EUROGRAPHICS 2012 / P. Cignoni, T. Ertl
(Guest Editors)

Volume 31 (2012), Number 2

Rasterized Bounding Volume Hierarchies

Jan Novák and Carsten Dachsbacher

Computer Graphics Group / Karlsruhe Institute of Technology

Abstract

We present the rasterized bounding volume hierarchy (RBVH), a compact data structure that accelerates approximate ray casting of complex meshes and provides adjustable level of detail. During construction, we identify subtrees of BVHs containing surfaces that can be represented by height fields. For these subtrees the conventional ray-surface intersection, which possibly involves a large number of triangles, is replaced by a simple ray marching procedure to find the intersection with the surface. We describe GPU algorithms for construction, ray casting, and data querying of the RBVH that achieve comparable or higher performance than state of the art acceleration structures for triangle meshes. Moreover, RBVHs provide an inherent surface parameterization for storing data on the surfaces and natively handle triangle and point-based surface representations. We also show that RBVHs support adaptive level-of-detail and can be combined with traditional BVHs to handle complex scenes.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

Interactive ray tracing has been an intensive area of research in recent years. While the first challenge was to develop appropriate acceleration structures for static scenes that accelerate ray casting on modern parallel hardware, research next began to focus on efficiently building these data structures for fully dynamic scenes in every frame [WMG*07]. The main advantage of ray tracing, compared to rasterization, is that it naturally handles effects such as reflection or refraction, as secondary rays can be cast easily. For primary rays, both essentially produce identical results; rasterization can also be accelerated using spatial data structures, e.g. for culling (see [COCSD03] and [Dac10] for overviews).

A common consensus is that the intersection computation for secondary rays is often not required to be accurate, e.g. when computing indirect lighting or glossy reflections [YCK*09]. Voxel representations [CNLE09], for instance, can be used for approximate ray casting as they can be created with arbitrary accuracy, and reach speed comparable to that of triangle-based acceleration structures. However, their construction typically requires significant time and memory [LK10]. Other examples of approximate scene representations used for global illumination are point hierarchies [REG*09] and multiple depth cube maps [YWC*10]. These approaches have in common that the input surfaces, typically triangle meshes, are resampled.

In this paper we present a novel acceleration structure primarily targeted at fast, approximate ray casting. The main difference to standard bounding volume hierarchies, e.g. as created by [LGS*09], is that during construction we identify subtrees containing surfaces which can be represented by, and thus rasterized to, a height field. For these subtrees the conventional ray-surface intersection, possibly involving a large number of triangles, is replaced by a simple ray marching procedure to find the intersection with the surface. In general, rasterized bounding volume hierarchies (RBVHs) are shallow structures with the leaves storing the scene geometry in the form of height fields (see Fig. 1). These can be rasterized at an arbitrary resolution and thus, thanks to decoupling from the input surfaces, inherently provide means to adjust the level of detail (LOD).

RBVHs are best suited for complex scenes consisting of large numbers of primitives, e.g. obtained from subdivision surfaces or scanned environments. We show that in these cases an RBVH can achieve better *approximate* ray tracing

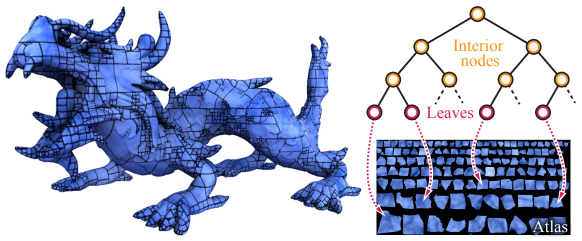


Figure 1: *A Rasterized Bounding Volume Hierarchy (RBVH) represents surfaces using a set of height fields (left) that are organized into a hierarchy and stored in an atlas (right).*

performance than other accelerating structures. In addition, RBVHs have further beneficial properties: the height fields are stored in a texture atlas, which automatically provides a parameterization of the surfaces and can be used to store on-surfaces signals, e.g. for interactive painting or photon mapping. RBVHs are not restricted to polygonal meshes: all representations that can be rasterized, e.g. point clouds [GP07], can directly be used during construction. Finally, the memory footprint of RBVHs is typically much lower than that of the original geometry as they can be created with just the required amount of detail. This enables ray casting of large scenes with limited memory, e.g. on GPUs. We describe a parallel construction scheme to build RBVHs for point representations and triangle meshes suitable for GPUs. Furthermore, we introduce a hybrid RBVH that stores triangles in leaves (like a traditional BVH) whenever height fields are not well-suited to represent the geometry. We demonstrate the use of RBVHs for applications such as rendering caustics with photon mapping, glossy reflections, interactive painting, and ray casting of point clouds.

2. Previous Work

Acceleration Structures for Ray Tracing Hierarchical and non-hierarchical spatial index structures, with all their advantages and disadvantages, have been explored for accelerating ray casting on various architectures, different scenes and applications; Wald et al. [WMG*07] have an excellent overview. Recent work focuses on building BVHs and kD-trees on the GPU, where they can be directly used for ray casting. Lauterbach et al. [LGS*09] construct LBVHs by linearizing primitives along a space filling Morton curve combined with the surface area heuristic (SAH) yielding close to optimal hierarchies, and thus good overall performance for both construction and traversal. This approach has been improved using a hierarchical formulation [PL10] and further accelerated with work queues while using less memory [GPM11]. Ray tracing performance of BVHs can be improved by creating tighter axis-aligned bounding boxes (AABBs), e.g. by split clipping [EG07], subdividing triangles recursively [DK08], or adapting the SAH [SFD09]. The potential of these approaches has been analyzed by Popov et al. [PGDS09] who also present a generic algorithm. Aila and Laine [AL09] analyze the traversal of BVHs on GPUs and their work can be considered as state of the art in terms of ray casting performance. kD-trees can also be efficiently built on the GPU using a data-parallel spatial median algorithm for the upper levels to partition the workload between streaming processors [ZHWG08]. In contrast, Choi et al. [CKL*10] focus on precise SAH-optimized kD-trees on architectures with less cores. Various two-level hierarchies were proposed for ray tracing dynamic scenes with nested grids [KBS11] and handling tessellated and displaced patches [HKL10]. The benefits of using shallow hierarchies were explored in [DHK08], but only in the context of multicore CPUs. Note that RBVHs are also shallow, as entire subtrees are replaced by single height fields.

