Compact BVH Storage for Ray Tracing and Photon Mapping

Bartosz Fabianowski and John Dingliana

GV2 Group, Trinity College, Dublin, Ireland

Abstract

*The bounding volume hierarchy (BVH) is becoming an increasingly popular spatial index in high-performance ray tracing [WMH*07]. Specifically, binary hierarchies of axis-aligned bounding boxes are used. We present a compact representation that eliminates redundant information, storing the same BVH nodes in 43%–50% less memory. Algorithms are described that allow the compact representation to efficiently be traversed. We demonstrate and analyze the application to ray tracing and photon mapping on NVidia's CUDA platform [NVI09b], an example of the emerging trend toward manycore architectures. In both cases, memory and bandwidth requirements are reduced. For photon mapping, a significant speed-up is obtained.*

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Ray tracing [Whi80] and photon mapping [Jen96] make extensive use of geometric searches to locate visible surfaces and incident illumination. By building a hierarchical spatial index, the complexity of such searches can be reduced from $O(n)$ to $O(\log n)$. In ray tracing, bounding volume hierarchies (BVHs) [RW80] are gaining popularity, supplementing kd-trees [Ben75] as the indexing method of choice [WMH*07] due to their ability to accommodate dynamic scene changes [LYT06]. Photon mapping, traditionally based on kd-trees, has also recently been shown to efficiently work with BVHs [FD09].

While the term BVH can refer to any hierarchy of arbitrary bounding volumes, in the context of ray tracing, it has become virtually synonymous with a *binary* hierarchy of *axis-aligned bounding boxes* (AABBs) [WMH*07]. In this paper, BVH always refers to a binary hierarchy of AABBs. Such a BVH requires less memory than a kd-tree [GPSS97] but still constitutes an overhead over storing only raw data.

We propose a method that reduces the BVH's memory footprint by 43%–50% while eliminating only redundant information. The lower storage requirements translate into fewer memory reads while searching through the BVH. We demonstrate the effects of this on NVidia's CUDA platform [NVI09b] where main memory accesses suffer from high latency. Ray-tracing is accelerated slightly, photon mapping significantly by our BVH representation.

2. Related work

Hierarchical spatial indexes can significantly speed up geometric searches. As the index is recursively traversed from its root, any node falling outside the query domain may be safely ignored along with its children. An acceleration is obtained if traversal is cheap and culls away large subsets of data. For ray tracing, indexes built according to the surface area heuristic (SAH) [MB90] are highly successful.

An index may partition space or objects. Space partitioning using kd-trees has long been the preferred method for ray tracing. The simplicity of kd-tree traversal enabled real-time ray tracing on clusters of PCs [WSBW01] and in CUDA [BAGJ08]. Kd-trees, however, cannot adapt to dynamic changes [WMH*07]. This has led to a renewed interest in object subdivision using BVHs. Initially proposed as a hierarchy of oriented bounding boxes [RW80], BVHs are now virtually synonymous with binary trees of axis-aligned bounding boxes (AABBs). A BVH can efficiently be refitted or partially rebuilt when the scene changes [LYT06, WBS07] and requires less memory than a kd-tree.

Aiming to combine the simple traversal of kd-trees with the lower memory requirements of BVHs, several authors

have proposed hybrid indexing structures. Bounding interval hierarchies (BIHs) [WK06], H-trees [HHS06] and B-kd-trees [WMS06] follow the same basic idea: Instead of storing the full bounding box of each child, only the bounding planes for a single splitting axis are recorded. Traversal is similar to a kd-tree but refitting and selective rebuilding are possible as sibling nodes may now overlap. Memory requirements are lower than those of a BVH. The main disadvantage is a looser bounding of space. To improve tightness, H-trees contain interspersed AABB nodes cutting off excessive empty space; the single slab hierarchy [EWM08] allows the planes recorded for sibling nodes to lie on different axes; ray-strips [LYM07] and ReduceM [LYTM08] combine a tightly fitting BVH over triangle strips with more lightweight BIHs inside the strips.

BVH memory requirements can significantly be reduced without such fundamental changes to the indexing structure. By reordering the nodes to implicitly encode the hierarchy, some [Smi98] or all [CSE06] child pointers are eliminated. If leaves contain small numbers of elements, it may be more efficient to store inner nodes only and visit all elements when their parent node is traversed [FM86]. A quantized, lower precision encoding of node bounds relative to their parent [Mah05, CSE06] reduces the memory footprint at the cost of introducing conversion operations during traversal. These techniques are also proposed in the context of BVHs for collision detection [Ter01] and coordinate quantization is used with kd-trees as well [HMHB06]. Further savings in storage space are possible by applying a compression algorithm to the BVH [KMKY09], allowing very large models to be processed but incurring a decompression cost.

