GPU Ray Tracing using Irregular Grids

Preprint version

Arsène Pérard-Gayot[1], Javor Kalojanov[1], Philipp Slusallek[1,2]
[1]Saarland University & Intel VCI, Germany
[2]German Research Center for Artificial Intelligence, Germany

Abstract
We present a spatial index structure to accelerate ray tracing on GPUs. It is a flat, non-hierarchical spatial subdivision of the scene into axis aligned cells of varying size. In order to construct it, we first nest an octree into each cell of a uniform grid. We then apply two optimization passes to increase ray traversal performance: First, we reduce the expected cost for ray traversal by merging cells together. This adapts the structure to complex primitive distributions, solving the "teapot in a stadium" problem. Second, we decouple the cell boundaries used during traversal for rays entering and exiting a given cell. This allows us to extend the exiting boundaries over adjacent cells that are either empty or do not contain additional primitives. Now, exiting rays can skip empty space and avoid repeating intersection tests. Finally, we demonstrate that in addition to the fast ray traversal performance, the structure can be rebuilt efficiently in parallel, allowing for ray tracing dynamic scenes.

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

1. Introduction

Today's high performance ray tracers rely on high quality acceleration structures for efficient ray traversal and fast rendering times. A number of types of these structures have been explored and adapted to various application scenarios, considering aspects such as hardware architecture, type of rendering applications (e.g. offline, interactive), or type of scenes (e.g. static, deformable, dynamic). For example, state of the art ray traversal performance for static scenes on GPUs [AL09] can be obtained using spatial bounding volume hierarchies (SBVHs) [SFD09, PGDS09]. For dynamic scenes, different types of bounding volume hierarchies (BVHs) like LBVHs [LGS*09] and HLBVHs [PL10] are preferred due to their fast build times, which allow per-frame rebuild of the structure even for very large scenes.

In this paper, we demonstrate that efficient ray traversal on modern GPUs is also possible with a new type of acceleration structure, which combines characteristics of hierarchical grids and bounding volume hierarchies. Similar to the standard construction algorithms for LBVHs and two-level grids, we initially subdivide the scene into small cells. To make this initial stage of the construction efficient, we use a two-level approach. We build a coarse uniform grid for the top level and then subdivide each of its cells independently and adaptively, based on the local primitive density. The resulting structure was first proposed by Jevans and Wyvill [JW89]. Similar to previous work on two-level grids [KBS11], this acceleration structure is very fast to build, but offers limited traversal performance—the main problem our work addresses.

We provide a two-stage algorithm to transform the initial acceleration structure into an *irregular grid*—a non-hierarchical spatial subdivision into axis aligned bounding boxes of varying size. We start by iteratively merging the leaf cells of the two-level hierarchy, thereby optimizing the quality of the structure w.r.t. to the Surface Area Heuristic (SAH) [MB90, Hav01]. This adapts the spatial subdivision to the distribution of primitives in the scene, without constructing a deep hierarchy. We then further reduce the amount of work during ray tracing by decoupling the cell boundaries used for entering and exiting rays. This makes it possible to use different, extended cell boundaries for outgoing rays. These new exiting cell boundaries can be placed beyond neighbors that contain only a subset of the primitives inside it. Rays that have entered a cell can then skip overlapped neighbors and thus avoid unnecessary traversal steps and intersection tests.

Our evaluation indicates that, when optimized for static geometry, our acceleration structure can often provide traversal performance superior to standard SBVHs. Without modifying the build algorithm, we can trade off quality for faster build times, enabling ray-tracing dynamic scenes with complete per-frame rebuild of the structure. In this scenario, our method achieves interactive performance without substantially reducing the ray budget per frame. The trade-off between build times and acceleration structure quality can be manually or automatically controlled through the maximum grid density—an intuitive parameter used for uniform and two-level grid construction. This enables support for both static and dynamic scenes with the same construction algorithm.

Our approach can also be viewed as an efficient strategy for empty space skipping in uniform or hierarchical grids. The regularity of these acceleration structures forces unnecessary subdivision of empty regions, which usually creates a memory latency bottleneck during traversal. Both the merge step and expansion phase of our construction algorithm address this issue by reducing the number of traversal steps and intersection tests. As a result, the traversal performance is no longer memory latency limited for coherent rays, as opposed to uniform and two-level grids.

2. Background

Acceleration structures for ray tracing have been researched intensively and played an important role for the increased popularity of rendering solutions based on ray tracing. The purpose of an acceleration structure (a.k.a. spatial index structure) is to spatially organize (sort) the geometric primitives in a scene and thus eliminate most of them as intersection candidates for a given ray. To this end, each ray is traversed within the structure, quickly identifying the subset of intersected cells (or nodes). Only the primitives contained inside these cells are tested for intersection and all the others can be ignored. Hence, a good acceleration structure for ray tracing should eliminate a large number of intersection candidates, while at the same time providing fast traversal and, in some applications, fast build times.

Some spatial index structures variants have received special attention. Most notably, hierarchical structures such as kd-trees [RSH05, WH06, PGSS06, HMS06] and bounding volume hierarchies [WMG*07, Tsa09, PGDS09, SFD09, AL09] are able to provide the best traversal performance on modern hardware. They offer computationally inexpensive traversal and, being hierarchical subdivisions, can adapt to the local size and density of primitives. More importantly, the expected cost of traversing a random ray trough the structure is minimized using the Surface Area Heuristic [MB90, Hav01] when subdividing nodes, greatly improving performance compared to a naïvely constructed tree.

A common drawback of high quality acceleration structures optimized according to the SAH are the long build times, which limit their use to ray tracing static scenes. To address this, researchers have tried various build algorithms [WMG*07, Wal10, GPBG11, KA13] and variations in the structure [WMS06, WK06, LGS*09, PL10] in order to trade off some of the traversal performance for faster build or update times. It is important to note that these previous approaches require fundamentally different construction algorithms to achieve fast build times. Our technique differs as it allows for controlling the quality of the structure with two user-defined parameters.

