Ray Tracing with the BSP Tree

Thiago Ize
University of Utah

Ingo Wald
Intel Corp

Steven G. Parker
NVIDIA
University of Utah

Abstract

One of the most fundamental concepts in computer graphics is binary space subdivision. In its purest form, this concept leads to binary space partitioning trees (BSP trees) with arbitrarily oriented space partitioning planes. In practice, however, most algorithms use kd-trees—a special case of BSP trees that restrict themselves to axis-aligned planes—since BSP trees are believed to be numerically unstable, costly to traverse, and intractable to build well. In this paper, we show that this is not true. Furthermore, after optimizing our general BSP traversal to also have a fast kd-tree style traversal path for axis-aligned splitting planes, we show it is indeed possible to build a general BSP based ray tracer that is highly competitive with state of the art BVH and kd-tree based systems. We demonstrate our ray tracer on a variety of scenes, and show that it is always competitive with—and often superior to—state of the art BVH and kd-tree based ray tracers.

1 Introduction

High performance ray tracing usually requires a spatial and/or hierarchical data structure for efficiently searching the primitives in the scene. One of the most fundamental concepts in these data structures is "binary space partitioning" — successively subdividing a scene's bounding box with planes until certain termination criteria are reached. The resulting data structure is called a binary space partitioning tree or BSP tree.

The flexibility to place splitting planes where they are most effective allows BSP trees to adapt very well even to complex scenes and highly uneven scene distributions, usually making them highly effective. For a polygonal scene, a BSP tree can, for example, place its splitting planes exactly through the scene's polygons, thus exactly enclosing the scene. In addition, their binary nature makes them well suited to hierarchical divide-and-conquer and branch-and-bound style traversal algorithms that in practice are usually quite competitive with other algorithms. Thus, in theory BSP trees are simple, elegant, flexible, and highly efficient.

However, "real" BSP trees with arbitrarily oriented planes are used very rarely, if at all in practice. In collision detection, BSP ray-shooting queries [1] do exist, but if applied to generating images with ray tracing, would perform quite poorly. In ray tracing, we are not aware of a single high-performance ray tracer that uses arbitrarily oriented planes. Instead, ray tracers typically use kd-trees, which are a restricted type of BSP tree in which only axis-aligned splitting planes are allowed[1]. Kd-trees provide storage and computation advantages but do not conform as well to scene geometry.

This focus solely on axis-aligned planes is somewhat surprising, since in theory, a BSP should be far more robust to arbitrary geometry. In particular, it is relatively easy to "break" a kd-tree with non-axis aligned geometry: consider a long, skinny object that is rotated. While aligned to an axis, the kd-tree would be highly efficient,

Figure 1: The 283 thousand triangle conference room and three views of the 2.2 million triangle UC Berkeley Soda Hall model, rendered with a path tracer. For these views, our BSP outperforms a kd-tree by 1.1×, 1.3×, 1.2×, and 2.5× respectively (and by 1.1×, 1.8×, 1.2×, and 2.4× respectively, for ray casting)

but when oriented diagonally the kd-tree would be severely hampered. With a BSP, in contrast, rotating the object should not have any effect at all. In addition, since every kd-tree is expressible as a BSP tree, a properly built BSP tree could never perform worse than a kd-tree, while having the additional flexibility to place even better split planes that a kd-tree could not. Despite being theoretically superior on all counts, they are nevertheless not used in practice.

We believe that this discrepancy between theory and praxis stems from three widespread assumptions: First, it is believed that traversing a BSP tree is significantly more costly than traversing a kd-tree, since computing a ray's distance to an arbitrarily oriented plane requires a dot product and a division, while for a kd-tree it requires only a subtraction and a multiplication. Furthermore, a BSP tree node requires more storage than a kd-tree node. Thus, any potential gains through better oriented planes would be lost due to traversal cost. Second, the limited accuracy of floating point numbers is believed to make BSP trees numerically unstable, and thus, useless for practical applications. Third, the (much!) greater flexibility in placing a split plane makes building (efficient) BSP trees much harder than building kd-trees: For a kd-tree, when splitting a node with n triangles there are only $6n$ plane locations where the number of triangles to the left and right of the plane changes; for a BSP tree, the added two dimensions for the orientation of the plane produces $O(n^3)$ possible planes. Instead, recursively placing planes through

[1] Somewhat surprisingly, while there are many papers on ray tracing with "BSP trees" ([8, 10, 18] to name just a few), *none* of the papers we found actually uses arbitrarily oriented planes!

randomly picked triangles is likely to produce bad and/or degenerate trees, and would likely result in poor ray tracing performance. Thus, it is generally believed that naively building BSPs results in bad BSPs, while building good BSP trees is intractable.

In this paper, we show that none of these assumptions is completely true, and that a building a BSP based ray tracer—when using the right algorithms for building and traversing—is indeed possible, and that it can be quite competitive to BVH or kd-tree based ray tracers: While the numerical accuracy issues requires proper attention during both build and traversal, we will demonstrate our ray tracer on a wide variety of scenes, and with both ray casting and path tracing. We also show that the well-known surface area heuristic (SAH) easily generalizes to BSP trees, and that SAH optimized BSP trees—though significantly slower to build than kd-trees—can indeed be computed with tractable time and memory requirements. Based on a number of experiments, we show that our BSP based ray tracer is at least as stable as a kd-tree based one, that it is always at least roughly as fast as a best-of-breed kd-tree, and that it can often outperform it.

2 Background

2.1 Binary Space Partitioning Trees

BSPs were first introduced to computer graphics by Fuchs et al. to perform hidden surface removal with the painter's algorithm [4]. As its name describes, a BSP is a binary tree that partitions space into two half-spaces according to a splitting plane. This property allows BSPs to be used as a binary search tree to locate objects embedded in that space. If a splitting plane intersects an object, the object must be put on both sides of the plane. This property can in theory lead to poor quality BSP trees with $\Omega(n^2)$ nodes for certain configurations of n non-intersecting triangles in $\Re^3$ [13]. However, in practice this will not occur and space should be closer to linear. In fact, if we only have *fat* triangles — which means there are no long and skinny triangles — then there exist BSP trees with linear size [3]. Assuming the tree is well balanced, query time is usually $O(\log n)$. Build time depends on the algorithm used to pick the splitting planes. An algorithm that chose the optimal splitting planes would likely take at least exponential time which is not feasible. For this reason, splitting planes are usually chosen at random, to divide space or the elements in half, or using some other heuristic such as the greedy SAH [5, 12].