Ray Tracing with Sample-Based Representations This topic is intensively studied and closely related to our work. A classic sample-based representation is the voxelization of a scene, possibly stored as a hierarchy in an octree, e.g. [CNLE09, CNS*11]. Voxel data structures allow for high ray casting performance and adaptive accuracy [LK10], but often require significant construction time and memory. Detailed displacements of smooth surfaces can be represented as height fields and ray casted very efficiently (see Szirmay-Kalos and Umenhoffer [SKU08] for an overview). Ray casting height fields has further been extended to handle arbitrary geometry using non-orthogonal projections [BD06]. A set of depth cube maps, rendered from well-chosen locations within a scene, can also be used to accelerate ray casting for photon mapping [YWC*10]. Carr et al. [CHCH06] use geometry images which allow for efficient ray casting, since AABBs can be easily obtained from min/max-mipmaps. Note that this approach handles deforming geometry but the topology is not allowed to change, as it is too costly to recompute the parameterization on-the-fly. Closely related to our work, de Toledo et al. [dTWL08] partition an object's surface and represent the individual parts as height fields. In contrast to our work, they do not construct a hierarchical data structure and the build process is too costly for frequent updates. Although used for storing textures, the partitioning scheme used in TileTrees [LD07], which splits the surfaces into parts that can be bijectively projected onto a cube's faces, is in spirit similar to ours, but not feasible for updating the data structure on-the-fly.

Level of Detail An important feature of RBVHs is the inherent possibility to adjust the LOD of the representation. Pantaleoni et al. [PFHA10] report that voxelization is well-suited as an approximate representation for small and medium sized scenes, but fails to handle large, complex scenes. They propose to combine acceleration structures with a multiresolution scheme for LOD. In principle any mesh decimation method, e.g. progressive meshes [HSH09], could be used; however, this requires maintaining and implementing two intricate algorithms for LOD and construction. Related to our work are the volume surface trees [BHGS06] that combine an octree and a set of quadtrees to represent surfaces. However, their goal is to resample surfaces for reconstruction and mesh simplification; the resulting structure is not suitable for ray casting and adaptive level of detail.

3. RBVH Construction

In contrast to a triangle-based BVH, our RBVH can be seen as a two-level data structure, where the upper part consists of a shallow tree, and the lower part represents the geometry using height fields. We build the RBVH in a top-down manner, i.e. we start from the scene's bounding box representing the root node and continue with the inner nodes towards the leaves. In general, each node is split and the primitives (e.g. triangles or points) partitioned into child nodes until: (1) the geometry can be faithfully represented as a single

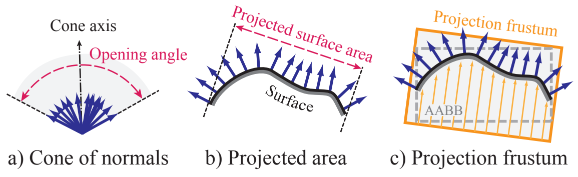


Figure 2: *During the RBVH construction, nodes are further refined if the cone of normals (a) or the projected area (b) (depending on which one is used) exceeds a user-defined quality threshold. In the opposite case, the surfaces within the node are rasterized using an orthogonal projection (c).*

height field, and (2) the cost of intersecting the height field is smaller than the traversal of an interior node (resembling the idea of the SAH). Once the geometry is partitioned, we rasterize the primitives of every individual leaf using an orthogonal projection into a texture atlas, where each tile stores the depth values of a height field that represents the surfaces in the corresponding leaf.

The RBVH is an approximate, sample-based structure providing several means to control the quality of the representation: in Sect. 3.1 we describe two *local* criteria to measure and control the accuracy of representing a surface by a height field. Then we introduce heuristics for node splitting and minimizing traversal costs in Sect. 3.2, and finally, in Sect. 3.3, we detail the *global* quality control (i.e. the sampling density) achieved through varying the resolution of the rasterized height fields.

3.1. Refinement Criteria

One integral component of the RBVH construction is an efficient way to determine whether we can, and should, represent a part of a surface as a single height field. Such representation is only possible without loss of information, if we find a projection of the surface onto a plane without folding. Additionally, we strive to sample surfaces as uniformly as possible and thus we should avoid rasterizing surfaces from grazing angles. To find a suitable direction and to minimize the projection error we consider one of the following measures: the minimum cone subtended by the normals of the surface, and the area of the projected surface. Both measures are shown in Fig. 2 and detailed in the following paragraphs.

Cone of Normals The cone of normals of the primitives within a node can be used to determine if there exists an orthogonal projection where all surfaces are front-facing: if the opening angle of the cone is less than π, such directions exist. Otherwise, we should split the surface and represent it with multiple height fields. In practice we compute an approximation [SAE93] and use even narrower cones (e.g. $\pi/2$) enforcing more uniform sampling of surfaces. To maximize the *minimum* sampling density (of the most diverted surface) we orient the projection frustum along the cone axis.

Projected Surface Area Another good projection direction is the average surface orientation, computed as the area-weighted sum over all primitives' normals. It maximizes the *average* sampling density, but does not guarantee sampling of the entire surface, as some primitives might be back-facing and thus occluded. As a quality metric we use the projected area $A^{\perp}$ of the primitives, which equals to the length of the summed area-weighted normals. We rasterize the surfaces if the ratio $A^{\perp}/A$, where A is the surface area, is greater than a user-defined threshold α; otherwise we split the node.

Discussion The cone of normals is a restrictive criterion, splitting a node whenever there is no projection possible without back-facing primitives. According to our experiments it is best-suited for (manually) modeled scenes, e.g. from subdivision surfaces. The relative projected area is robust against noise in the primitives' orientation, which is often present in scanned geometry (see Fig. 10). In either case the projection frustum is defined by the the minimum bounding box that is oriented along the projection direction and contains the AABB of the surfaces. We also consider whether it is beneficial to split the surface, even if it can already be represented as a height field. Heuristics for splitting and an their analysis are discussed in Sect. 3.2 and 3.4.

3.2. Subdivision Strategies

The previously described refinement criteria determine *whether* a surface has to be split. In this section we discuss heuristics determining *how* to actually split the surface, i.e. how to partition the primitives within a node. We discuss and compare two approaches partitioning the primitives with respect to a split plane: the RBVH pendant of the Surface Area Heuristic (SAH) [Hav00] and the simple spatial median. We also introduce a complementary object split strategy that avoids fusing surfaces of different objects.

Surface Area Heuristic Ray casting accelerators are often built according to the SAH, which defines the cost of a partition by summing up the costs of intersecting each child, weighted by the respective probability that a ray passes through them. The minimum is usually greedily searched by evaluating the cost of several candidate split-planes at the scope of the current node. In case of the RBVH, the cost of intersecting a surface amounts to ray marching the respective height field. The number of ray marching steps depends on the resolution of the height field, which is in turn linked to the surface area, as we strive to achieve uniform sampling (described in Sect. 3.3). Therefore, the area of the childrens' surfaces is already a good estimate for the intersection cost. Note that we use the SAH only to determine where to split, not whether to split. The modified SAH for finding the best partitioning of a node a into two children b and c is then:

$$C(b,c) = p(b|a)A(b) + p(c|a)A(c), \qquad (1)$$

where $A(b)$ is the area of the surface in b, and $p(b|a)$ is the geometric probability of intersecting b: $p(b|a) = S(b)/S(a)$, where $S(.)$ is the surface area of a nodes' bounding box.