A different approach to optimizing the trade off between build and traversal times is to consider regular (non-SAH) structures with fast, parallel construction such as uniform grids [WIK*06, IWRP06, KS09], hierarchical grids [KBS11], or LBVHs [LGS*09, PL10]. These methods rely on a more regular space (or primitive) subdivision, which limits traversal performance, especially in scenes with complex primitive distributions (the "teapot in a stadium problem"). However, their fast construction makes them practical for dynamic scenes with rapidly changing geometry. Similar to the core idea of LBVHs we start with a dense regular subdivision of the scene into cells, which are consecutively merged into larger cells, which offer better traversal performance.

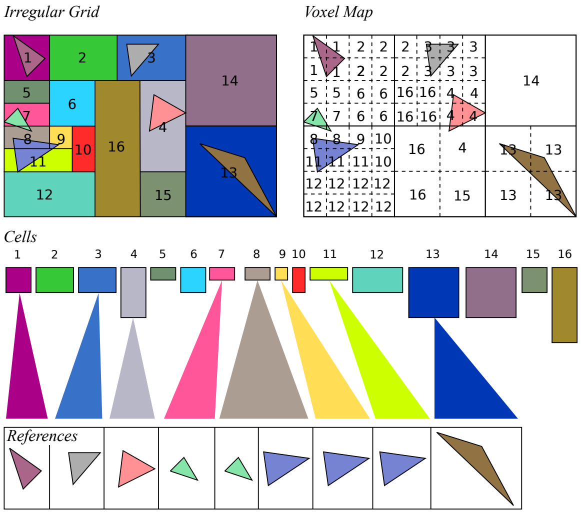


Figure 1: *Top: Our structure subdivides space into irregular cells (left), and we store a two-level voxel map to link voxels to their respective cells (right). Bottom two rows: the cells and the corresponding ranges of references.*

Empty space skipping methods address the inability of regular spatial subdivisions, e.g. uniform grids, to adapt to the local primitive density. Some solutions [CS94, EI07] are based on distance fields: The number of adjacent empty cells that can be skipped with a single traversal step is stored alongside each cell. The merge and expansion steps of our method can be seen as a generalization of the approach by Devillers [Dev89], which uses bounding boxes to enclose large regions of empty space. The difference is that we do not restrict macro regions to empty space, but instead base our merging strategy on the SAH and apply it to a fine-grained two-level initial subdivision, which results in several times faster traversal times.

Our data structure and traversal algorithm share some similarities with rope trees [PGSS07]. Since advancing to the next bounding box along a given ray is non-trivial, we store a map from fine resolution voxels to the cells of the structure. This resembles the idea of Popov et al. [PGSS07] to store pointers to the neighbors inside kd-tree nodes for stackless traversal.

3. Data Structure

Our acceleration structure (see Figure 1) is an irregular spatial subdivision into axis-aligned cells. These cells contain indices into an array of primitive references. Each cell has to be aligned on a *virtual base grid*, but the size of individual cells is otherwise unrestricted. This allows for adapting the structure to non-uniform primitive distributions (i.e. solving the "teapot in a stadium" problem), and makes empty space skipping efficient. On the other hand, this property complicates traversal, since neighboring cells are no longer located trivially with respect to each other. We solve this by computing and storing a map between base grid voxels and the cells they belong to. As illustrated in Figure 1, an irregular grid is composed of:

- A *voxel map* $E : \mathbb{N}^3 \longrightarrow \mathbb{N}$, which links the virtual base grid voxels to the cells of the structure,
- An array of *cells*, containing for each cell the associated bounding box and range of references,
- An array of *references*, pointing to the array of primitives.

Since using a uniform grid would require to store N^3 values for the voxel map, we use a substantially more efficient two-level hierarchical grid. This allows a fast, stackless traversal and an efficient construction. To summarize, our structure is the combination of a set of axis-aligned cells and a two-level grid—the voxel map—that contains the connectivity information.

4. Parallel Construction

Our construction algorithm consists of three phases: an initialization, a merge step, and a cell expansion step. The initialization step subdivides the scene into small voxels. This is done using a two-level strategy that allows for adapting the voxel resolution to the local primitive density. We then merge cells together in order to optimize the overall SAH cost of the structure (see Figure 2). In the expansion step, cell bounding boxes can be further extended in order to skip some cells during traversal (see Figure 3).

4.1. Initialization

In the initialization step, we generate the initial cells of the structure, the voxel map, and the array of references. Because the quality of the acceleration structure depends on the resolution of the voxelization, it is critical that this initialization subdivides the scene adaptively.

Building the cells: First, we build a *coarse* top-level uniform grid. The resolution of this grid is computed according to a commonly used heuristic [CWVB83]:

$$R_x = d_x \sqrt[3]{\frac{\lambda_1 N}{V}},\ R_y = d_y \sqrt[3]{\frac{\lambda_1 N}{V}},\ R_z = d_z \sqrt[3]{\frac{\lambda_1 N}{V}} \qquad (1)$$

The variables λ_1, V, N and d correspond to the top-level density, volume, number of primitives, and bounding box of the scene. Once the top-level grid is constructed, we build an octree in each of its cells. The depth of one octree is derived from the same formula, by using a different density λ_2 common to all cells and substituting V, N and d by the volume, number of primitives, and bounding box of the corresponding cell.

The depth D for a given octree is then simply the resulting resolution rounded to the next power of two:

$$D = \lceil log_2(max(R_x, R_y, R_z)) \rceil$$

The granularity of our two-level subdivision is hence completely controlled by λ_1 and λ_2, the top- and second-level resolutions, exactly as in [KBS11]. Likewise, both levels use sorting to find which primitives lie in each cell (each level using exactly one sort pass, independently of the top-level resolution or the octree depths). This allows for fast build times, and more importantly, an even work distribution regardless of the primitive distribution. However, the octree construction differs from the grid construction in the way the primitive references are generated. In the octree construction, we encode a primitive reference as a triplet $(\textit{top-level cell ID}, \textit{octree cell ID}, \textit{primitive ID})$, instead of a pair as in [KBS11]. The references are split in several passes: Each pass splits every reference until the maximum depth for the corresponding top-level cell is reached. Splitting a reference consists in testing the primitive against the 8 children of the octree cell and emitting a new reference only if there is an intersection. Therefore, we need to perform as many passes as the maximum octree depth to emit all the references, instead of only two for a two-level grid. In large scenes, this additional cost is compensated by the reduced amount of primitive-cell intersection tests.