On previous computer generations, BVH traversal may be dominated by the cost of floating point operations. When a ray's entry point into a node is sought, intersections with the three back-facing bounding planes are irrelevant and may be skipped [Woo90]. For both entry and exit point, intersecting the ray with bounding planes coinciding with the bounds of a parent node is redundant and can be omitted [ST94]. Neither technique provides a reduction in memory footprint or a speed-up on current hardware with fast floating point units.

The motivation for our work is photon mapping. In this algorithm, photons emitted by the light sources are traced through the scene and their hit points recorded in the photon map, a balanced kd-tree [Jen96] or one built according to the voxel volume heuristic [WGS04]. Illumination at a visible surface point is computed by locating the k nearest hits and summing their contributions.

We assign each photon hit a contribution radius during the tracing phase instead [FD09], using a combination of photon path densities [HHK*07] and photon differentials [SFES07]. When computing illumination, not the k nearest photon hits but all those whose contribution radii overlap the visible point are sought. As every photon hit's region of contribution is precisely known, the kd-tree can be replaced with a tight BVH. We employ a fast linear BVH build method originally proposed for ray tracing [LGS*09].

Our target platform is CUDA [NVI09b], a manycore architecture built on current NVidia GPU hardware. High computational speed is provided but memory is scarce and slow. Each main memory access carries a latency of approximately 400 cycles with only minimal caching if it is accessed through the GPU's texturing units [NVI09a]. A BVH representation is therefore needed that has a low memory footprint and requires few reads during traversal.

3. BVH and hybrid structures

In a naïve BVH representation, each node holds six floating point values representing axis-aligned bounding planes and either two child pointers (inner node) or an element pointer and an element count (leaf node). Using 32-bit floating point numbers and pointers, this corresponds to 32 bytes per node. Replacing the pointers with 32-bit indexes yields the same size but allows a few bits to be stolen for bitmasks, as done extensively in more compact representations.

Marking the last element in a leaf instead of storing an element count enables two optimizations. If leaf bounding boxes are not to be recorded [FM86], leaf nodes can be omitted, storing their element index in the parent instead. Alternatively, one of the two child indices may be eliminated by allocating sibling nodes together and referencing both with a single index [Smi98], leading to a node size of 28 bytes. Replacing the 32-bit floating point bounds with quantized values allows for a further reduction in node size to 20 bytes [Ter01] or even just 12 bytes [Mah05].

In the single slab hierarchy [EWM08], only one bounding plane instead of six is stored per node, leading to a size of 8 bytes. The H-tree [HHS06] uses nodes of varying types, each occupying either 16 or 32 bytes. BIH [WK06] and B-kd-tree [WMS06] have a different storage organization, recording both children's bounds in the parent node. The BIH uses one plane per child, translating into 12-byte nodes. In the B-kd-tree, each child is bounded by two planes with 22-bit precision, nodes occupying 16 bytes each.

3.1. Ray tracing traversal

The slabs test [KK86] makes BVH traversal simple and efficient. For each coordinate axis, a node's two bounding planes enclose an interval $t \in [a_k, b_k]$ of the ray $\vec{x} = \vec{o} + t\vec{d}$. The intersection $[a, b]$ of these intervals is the part of the ray that passes through the node (fig. 1, left). If the intervals are disjunct and their intersection is empty, the node is missed (fig. 1, right). For rays of finite length, the intersection is furthermore bounded by $a \geq 0$ and $b \leq b_{max}$.

Algorithm 1 gives a pseudocode implementation. Vectors $\vec{m}$ and $\vec{M}$ describe the node's left and right bounding planes for the three coordinate axes. Where the ray direction has

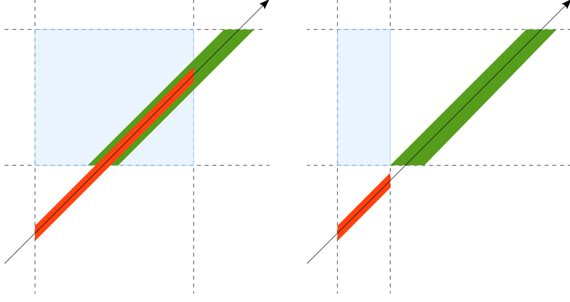

Figure 1: 2*D slabs test. Node bounding planes enclose ray intervals on the x (red) and y (green) axes. Left: The intervals overlap where the ray passes through a node. Right: If the node is missed, they are disjunct.*

negative sign, the front and back planes must be swapped. This is done branchlessly by using the minimum and maximum of t_1, t_2 as the interval bounds (lines 6–7).