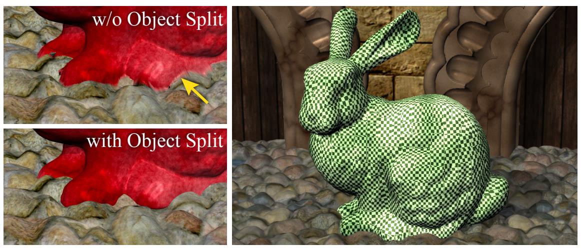


Figure 3: *Left: subdividing nodes according to object IDs prevents excessive refinement and avoids fusing of surfaces. Right: the RBVH achieves almost uniform sampling across all surfaces (shown as a checker board on the bunny model).*

Spatial Median This simple strategy always places the split plane in the middle of the bounding box perpendicular to the longest axis. This can be highly suboptimal in the case of regular BVHs, as the number of primitives on both sides can differ significantly. Since RBVHs decouple from the actual tessellation, the spatial median somewhat resembles the idea of Eq. 1, and, for smooth surfaces in particular, also tends to split the surface into roughly equal areas (see Sect. 3.4 for analysis).

Object Split When the surfaces of two or more objects intersect, it is reasonable to split them according to their object IDs (which is typically available from the scene modeling or hierarchy). By this, we can avoid excessive refinement due to a low projected area or diverging normals of nearby objects. Fig. 3 (left) shows an example where the object splitting also avoids fusing of two meshes. We perform the object split whenever a node contains surfaces of exactly two objects.

3.3. Rasterization to the Atlas

After splitting the nodes and creating the tree hierarchy of the RBVH, we rasterize the surfaces to the atlas. For every leaf node, we compute an orthogonal bounding frustum that is aligned with the projection direction and contains the axis-aligned bounding box (AABB) of the corresponding surface (see Fig. 2(c)). Next we determine the resolution allocated to the respective atlas tile, i.e. how densely we sample the surface. Our goal is to retain a (roughly) uniform sampling of surfaces, which can be intuitively controlled by the user. For this, we use a *global* pixel-to-area ratio ρ to compute the resolution R of a square tile as: $R = \sqrt{\rho A^2 / A^{\perp}}$, where ρA is the total number of pixels for representing the surface scaled by the inverse relative projected area $A/A^{\perp}$. Although the actual number of samples used to store the height information is usually smaller (as a tile is typically not fully covered by the projection), we found that this approach still provides almost uniform sampling of the entire surface (Fig. 3 right). Complementing the local refinement criteria, the pixel-to-area ratio is a means to adjust the quality globally.

As we create square tiles, we can easily and tightly pack them into the atlas. For that we sort them according to their

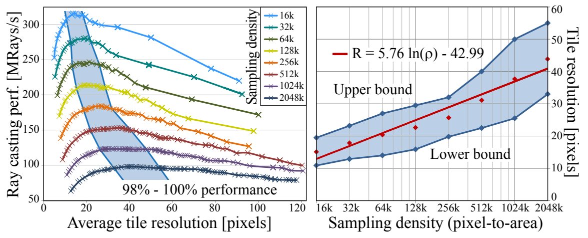


Figure 4: *Left: the horizontal axis shows the average tile resolution and individual series depict the dependency of the ray casting performance on the sampling density. Right: we identify intervals of the resolution with at least 98% of the peak performance and approximate the midpoints of these intervals with a logarithmic function, that is used to select the optimal tile resolution during the RBVH construction.*

descending resolution R and pack them row-wise. Within each row the tiles are placed from the largest to the smallest from left to right. Resolution of the first (largest) tile in a row determines the vertical offset to the next row. For each leaf, we store a final projection matrix that is computed from the bounding frustum, tile resolution, and position in the atlas. Lastly, we transform all primitives using the corresponding matrices and rasterize them to the atlas. Note that we can use the atlas to store arbitrary on-surface signals, e.g. normals or surface colors, by rasterizing them to additional atlas layers.

As we rasterize surfaces from different directions and at different resolutions, the height fields typically do not match exactly at their boundaries. To avoid cracks in the reconstruction, we ensure that the height fields of neighboring surface parts slightly overlap by duplicating primitives within a certain region around the split plane ($\pm 5\%$ in our scenes), and assigning them to both children. After rasterization we also apply a dilation filter on the atlas that creates an additional “safety-border” for all tiles. Note that this process is common to many atlas-based techniques (e.g. see [dTWL08]). In order to create meaningful data, we compute the gradient of the surrounding pixels to ensure that the dilated surfaces do not create distracting extrusions.

3.4. Analysis of the Subdivision Strategies

In addition to the subdivision strategies, the RBVH has another degree of freedom that trades off between the size of the node hierarchy and the resolution of tiles: despite a surface may not require subdivision due to the refinement criterion, we might still split it, if it improves the traversal performance. Note that this consideration is similar to the choice of how many primitives should be stored in the leaves of regular BVHs. To this end, we ran a series of benchmarks (primary and secondary rays separately) to find a good balance between the depth of the node hierarchy and the tile resolution. In each test we subdivided the nodes until the atlas tiles had resolutions lower than a specified threshold. Note that higher tile resolution results in shallower hierarchies and

Scene	Heuristic	Nodes	CPU Build	Trac. [MRays/s]	
Dragon (699 k tris.)	Median	2999	0.9 s	146.3	= **91%**
	SAH	2674	26.8 s	161.3	
Happy Buddha (1.37 M tris.)	Median	8065	1.8 s	138.1	= **82%**
	SAH	3266	50.6 s	169.3	
Beast (2.82 M tris.)	Median	7639	3.5 s	109.6	= **78%**
	SAH	4696	118.1 s	140.5	
AsianDragon (7.22 M tris.)	Median	6345	5.9 s	159.9	= **89%**
	SAH	5227	173.0 s	180.2	

Table 1: *Number of nodes, CPU construction time (using 1 core), and tracing performance of our RBVH built using either the spatial median along the longest axis, or the SAH selecting the best from 32 split candidates along each axis of the bounding box.*

vice versa. Fig. 4 illustrates that the peak performance is obtained with different tile resolutions for different pixel-to-area ratios (curves in Fig. 4, left), i.e. depending on the sampling density, there is an optimal resolution and we should refine the nodes until their tiles reach it. To derive the *optimal* tile resolution we fit a logarithmic function to the midpoints of parameter intervals with at least 98% of the peak performance. This has proven to be more reliable than fitting to the absolute maxima. For coherent rays the optimal tile resolution is $5.76\ln(\rho) - 42.99$; running the benchmark for secondary rays yielded $1.23\ln(\rho) - 0.17$. That is, depending on the application, different parameters yield optimal RBVHs.