2.2 Ray Tracing Acceleration Structures

There are currently three main classes of acceleration structures used in modern ray tracers: kd-trees [16, 17], uniform grids (possibly with multiple levels) [23], and axis-aligned bounding volume hierarchies (BVHs) [11, 21]. These acceleration structures can be further accelerated by tracing packets of coherent rays, using SIMD extensions on modern CPUs [19], and by using the bounding frustum [16, 23] or interval arithmetic (IA) [21] to traverse the packet through the acceleration structure by using only a small subset of the rays in the packet. However, it is unclear how well SIMD, packet, and frustum techniques will perform for secondary rays and so it is important to still look into what the best single ray acceleration structure can do.

Kd-trees are a variant of BSP trees where the splitting planes are only axis-aligned. This leads to an extremely fast tree traversal. However, unlike an optimal BSP, an optimal kd-tree will not be able to partition all the objects into individual leaves since axis-aligned splits might not exist that cleanly partition triangles. This results in more ray-object intersection tests (albeit potentially fewer traversal steps) that must be performed. The prevailing belief has been that the faster traversal of the kd-tree makes up for the increase in intersection tests; however, in this paper we show that is not always the case.

Kammaje and Mora showed that using a restricted BSP with many split planes resulted in fewer node traversals and triangle intersections than their kd-tree, but for almost all their test scenes, the BSP resulted in slower rendering times [10]. One possible reason for this is that the traversal differs from a kd-tree traversal too much and ends up being too expensive. Another reason might be that at the lower levels of the tree, the restricted BSP still suffers from the same problem as the kd-tree, in that it is not able to cleanly split triangles apart. In a certain sense their data structure inherits the disadvantages of both kd-trees (not being able to perfectly adapt) and BSP trees (costly traversal).

2.3 Surface Area Heuristic

When building an adaptive data structure like a BSP, kd-tree, or BVH, the actual way that a node gets partitioned into two halves can have a profound impact on the number of expected traversal steps and primitive intersections. For kd-trees, the best known method to build trees with minimal expected cost is the greedy surface area heuristic (SAH). The greedy SAH estimates the cost of a split plane by assuming that each of the two resulting halves l and r would become leaves. Then, using geometric probability (see [6] for a more detailed explanation) the expected cost of splitting node p is

$$C_p = \frac{SA(v_l)}{SA(v_p)} n_l C_i + \frac{SA(v_r)}{SA(v_p)} n_r C_i + C_t$$

where $SA()$ is the surface area, n is the number of primitives, C_i is the cost of intersecting a primitive, and C_t is the cost of traversing a node. Though originally invented for BVHs and commonly used for kd-trees, Kammaje and Mora recently showed how it can also be applied to restricted BSPs [10] and the same theory works for BSP trees too, so we will use it here.

Wald and Havran showed that an SAH kd-tree can be built in $O(n \log n)$, although not at interactive rates [22]. Independently, Hunt [9] and Popov [14] showed that the SAH kd-tree build can be made faster by approximating the SAH cost with a subset of the candidate splits at the expense of a slight hit in rendering performance.

3 Building the BSP

The BSP build is conceptually very similar to the standard kd-tree build. The differences occur mainly in the implementation in order to handle the decreased numerical precision, computing the surface areas of a general polytope (as opposed to an axis-aligned box), and deciding which general split planes to use.

3.1 Intersection of Half-Spaces

The SAH requires the surface area of a node. For kd-trees this is trivial to calculate, but a node in a BSP is defined by the intersection of the half-spaces that make up that node. The intersection of half-spaces defines a polyhedron, and more specifically in the case of BSPs, a convex polytope since it can never be empty or unbounded (we place a bounding box over the entire object). We thus need to find the polytope in order to compute the surface area. As with Kammaje and Mora, we form the polytope by clipping the previous polytope with the splitting plane to get two new polytopes, and then simply sum the areas of the polygonal faces [10].

3.2 Handling Numerical Precision

We use the SAH, which requires us to compute the surface area of each node. As in [10], we find the surface area by computing the two new polytopes formed by cutting the parent node with the splitting plane. Care must be taken to make sure that the methods for computing the new polytope are robust and do not break down for very thin polytopes, which are quite common. Numerical imprecision makes it difficult to determine whether a vertex lies exactly on a plane. Assume we have a triangulated circle and a plane that is supposed to lie on that circle. If it is axis-aligned, for instance on the y-axis, determining whether each vertex lies on the plane is simple since all

the vertices will have the same y value. But if we rotate the circle by 45 degrees so that the plane is now on $x = y$, the plane equation may not evaluate exactly to zero for these vertices. Consequently, we need to check for vertices that are within some epsilon of the plane. The distance of the vertices from the true plane is small; however, because we must use the planes determined from the triangles, the error can become much larger. If the triangles are very small and the circle they lie on very large, then a plane defined by a triangle on one side of the circle might end up being extremely far away from a triangle on the opposite side. If the epsilon is too small, tree quality will suffer since many planes will be required to bound the triangle faces when a single plane would have sufficed. This forces us to pick a much larger epsilon. However, if the epsilon is too large then a node will not be able to accurately bound its triangles. This can cause rendering artifacts during traversal. For this reason we must also include this epsilon distance in the ray traversal.