Algorithm 1 Slabs test for ray $\vec{o}+t\cdot\vec{d}$ and node $\left(\vec{m},\vec{M}\right)$

1: $a \leftarrow 0$
2: $b \leftarrow b_{max}$
3: **for** $k \in \{x,y,z\}$ **do**
4: $\quad t_1 \leftarrow (\vec{m}_k - \vec{o}_k)\,/\vec{d}_k$
5: $\quad t_2 \leftarrow \left(\vec{M}_k - \vec{o}_k\right)/\vec{d}_k$
6: $\quad a \leftarrow \max(a, \min(t_1,t_2))$
7: $\quad b \leftarrow \min(b, \max(t_1,t_2))$
8: **end for**
9: **if** $a \leq b$ **then**
10: $\quad$ visit node's children
11: **end if**

When the ray intersects a node, both of its children must be visited. One child is tended immediately, the other's index is pushed onto a stack. It is desirable to visit the closer child first. If a surface hit is found there, the ray's b_{max} decreases to the hit distance, culling any nodes behind it. Although the actual order along the ray is unknown, there exists a simple heuristic. The coordinate axis on which each node was originally split to produce its two children is recorded. Children are then visited left-right or right-left according to the ray direction's sign for that coordinate [Mah05].

Quantized BVH representations store child AABBs relative to their parent. In addition to a child index, the parent's complete bounds must therefore be pushed onto the stack during traversal. Other indexing structures that do not record full AABBs also put additional items on the stack. The slabs test is not directly applicable to them as the information to compute an interval $[a_k, b_k]$ for each coordinate axis is not available. Instead, the parent's interval $[a, b]$ is passed on via the stack and clamped by a child's bounding plane(s).

The BIH and the B-kd-tree, which store child bounding planes at their parent, obtain intervals $[a_l, b_l]$ and $[a_r, b_r]$ for both children in this way. Having these intervals available at the parent allows children missed by the ray to be skipped immediately. Additionally, children can be visited in the actual order of their entry points a_l and a_r. By performing the slabs test for two sibling nodes together, the same effect can be achieved for BVHs [AL09].

3.2. Photon mapping traversal

The aim of a BVH photon map traversal is to find the photon hits that overlap a query point $\vec{p}$ [FD09]. To determine whether a node's children are to be visited, the position of $\vec{p}$ relative to the bounds $\vec{m}$, $\vec{M}$ is computed (alg. 2). If $\vec{p}$ lies inside the node, both children are visited, one immediately and the other by pushing its index onto the stack. Traversal order is irrelevant as no further culling can be achieved.

Algorithm 2 Photon map test for point $\vec{p}$ and node $\left(\vec{m},\vec{M}\right)$

1: **if** $\vec{m}_x < \vec{p}_x$ **and**
$\quad \vec{m}_y < \vec{p}_y$ **and**
$\quad \vec{m}_z < \vec{p}_z$ **and**
$\quad \vec{M}_x > \vec{p}_x$ **and**
$\quad \vec{M}_y > \vec{p}_y$ **and**
$\quad \vec{M}_z > \vec{p}_z$ **then**
2: $\quad$ visit node's children
3: **end if**

---

4. Compact BVH

We propose a compact BVH representation based on the insight that in a hierarchy of AABBs, each parent bounding plane is inherited by at least one of the children. As we will show, this limits the number of new bounding planes to six when a node's two children are considered together.

For every coordinate axis, a child may inherit the parent's left bounding plane, right bounding plane, both or none. The number of bounding planes a node does not inherit from its parent thus varies between zero and six. Considering both children together, however, we observe that whenever one introduces a new bounding plane, the other *must* inherit the corresponding plane from the parent (see fig. 2 for 2D examples). If it were not to, the second child would either have to extend outside the parent or be entirely interior like its sibling, the parent thus not providing the tightest possible AABB around them. Neither is allowed in a BVH.

Our BVH representation encodes two sibling nodes in 32 bytes: Six 32-bit floating point values ($\vec{m}$, $\vec{M}$) represent the new minimum and maximum bounding planes for each axis. Two 28-bit indices link to the children of both nodes. A pair of 3-bit masks ($\vec{l}$, $\vec{L}$) assigns each new plane to either the left or right child. The final 2 bits mark the children as either

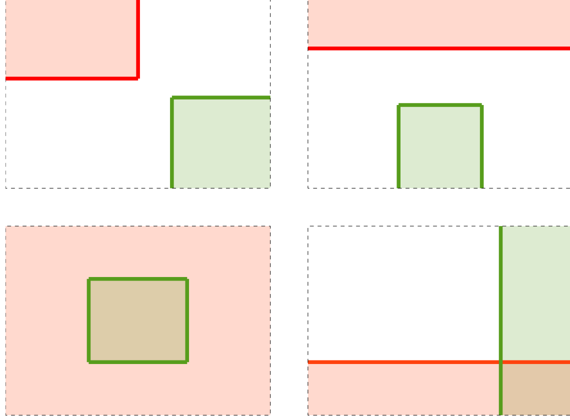

Figure 2: *2D illustration of the new bounding planes introduced (bold) by the two children (pink, green) of a parent node (dashed) in four different cases.*

inner nodes or leaves. In the case of a leaf, the 28-bit index points to the first element it contains.