We also compare the impact of the SAH and spatial median heuristics in terms of ray tracing performance and the number of RBVH nodes in Tab. 1. Although the total number of nodes created using the spatial median was sometimes up to 2.5× higher, the ray tracing was always at least 78%, and 86% on average, of the performance of the RBVH built with SAH. Considering the build times, it is obvious that the spatial median is the better choice for dynamic scenes.

3.5. Adaptive, Varying Level of Detail

In Sect. 3.1 and 3.3 we described two means to control the accuracy and memory requirement of the RBVH. Altering these parameters also influences the performance: (1) loosening the refinement criterion, e.g. allowing larger cones of normals, effectively prunes the tree by representing surfaces, that would be otherwise refined, with a single height field. This sacrifices uniform sampling (eventually even bijective projections) for reducing the number of nodes in the tree; (2) decreasing the sampling density when rasterizing the tiles reduces memory footprint and speeds up the ray-height field intersection. Instead of setting these parameters once for the entire RBVH, we can determine them for every node during construction. This enables us to adapt the level of detail according to a quality function that, for a given point in space, defines the desired quality (which is mapped to construction parameters). By this, for instance, we can locally adjust the accuracy of the RBVH depending on the distance to the viewer or any other point of interest (Fig. 5).

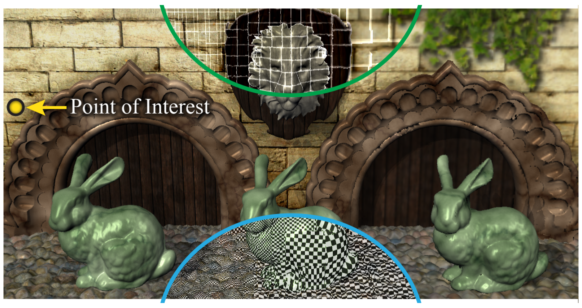


Figure 5: *RBVHs support level of detail rendering. Here the refinement criterion is loosened and the tile resolution decreases with the distance to the point of interest. The top inset depicts the boundaries of tiles, the bottom inset shows the varying sampling density.*

In situations when the reconstruction of the entire RBVH is not desired, we adapt the level of detail by mip-mapping the atlas, and select an appropriate mip level during traversal. Since the tree structure does not change, this strategy is suboptimal to a full rebuild, but faster and well-working as long as the atlas tiles do not fuse during the mip-map creation.

4. Hybrid RBVH

RBVHs work best when the input surfaces are highly tessellated and can be faithfully represented by height fields. Obviously this is not always the case and would restrict the applicability of the RBVH in many cases. To this end, we propose a hybrid RBVH: whenever the surfaces of a node cannot be efficiently represented as a height field (e.g., few triangles with large area, or surfaces with sharp bends) we build the subtree as a traditional BVH. This creates hybrid RBVHs where surfaces are partly represented as height fields and partly by triangles. Fig. 6 shows a car model with surfaces color-coded according to whether they are represented as height fields, triangles, or both (similar to two height fields, triangles and height fields of adjacent nodes may overlap). For the hybrid RBVH we precompute the surface curvature for the input mesh vertices and resort to triangles when (1) the percentage of vertices above a certain curvature exceeds 80%; (2) there are less than 8 triangles in the node.

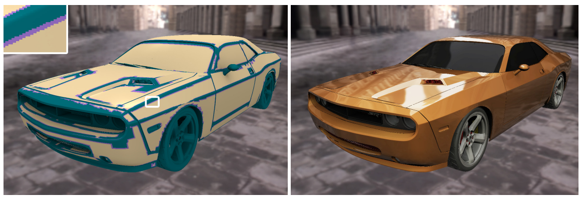


Figure 6: *Rendering with a hybrid RBVH – Left: visualization of surfaces represented by height fields (yellow), triangles (green), or both (purple). Right: ray tracing of primary and secondary rays without any visible artifacts.*

5. Traversal of the RBVH

Using RBVHs for ray casting is similar to the traversal of traditional BVHs: the traversal starts at the root node, tests the ray against the childrens' bounding boxes, and stores the nodes to be visited on a stack. In the case of a leaf node, the ray segment overlapping the leaf's bounding box is transformed into atlas space using the stored projection matrix. Then we march along the corresponding line segment and test for an intersection with the height field using linear plus secant search (as in [SKU08]). If an intersection is found, we transform the location back to world space. Note that the texture coordinates of the intersection can be used to look up surface attributes from other layers of the atlas. To reconstruct these attributes without seams, filtering across tiles can be implemented [LD07], but in our examples, increasing the resolution of the attribute layer provided sufficient quality.

Retrieving stored on-surface information for a given point in world space (on the surface) is also easy: we first search for the leaf node whose bounding box contains this point, and then transform the point's coordinates into atlas space using the projection matrix. Note that due to the overlaps a surface point may be represented by two or more height fields. We account for this when writing data into the atlas and set all the texels corresponding to a single point.

6. GPU Construction

In this section we describe an RBVH construction algorithm for massively parallel architectures, such as GPUs. We favor simplicity and fast construction using the inexpensive spatial median for partitioning the primitives. The slightly lower traversal performance is compensated by the faster construction, which is beneficial especially for dynamic scenes. As the refinement criterion we use the more robust projected surface area, which also requires only one pass over the primitives (cone of normals needs two).

The construction proceeds in a breadth-first manner starting from the root node (containing all primitives) and creating nodes level by level. To construct a single level we first compute the axis-aligned bounding boxes (AABB) and the average surface orientation for each node in the level. Next we determine which nodes can be rasterized and remove the corresponding primitives from the subsequent construction steps. All other primitives are assigned to the respective child nodes. We show a pseudocode and meaning of the variables in Fig. 7 and detail the construction in the next paragraphs.