The choice of an octree for the second level is motivated by the fact that Equation 1 favors subdivision such that each top-level cell is almost a cube. Consequently, in a given cell, the second-level resolution computed from Equation 1 is usually equal on each axis (in our test scenes, this is true for more than 95% of the non-empty cells).

The next step is to compute the resolution of the virtual base grid: We obtain it naturally by multiplying the top-level resolution by the maximum octree resolution. Each cell is, by design of our construction algorithm, aligned on this virtual grid. We would like to stress that this grid is *virtual*: We do not store its cells, but we use it to *index* into the voxel map.

Building the voxel map: Since the voxel map transforms a voxel position on the virtual base grid (which can reach extremely high resolutions) to an index in the array of cells, it is impractical to store it as a uniform grid. We therefore use a two-level grid with the same top- and second-level resolutions as the one used for the cells:

- For each top-level voxel: 4 bits for the logarithm of the resolution D and 28 bits for the index to the first cell of the second level.
- For each second-level voxel: the corresponding index into the cell array.

These two levels are stored linearly into one array. Using only 4 bits for the logarithm of the resolution allows us to reach a sub-level resolution of $2^{2^4-1} = 32768$. Previous works use 24 bits and restrict the maximum resolution in each dimension to 255 [KBS11].

The resulting structure is essentially an adaptive voxelization indexed by a two-level grid (see Figure 2, middle), and can already be used for traversal. Its performance can be further improved by merging redundant cells, which is the next step of our construction algorithm.

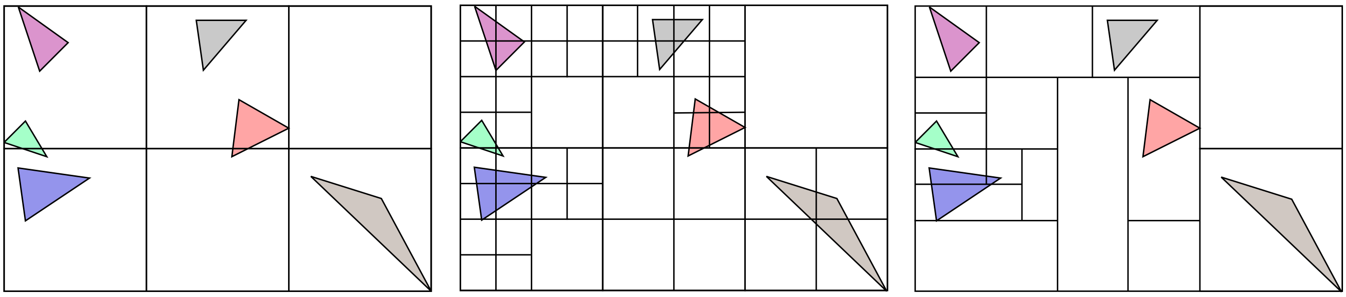


Figure 2: *Construction stages without cell expansion. We initially construct a* coarse *uniform top-level grid (left). Then, the cells are subdivided into octrees (middle). Finally, the cells from both the top and second level are merged in order to minimize the SAH cost (right).*

4.2. Cell Merging

The second step of our build algorithm consists in merging adjacent cells according to the SAH in a way that reduces the expected traversal cost.

The expected cost for traversing a cell equals the probability of a ray hitting it times the cost for testing the triangles (or primitives) it contains. We can compute the costs for a pair of cells if they are adjacent and if the union of their bounding boxes is itself a box. A simple criterion to decide when to merge cells is to test if the sum of the cost of two cells taken *individually* is greater than the cost of the resulting merged cell. By iteratively merging cells according to this criterion, we can greedily improve the overall traversal cost of the structure.

More formally: Let a cell c be defined by the set of triangles T it contains and its bounding box B, and let $\mathcal{SA}(\cdot)$ be the surface area of $\cdot$. Furthermore, let C_t be the cost for traversing an empty cell of the acceleration structure, and C_i the cost for testing a triangle for intersection. The resulting cost functions C_{cell} for one individual cell and C_{merge} for two merged cells are then proportional to:

$$C_{cell}(c) \propto (C_i|T| + C_t)\,\mathcal{SA}(B)$$
$$C_{merge}(c_1, c_2) \propto (C_i|T_1 \cup T_2| + C_t)\,\mathcal{SA}(B_1 \cup B_2)$$

We determined experimentally that setting $C_t = 1$ and $C_i = 1$ gives the best performance with our traversal algorithm, and we use these values for our tests. Also, for simplicity, we omit a division by the surface area of the bounding box of the scene in both cases.

During the whole construction, in order to merge cells and their contained primitives efficiently, we keep the primitive references sorted by cell (globally), and, within each cell, by triangle reference. This allows for computing the union in one linear pass similar to the merge step in merge sort, except that we discard double occurrences.

We perform merge passes along a single axis (x, y or z), which we alternate in a round-robin fashion. Each pass is executed as follows:

1. For each cell, in parallel, if a merge is possible with the neighbor on the axis, we compute the cost of merging with it. If the sum of the individual costs of the cell and its neighbor is greater than the cost of merging them together, then the cell becomes a *merge candidate*. Merge candidates can form chains, which we call *merge chains*.
2. For every merge chain in parallel, we mark the cells at odd positions as *residue* (we only perform pairwise merges). The length of the chain can therefore be reduced by half in each pass.
3. For each cell, in parallel,
   - If the cell is a merge candidate and is not residual, a new cell containing the merged result is created and the references to the primitives are updated.
   - If the cell is not a merge candidate, it is kept without change.
   - If the cell is residual, it is naturally removed.
4. The entries of the voxel map pointing to removed cells are updated to their respective new (and larger) cells.