As described in section 3, storing the same information in a traditional BVH requires at least 28 bytes per node (six bounding planes and a child index). By encoding two children in 32 bytes, we reduce BVH storage requirements by 42.8%. It is important to note that no information is lost and bounding tightness maintained. Only redundant information is eliminated. The overall BVH construction process is unaffected and no additional cost is incurred. Only the way in which sibling node pairs are stored in memory is changed.

Whenever both children inherit the same bounding plane from their parent, no new plane is introduced. To maintain constant storage size per node and avoid any special cases, we always allocate 32 bytes for a pair of siblings and store six planes. Should no new plane be introduced, the parent's bound is simply replicated and assigned to the left child.

4.1. Ray tracing traversal

In our compact representation, inherited bounding planes are not readily available when traversing a node. The slabs test (alg. 1) must be modified to account for this and to compute two intervals $[a_l, b_l]$, $[a_r, b_r]$ for the two children stored together. As with hybrid indexing structures, we maintain the current node index and interval $[a, b]$ during traversal, pushing all three values onto the stack when traversal branches (see section 3.1).

Traversal of a sibling node pair begins by initializing the intervals $[a_l, b_l]$ and $[a_r, b_r]$ to the parent's $[a, b]$ (alg. 3, lines 1–2). If this was popped off the stack, the intervals are additionally clamped against the ray's current length b_{max}. For each of the three coordinate axes, values t_1 and t_2 are computed exactly as in the slabs test (alg. 1 and 3, lines 4–5). Lines 7–8 distribute these interval bounds to the two children. When $\vec{l}_k = 1$, the stored minimum bounding plane belongs to the left child and t_1 is written to $t_{1,l}$. Otherwise, the plane belongs to the right child and t_1 is written to $t_{1,r}$. The value of t_2 is analogously copied into $t_{2,l}$ or $t_{2,r}$.

The child inheriting the parent's plane is assigned $-t_3$ instead of t_1 and t_3 instead of t_2. This value, calculated in line 6, leads to an unbounded interval with the open side where the inherited plane would have been. Ignoring an inherited bound in this way introduces no error as its effect is already contained in the interval $[a, b]$ passed down during traversal. Lines 9–12 are identical to lines 6–7 of the original slabs test, replicated to cover both children. Results are evaluated from line 14 onward, visiting the children for which a non-empty interval has been computed in the order of their entry points a_l, a_r along the ray.

Algorithm 3 Compact BVH slabs test for ray $\vec{o} + t \cdot \vec{d}$, parent interval $[a, b]$ and child nodes encoded as $\vec{m}$, $\vec{M}$, $\vec{l}$, $\vec{L}$

1: $a_l, a_r \leftarrow a$
2: $b_l, b_r \leftarrow \min(b, b_{max})$
3: **for** $k \in \{x, y, z\}$ **do**
4: $\quad t_1 \leftarrow (\vec{m}_k - \vec{o}_k) / \vec{d}_k$
5: $\quad t_2 \leftarrow \left(\vec{M}_k - \vec{o}_k\right) / \vec{d}_k$
6: $\quad t_3 \leftarrow \infty / \vec{d}_k$
7: $\quad (t_{1,l}, t_{1,r}) \leftarrow$ **if** $\vec{l}_k$ **then** $(t_1, -t_3)$ **else** $(-t_3, t_1)$
8: $\quad (t_{2,l}, t_{2,r}) \leftarrow$ **if** $\vec{L}_k$ **then** (t_2, t_3) **else** (t_3, t_2)
9: $\quad a_l \leftarrow \max\left(a_l, \min\left(t_{1,l}, t_{2,l}\right)\right)$
10: $\quad b_l \leftarrow \min\left(b_l, \max\left(t_{1,l}, t_{2,l}\right)\right)$
11: $\quad a_r \leftarrow \max\left(a_r, \min\left(t_{1,r}, t_{2,r}\right)\right)$
12: $\quad b_r \leftarrow \min\left(b_r, \max\left(t_{1,r}, t_{2,r}\right)\right)$
13: **end for**
14: **if** $a_l \leq b_l$ **and** $a_r \leq b_r$ **then**
15: $\quad$ visit both children
16: **else if** $a_l \leq b_l$ **then**
17: $\quad$ visit left child
18: **else if** $a_r \leq b_r$ **then**
19: $\quad$ visit right child
20: **end if**

When popping an interval $[a, b]$ off the stack, we first check whether $a > b_{max}$. If so, a surface hit has already been found closer to the ray origin than the entry point and the node at the top of the stack can safely be ignored.