We start the construction of the node hierarchy by initializing two arrays storing references to the primitives, *primRefs*, and the node to which every primitive belongs to, *nodeRefs*. Since we want the child nodes to slightly overlap (to prevent cracks), both arrays have to be large enough to allow duplicating references (in all our examples 150% of the original size was sufficient). We also keep track of the number of nodes N, the number of remaining references R, and the references stored in the final arrays of leaves F.

```
BuildNodeHierarchy(primBounds, primNorm, primA)
  nodeBounds[], nodeA[]  // AABB and surface area of nodes
  nodeANorm  // sum of area weighted normals in each node
  primRefs ← {1, 2, ...N}  // references to primitives
  nodeRefs ← {1, 1, ...1}  // references to nodes
  finPrimRefs  // final array with references to primitives
  finNodeRefs  // final array with references to nodes
  N  // (maximum) number of nodes potentially created so far
1  $N \leftarrow 1$, $R \leftarrow$ number of primitives, $F \leftarrow 0$
2  while $R > 0$ :
3    for all $i$ in $[0,R)$ in parallel :
4      $j \leftarrow$ primRefs$[i]$
5      bounds$[i] \leftarrow$ primBounds$[j]$
6      ANorm$[i] \leftarrow$ primA$[j] *$ primNorm$[j]$
7      A$[i] \leftarrow$ primA$[j]$
     // compute per-node information
8    nodeBounds$[N..2N] \leftarrow$ reduceByKey(bounds, nodeRefs)
9    nodeANorm$[N..2N] \leftarrow$ reduceByKey(ANorm, nodeRefs)
10   nodeA$[N..2N] \leftarrow$ reduceByKey(A, nodeRefs)
     // decide whether to split or rasterize
11   rasterize$[0..R] \leftarrow$ count$[0..R] \leftarrow \{0,0,\dots 0\}$
12   for all $i$ in $[0,R)$ in parallel :
13     $n \leftarrow$ nodeRefs$[i]$
14     if not requiresSplit(n) and not shouldBeSplit(n) :
15       rasterize$[i] \leftarrow 1$
16     else :
17       count$[i] \leftarrow$ toNChildren$(i)$
18       child$[i] \leftarrow$ toWhichChildren$(i)$
19   finRank$[0..R] \leftarrow$ scan(rasterize)
20   cRank$[0..R] \leftarrow$ scan(count)
     // filter primitives for rasterization; compact the others
21   for all $i$ in $[0,R)$ in parallel :
22     if rasterize$[i]$ :
23       finPrimRefs$[F +$ finRank$[i]] \leftarrow$ primRefs$[i]$
24       finNodeRefs$[F +$ finRank$[i]] \leftarrow$ nodeRefs$[i]$
25     else :
26       primRefs[cRank$[i]] \leftarrow$ primRefs$[i]$
27       nodeRefs[cRank$[i]] \leftarrow$ nodeRefs$[i]*2+$child$[i]\&1$
28       if child$[i] = 3$ :
29         primRefs[cRank$[i]-1] \leftarrow$ primRefs$[i]$
30         nodeRefs[cRank$[i]-1] \leftarrow$ nodeRefs$[i]*2$
     // sort primitives according to the node; update counters
31   sortByKey(primRefs, nodeRefs)
32   update$(R,F)$
33   $N \leftarrow 2N+1$
```

Figure 7: *Parallel algorithm for constructing the upper part (hierarchy of nodes) of the RBVH.*

The construction of the RBVH continues until all remaining references R are processed, i.e. placed in the leaves (while $R > 0$, line 2). We assume that, at the beginning of each iteration, the primitive references are sorted according to the node they belong to (stored in *nodeRefs*), thus forming segments with the same node reference. In order to evaluate the refinement criterion, we compute the AABB, total area, and the sum of area-weighted normals for every node. For this, we fill three auxiliary arrays (*bounds*, *ANorm*, and A) with the primitive data (lines 3-7), and perform a parallel segmented reduction to obtain a single value per node (lines 8-10).

Next, using this per-node information we determine if primitives will be rasterized or further split, thus going into new child nodes: we evaluate the splitting criterion (Sect. 3.1) and optimal subdivision (Sect. 3.4) (line 14) and mark every primitive either for rasterization (line 15) or splitting (lines 17-18). In the latter case, we store two flags per primitive: *count* contains the number of children the primitive will go to (1 or 2), and *child* refers to the actual children (1 - left child, 2 - right child, 3 - both).

These flags are used to send the primitives to the final array (when they are flagged for rasterization) or to the construction array of the next RBVH level. To determine the locations within these array, we first compute parallel prefix sums (lines 19-20). Note that computing the prefix sum on the *count*-array automatically reserves space for duplicating primitives. Then we again process all primitives: those flagged for rasterization will be simply appended to the final arrays according to the prefix sum (lines 22-24); primitives that go into child nodes are kept in the construction arrays, but are compacted to remove unused entries (we double-buffer the arrays to avoid write-after-read hazards). Primitives that are sent to both child nodes (recall that we accounted for that in the prefix sum) are duplicated in lines 29-30. Note that we use implicit addressing to avoid computing and storing pointers during the construction. In contrast to regular BVHs, the upper part of the RBVH is very shallow (hundreds or thousands of nodes), and thus the memory required due to storing a full tree for implicit addressing is small.

The last step of constructing a single RBVH level is to sort the references again according to the node indices (for the next iteration), and to update the number of remaining primitive references and references in leaves (lines 31-33).

Finalizing the RBVH After the hierarchy construction, per-node attributes (e.g. the bounding box) are stored in the *node**** arrays. We first separate interior and leaf nodes into two arrays, and remove the unused entries that were introduced due to the implicit addressing. Lastly, we sort all leaf nodes according to their tile resolution, compute transformation matrices and rasterize all primitives referenced by *finPrimRefs* as described in Sect. 3.3.

7. Implementation Details

We implemented the RBVH construction on the CPU (for evaluating the subdivision strategies) and on the GPU. On the GPU we use CUDA for the hierarchy construction and ray casting/traversal, and Direct3D 10 for rendering the primitives into the atlas. For all data parallel primitives (**reduceByKey**, **sortByKey**, **scan**) we used the Thrust CUDA library [HB11].

After the hierarchy construction, we compact the nodes into a single tightly packed array, obtaining an efficient representation for fast traversal. Each interior node occupies $48+8$ bytes for both child-AABBs (6 floats each) and references to the child nodes (2×4 bytes). For leaves we only need to store the projection matrix. As we use orthogonal projections only, the last row is always $\{0,0,0,1\}$, thus we can store the matrix as 12 floats, or 48 bytes, respectively.

8. Results and Discussion

In this section we evaluate the RBVH regarding its ray casting performance, level of accuracy and approximate nature, construction time, and memory requirements. All results were measured on an Intel Core i7 system running at 2.8GHz with an NVIDIA GTX 470 GPU.