3.3 BSP Surface Area Heuristic

We support a tree with a combination of general BSP and axis-aligned kd-tree nodes. Therefore we have two node traversal costs, C_{BSP} and $C_{\text{kd-tree}}$, that we must use with the SAH. However, using these directly in the SAH results in BSP nodes being used predominantly over the kd-tree nodes even when C_{BSP} is set to be many times greater than $C_{\text{kd-tree}}$. The reason for this is the assumption that a split creates leaves with costs linear in the number of triangles. When there is only one constant traversal cost, this works well since the traversal cost affects only the decision of performing the split versus terminating and creating a leaf node. However, in our case we need to use the traversal cost not just to determine when to create a leaf, but also whether a BSP split, with its more expensive traversal, is worth using over a cheaper kd-tree traversal. Unfortunately, this linear intersection cost will quickly dwarf the constant traversal cost, causing the optimal split to be based almost entirely on which splitting plane results in fewer triangle intersection tests. We handle this problem by making C_{BSP} vary linearly with the number of primitives so that $C_{\text{BSP}} = \alpha C_i(n-1) + C_{\text{kd-tree}}$, where α is a user tunable parameter (we use 0.1). If after evaluating all splitting planes we find that the cost of splitting is not lower than the cost of creating a leaf node, then we evaluate all the BSP splitting planes again but this time with a fixed C_{BSP}. In practice this works much better than using a fixed cost, however it is possible that an even better heuristic exists.

3.4 Which Planes to Test

At each node we must decide with which splitting planes to evaluate the SAH. While a kd-tree has infinitely many axis-aligned splitting planes to choose from, Havran showed that only those splitting planes that were tangent to the clipped triangles (perfect splits) were actually required [6]. Since there are only 6 axis-aligned planes per triangle that fulfill this criteria, this allowed for $O(n)$ split candidates per node. Directly extending Havran's results to handling the additional degrees of freedom in choosing the normal would result in $O(n^3)$, or possibly even higher, split candidates, which is often impractical.

To maintain a practical build time, we limit ourselves to only $O(n)$ split candidates for each BSP node, at the expense of possibly missing some better splitting planes. For each triangle in a node, the split candidates we pick are: the plane that defines the triangle face (auto-partition), the three planes that lie on the edges of the triangle and are orthogonal to the triangle face, and the same six axis-aligned splitting planes used by the kd-tree. Note that the first four splitting planes require a general BSP and would not work with a RBSP.

3.5 Build

Our build method is similar to those for building kd-trees with SAH, except that we are not able to achieve the $O(n \log n)$ builds that are possible for kd-trees [22] since we need to partition the triangles along an arbitrary direction. We use the same build algorithm as for the standard naive $O(n^2)$ kd-tree build [22]. At this point, we cannot use the faster lower complexity builds since we cannot sort the test planes. However, we are able to lower the complexity by using a helper data structure for counting the number of triangles on the left, right, and on the plane. This structure is a bounding sphere hierarchy over the triangles, where each node has a count of how many triangles it contains, so that if the node is completely to one side of the plane the triangle count can be immediately returned. We build this structure using standard axis-aligned BVH building algorithms, so the tree can be computed extremely quickly [20] compared to the BSP build time. If all the triangles lie on the plane, then our current structure will end up traversing all the leaves and take linear time for that candidate plane. This worst case time complexity however will not increase the BSP build complexity since we would have had to spend linear time counting anyway. This can occur for portions of a scene, such as the triangles that tesselate a wall. However, most of the scene will not lie on the same plane and so for well-behaved scenes, this helper structure will give the location counts in sub-linear time. While we must update this structure after every split, and that takes $O(n)$, that cost is the same as the $O(n)$ splitting plane tests performed, so it does not increase the complexity. Thus, for almost any scene we are able to achieve a sub-quadratic build. Like in kd-trees, clipping triangles to the splitting plane results in a higher quality build [7].

4 Ray Tracing using Generalized BSPs

4.1 Traversal

A ray is traversed through a kd-tree by intersecting the ray with the split plane giving a ray distance to the plane which allows us to divide the ray into segments. The initial ray segment is computed by clipping the ray with the axis-aligned bounding box. A node is traversed if the ray segment overlaps the node. Since the two child nodes do not overlap, we can easily determine which node is closer to the ray origin and traverse that node first (provided it is overlapped by the ray segment), thereby allowing an early exit from the second node.

We use exactly the same traversal algorithm for the BSP tree except for two modifications. First, computing the distance to the plane now requires two dot products and a float division instead of just one subtraction and a multiplication with a precomputed inverse of the direction. Second, due to the limited precision of floating point numbers, the stored normal for any non-axis-aligned plane will always slightly deviate from the actual "correct" plane equation; thus we cannot use the computed distance value directly, but have to assume that all distances within epsilon distance of the plane could reside on either side and so should traverse both nodes. Primarily because of the two dot products and the division, a BSP traversal test is significantly more costly; in our implementation, profiling shows the BSP traversal is roughly $1.75\times$ slower than the kd-tree traversal. We thus use a hybrid approach where axis-aligned splits can still use the faster kd-tree style traversal.

The initial ray segment is still computed by clipping the ray with the original axis-aligned bounding box, exactly as with kd-trees. Though we could in theory use an arbitrary or more complex bounding volume, such as in the RBSP, we use the bounding box for reasons of simplicity. This has the added advantage that it provides a very fast rejection test for rays that completely miss the model.

4.2 Packet-Traversal

While we are primarily interested in single-ray traversal, as a proof of concept we have also implemented a SIMD packet traversal variant. While for single-ray traversal the BSP and kd-tree traversals are nearly identical except for the distance computation, the packet traversal has a few significant differences. In particular, for kd-tree

traversal one usually assumes that packets have the same direction signs, and split packets with non-matching signs into different sub-packets. Packets must have rays with matching direction signs so that the signs of the first ray can then be used to determine the traversal order of any given kd-tree node — since kd-tree planes are axis-aligned, once all rays in a packet have the same direction signs they will all have the same traversal order.