Our traversal algorithm performs more operations than the original slabs test but also handles two children at once. Main memory bandwidth is reduced as 32 bytes instead of 2×28 bytes are read per pair of nodes. The conditional statements in lines 7–8 can be expressed using the ternary operator and compiled into branchless code on hardware supporting predicated instructions.

4.2. Photon mapping traversal

The main motivation for our work are BVH photon maps. Algorithm 4 illustrates the traversal efficiency of our compact BVH representation in photon mapping. Compared to algorithm 2, only ten additional bitwise operations (six shifts $\ll$, one negation $\neg$, one conjunction $\vee$, two disjunctions $\wedge$) are necessary to traverse two sibling nodes instead of one.

Line 1 constructs a bitmask *miss* indicating which of the six bounding planes the query point $\vec{p}$ lies outside. The result of each comparison, when cast into an integer, is either 0 or 1. By shifting these into position, the mask is built without branching. In line 2, a corresponding bitmask $left$ is assembled that indicates whether each of the planes bounds the left or right child. This step is necessary as we store the information in two 3-bit fields $\vec{l}$, $\vec{L}$ stolen from the child indices.

If *miss* contains only zeroes, neither of the children is missed and both must be visited (lines 3–4). Otherwise, the query point lies outside at least one of the children. If no bit is set in both $left$ and *miss*, $\vec{p}$ is inside all planes bounding the left child (lines 5–6). Similarly, if no bit is set in both the complement of $left$ and *miss*, $\vec{p}$ is inside the right child (lines 7–8). Should neither case apply, both children are missed.

Algorithm 4 Compact BVH photon map test for query point $\vec{p}$ and child nodes encoded as $\vec{m}$, $\vec{M}$, $\vec{l}$, $\vec{L}$

1: $miss \leftarrow \vec{m}_x \geq \vec{p}_x \ll 5 \vee$
   $\vec{m}_y \geq \vec{p}_y \ll 4 \vee$
   $\vec{m}_z \geq \vec{p}_z \ll 3 \vee$
   $\vec{M}_x \leq \vec{p}_x \ll 2 \vee$
   $\vec{M}_y \leq \vec{p}_y \ll 1 \vee$
   $\vec{M}_z \leq \vec{p}_z$
2: $left \leftarrow \vec{l} \ll 3 \vee \vec{L}$
3: **if not** *miss* **then**
4: visit both children
5: **else if not** $left \wedge miss$ **then**
6: visit left child
7: **else if not** $\neg left \wedge miss$ **then**
8: visit right child
9: **end if**

Contrary to ray tracing traversal (section 4.1), no information other than the node index needs to be pushed on the stack when traversal branches. Whenever algorithm 4 is executed for a pair of sibling nodes, it is already known that the query point $\vec{p}$ lies inside the six planes bounding their parent. For any planes inherited by the children, the corresponding *miss* bit, if computed or stored, would thus always be zero.

5. Results

We have evaluated our compact BVH representation in the context of both ray tracing and photon mapping, using six scenes of varying complexity (fig. 3). Results are averaged over a flight through each scene, ensuring that the impact of different viewpoints is taken into account. The benchmark platform is CUDA [NVI09b] running on an NVidia GeForce GTX 280 under GNU/Linux.

5.1. Ray tracing

For ray tracing, we evaluate our BVH representation in a reimplementation of the currently fastest CUDA ray tracer [AL09]. The BVH is built offline, using the SAH [MB90] and early split clipping [EG07] to prevent degradation when large triangles are present. Images are rendered at 1024^2 resolution with one eye ray and one shadow ray per pixel.

Each ray is traced by an independent thread using a separate traversal stack. As the fast shared memory available in CUDA is too small, stacks are located in slow main memory [BAGJ08]. Persistent threads [AL09] avoid an inefficiency in the GPU's scheduler, launching only as many threads as are required to saturate the parallel processing units and then fetching rays from a global job queue.

The baseline BVH representation we compare against [AL09] stores and traverses pairs of sibling nodes together. For each pair, 2×6 bounding planes and two child indices are recorded, occupying 56 bytes. Our representation requires only 32 bytes per node pair, reducing BVH memory usage by 43% (tab. 1, columns 2–3). As noted in section 4, there is no cost overhead for constructing this representation.

By considering sibling nodes in pairs, both representations yield the same traversal order. However, bandwidth requirements differ. Loading a bounding box pair from main memory uses either 56 or 32 bytes. The information is read via the GPU's texturing units, providing a minimal amount of caching [NVI09a]. As a texture read can transfer up to 16 bytes, the baseline needs 4 reads per node pair, our representation only 2. The average bandwidth consumed per pixel for loading BVH nodes is shown in columns 4–5.