This merging procedure is repeated (alternating the three axes) until the number of merged cells is small enough. If N_{before} is the number of cells before merging and N_{after} is the number of cells after merging, the termination criterion is:

$$N_{after} \geq \alpha N_{before} \text{ with } \alpha \in [0, 1]$$

When α is 1, the merging procedure will loop until no more merge is possible, or none of the remaining possibilities improves the SAH cost of the structure. When α is 0, the merge procedure will only be executed once. Our experiments show that setting $\alpha = 0.995$ reduces the number of iterations while providing close to optimal performance (w.r.t. $\alpha = 1$). The number of merged cells varied from 15 to 70%, depending mainly on the scene complexity and amount of empty space.

4.3. Cell Expansion

After the merge step, empty regions of the scene become cheap to traverse, and, similarly to most other acceleration structures, the remaining expensive areas in a viewport are at object boundaries. These parts of the scene are difficult to handle by simply adjusting the density of the spatial subdivision: making the cells smaller reduces the amount of ray-primitive intersection tests for rays that travel close to the object boundaries without hitting the object. On the other hand, the smaller the cells, the more traversal steps are necessary for rays barely missing the object.

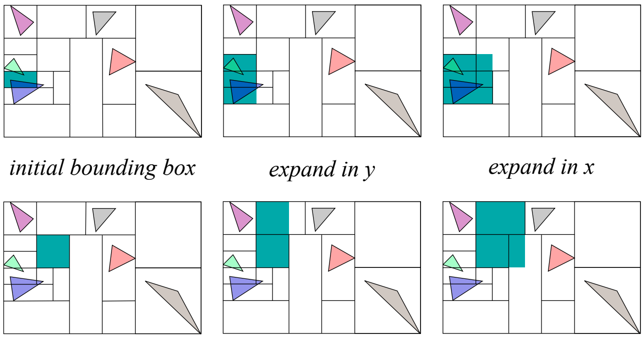


Figure 3: *Example expansion of cell bounding boxes. We expand both empty (bottom figures) and non-empty (top figures) cells, extending their boundaries along each dimension in turn, one cell at a time, if the set of contained triangles remains unchanged.*

To address this, we take advantage of the fact that entering and exiting a cell are two decoupled operations: During traversal, we compute the cell to *enter* by performing a lookup in the voxel map at the position where we left the previous cell. However, we compute where a given ray *exits* a cell by intersecting it with the bounding box of the cell. Hence, increasing the size of the bounding box of a cell will allow the ray to exit farther away, skipping neighboring cells and reducing the number of traversal steps.

In order to preserve traversal correctness, we only expand a cell if all its neighbors on a given side contain a subset of the primitives already contained in this cell (see Figure 3). The amount by which the bounding box is expanded is taken from the smallest neighboring cell. A single expansion pass consists in repeating this process for all cells once along x, then y and finally z.

Expansions passes can be performed repeatedly, taking into account neighbors that are farther away. In Figure 3 for example, the cell in the top row will benefit from a second expansion pass, which would increase its size in x and y. Overall, each successful expansion pass can extend the boundaries of a cell over its closest neighbors, and allow an exiting ray to skip up to three traversal steps (one per dimension). However, we observed that the improvement in traversal performance gets negligible after only a few iterations. In our tests, doing more than 3 expansion passes yielded less than 1% in performance compared to only doing 3. Therefore, we opted for doing 3 expansion passes after merging when rendering static geometry and a single pass for dynamic scenes.

After expansion based on this criterion, the traversal performance increases by 5-20% for our test scenes. One part of the performance increase comes from expanding cells over their empty neighbors, which reduces the number of traversal steps for empty space. The remaining benefit is due to extending the boundaries over non-empty neighbors, which also reduces the number of redundant ray-primitive intersection tests.

Note that cell expansion makes discovering neighbors a nontrivial task, which is why we only perform it after merging cells. Merging cells beforehand also makes cell expansion faster by reducing the number of cells to expand.

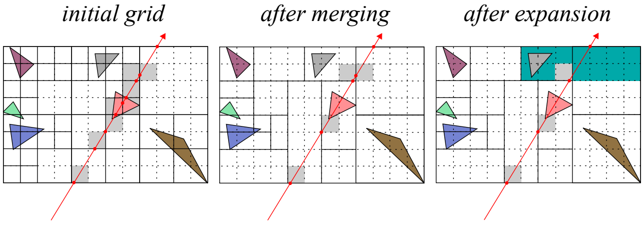


Figure 4: *Traversal steps using the initial grid, cell merging, and cell merging with expansion. At each intersection with a cell (red dot) the next virtual base grid voxel (gray) is computed and the corresponding cell is looked up.*

5. Traversal

The core idea behind the traversal algorithm is the same as the standard 3D-DDA algorithm [AW87], except that we recompute the increment along the ray at each step. The virtual base grid—which, as explained in Section 4, is *not* stored—serves as a support for the traversal: The current voxel position moves on this grid and the voxel map is used to link this position to the current cell.

The algorithm can be summarized as follows:

1. We locate the origin of the ray on the virtual base grid.
2. We then repeatedly:
    a. Look up the cell index at the current virtual grid voxel position using the voxel map. This only requires two dependent memory lookups (one for the top level, and one for the second level).
    b. Intersect the primitives contained in the cell, if any.
    c. Intersect the bounding box of the cell to compute the increment along the ray. In practice, we only need the exit distance, so it is enough to intersect only the three farthest bounding box planes with respect to the ray direction.
    d. Find the next virtual base grid position based on the exit distance.
    e. If the next voxel position is outside the grid or if an intersection was found, the traversal stops.

As discussed in the following section, the traversal performance we obtain using our acceleration structure is comparable to SBVHs and superior to two-level grids. The performance difference with two-level grids is especially large for rays traveling long distances trough empty space or traversing areas close to complex objects. Additionally, our measurements show that the traversal of coherent rays inside an irregular grid is not memory latency limited as is the case with two-level grids and to some extent with SBVHs [Gut14]. This property is one of the main factors contributing to the traversal performance on graphics hardware.