Since BSP planes can be arbitrarily oriented, this method of guaranteeing that a packet will always have the same traversal order no longer applies. Thus, traversal order must be handled explicitly in the traversal loop, including the potential case that different rays in a packet might disagree on the traversal order. For arbitrary packets, this would actually require us to be able to split packets during traversal. We avoid this by currently considering only packets with common origin, in which case one can again determine a unique traversal order per packet based on which side of the plane the origin lies. This test is actually the same as the one proposed by Havran for single-ray traversal [6], except in packetized form. The resulting logic is slightly more complicated than the "same direction signs" variant, but the overhead is small compared to the more costly distance test, and it is independent of the orientation of the plane.

Since currently all our packets do have a common origin, the current implementation is sufficient for our purposes (the kd-tree code is optimized for common origins, too), but for practical applications this limitation would need to be addressed, perhaps by using Reshetov's omnidirectional kd-tree traversal algorithm [15].

4.3 Traversal Based Triangle Intersection

Since the BSP is able to perfectly bound most triangles and performs ray-plane intersection tests at each traversal, we are able to do ray-triangle intersection tests in the BSP for very little extra cost. This requires a small amount of extra bookkeeping during traversal, but by the time a ray reaches a leaf the hit point can be found for free. We accomplish this by storing into each leaf node a flag for whether the node can use traversal based intersection and the depth of the splitting plane that corresponded to the triangle face. These values can be stored into the node without using any extra space. During traversal we keep track of the current tree depth and the depth of the current near and far planes. With this we are able to determine whether the triangle lies on the near or far plane, and if so, we get the hit point for free from the already computed ray-plane hit points used in the standard BSP/kd-tree traversal. This removes the need for most explicit ray-triangle intersection tests.

4.4 Data Structure

Optimized kd-trees generally use an 8 byte data structure where 2 bits specify the plane normal (essentially the axis), 1 bit for a flag specifying whether the node is internal or leaf, and 29 bits specify the index of the child nodes in the case of internal nodes, or the number of primitives in the case of leaf nodes. The remaining 4 bytes give the floating point normal offset. The BSP node is essentially the same, except that we must store 3 floats for the full plane normal. The data required by the traversal based triangle intersection is only needed by the leaf nodes, so we can easily store that in the 12 bytes used the by normals in the internal nodes. BSP nodes therefore require 20 bytes per node. The total storage requirements can be computed by multiplying the number of nodes by 20 bytes.

5 Results

Measurements were taken on a dual 2 Ghz clovertown (8 cores total). Build times are using a single core and render times using all 8 cores. We render both ray cast images using only primary rays with no shadows or other secondary rays, and simple brute force Kajiya style path traced images with multiple bounces off of lambertian surfaces. The ray casting lets us test very coherent packets, in contrast to the path tracing which exhibit extremely incoherent secondary ray packets. Using an optimized path tracer might result in faster render times, but it would not affect how the BSP compares with other acceleration structures.

We compare the BSP against an optimized kd-tree acceleration structure built using the $O(n \log^2 n)$ SAH with triangle clipping, and against the highly optimized SAH interval arithmetic (IA) BVH from [21]. Both the BSP and kd-tree can traverse rays in either single ray traversal or using SIMD packet traversal, but it cannot switch between the two during traversal. Unlike Reshetov's MLRTA [16], neither our BSP nor our kd-tree use frustum traversal or entry point traversal. Were they to do so, it is expected that the kd-tree would be faster than the BVH [21], and those techniques could also be applied to BSP traversal. The BVH always uses interval arithmetic traversal with SIMD packets, with the traversal performance often gracefully degenerating to single ray BVH performance for incoherent packets.

We use a variety of scenes (Figures 1 and 2) for our tests, some of which were chosen because they highlight the BSP's strengths and others because they highlight the kd-tree's strengths. Section 9 of the UNC power plant and the UC Berkeley Soda Hall show how real scenes can contain many non-axis-aligned triangles for which the BSP is able to significantly outperform competing acceleration structures, in this case, by up to 3×.

The conference room was chosen as an example of a scene for which axis-aligned acceleration structures excel since many of the triangles are axis-aligned. As expected, the BSP does not show a strong advantage over other structures, however it still proves to be slightly faster than the kd-tree since most of the tree ends up using kd-tree nodes and near the leaves the BSP is able to further split triangles whereas the kd-tree and BVH would normally have stopped and created leaves containing multiple triangles.

The Stanford bunny is a traditional example of a character mesh. It is a well behaved scene that doesn't favor any particular acceleration structure since most triangles are of similar size, not axis-aligned, and not long and skinny. Note that the BVH does better than the BSP and kd-tree at path tracing the bunny because the average ray packet is still coherent because most secondary rays will immediately hit the background and terminate. For enclosed (indoor) scenes, rays can bounce around multiple times and therefore impact the overall time more than the primary rays.

The pickup truck used for bullet ray vision by Butler and Stephens [2] is a good example for where the expensive BSP build cost is computed once in a preprocess and then easily offset by having a faster interactive visualization. While the bullet vision paper used a more complicated material to determine the amount of transparency, we use a fixed value for the transparency for these tests. However this does not affect the overall results when comparing acceleration structures.

As a stress case for other axis-aligned structures, and to showcase the abilities of the BSP, we compare against a non-axis-aligned cylinder tessellated with many long skinny triangles. While this is clearly not realistic and strongly favors the BSP, it does help to illustrate the strengths the BSP has over other acceleration structures, and helps to explain why the BSP can outperform other acceleration structures.

Impressively enough in all the scenes with secondary rays, with the exception of the bunny, the single ray BSP is able to outperform all other acceleration structures including the SIMD packetized IA BVH acceleration structure. In the bunny scene the BSP is able to outperform only the kd-tree. The single ray BSP is always faster than the single ray kd-tree and the SIMD BSP is usually as fast or faster than the SIMD kd-tree.

The ray packet improvements for the BSP over the kd-tree are not as pronounced as the single ray improvements because ray packets benefit less from having triangles completely subdivided into separate nodes. This is because the ray packet will often be larger than the individual triangles.

