consuming 10-20x less bandwidth and producing a considerably more compact tree memory layout (2-4x). This algorithm is capable of building the spatial index for models with one million triangles in less than 35ms on consumer GPUs available today, and less than 25ms on early samples of the next generation of NVIDIA GPUs. This is the fastest BVH builder available to date. We believe this algorithm will allow future games to use real time ray tracing on dynamic geometry and possibly find applications in other fields, such as physics simulation.

Our second contribution is a SAH-optimized HLBVH building procedure, utilizing the greedy SAH sweep build algorithm to build the top level tree of our HLBVH hierarchies. This is the fastest SAH-based builder available to date, and we believe it will prove superior to other hybrid algorithms which use the surface area heuristic to optimize the bottom levels of the hierarchies only.

Both algorithms rely on a combination of the CSD scheme introduced in [GL10] and a hierarchical problem decomposition to heavily reduce the amount of work needed for the top level Morton curve ordering and SAH sweep build respectively. The hierarchical decomposition enables us to extract and exploit coarse spatial coherence available in the input meshes and greatly accelerates BVH construction on dynamic geometry.

Both algorithms have been implemented in CUDA using

Thrust, a collection of general purpose parallel primitives, bypassing the need to write CUDA code directly except for the case of the odd-even intra-CTA sorting kernel. The resulting source code is simple and terse, showing that high performance parallel computing can be made easy by using the proper abstractions.

5.1. Future Work

We plan to optimize all our algorithms for the NVIDIA GF100 architecture, making use of its new instructions, larger shared memory and new cache hierarchy to reduce the amount of work and bandwidth to external memory. In particular, we believe that our top level SAH building procedure contains many opportunities for further optimization. Further on, we plan to integrate more hierarchical steps for larger models and to explore lazy builds for on-demand BVH construction.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High-Performance Graphics 2009* (2009). 1, 7

[Bia69] BIALLY T.: Space-filling curves: Their generation and their application to bandwidth reduction. In *IEEE Transactions on Information Theory* (1969), vol. 15, 6, pp. 658 – 664. 2

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Fast gpu ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of Graphics Interface 2006* (Toronto, Ont., Canada, Canada, 2006), Canadian Information Processing Society, pp. 203–209. 1

[Gar08] GARANZHA K.: Efficient clustered bvh update algorithm for highly-dynamic models. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing* (2008), pp. 123 – 130. 2

[GGKM06] GOVINDARAJU N. K., GRAY J., KUMAR R., MANOCHA D.: Gputerasort: High performance graphics coprocessor sorting for large database management. In *Proceedings of ACM SIGMOD* (2006), pp. 325–336. 2

[GL10] GARANZHA K., LOOP C.: Fast ray sorting and breadth-first packet traversal for gpu ray tracing. In *Eurographics 2010 State ,of the Art Reports* (2010), vol. 29. 1, 2, 7, 8

[GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007* (Sept. 2007), pp. 113–118. 7

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7, 5* (1987), 14–20. 6, 7

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. 1, 6

[IWP07] 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). 2

[Knu68] KNUTH D. E.: *The Art of Computer Programming*, vol. 3, Sorting and Searching. Addison-Wesley, 1968, section 5.2.2. 3

[LeYM06] LAUTERBACH C., EUI YOON S., MANOCHA D.: Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 39–45. 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast bvh construction on gpus. *Comput. Graph. Forum 28*, 2 (2009), 375–384. 1, 2, 6

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable parallel programming with cuda. *ACM Queue 6*, 2 (2008), 40–53. 2

[PGSS06] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 89–94. 1

[SHG09] SATISH N., HARRIS M., GARLAND M.: Designing efficient sorting algorithms for manycore gpus. In *IPDPS* (2009), pp. 1–10. 2

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan primitives for gpu computing. In *Graphics Hardware 2007* (Aug. 2007), ACM, pp. 97–106. 2

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN E.: Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum 26, 3* (2007), 395–404. 1

[Wal07] WALD I.: On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (2007). 1, 2, 6

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (2007). 1, 2, 6, 7

[WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In *In Rendering Techniques 2006 - Proceedings of the 17th Eurographics Symposium on Rendering* (2006), pp. 139–149. 1, 2

[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. *ACM Trans. Graph. 27*, 5 (2008), 1–11. 1