Floating point precision is a common issue with grid traversal algorithms, and ours is no exception. In order to prevent the next voxel position to fall in a cell that has already been traversed, we keep the current voxel position from the previous step, and we make sure that the next voxel position is further along the ray. In practice, this only amounts to a *max* operation between the next voxel coordinate and the previous one along axes on which the ray direction

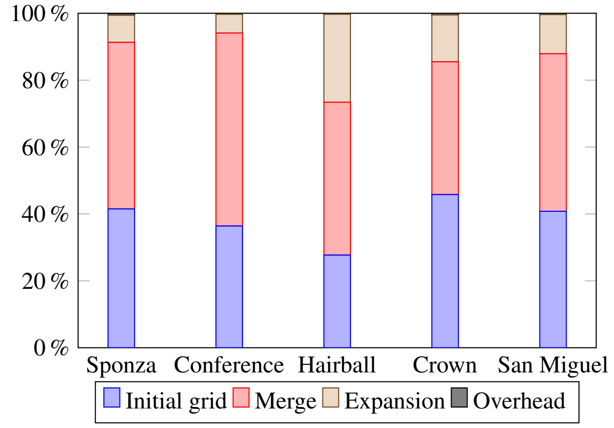


Figure 5: *Relative cost of each step in the total construction time. The time spent allocating memory and copying data to the GPU is considered as overhead.*

is positive (and similarly, a *min* operation for axes on which the ray direction is negative).

Sponza	N_T	N_I	MRays/s
SBVH	35 (+0%)	6 (+0%)	409 (+0%)
Initial grid	16 (-54%)	12 (+100%)	454 (+11%)
+ Merge	12 (-66%)	12 (+100%)	511 (+25%)
+ Merge & Exp.	7 (-80%)	11 (+83%)	653 (+60%)

Crown	N_T	N_I	MRays/s
SBVH	39 (+0%)	11 (+0%)	232 (+0%)
Initial grid	15 (-61%)	18 (+62%)	238 (+2%)
+ Merge	13 (-66%)	18 (+62%)	266 (+14%)
+ Merge & Exp.	11 (-72%)	17 (+59%)	296 (+28%)

San Miguel	N_T	N_I	MRays/s
SBVH	55 (+0%)	8 (+0%)	227 (+0%)
Initial grid	28 (-49%)	13 (+62%)	206 (-9%)
+ Merge	15 (-72%)	13 (+62%)	241 (+6%)
+ Merge & Exp.	10 (-81%)	13 (+62%)	291 (+28%)

Table 1: *Number of traversal steps (N_T) and triangle intersections (N_I) per primary ray on average for an SBVH compared to our structure after each construction phase (with $\lambda_1 = 0.12, \lambda_2 = 2.4$).*

6. Evaluation

We implemented our method in CUDA 7.0 [NVI07] and evaluated the performance comparing it to up-to-date ray tracers using SBVHs [ALK12] and two-level grids [KBS11]. The first reference offers state-of-the-art performance for ray tracing static scenes defined as a triangle soup. For dynamic scenes with per-frame rebuild of the acceleration structure, two-level grids and LBVHs offer similar build times and ray traversal speeds [KBS11]. We compare against two-level grids since they are closely related to our acceleration structure. We report test results on a GeForce Titan X (Maxwell), but the relative performance w.r.t. alternative methods was very similar on a GeForce GTX 970. We did not use third party implementations in our code, except for the scan and radix sort provided in the CUB [NVI13] library. Please note that we compare performance by tracing the exact same set of rays, and use the Möller-Trumbore ray-triangle intersector [MT97] with each ray tracing implementation.

6.1. Construction

In Figure 5, we give a breakdown of the time spent during each stage of our build algorithm. Because of the reduced amount of grid cells after the merge step, cell expansion typically takes around 20% of the construction time. While each of the two methods (merging and expansion) can provide substantial improvement in the traversal performance (see Table 1), we do not recommend using expansion only. Without the merging step, the expansion would have to process too many cells, resulting in very slow build times and smaller quality improvement.

Memory footprint: Assuming unconstrained build times, the main limiting factor for the quality of the acceleration structure is the memory necessary construct a dense voxelization. The finer the initial subdivision, the more ways to merge the voxels into cells of the irregular grid. This makes the search space for an optimal subdivision larger and enables the construction of a structure with a better expected traversal cost (see Table 2). Our build algorithm uses more memory than the final structure (between 2.5× and 3×), because of its massively parallel nature (the merge operation, for instance, does not work in-place). Note that such memory overheads are also present in the typical construction algorithms for SBVHs and two-level grids. In practice, however, this limitation did not prevent us from using optimal build parameters for all the scenes used in this paper. If memory is really a concern, one obvious solution is to merge the grid in several passes, or decrease the densities.

We show the impact of changing the top-level and second-level densities on build times and primary ray traversal time for the crown scene in Figure 6. The results for the remaining scenes are similar and indicate that the second-level density correlates with the traversal performance, and that choosing a top-level density that is lower than the second-level density reduces the build times without affecting the traversal performance significantly. Higher build times also seem to correlate with faster traversal, until the maximum performance is reached.

Even though the quality of the irregular grid is influenced by the choice of initial grid densities (λ_1, λ_2), our structure is less sensitive to the choice of parameters compared to a two-level grid (see Table 2). While the two-level grid construction terminates after the initial subdivision, our method only subdivides the scene to form a starting guess. In the case of a two-level grid, increasing the grid density will ultimately cause performance degradation, due to the increased number of empty cells. Thanks to the merging and expansion passes, we can locally optimize our structure and, for example, reduce traversal overhead in densely subdivided empty regions of the scene.