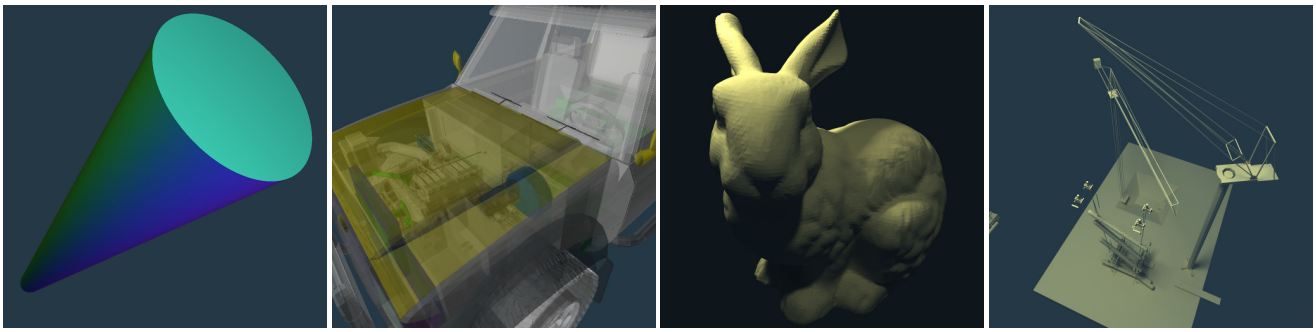

Figure 2: Ray cast cylinder (596 tri) and transparent pickup truck (183k tri), rendered $26\times$ and $1.3\times$ faster than the single ray kd-tree. Path traced bunny (69k), and section 9 of the UNC powerplant (122k) outperform the kd-tree by $1.2\times$ and $2.4\times$.

	Single Ray				SIMD Ray Packet		
	BSP-orig	BSP	BSP-tri	kd-tree	BSP	kd-tree	BVH
Path Traced (frames per minute)							
bunny	0.874	0.954	0.963	0.796	-	-	1.306
conf	0.296	0.347	0.358	0.333	-	-	0.218
soda room	0.233	0.349	0.363	0.314	-	-	0.171
soda art	0.217	0.236	0.250	0.102	-	-	0.116
soda stairs	0.170	0.224	0.242	0.188	-	-	0.084
pplant9	1.51	1.74	1.76	0.731	-	-	1.111
Ray Traced (frames per second)							
cylinder	22.0	21.9	23.0	0.868	49.2	5.73	5.28
bunny	12.2	13.3	13.7	11.2	20.2	22.6	38.0
truck	2.46	2.75	2.80	2.19	-	-	1.87
conf	8.87	10.9	11.6	10.3	26.2	25.7	31.9
soda room	4.17	7.43	7.6	6.33	23.7	21.9	25.0
soda art	6.09	9.06	9.7	4.01	30.6	16.9	26.0
soda stairs	6.47	8.07	8.7	4.79	25.8	17.3	11.6
pplant9	13.3	16.1	16.4	4.95	33.8	19.2	17.8

Table 1: Frame rate for rendering the 1024spp 512x512 path traced scenes and the 1024x1024 ray cast scenes. BSP-orig is the BSP without the kd-tree node optimization and BSP-tri is the BSP with traversal based intersection.

In order to measure how including kd-tree nodes in the BSP improves performance, we also compare our BSP with a BSP that uses only general splitting planes and is built using a standard SAH; we refer to this as BSP-orig. BSP-orig is still novel since we are not aware of any general BSPs that are built with the SAH, and it still performs better than the kd-tree for some scenes. However, we do not advocate using it since the optimized BSP we present in this paper is almost always faster, easy to implement, and the trees are not much larger in size.

5.1 Statistical Comparison

From Table 2 we see that the BSP uses roughly as many traversal steps as the kd-tree, with a fraction of those traversals going through the more expensive BSP nodes. For this reason, the BSP will usually take roughly as much, to slightly more time traversing than the kd-tree. However, the advantage of the BSP is that it performs many times less triangle intersections. Unfortunately, for scenes that a kd-tree does well on, such as the conference room, the possible speedup that the BSP can achieve over the kd-tree is limited since the kd-tree only spends about 30% of the time intersecting triangles and roughly 55% of the time traversing nodes. Amdahl's law thus states that even if we eliminated the intersection cost completely without slowing down the traversal, the overall speedup would only be $1.4\times$. We reduced the number of triangle intersections by $4\times$, which would have resulted in a $1.3\times$ speedup if the traversal cost stayed the same. However, instead the traversal cost also went slightly up so in the end we got a $1.1\times$ speedup. This means the BSP cannot be significantly faster than the kd-tree unless it is also able to do many fewer traversals or the cost of a BSP traversal went down. Since our BSP is already very close to the minimum number of triangle intersections (even fewer with traversal-based intersection), a higher quality build, for instance from testing $O(n^3)$ possible splitting planes at each node, would only be able to significantly improve performance by reducing the number of traversals. However, for certain scenes or viewpoints, for instance the soda hall art or section 9 of the power plant, the amount of time the kd-tree spends on triangle intersections becomes quite high (two orders of magnitude in these examples) and for these situations the BSP can offer very significant improvements; this is where the BSP is truly superior.

Increasing C_i will result in more aggressive splits which ends up further reducing the number of triangle intersections, at the expense of more node traversals. When not performing traversal-based intersections, this causes a slight negative performance hit. Overall, the traversal-based intersection performs better since more triangles end up being tightly bound by splitting planes. The results presented in this paper are all using the lower intersection cost to build the BSP tree.

Table 2 shows that the BSP-orig is able to perform fewer triangle tests than the optimized BSP and often performs less total traversal steps. However, since all those traversal steps are done using the more expensive BSP traversals, the actual render time ends up increasing as seen in Table 1.

Table 3 shows that the BSP is able to subdivide most triangles into individual leaves. Contrast this with the kd-tree build for the conference room which has 13 leaves with more than 100 triangles and about 100K nodes with more than four triangles, and roughly as many nodes with one triangle as with two or three triangles. Another interesting point is that while the majority of nodes in the BSP tree use general BSP nodes, Table 2 shows that most of the nodes actually visited during traversal use the kd-tree style node. This is explained by noting that most kd-tree nodes are higher up in the tree where they are more likely to be visited, while the BSP nodes are usually found near the leafs.