When pushing onto or popping off the stack, the basline needs to transfer only a 4-byte node index. Our BVH representation requires $3 \times 4 = 12$ bytes to be stored, consuming more bandwidth per stack access. However, 12 bytes can still be read or written as a single transaction, incurring main memory latency only once. We find that the baseline attains a small speed-up if the entrypoint is stored on the stack, allowing nodes with $a > b_{max}$ to be skipped (see section 3.1). The bandwidth figures for stack access in columns 6–7 reflect this use of 8-byte stack entries in the baseline.

Columns 8–9 summarize the difference in bandwidth requirements during BVH traversal. In the final three columns, the resulting frame rates are given. We see a bandwidth reduction of 37% on average, coinciding with a slight increase in frame rate. As BVH traversal is only one component of the rendering algorithm, its potential influence on the overall frame rate is limited. Furthermore, speed-ups may be hampered by one of three effects: The ray tracer is not bandwidth

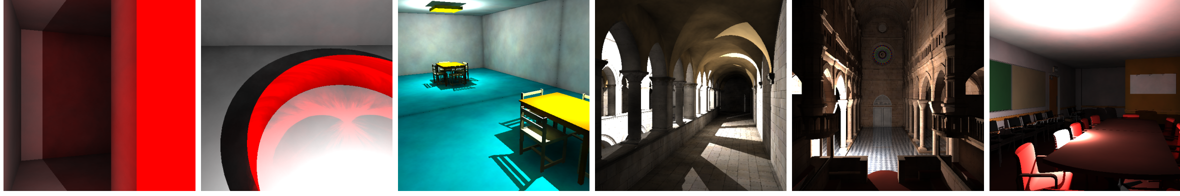

Figure 3: *The six benchmark scenes used, shown here with photon mapping: Wall, Ring, Scene-6, Sponza, Sibenik, Conference.*

Scene	Triangles	BVH memory		BVH traversal bandwidth per pixel						FPS		
		Base	Compact	BVH_B	BVH_C	$Stack_B$	$Stack_C$	Δ		Base	Compact	Δ
Wall	30	504	288	489.0	279.4	29.1	55.5	-183.2	-35%	103.00	106.47	3.4%
Ring	138	2576	1472	682.4	389.9	25.7	47.2	-270.9	-38%	107.65	107.74	0.1%
Scene 6	804	16296	9312	1101.6	629.5	59.7	114.5	-417.3	-36%	83.65	85.54	2.3%
Sponza	76107	1314376	751072	2927.9	1673.1	154.6	299.6	-1109.7	-36%	42.45	42.64	0.4%
Sibenik	76643	1592360	909920	3779.9	2159.9	175.2	320.2	-1474.9	-37%	34.05	34.80	2.2%
Conference	282755	4181856	2389632	2343.7	1339.2	103.5	195.9	-912.0	-37%	50.19	51.00	1.6%
Column	1	2	3	4	5	6	7	8	9	10	11	12

Table 1: *Ray tracing results, memory and bandwidths in bytes: BVH footprints; bandwidth requirements per pixel to read nodes (BVH) and access the stack (Stack) for the baseline (B) and compact (C) BVH representations; frame rates for both.*

bound; texture caching successfully eliminates main memory reads; stack accesses incur more latency than expected.

If the ray tracer is computation-bound, a more compact BVH representation requiring additional operations to traverse must necessarily make it slower. The advantage is in reduced storage, allowing larger models to be handled. Texture caching may be successful because of our benchmark scenes' relatively low complexity. With larger models, cache misses become more likely. These will cause a higher number of slow main memory reads with the baseline than with our compact BVH representation. Finally, transferring three 32-bit values to and from the stack should occur in a single transaction. It is unclear whether the CUDA compiler actually issues one or three transactions in this case. If more than one transaction is used, our traversal algorithm incurs the unnecessary penalty of additional latencies.

5.2. Photon mapping

BVH photon maps were the initial motivation for our work. The photon mapping testbed is therefore older, based on an earlier incarnation of our CUDA ray tracer that uses a kd-tree for scene traversal and no persistent threads. Images are rendered at 512^2 resolution with one eye ray, one specular reflection ray and a shadow ray for each per pixel. At the eye and reflection ray hit points, photon hits are retrieved from a photon map. To highlight traversal performance, photon maps are constructed offline using a linear BVH build method [LGS*09]. As previously demonstrated [FD09], the build can also be done per-frame at interactive rates.

The baseline representation is a classical BVH, storing six bounding planes and a child index for each node. By padding the node size from 28 to 32 bytes, it can more efficiently be loaded using two texture reads. The compact BVH requires 32 bytes per pair of sibling nodes. Photon hit counts and BVH sizes are given in columns 1–3 of table 2. Our representation reduces the BVH memory footprint by nearly 50%.