		Build times (s)			Traversal (MRay/s)		Memory (MB)		
Scene	#Tris	Ours	2L Grid	SBVH	Ours	2L Grid	Ours	2L Grid	SBVH
Sponza	262K	[0.012, 0.026]	0.007	9	[201, 653]	145	[4, 23]	24	20
Conference	283K	[0.016, 0.022]	0.007	7	[182, 597]	77	[4, 12]	27	21
Hairball	2.9M	[0.349, 0.893]	0.177	336	[79, 148]	37	[138, 779]	668	413
Crown	3.5M	[0.066, 0.203]	0.039	44	[115, 296]	74	[53, 278]	182	241
San Miguel	7.9M	[0.162, 0.492]	0.071	95	[97, 291]	63	[107, 565]	323	510

Table 2: *Build times, memory, and traversal statistics for grid densities ranging in:* $\lambda_1 \in [0.012, 0.12], \lambda_2 \in [0.24, 2.4]$. *For the two-level grid, we select the pair* (λ_1, λ_2) *that yields the best traversal performance. The traversal algorithms are tested using primary rays with the viewpoints in Figure 7. The SBVH build times are measured on a single CPU core.*

			Primary		AO		Random	
Scene	#Tris	Fig. 7	SBVH	Ours	SBVH	Ours	SBVH	Ours
Sponza	262K	(a)	409	653 (+60%)	270	386 (+43%)	166	274 (+65%)
		(f)	265	473 (+78%)	187	234 (+25%)		
Conference	283K	(b)	583	597 (+2%)	303	332 (+10%)	295	312 (+6%)
		(g)	523	526 (+1%)	326	338 (+4%)		
Hairball	2.9M	(c)	100	148 (+48%)	53	69 (+30%)	19	26 (+37%)
		(h)	79	93 (+18%)	63	61 (-3%)		
Crown	3.5M	(d)	232	296 (+28%)	108	120 (+11%)	221	238 (+8%)
		(i)	181	191 (+6%)	112	125 (+12%)		
San Miguel	7.9M	(e)	227	291 (+28%)	119	119 (+0%)	119	160 (+34%)
		(j)	157	180 (+15%)	125	115 (-8%)		

Table 3: *Ray traversal performance in Mrays/s for static scenes on a GeForce Titan X (Maxwell). The top-level density* $\lambda_1 = 0.12$ *and leaf-level density* $\lambda_2 = 2.4$ *of the initial grid are identical for all scenes.*

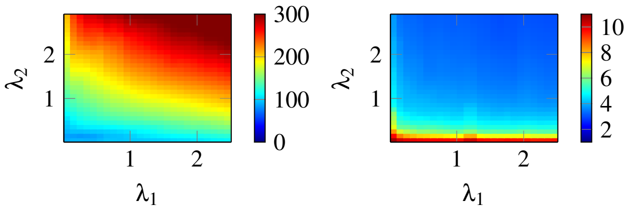


Figure 6: *Build times (left) and primary ray traversal times for a viewport of 1024×1024 (right), in milliseconds, for the crown scene, depending on the top-level (λ_1) and second-level (λ_2) densities.*

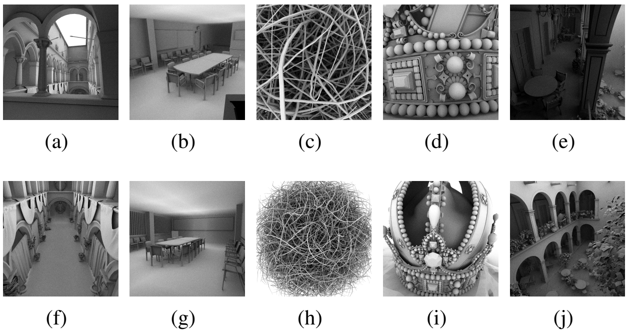
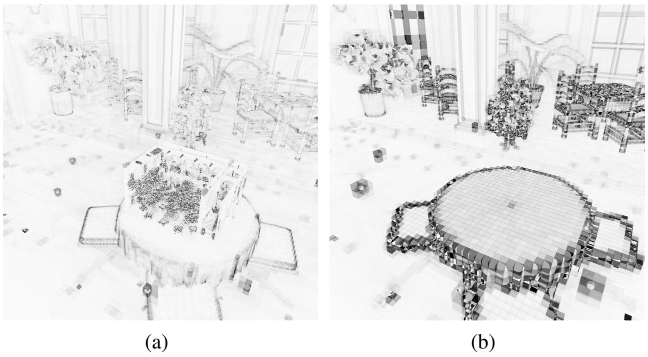


Figure 7: *View points used to benchmark the traversal algorithm on static scenes.*

6.2. Static Scenes

We compare the performance of our traversal algorithm for static scenes against the SBVH ray traversal implementation of Aila et al. [ALK12] in Table 3. We would like to thank the authors for providing the code for their ray tracer, which enables a fair comparison of traversal performance. A GPL'ed version of our implementation is also available as supplemental material. The approach by Aila et al. is used as a baseline for traversal performance in multiple related works (e.g. [KA13, ÁSK14, Gut14]). Examining the relative increase or decrease of performance reported in these works indicate that our method should perform similarly.



	Mem.	Fig. 8(a)	Fig. 8(b)	Random
SBVH	1.0 GB	200 (+0%)	147 (+0%)	122 (+0%)
Ours (0.12, 2.4)	0.84 GB	107 (-46%)	144 (-2%)	176 (+44%)
Ours (0.24, 4.8)	1.5 GB	186 (-7%)	204 (+38%)	167 (+37%)

Figure 8: *San Miguel inside San Miguel. The table compares the performance for primary and random rays (in Mrays/s) for an SBVH and an irregular grid using different initial densities. The two images represent the amount of computation for traversal and intersection with* $(\lambda_1, \lambda_2) = (0.12, 2.4)$. *The second viewpoint is inside the embedded scene.*

The scenes have been chosen to highlight the performance of the structure for different polygon distributions and various scene sizes. Although not the largest, the Hairball scene is the most difficult to traverse using both an irregular grid or an SBVH. This is caused by the very irregular primitive distribution, especially in the center of the hairball, where a lot of the geometry is intertwined.