5.2 Absolute Performance Comparison

The BSP should always be roughly as fast or faster than the kd-tree since the BSP can use kd-tree style traversal. Compared to the IA BVH, the BSP tends to be faster only for non-coherent rays, as evident in the path traced benchmarks, and for scenes with many non-axis-aligned triangles. Performing the triangle intersection as part of the BSP traversal will result in performance improvements if most rays are intersecting triangles (indoor scenes or closeups of polygonal characters).

	cylinder 596	pplant9 122K	conf 283K	bunny 69K	truck 183K	sodahall 2.14M
	BSP / kd	BSP / kd	BSP / kd	BSP / kd	BSP / kd	BSP / kd
build time	9.2s / 0.24s	36m / 29.2s	112m / 1.2m	19m / 14.3s	65m / 46s	23.6h / 6.3m
max tree depth	26 / 33	48 / 75	47 / 89	33 / 59	44 / 135	60 / 108
# nodes	15K / 13K	3386K / 1645K	9246K / 3921K	1470K / 988K	3363K / 2870K	28.9M / 27.9M
% kd-tree nodes	15% / 100%	37% / 100%	21% / 100%	27% / 100%	27% / 100%	31% / 100%
leaf (0 tris)	3.3K / 3.7K	734K / 181K	2128K / 746K	423K / 250K	774K / 376K	6.73M / 3.36M
leaf (1 tri)	3.9K / 412	783K / 171K	2196K / 279K	309K / 24K	713K / 277K	6.41M / 2.29M
leaf (2 tris)	142 / 333	125K / 264K	275K / 383K	3.3K / 158K	176K / 440K	1.13M / 5.29M
leaf (3 tris)	30 / 1.2K	45K / 130K	21K / 333K	107 / 51K	15K / 217K	130K / 2.39M
leaf (4 tris)	0 / 131	3.5K / 41K	3.1K / 108K	2 / 9.6K	3.0K / 73K	20K / 403K
leaf (>4 tris)	0 / 603	3.7K / 36K	493 / 111K	0 / 905	618 / 52K	8.7K / 209K
max tris in leaf	3 / 301	17 / 64	8 / 101	4 / 7	20 / 119	600 / 600

Table 3: Build statistics.

In the path traced bunny many primary rays miss the bunny, and of those that do hit the bunny, most cast new rays that hit the background. As such, the ray packets are much more coherent than in interior path traced scenes, such as in the conference scene, and so the IA BVH is able to outperform the BSP.

In Figure 3 we compare the per pixel rendering time when performing just ray casting. In the first two columns, each pixel intensity corresponds to the amount of time taken to render that pixel when using a kd-tree and a BSP. The third column is the normalized difference of the first two columns, with white corresponding to the kd-tree taking longer and red with the BSP. In the power plant, soda hall, and cylinder scenes, the BSP clearly performs much better than the kd-tree due to the kd-tree not being able to handle the long skinny non-axis-aligned triangles. The brightened silhouettes on the bunny comparison shows how the BSP is able to quickly get a tight bound over the mesh. The BSP shows much less variation in render time compared to the kd-tree which has substantial variation with some regions of the scene clearly showing "hot-spots" where the kd-tree breaks down.

If we position the camera on one of these hot-spots (white pixels), the BSP will have a much bigger advantage. For instance, even though for the camera view we used in the conference room the BSP is only slightly faster than the kd-tree, the BSP can get an order of magnitude improvement over the kd-tree simply by moving the camera to look at just the chair arms. The same principle applies to all the other scenes.

5.3 Build Time and Build Efficiency

The complexity of our BSP build algorithm for well behaved scenes is sub-quadratic. From experimental observation it appears to be around $O(n\log^2 n)$, which makes the complexity fairly close to that of kd-trees which can currently be built in $O(n\log n)$. But the BSP is still almost 2 orders of magnitude slower. This is mainly due to the large cost in calculating the surface area of a node. Since the focus of this paper is not on fast builds, the BSP builder is left unoptimized.

These build times are clearly non-interactive and are only useful as a preprocess. However, note that the full quality kd-tree build is also still non-interactive. The $O(n\log n)$ kd-tree build from Wald and Havran [22] is $2-3\times$ faster than our $O(n\log^2 n)$ kd-tree build, but also still a couple orders of magnitude too slow for interactive builds. In order to get interactive kd-tree and BVH builds, the tree quality must suffer. For coherent packet tracing, tree quality near the leaves is not as important since the packets are often as large as each node, and so using a lower quality approximate build will often result in only a slight increase in intersection tests. However, for incoherent packets where each ray intersects different primitives, the faster approximate builds will result in a very noticeable performance penalty as rays perform many more unneeded intersection tests.

6 Summary and Discussion

We have shown that a general BSP is as good or better than a kd-tree if build time can be ignored. For scenes with non-coherent rays or with many long non-axis-aligned triangles, the single ray BSP is even able to outperform the packetized SIMD IA-BVH. Furthermore, the BSP is able to handle difficult scenes that other axis-aligned based acceleration structures break down on. Even though the complexities of kd-trees and BSPs are roughly the same, in practice, the BSP tree allows for fewer triangle-ray intersection tests with only a minimal increase in the traversal cost. The complexity of our current SAH build is sub-quadratic, like the kd-tree, but it has a rather large constant term due to the expensive computation of the surface area formed by the intersection of half-spaces, which leads to slow build performance. However, there are still many ways to improve the build performance and we expect the BSP build to easily become much faster than it is currently. Furthermore, as interactive ray tracing begins to focus more on incoherent secondary rays, the fast approximate interactive builds will not perform as well and so for static scenes, slower but full quality builds will once again be required. If the tree will be built offline as a preprocess, then spending an hour instead of a minute in exchange for more consistent frame rates with no trouble hot-spots can be a very worthwhile trade-off for certain applications. Clearly, the BSP is currently not suitable in situations where fast build times are required.