During each photon map traversal, an average of 344 nodes are visited (column 4). By storing sibling nodes together, our BVH representation significantly reduces the number of stack accesses required for this traversal (columns 5–6). In a classical BVH, an inner node found to contain the query point $\vec{p}$ causes one of its children to be visited immediately, the other pushed onto the stack. With our representation, the children's bounds are tested immediately. Only if both actually contain $\vec{p}$ is a child pushed onto the stack. Otherwise, traversal proceeds directly with the child that contains $\vec{p}$ or, if neither does, by popping from the stack.

For both BVH representations, only a 4-byte node index needs to be put on the stack. The more compact node storage and lower number of stack accesses translate into bandwidth savings of 53% on average for BVH traversal with our representation (columns 7–10). Columns 11–13 demonstrate that besides reducing storage and bandwidth requirements, rendering speeds are also improved.

Photon map traversal is again only one part of a more complex rendering algorithm. To better illustrate its speedup, we additionally compare frame rates with photon retrieval disabled. Rays are traced and the photon map is traversed as before. However, the computation of each photon's actual illumination contribution is omitted. By removing this

Scene	Photon	BVH memory		Operations / trav			BVH BW / trav				FPS			
	hits	B	C	V	S_B	S_C	B	C	Δ		B	C	Δ_1	Δ_2
Wall	20256	249k	125k	201.9	100.9	31.7	7268	3484	-3784	-52%	7.04	7.46	6.0%	52%
Ring	74101	772k	386k	492.7	246.3	70.8	17737	8450	-9287	-52%	4.91	5.42	10.4%	88%
Scene 6	160298	703k	351k	303.6	151.8	47.0	10930	5234	-5697	-52%	3.52	3.71	5.4%	88%
Sponza	292511	5479k	2740k	457.8	228.9	62.4	16482	7824	-8657	-53%	5.02	6.38	27.1%	113%
Sibenik	413316	7789k	3894k	160.5	80.2	31.7	5776	2821	-2956	-51%	9.12	11.32	24.1%	87%
Conference	97911	1153k	577k	445.6	222.8	63.8	16042	7640	-8402	-52%	4.83	5.50	13.9%	65%
Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Table 2: *Photon mapping results, memory and bandwidths in bytes: Number of photon hits; BVH footprints for baseline (B) and compact (C) representations; nodes visited (V) and pushed onto the stack (S); total bandwidth requirements per photon map traversal; frame rates; frame rate difference with photon retrieval disabled.*

operation's masking effect, the speed-up in the traversal becomes more apparent, as evidenced by the final column.

6. Conclusions and future work

We have described a new compact representation for binary hierarchies of AABBs, a form of BVHs frequently used in ray tracing [WMH*07] and recently also introduced for photon mapping [FD09]. By eliminating only redundant information, full bounding tightness is maintained while reducing the BVH's memory footprint by 43%–50%. BVH construction time is not adversely affected.

We have presented efficient traversal algorithms for ray tracing and photon mapping. Their implementations on a current NVidia GPU show reduced bandwidth requirements over existing BVH representations. The resulting speed-ups are slight for ray tracing and significant for photon mapping. With processing power continuing to increase faster than memory speeds, the lower bandwidth requirements can be expected to have an even more significant impact in the future, translating into larger speed-ups. We plan to reevaluate our traversal algorithms on NVidia's next generation Fermi architecture [NVI09c] when it is released.

Our compact representation reduces the memory occupied by BVH nodes. The data that is indexed by the BVH remains unaffected and retains its full storage and bandwidth requirements. We would like to combine the compact BVH with a more compact representation of actual data, such as ray strips [LYM07], to efficiently visualize larger models.

Early split clipping [EG07] improves the quality of the spatial index. However, it also introduces an additional level of indirection between the BVH nodes and the scene elements. These added references are not compacted in our current representation. In the future, we would also like to investigate their more efficient storage.

Finally, our compact representation can be applied outside of computer graphics. Collision detection is the most prominent field also using binary hierarchies of AABBs [Ter01]. It will be interesting to evaluate the impact of our BVH representation on a high-performance collision detection system.