Quality vs. Performance One key feature of the RBVH is that it allows trading quality for performance (Fig. 9, please zoom in using the electronic version). Tab. 2 shows the rendering performance for primary and secondary rays (measured separately) for five different quality settings (finest Q0 to coarsest Q4). For benchmarking secondary rays, we computed environmental lighting by randomly sampling the hemisphere with 4000 rays per pixel. To compare against a triangle-based BVH, we integrated Aila and Laine's [AL09] GPU ray tracer (which can currently be considered as the state of the art) into our application to ensure that we cast exactly the same rays with both acceleration structures. Tab. 2 also includes the comparison to this method.

As the RBVH is a sample-based structure its ray casting performance does not depend on the number of triangles. We adjusted the quality levels such that Q0 is visually indistinguishable from the reference for primary rays at a resolution of 1024×1024. The coarser levels, Q1 to Q4, are appropriate for secondary rays, or for primary rays if the object is further from the camera. Here we keep the same distance to better visualize the approximate nature of the RBVH. For primary rays and complex scenes (more than 1.5 million triangles in our examples) the RBVH outperforms the regular BVH, as it does not store more information than actually necessary. The absolute performance of the RBVH depends on the curvature of the surfaces; nevertheless, it linearly grows with decreasing quality for all tested scenes. The relative speed-up of RBVHs to BVHs increases with the scene size.

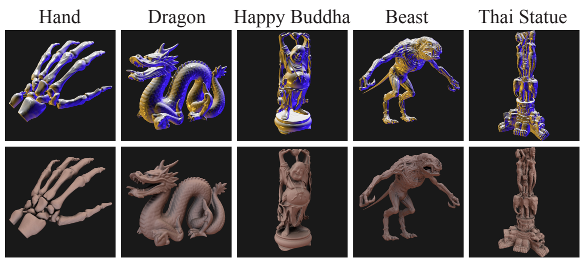


Figure 8: *Test scenes listed in Tab. 2 using an RBVH with quality level Q2 for primary (top row) and secondary rays (bottom row); please use the electronic version to zoom in.*

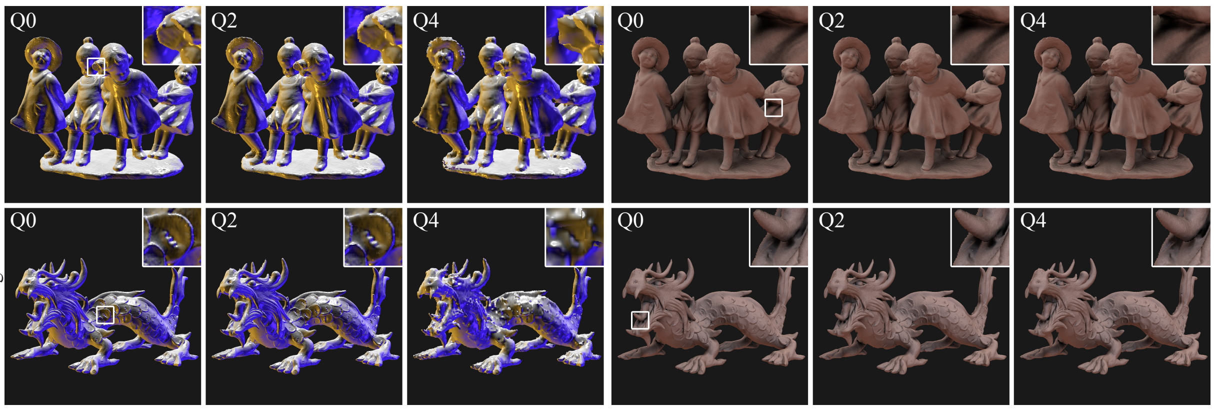


Figure 9: *RBVHs can trade performance for quality; shown with close-ups for three different quality settings. The respective ray casting performance is given in Tab. 2. Left: the RBVHs are used for casting primary rays; the approximate nature becomes visible for low quality settings such as Q4, but not for high quality settings. Right: objects are rendered using rasterization and the RBVHs are used for secondary rays only. Note that even for the coarse Q4 settings we obtain satisfying results.*

		Primary (Coherent) Rays [MRays/s]							Diffuse (Incoherent) Rays [MRays/s]						
Scene	# of triangles	BVH	Q0	Q1	Q2	Q3	Q4	Speedup	BVH	Q0	Q1	Q2	Q3	Q4	Speedup
Hand	655 k	178.7	145.3	176.6	209.9	266.0	300.0	**0.8 - 1.7**	55.9	32.5	37.1	44.3	56.6	69.1	**0.6 - 1.2**
Dragon	699 k	137.2	111.6	139.8	169.4	229.4	287.2	**0.8 - 2.1**	38.2	20.7	21.7	27.6	38.7	53.3	**0.5 - 1.4**
Happy Buddha	1.37 M	144.6	147.5	183.9	226.2	307.2	386.4	**1.0 - 2.7**	37.2	21.7	24.9	30.5	41.4	54.6	**0.6 - 1.5**
Children	1.45 M	123.2	121.2	148.8	177.6	232.2	282.8	**1.0 - 2.3**	36.1	21.2	23.6	28.1	39.1	51.0	**0.6 - 1.4**
Beast	2.82 M	120.4	131.9	163.1	195.4	256.0	298.3	**1.1 - 2.5**	37.8	20.7	23.3	27.8	37.9	47.4	**0.5 - 1.3**
Asian Dragon	7.22 M	85.9	158.5	190.2	216.1	263.7	304.4	**1.8 - 3.5**	36.7	29.4	31.9	35.1	43.7	53.9	**0.8 - 1.5**
Thai Statue	10.0 M	122.9	197.4	250.5	302.7	375.4	454.6	**1.6 - 3.7**	32.9	30.7	35.5	41.9	51.7	59.9	**0.9 - 1.8**

Q0: $\rho_s = 2\text{M}, \{\alpha = 0.85 \mid \varphi = 45°\}$	Q1: $\rho_s = 1\text{M}, \{\alpha = 0.80 \mid \varphi = 50°\}$	Q2: $\rho_s = 512\text{k}, \{\alpha = 0.75 \mid \varphi = 55°\}$
Q3: $\rho_s = 128\text{k}, \{\alpha = 0.70 \mid \varphi = 60°\}$	Q4: $\rho_s = 32\text{k}, \{\alpha = 0.65 \mid \varphi = 65°\}$	

Table 2: *Detailed performance (in million rays per second) for the scenes shown in Fig. 8 and 9 with different quality settings (Q0 - finest, Q4 - coarsest). All RBVHs have been constructed using the projected area refinement criterion (except for the Hand model). The construction parameters are: ρ_s is the pixel-to-area ratio (all scenes were normalized to a total area of 1 to achieve equal sampling density), α is the ratio of the projected area, and φ denotes the cone of normal. The "BVH" column reports the performance using Aila and Laine's [2009] method, "Speedup" shows the range of relative performance of the RBVHs.*

Memory Requirements The node hierarchy of the RBVH is typically shallow and the memory requirements are dominated by the texture atlas. The atlas size directly depends on the desired sampling density, which enables us to build compact acceleration structures at the expense of lower accuracy. Tab. 3 shows memory requirements of a triangle-based BVH, where the geometry is represented by indexed vertices, and our RBVH for four different models consisting of 699k (Dragon) to 10.0 million triangles (Thai Statue). In all cases, the RBVH requires less memory even at the highest quality level. The most significant *compression* can be observed for the Thai Statue, where Q0 and Q4 require only 4% and 0.2% memory of the regular BVH, respectively. This is obviously achieved by decoupling from the input geometry, which is desirable for removing detail not required for a given resolution or rendering task.