Numerical precision issues cause problems in the build. During rendering, these precision issues can be easily handled by using an epsilon test when doing a standard check for whether a ray-plane intersection occurs on the near or far plane. This adds a negligible two additional addition instructions and occasionally result in a few extra unneeded traversals, but rendering quality is not affected.

In all of our tests, the BSP tree was always the fastest option for single ray traversal and is robust to scene geometry, unlike other acceleration structures which might perform well in some conditions but poorly in others. Looking at the time visualizations from Figure 3 we saw that many scenes exhibit hot-spots when rendered with the kd-tree, but not with the BSP. If the user were to by chance zoom into one of these hot spots, the frame rate could go down by an order of magnitude as in the soda hall, cylinder and conference room chair arm examples. In some applications this is unacceptable and the expensive BSP build would be fully justified, especially when used as a one-time preprocess.

6.1 Future Work

Our implementation does not make use of frustum/IA-based traversal present in the fastest acceleration structures. While this might be fine when there is low ray coherence, such as in path tracing, for primary rays this can sometimes make up to an order of magnitude difference in performance [16, 21, 23]. Extending Reshetov's MLRTA from kd-trees to BSPs would likely be the best method. Another optimization

would be to dynamically use the packetized traversal for coherent packets and single ray traversal for incoherent packets. This would allow the BSP (and the kd-tree) to traverse the primary rays much faster. This would likely remove the advantage the BVH had in some scenes that were predominately primary rays, such as the path traced bunny.

We would also like to improve the build performance. This can be done in many ways. Optimizing the surface area calculation and parallelizing the build are two obvious options and would likely result in substantial speedups over our non-optimized code. Randomly testing only a subset of the possible splitting planes would trade build quality for build speed. Handling axis-aligned splits with a traditional kd-tree build should give a significant improvement, especially since many of the kd-tree nodes are near the top of the tree where nodes have more splitting plane candidates and so are more expensive. Since the top of the tree consists mainly of kd-tree nodes, we could also only use a kd-tree build for the top of the tree and switch over to general splits when there are less than a certain number of triangles in a node or if performing a kd-tree split results in a small improvement to the SAH cost.

Improving the selection of splitting planes could allow for further improvements in tree quality. For instance, when using a triangle edge as the split location, we set the plane normal to be orthogonal to the edge and triangle normal. This is an arbitrary decision and will not always result in an optimal split. For instance, if two triangles share an edge and form an acute angle, the splitting plane on that edge will place both triangles on the same side, which might not be desirable. Likewise, if all the triangles are axis-aligned, but form diagonal lines through stair-stepping, then the optimal split will be those that are tangent to as many triangles as possible.

Our heuristic for combing C_{BSP} and $C_{\text{kd-tree}}$ is an improvement over the standard SAH, but it is likely that an even better heuristic exists that would further result in performance improvements.

Extending this to handle more expensive primitives, such as spline patches could result in substantial improvements since minimizing the number of ray-patch intersection tests would make a noticeable difference. The only required modification would be in choosing the candidate splitting planes and in removing the traversal based triangle intersection. One possible way to select the planes would be to use the axis-aligned planes that make up the bounding box as well as randomly selecting planes that are tangent to the patch.

7 Acknowledgments

We would like to thank Peter Shirley for his insightful advice. The truck model is courtesy of the US Army Research Laboratory, and the bunny comes from the Stanford 3D Scanning Repository.

References

[1] Sigal Ar, Gil Montag, and Ayellet Tal. Deferred, Self-Organizing BSP Trees. *Computer Graphics Forum*, 21(3):269–278, 2002.

[2] Lee Butler and Abe Stephens. Bullet Ray Vision. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing*, pages 167–170, 2007.

[3] M. de Berg. Linear Size Binary Space Partitions for Fat Objects. *Proceedings of the Third Annual European Symposium on Algorithms*, pages 252–263, 1995.

[4] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph.*, 14(3):124–133, 1980.

[5] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[6] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[7] Vlastimil Havran and Jirí Bittner. On Improving Kd Tree for Ray Shooting. In *Proceedings of WSCG*, pages 209–216, 2002.

[8] Vlastimil Havran, Tomas Kopal, Jiri Bittner, and Jiri Žára. Fast robust BSP tree traversal algorithm for ray tracing. *Journal of Graphics Tools*, 2(4):15–23, 1998.

[9] Warren Hunt, Gordon Stoll, and William Mark. Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[10] Ravi P. Kammaje and Benjamin Mora. A study of restricted BSP trees for ray tracing. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing*, pages 55–62, 2007.

[11] Christian Lauterbach, Sung-Eui Yoon, David Tuft, and Dinesh Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–45, 2006.

[12] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(6):153–65, 1990.

[13] Michael S. Paterson and F. Frances Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete and Computational Geometry*, 5(1):485–503, 1990.

[14] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Experiences with Streaming Construction of SAH KD-Trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006.

[15] Alexander Reshetov. Omnidirectional ray tracing traversal algorithm for kd-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 57–60, 2006.

[16] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics*, 24(3):1176–1185, 2005. (Proceedings of ACM SIGGRAPH 2005).

[17] Gordon Stoll, William R. Mark, Peter Djeu, Rui Wang, and Ikrima Elhassan. Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Technical Report 06-21, University of Texas at Austin Dep. of Comp. Science, 2006.

[18] Kelvin Sung and Peter Shirley. Ray Tracing with the BSP Tree. In David Kirk, editor, *Graphics Gems III*, pages 271—274. Academic Press, 1992.

[19] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[20] Ingo Wald. On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing*, pages 33–40, 2007.

[21] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):1–18, 2007.

[22] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–70, 2006.

[23] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. (Proceedings of ACM SIGGRAPH).