7. Acknowledgments

The models used are courtesy of Peter Shirley (Scene 6), Marko Dabrovic (Sponza, Sibenik), Anat Grynberg and Greg Ward (Conference Room). We thank the reviewers for their comments and suggestions.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *HPG* (2009), pp. 145–149. 3, 5

[BAGJ08] BUDGE B., ANDERSON J., GARTH C., JOY K.: A straightforward CUDA implementation for interactive ray-tracing. In *RT* (2008), p. 178. 1, 5

[Ben75] BENTLEY J.: Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517. 1

[CSE06] CLINE D., STEELE K., EGBERT P.: Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools 11*, 4 (2006), 61–71. 2

[EG07] ERNST M., GREINER G.: Early split clipping for bounding volume hierarchies. In *RT* (2007), pp. 73–78. 5, 7

[EWM08] EISEMANN M., WOIZISCHKE C., MAGNOR M.: Ray tracing with the single slab hierarchy. In *VMV* (2008), pp. 373–381. 2

[FD09] FABIANOWSKI B., DINGLIANA J.: Interactive global photon mapping. In *EGSR* (2009), pp. 1151–1159. 1, 2, 3, 6, 7

[FM86] FABBRINI F., MONTANI C.: Autumnal quadtrees. *The Computer Journal 29*, 5 (1986), 472–474. 2

[GPSS97] GÜNTHER J., POPOV S., SEIDEL H.-P., SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *RT* (2997), pp. 113–118. 1

[HHK*07] HERZOG R., HAVRAN V., KINUWAKI S., MYSZKOWSKI K., SEIDEL H.-P.: Global illumination using photon ray splatting. In *Eurographics* (2007), pp. 503–513. 2

[HHS06] HAVRAN V., HERZOG R., SEIDEL H.-P.: On the fast construction of spatial hierarchies for ray tracing. In *RT* (2006), pp. 71–80. 2

[HMHB06] HUBO E., MERTENS T., HABER T., BEKAERT P.: The quantized kd-tree: Efficient ray tracing of compressed point clouds. In *RT* (2006), pp. 105–113. 2

[Jen96] JENSEN H.: *The Photon Map in Global Illumination*. PhD thesis, Technical University of Denmark, Lyngby, Denmark, 1996. 1, 2

[KK86] KAY T., KAJIYA J.: Ray tracing complex scenes. In *SIGGRAPH* (1986), pp. 269–278. 2

[KMKY09] KIM T.-J., MOON B., KIM D., YOON S.-E.: RACBVHs: Random-accessible compressed bounding volume hierarchies. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009). 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. In *Eurographics* (2009), pp. 375–384. 2, 6

[LYM07] LAUTERBACH C., YOON S.-E., MANOCHA D.: Ray-strips: A compact mesh representation for interactive ray tracing. In *RT* (2007), pp. 19–26. 2, 7

[LYT06] LAUTERBACH C., YOON S.-E., TUFT D.: RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *RT* (2006), pp. 39–46. 1

[LYTM08] LAUTERBACH C., YOON S.-E., TANG M., MANOCHA D.: ReduceM: Interactive and memory efficient ray tracing of large models. In *EGSR* (2008), pp. 1313–1321. 2

[Mah05] MAHOVSKY J.: *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, Calgary, Alberta, Canada, 2005. 2, 3

[MB90] MACDONALD J., BOOTH K.: Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3 (1990), 153–166. 1, 5

[NVI09a] NVIDIA CORPORATION: NVIDIA CUDA C programming best practices guide CUDA toolkit 2.3. 2009. 2, 5

[NVI09b] NVIDIA CORPORATION: NVIDIA CUDA programming guide version 2.3. 2009. 1, 2, 5

[NVI09c] NVIDIA CORPORATION: NVIDIA's next generation CUDA compute architecture: Fermi. 2009. 7

[RW80] RUBIN S., WHITTED T.: A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH* (1980), pp. 110–116. 1

[SFES07] SCHJØTH L., FRISVAD J., ERLEBEN K., SPORRING J.: Photon differentials. In *GRAPHITE* (2007), pp. 179–186. 2

[Smi98] SMITS B.: Efficiency issues for ray tracing. *Journal of Graphics Tools 3*, 2 (1998), 1–14. 2

[ST94] STÜRZLINGER W., TOBLER R.: Two optimization methods for raytracing. In *SCCG* (1994), pp. 104–107. 2

[Ter01] TERDIMAN P.: Memory-optimized bounding-volume hierarchies. 2001. 2, 7

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 6:1–6:18. 1

[WGS04] WALD I., GÜNTHER J., SLUSALLEK P.: Balancing considered harmful – faster photon mapping using the voxel volume heuristic –. In *Eurographics* (2004), pp. 595–603. 2

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (1980), 343–349. 1

[WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In *EGSR* (2006), pp. 139–149. 2

[WMH*07] WALD I., MARK W., HUNT W., GÜNTHER J., PARKER S., BOULOS S., SHIRLEY P., IZE T.: State of the art in ray tracing animated scenes. In *Eurographics State of the Art Reports* (2007), pp. 89–116. 1, 7

[WMS06] WOOP S., MARMITT G., SLUSALLEK P.: B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *GH* (2006), pp. 67–77. 2

[Woo90] WOO A.: *Graphics Gems*. Academic Press, 1990, ch. Fast Ray-Box Intersection, pp. 395–396. 2

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. In *Eurographics* (2001), pp. 153–164. 1