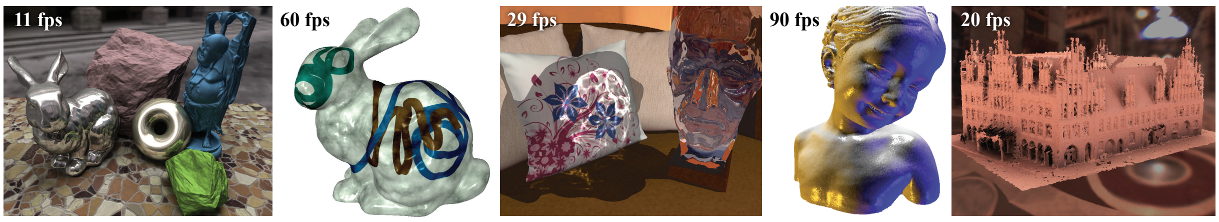
GPU Construction Our CUDA-based construction algorithm enables using the RBVH for interactive rendering of fully dynamic scenes. Tab. 4 reports the CPU and GPU

	BVH			RBVH Q0			Q2	Q4
Scene	Tree	Tris.	**Total**	Tree	Atlas	**Total**	**Total**	**Total**
Dragon	14.5	12.0	**26.5**	0.65	15.2	**15.8**	**3.86**	**0.38**
Happy Bud.	25.5	23.5	**48.9**	1.04	15.9	**17.0**	**4.13**	**0.40**
Asian Drag.	153.0	123.9	**276.9**	0.90	12.2	**13.1**	**3.43**	**0.40**
Thai Statue	204.4	171.1	**375.5**	2.29	14.3	**16.6**	**4.21**	**0.56**

Table 3: *Memory consumption (in megabytes) of a triangle-based BVH and our RBVH with quality levels Q0, Q2, and Q4. Bold numbers show the total required memory; other numbers denote the memory required for the tree hierarchy and the geometry, i.e. triangles or atlas, respectively.*

	CPU Construction			GPU Construction		
Scene	Q0	Q2	Q4	Q0	Q2	Q4
Dragon	663	530	431	121	101	82
Happy Buddha	1160	901	704	182	152	120
Beast	2395	1758	1472	330	281	239

Table 4: *Build times (in milliseconds) of the CPU and GPU construction using the spatial median and Q0, Q2, and Q4 quality settings.*



a) Glossy refl. + cached diffuse illum. b) Painting on surface c) Photon mapping d) Point cloud e) Point cloud + cached diffuse illum.

Figure 10: *RBVHs have advantages over traditional acceleration structures in several applications: (a) two-bounce global illumination: glossy reflections are computed using 16 secondary rays; diffuse interreflections are progressively computed using 1 sample per frame and cached in the atlas. (b) interactive painting stored directly into the RBVHs atlas. (c) photon mapping where the RBVH is used for tracing* 400*k photons, casting shadow and specular rays; caustics are generated using density estimation of photons stored in the atlas. RBVHs can also be directly created for point cloud representations: (d) and (e) show ray casting of the Bimba con Nastrino and the Old Town Hall in Hannover point sets; Town Hall rendered with cached diffuse environmental lighting. All images were rendered at* 1280×720 *and cropped for this figure. Please see the accompanying video.*

construction timings of the RBVH for three different quality levels. Compared to existing methods for BVH construction on the GPU, our algorithm seems to build RBVHs for comparable scenes faster than the hybrid LBVH [LGS*09], on par with HLBVH [PL10] (the original algorithm constructs an HLBVH for the Dragon model in 81 ms on an NVIDIA GTX 480), but not as fast as the recently introduced more efficient HLBVH [GPM11]. As most of the concepts introduced in [GPM11] (e.g. the work queues) are general, we believe that they can be used to further accelerate our current straightforward GPU construction.

9. Applications

In this section we outline some of the applications of RBVHs. In addition to “just accelerate” ray casting, the representation as height fields and the inherent surface parameterization allow several applications, that otherwise require dedicated methods.

Approximate Ray Tracing The RBVH enables fast, approximate ray tracing with adaptive accuracy. Note that the term “approximate” does not necessarily mean low-quality: the renderings obtained with quality settings Q0 and Q1 are visually almost indistinguishable from triangle-based rendering even for primary rays. However, in certain global illumination computations, e.g. computing indirect lighting or glossy reflections, the full accuracy is not required. In such cases, primary rays can be efficiently replaced by rasterization, and complemented by an RBVH for secondary rays. Fig. 10(a) shows an example of using the RBVH for glossy reflections and caching diffuse interreflections; please see the accompanying video for more examples using the RBVH for progressive, cached ambient occlusion and diffuse environmental lighting.

Using the Atlas for Texturing Due to the implicit surface parameterization, RBVHs inherently provide means to store surface information. For testing purposes, we implemented two practical applications: on-surface painting and photon mapping (Fig. 10 (b) and (c)). Both would require additional parameterizations or data structures with a traditional BVH (or similar accelerator). With our RBVH we can store surface data by casting rays to determine the corresponding atlas texture coordinates, e.g. for a photon-surface intersection, and retrieve the information later during rendering using the algorithm described in Sect. 5.

Point Rendering The RBVH construction supports all primitive types that can be rasterized. We built RBVHs for point clouds obtained by resampling a scanned bust and directly from a 3D laser (Fig. 10 (d) and (e)). During rasterization of the atlas tiles, the point primitives were simply rendered as discs, but any other more sophisticated point rendering technique can be used to improve the surface quality (see [GP07] for an overview). In contrast to other accelerators, RBVHs provide ray casting of different primitives in a unified way.

10. Conclusion and Future Work

In this paper we presented rasterized bounding volume hierarchies for approximate ray casting of triangle and point-based surface representations with adjustable level of detail. Our data structure can be efficiently constructed on the CPU and GPU, and provides an inherent surface parameterization for storing data on the surfaces. We described several means to control the accuracy of the resulting RBVH locally and globally, and determined optimal construction parameters for primary and secondary rays. Hybrid RBVHs avoid excessive ray marching and provide high accuracy by partly keeping the input geometry.