The relative performance compared to the SBVH varies with the viewpoint, probably due to the different nature of the two acceleration structures: We compare a binary tree and a flat, non-hierarchical structure. We therefore report results on two representative (non-trivial) viewpoints for each scene, and present both favorable and non-favorable situations for our method. Overall, our impression is that the irregular grid offers faster traversal than the SBVH for small scenes like Sponza, and for coherent rays regardless of the scene size. Even for scenes that remain difficult for our method such as the Conference, Hairball, or San Miguel, and for incoherent sets of rays, we achieve comparable and often better performance.

Parameter selection: Our goal is to estimate the traversal performance for static geometry that can be obtained using our acceleration structure. Since we compare against an approach using an SBVH, we select the highest grid densities such that the memory footprint of both structures are similar. It turns out that the densities found are close for most scenes, hence we use the same top-level density $\lambda_1 = 0.12$ and second-level density $\lambda_2 = 2.4$ for all tests if not mentioned otherwise.

The teapot in a stadium: Non-hierarchical spatial subdivision structures such as uniform grids are considered inferior to hierarchical alternatives like kd-trees and BVHs for ray tracing scenes with non-uniform primitive distributions. This is referred to as the "teapot in a stadium problem". We constructed an artificial test scenario in order to analyze how well the irregular grid presented here can handle extreme primitive distributions. We inserted a scaled down copy of the San Miguel scene inside itself and tested traversal performance for our method and Aila's ray tracer (see Figure 8). Using the default grid densities, our method compares favorably for incoherent rays but we are not able to match the performance for primary rays for specific viewpoints (Figure 8(a) was the worst viewpoint for our method). This is easily solved by increasing the densities λ_1 and λ_2. Making the initial subdivision twice as dense results in 50% increase in the required storage for the structure and allows to match the traversal performance for any type of ray throughout the scene. If necessary, one can eliminate the memory overhead due to the higher resolution by using a voxel map with more than two levels.

Comparison with Macro Regions: In order to provide further insight into how the different stages of our construction algorithm influence the traversal performance, we evaluated the merge and expansion on uniform grids (see Table 5). Efficient algorithms for empty space skipping in uniform grids have been of interest previously, and we implemented the macro region construction algorithm by Devillers [Dev89] for a comparison baseline. This related approach could be viewed as a simplification of ours, where the expansion and merge phase are only applied for the empty cells of a uniform grid.

The statistics in Table 5 demonstrate that our method is able to provide faster traversal, owing to the ability to optimize the expected traversal cost for both empty *and* non-empty cells of the acceleration structure: Efficient traversal of non-empty cells is critical when looking at geometry from shallow angles, and even more so with increasing grid resolutions. We also confirm that performing expansion after a merge step delivers the best results in terms of both ray traversal and memory footprint. Furthermore, comparing the results in Table 5 and Table 3 shows that a major amount of the traversal performance is obtained by designing our method to efficiently handle a dense, *multi-level* initial space subdivision.

6.3. Dynamic Scenes

For dynamic scenes, we compare performance against a standard two-level grid ray tracer with per-frame rebuild of the acceleration structure. We measure the improvement in image quality when using irregular grids by rendering images using ambient occlusion. The number of ambient occlusion samples per pixel that can be traced without exceeding the pre-determined time per frame is listed in Table 4. While a two-level grid will always be faster to construct than our acceleration structure (we perform additional build steps) the added build time is usually balanced by a faster traversal for irregular grids. If the available render time is split equally between construction and traversal, our method outperforms the two-level grid ray tracer with respect to image quality. While this distribution of build time and traversal time might not always be optimal, it performs well in practice and can be used to automatically compute optimal grid densities as described below.

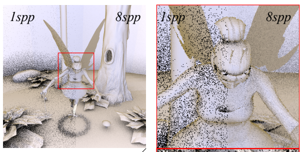
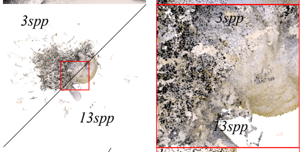
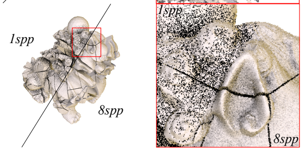
Scene	#Tris		10FPS (100ms) 2L Grid	Ours	20FPS (50ms) 2L Grid	Ours	30FPS (33ms) 2L Grid	Ours
	174K	λ_1, λ_2	0.2, 2.0	0.3, 2.4	0.2, 2.0	0.3, 2.4	0.2, 2.0	0.3, 2.4
		AO spp	2	20	1	8	0	3
	252K	λ_1, λ_2	0.2, 2.0	0.3, 2.4	0.2, 2.0	0.3, 2.4	0.2, 2.0	0.3, 2.4
		AO spp	21	57	8	24	3	13
	1.6M	λ_1, λ_2	0.03, 0.6	0.3, 2.4	0.03, 0.6	0.02, 0.16	0.03, 0.6	0.01, 0.08
		AO spp	1	8	0	1	0	0

Table 4: *AO spp is the (rounded down) number of Ambient Occlusion samples per pixel we can render for dynamic scenes (fairy, exploding dragon, breaking lion) at a resolution of 1024x1024 with a specific frame rate. We use the time budget for build and traversal (including primary rays), but ignore additional overheads like framebuffer display or keyframe interpolation.*

Scene Resolution		Traversal MRay/s	Memory MB
Sponza	Macro regions	167 (+0%)	10
	Ours (M)	169 (+1%)	8
$172 \times 71 \times 105$	Ours (E)	182 (+9%)	47
	Ours (M + E)	185 (+11%)	8
Conference	Macro regions	108 (+0%)	9
	Ours (M)	116 (+7%)	8
$210 \times 133 \times 50$	Ours (E)	121 (+12%)	50
	Ours (M+E)	135 (+25%)	8

Table 5: *The merge (M) and expansion (E) steps of the irregular grid construction can be used in combination with uniform instead of hierarchical grids. We compare the performance for primary rays for the viewpoints in Figure 7(a) and Figure 7(b) to the macro regions in [Dev89]. The resolution is computed with $\lambda = 5$.*

Parameter selection: We determine build parameters fully automatically by setting a fixed ratio $\lambda_1 : \lambda_2 = 1 : 8$ and testing our build implementation on each input scene using $\lambda_1 \in [0.01, 0.3]$, i.e. $\lambda_2 \in [0.08, 2.4]$. We choose this ratio in order to approximately double the resolution in each dimension in the second level of the initial grid. For each test scene, we perform test runs using multiple densities, starting at the minimum and increasing it as long as the total build time does not exceed half of the time budget. We then select the parameters that yield the highest acceleration structure quality.