	Ray-Triangle Tests		Node Traversals	
	standard	traversal-based	BSP	kd-tree
raycast bunny				
BSP-orig	0.07	0.40	21.6	-
BSP	0.11	0.36	5.8	18.4
kd-tree	2.3	-	-	27.8
path traced bunny				
BSP-orig	0.26	0.28	30.2	-
BSP	0.32	0.26	8.3	25.1
kd-tree	4.6	-	-	36.8
raycast conference				
BSP-orig	0.77	0.52	32.8	-
BSP	0.82	0.52	3.2	31.8
kd-tree	3.5	-	-	33.5
pathtraced conference				
BSP-orig	0.87	0.35	29.0	-
BSP	0.98	0.35	2.9	25.5
kd-tree	4.2	-	-	29.2
raycast power plant 9				
BSP-orig	0.04	0.24	23.3	-
BSP	0.09	0.21	4.9	19.9
kd-tree	18.5	-	-	25.7
path traced power plant 9				
BSP-orig	0.11	0.18	24.4	-
BSP	0.12	0.17	5.1	20.8
kd-tree	17.5	-	-	26.6
raycast cylinder				
BSP-orig	0.07	0.33	10.9	-
BSP	0.08	0.34	8.6	3.1
kd-tree	182.4	-	-	15.2
truck				
BSP-orig	1.4	0.40	27.4	-
BSP	1.4	0.45	7.1	22.3
kd-tree	7.5	-	-	26.5
raycast sodahall stairs				
BSP-orig	0.12	0.95	48.1	-
BSP	0.18	0.91	8.5	42.8
kd-tree	13.7	-	-	58.6
path traced sodahall stairs				
BSP-orig	0.24	0.57	39.2	-
BSP	0.27	0.57	4.1	33.6
kd-tree	4.9	-	-	38.5
raycast sodahall room				
BSP-orig	0.26	0.86	75.8	-
BSP	0.40	0.81	7.7	51.2
kd-tree	5.4	-	-	61.9
path traced sodahall room				
BSP-orig	0.32	0.51	46.9	-
BSP	0.39	0.48	4.3	36.4
kd-tree	5.9	-	-	41.4
raycast sodahall art				
BSP-orig	0.17	0.89	52.3	-
BSP	0.14	0.94	4.0	42.9
kd-tree	20.6	-	-	41.8
path traced sodahall art				
BSP-orig	0.22	0.70	47.4	-
BSP	0.31	0.69	8.6	43.2
kd-tree	44.4	-	-	37.2

Table 2: Per ray statistics. The total number of ray-triangle tests is the sum of the expensive standard tests and the very cheap traversal-based tests. The total number of node traversals is also the sum of the BSP and kd-tree node traversals. If traversal-based triangle intersection tests are not used in the BSP, then the number of standard triangle tests is roughly the sum of the standard tests and the traversal-based tests mentioned in the table.

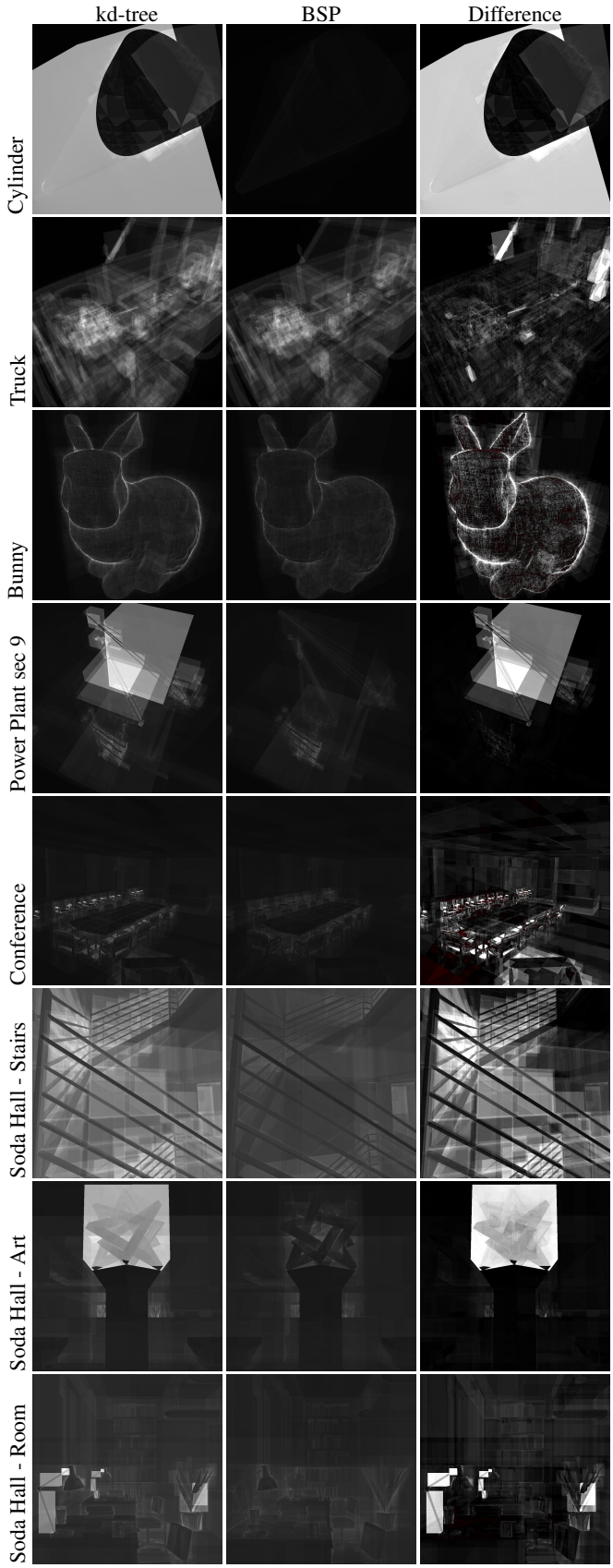

Figure 3: Rendering time comparison (right) of kd-tree (left) with BSP (middle). The time visualizations are rendered so that increasing intensity corresponds to increased rendering time for the pixel. The time comparisons are the normalized differences of the BSP and kd-tree timings, with white corresponding to the kd-tree being slower and red with the BSP being slower.