In the future, the RBVH could be further improved by using more elaborate projections, e.g. a reverse perspective [BD06], and signal-specialized tile resolutions. Incremental maintenance via local rasterization and refitting would be an interesting extension, as well as an out-of-core construction for fast visualization of extremely large scenes.

Acknowledgements Old Town Hall point cloud is from the Institute for Cartography and Geoinformatics Hannover; thanks to Michael Wand who provided a version with normals. Bimba con Nastrino and further models are courtesy of AIM@SHAPE, Stanford 3D Scanning Repository, NVIDIA.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proc. of High Performance Graphics* (2009), pp. 145–149. 2, 7

[BD06] BABOUD L., DÉCORET X.: Rendering geometry with relief textures. In *Proc. of Graphics Interface* (2006), pp. 195–201. 2, 9

[BHGS06] BOUBEKEUR T., HEIDRICH W., GRANIER X., SCHLICK C.: Volume-surface trees. *Computer Graphics Forum (Proc. of Eurographics)* (2006), 399–406. 2

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Fast GPU ray tracing of dynamic meshes using geometry images. In *Proc. of Graphics Interface* (2006), pp. 203–209. 2

[CKL*10] CHOI B., KOMURAVELLI R., LU V., SUNG H., BOCCHINO R. L., ADVE S. V., HART J. C.: Parallel SAH k-D tree construction. In *Proc. of High Performance Graphics* (2010), pp. 77–86. 2

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proc. of Symposium on Interactive 3D Graphics and Games* (2009), pp. 15–22. 1, 2

[CNS*11] CRASSIN C., NEYRET F., SAINZ M., GREEN S., EISEMANN E.: Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum (Proc. of Pacific Graphics)* (2011), 207–207. 2

[COCSD03] COHEN-OR D., CHRYSANTHOU Y. L., SILVA C. T., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9* (2003), 412–431. 1

[Dac10] DACHSBACHER C.: Analyzing visibility configurations. *IEEE Transactions on Visualization and Computer Graphics 17*, 4 (2010), 475–486. 1

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum (Proc. of EGSR) 27*, 4 (2008), 1225–1233. 2

[DK08] DAMMERTZ H., KELLER A.: Edge volume heuristic - robust triangle subdivision for improved BVH performance. In *Proc. of IEEE Symposium on Interactive Ray Tracing* (2008), pp. 155–158. 2

[dTWL08] DE TOLEDO R., WANG B., LÉVY B.: Geometry textures and applications. *Computer Graphics Forum 27*, 8 (2008), 2053–2065. 2, 4

[EG07] ERNST M., GREINER G.: Early split clipping for bounding volume hierarchies. In *Proc. of IEEE Symposium on Interactive Ray Tracing* (2007), pp. 73–78. 2

[GP07] GROSS M., PFISTER H.-P.: *Point-Based Graphics*. Elsevier, 2007. 2, 9

[GPM11] GARANZHA K., PANTALEONI J., MCALLISTER D.: Simpler and faster HLBVH with work queues. In *Proc. of High Performance Graphics* (2011), pp. 59–64. 2, 9

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2000. 3

[HB11] HOBEROCK J., BELL N.: Thrust CUDA library. http://code.google.com/p/thrust/, 2011. 7

[HKL10] HANIKA J., KELLER A., LENSCH H. P. A.: Two-level ray tracing with reordering for highly complex scenes. In *Proc. of Graphics Interface* (2010), pp. 145–152. 2

[HSH09] HU L., SANDER P. V., HOPPE H.: Parallel view-dependent refinement of progressive meshes. In *Proc. of Symposium on Interactive 3D Graphics and Games* (2009), pp. 169–176. 2

[KBS11] KALOJANOV J., BILLETER M., SLUSALLEK P.: Two-level grids for ray tracing on GPUs. *Computer Graphics Forum (Proc. of Eurographics) 30* (2011), 307–314. 2

[LD07] LEFEBVRE S., DACHSBACHER C.: Tiletrees. In *Proc. of Symposium on Interactive 3D Graphics and Games* (2007), pp. 25–31. 2, 6

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum (Proc. of Eurographics) 28*, 2 (2009), 375–384. 1, 2, 9

[LK10] LAINE S., KARRAS T.: Efficient sparse voxel octrees. In *Proc. of Symposium on Interactive 3D Graphics and Games* (2010), pp. 55–63. 1, 2

[PFHA10] PANTALEONI J., FASCIONE L., HILL M., AILA T.: PantaRay: fast ray-traced occlusion caching of massive scenes. *ACM Transactions on Graphics (Proc. of SIGGRAPH) 29* (2010), 37:1–37:10. 2

[PGDS09] POPOV S., GEORGIEV I., DIMOV R., SLUSALLEK P.: Object partitioning considered harmful: space subdivision for BVHs. In *Proc. of High Performance Graphics* (2009), pp. 15–22. 2

[PL10] PANTALEONI J., LUEBKE D.: HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proc. of High Performance Graphics* (2010), pp. 87–95. 2, 9

[REG*09] RITSCHEL T., ENGELHARDT T., GROSCH T., SEIDEL H.-P., KAUTZ J., DACHSBACHER C.: Micro-rendering for scalable, parallel final gathering. In *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)* (2009), pp. 132:1–132:8. 1

[SAE93] SHIRMAN L. A., ABI-EZZI S. S.: The cone of normals technique for fast processing of curved patches. *Computer Graphics Forum (Proc. of Eurographics) 12*, 3 (1993), 261–272. 3

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proc. of High Performance Graphics* (2009), pp. 7–13. 2

[SKU08] SZIRMAY-KALOS L., UMENHOFFER T.: Displacement mapping on the GPU - state of the art. *Computer Graphics Forum 27*, 6 (2008), 1567–1592. 2, 6

[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 *Computer Graphics Forum (Proc. of Eurographics)* (2007), pp. 89–116. 1, 2

[YCK*09] YU I., COX A., KIM M. H., RITSCHEL T., GROSCH T., DACHSBACHER C., KAUTZ J.: Perceptual influence of approximate visibility in indirect illumination. *ACM Transactions on Applied Perception 6*, 4 (2009), 24:1–24:14. 1

[YWC*10] YAO C., WANG B., CHAN B., YONG J., PAUL J.-C.: Multi-image based photon tracing for interactive global illumination of dynamic scenes. *Computer Graphics Forum (Proc. of EGSR) 29*, 4 (2010), 1315–1324. 1, 2

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia) 27*, 5 (2008), 126:1–126:11. 2