Together with the performance for static scenes, the results in Table 4 highlight another important contribution of this work. Using the same construction and traversal algorithm, we are able to obtain a rendering performance that is adequate for ray tracing both static and dynamic geometry. As opposed to supporting multiple variants of construction algorithms (e.g. LBVH, HLBVH, SBVH), our work makes this possible by *automatically* adjusting two intuitive parameters, greatly simplifying the development and maintenance of the actual implementation.

7. Conclusion

We presented a novel, grid-based acceleration structure for ray tracing on GPUs: The irregular grid. With this work, we demonstrate that efficient ray tracing is also possible with a non-hierarchical acceleration structure. We achieve this thanks to a few novel insights into ray traversal of regular acceleration structures like grids.

First, we propose a grid traversal algorithm that takes advantage of ray/box intersection tests—an operation that is very efficiently performed on a modern GPU. We take advantage of this and optimize the acceleration structure by repeatedly merging its cells together in order to improve the expected cost w.r.t. SAH. This eliminates the traversal bottleneck caused by empty space skipping in regular acceleration structures such as two-level grids.

Second, we separate the cell boundaries for entering and exiting rays, which gives us the opportunity to extend the exiting boundaries of cells. With this, we reduce the amount of necessary traversal steps and ray-primitive intersection tests, resulting in traversal performance that is not limited by memory latency.

Finally, depending on the selected density parameters, our method can provide both a ray traversal performance that rivals state-of-the-art methods for static scenes, as well as build times that allow for ray tracing dynamic scenes with per-frame rebuild of the structure. Altogether, we believe that thanks to its attractive properties, the irregular grid is an acceleration structure worth considering for ray tracing both static and dynamic scenes.

There are numerous opportunities for future work: For example, changing the initial subdivision scheme or varying the structure used to store the voxel map. Another potential for further research includes adapting the novel aspects of our traversal method to other acceleration structures like kd-trees or BVHs.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and suggestions. This work is co-funded by the European Union (EU), as part of the Dreamspace project, and by the Intel Visual Computing Institute Saarbrücken.

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.

[ALK12] AILA T., LAINE S., KARRAS T.: *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum.* NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June 2012.

[ÁSK14] ÁFRA A. T., SZIRMAY-KALOS L.: Stackless Multi-BVH traversal for CPU, MIC and GPU ray tracing. *Computer Graphics Forum 33*, 1 (2014), 129–140.

[AW87] AMANATIDES J., WOO A.: A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics '87*. Elsevier Science Publishers, 1987, pp. 3–10.

[CS94] COHEN D., SHEFFER Z.: Proximity Clouds&Mdash;an Acceleration Technique for 3D Grid Traversal. *Vis. Comput. 11*, 1 (Jan. 1994), 27–38.

[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.

[Dev89] DEVILLERS O.: The Macro-Regions: An Efficient Space Subdivision Structure for Ray Tracing. In *EG 1989-Technical Papers* (1989), Eurographics Association.

[EI07] ES A., İŞLER V.: Accelerated Regular Grid Traversals using Extended Anisotropic Chessboard Distance Fields on a Parallel Stream Processor. *J. Parallel Distrib. Comput. 67*, 11 (2007), 1201–1217.

[GPBG11] GARANZHA K., PREMOŽE S., BELY A., GALAKTIONOV V.: Grid-based SAH BVH construction on a GPU. *The Visual Computer 27*, 6 (2011), 697–706.

[Gut14] GUTHE M.: Latency Considerations of Depth-first GPU Ray Tracing. In *Eurographics 2014 - Short Papers* (2014), The Eurographics Association.

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[HMS06] HUNT W., MARK W. R., STOLL G.: Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (sep 2006).

[IWRP06] IZE T., WALD I., ROBERTSON C., PARKER S.: An Evaluation of Parallel Grid Construction for Ray Tracing Dynamic Scenes. *Symposium on Interactive Ray Tracing* (2006), 47–55.

[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.

[KA13] KARRAS T., AILA T.: Fast Parallel Construction of High-quality Bounding Volume Hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 89–99.

[KBS11] KALOJANOV J., BILLETER M., SLUSALLEK P.: Two-Level Grids for Ray Tracing on GPUs. In *EG 2011 - Full Papers* (Llandudno, UK, 2011), Min Chen O. D., (Ed.), Eurographics Association, pp. 307–314.

[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.

[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.

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for Ray Tracing using Space Subdivision. *Visual Computer 6*, 6 (1990), 153–65.

[MT97] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools 2*, 1 (1997), 21–28.

[NVI07] NVIDIA CORPORATION: *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[NVI13] NVIDIA CORPORATION: CUDA Unbound. `https://nvlabs.github.io/cub/`, 2013.

[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.

[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* (sep 2006), pp. 89–94.

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum 26*, 3 (Sept. 2007).

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

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics 24*, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH).

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial Splits in Bounding Volume Hierarchies. In *Proc. of High-Performance Graphics* (2009), pp. 7–13.

[Tsa09] TSAKOK J. A.: Faster Incoherent Rays: Multi-BVH Ray Stream Tracing. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 151–158.

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

[WH06] WALD I., HAVRAN V.: On Building Fast kd-Trees for Ray Tracing, and on Doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (sep 2006), pp. 61–69.

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. In *ACM SIGGRAPH 2006 Papers* (2006), pp. 485–493.

[WK06] WÄCHTER C., KELLER A.: Instant Ray Tracing: The Bounding Interval Hierarchy . In *Rendering Techniques 2006, Proceedings of the Eurographics Symposium on Rendering* (2006).

[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).

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware* (2006), pp. 67–77.