Interactive RayTracing of Dynamic Scenes

Tomas Davidovic*

Department of Computer Science and Engineering
Czech Technical University in Prague

Abstract

Raytracing of dynamic scenes requires not only fast ray traversal, but also fast update of acceleration structures. Fast ray traversal is usually achieved through traversing several rays at once in a packet. Update of acceleration structure can be done by gradually refitting the structure or by rebuilding it from the scratch every frame. The latter approach puts no requirements on amount or behavior of objects in the scene.

In this paper we will describe a raytracing system that combines fast packet traversal with Bounding Volume Hierarchy rebuilt from scratch using approximated Surface Area Heuristic. Vertex culling uses transient frustum in each leaf to quickly eliminate potential intersections with triangles lying strictly outside the frustum. This benefits from large leaves and allows shallow acceleration structure that can be built exceptionally fast with no impact on performance. We will show that combination of these two methods does provide interactive rates even on today's desktop computers. We will also demonstrate the interactive capabilities on fast relighting of scenes with moving point lights.

Keywords: Ray Tracing, Bounding Volume Hierarchy, Dynamic Scenes, Packet Traversal

1 Introduction

Raytracing at interactive frame rates can be roughly divided into three distinct categories. Static scenes with only camera and lights moving, dynamic scenes with constrains on objects movement and dynamic scenes without any constrains. The first approach uses precomputed acceleration structures of very high quality, but offers little in the means of moving objects. Those are usually placed outside the main acceleration structure and tested separately. The second approach is achieved by refitting the acceleration structure over existing geometry. This puts two constrains on the scene. The number of triangles should not change between frames and the triangles should not move too severely. Too severe movements would rapidly quality of the structure. However, even with a relatively slow movement, if it is not contained in a small area, the quality of the structure gradually degrades and has to be rebuilt eventually. We therefore consider the last option to be best suited for highly interactive applications, such as games. Games usually have geometry appearing and disappearing, e.g., explosions, and geometry with high degree of movement, e.g., projectiles and characters.

To put no constrains on geometry it is necessary to completely rebuild acceleration structure for every frame. The most common acceleration structures are kD-trees [1] and Bounding Volume Hierarchies (BVH) [4] using Surface Area Heuristics (SAH) cost function to find the best partitioning. This produces very efficient acceleration structures, but as the cost function is evaluated for every possible split plane it take average $O(N \log N)$ time to build them. We are however interested not in the pure ray traversal performance, but in optimization of build plus traversal time. It is therefore acceptable to build slightly worse acceleration structure if the speed up of build time outweighs the slowdown of traversal. We decided to use BVH acceleration structure for two main reasons. First, BVH is generally shallower than kD-tree and with fewer nodes to build it is safe to assume that the build process will be faster. Second, the BVH has exact upper bound $2N-1$ on the number of nodes and we can therefore completely avoid costly memory allocations during build process. To speed up the BVH build we use approximate SAH [3] and evaluate only a limited number of split planes in each node. We also aim for shallow data structure with fewer nodes and large leaves to further lower the number of splits plane evaluations performed.

Large leaves can seriously hamper performance if we do not implement a very effective intersection routine, because number of computed ray-triangle intersections increases significantly. We decided to use Vertex Culling packet-triangle intersection routine [2] to answer this problem. This method builds a transient frustum after reaching a leaf and using several simple checks discards triangles that are strictly separated from the packet and do not have to be intersected.

We show that using combination of these techniques it is possible to achieve interactive rates for moderately complex scenes. In Chapter 2 we describe implementation of approximate-SAH BVH, in Chapter 3 we add packet traversal and in Chapter 4 provide description of Vertex Culling and its various checks and provide summary of effect of those checks on performance of the whole system. In Chapter 5 we briefly describe necessary modifications aimed at using packet tracing for shadow rays and

*davidt2@fel.cvut.cz

offer methods application to interactive light positioning. In Chapter 6 we sum our measured results and in Chapter 7 we conclude.

2 Fast BVH

Bounding Volume Hierarchies one of the two acceleration structures used in raytracing. Unlike kD-tree, BVH partitions objects rather then the scene space. The two main distinctions are that in BVH each object belongs to exactly one leaf and that nodes of BVH can overlap. Each BVH leaf therefore contains pointer to its triangles and an Axis Aligned Bounding Box (AABB) of space its objects occupy. Each inner node contains pointers to its children and also AABB of the space its children occupy.

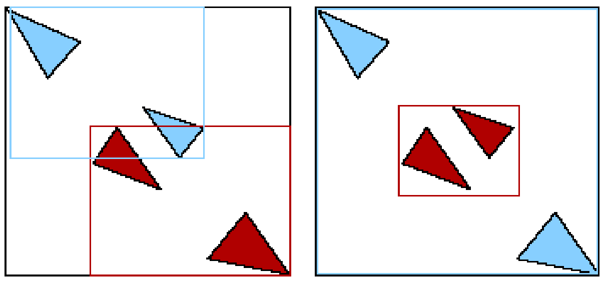

Figure 1: Possible BVH splits. Courtesy of [4]

When splitting leaf's triangles to create an inner node, it is possible to partition the triangles into an arbitrary number of groups by virtually any key (Fig. 1). Testing all $O(N^k)$ possible divisions is infeasible, so the most common approach is to use a split plane perpendicular of one of the world axis and divide triangles into two groups based on whether they are to the left or to the right of the split plane, very much like it is done in kD-tree build.

Position of the split plane is chosen based on Surface Area Heuristic cost function:

$$Cost = 2C_T + C_I \left(\frac{SA(V_L)}{SA(V)} N_L + \frac{SA(V_R)}{SA(V)} N_R \right) \quad (1)$$

where C_T is cost of traversal, C_I cost of intersection, N_L and N_R are number of triangles to the left, respectively right, of the split plane and $SA(V)$, $SA(V_L)$ and $SA(V_R)$ are surface area of the split node, and AABBs of triangles to the left and right of the split plane respectively. To determine whether a triangle is to the left or right of a split plane we need to use only one of its points. Usually its centroid is used.

2.1 Approximate SAH

Normal construction algorithm evaluates the cost function in split planes going through each triangle's centroid point. This however leads to $O(N \log N)$ time at best. We use an approximation introduces in [3]. For each node we consider only a limited number of equidistant split planes instead of considering all possibilities. We also consider only split planes perpendicular to the longest AABB axis, instead of considering all three directions. As only triangle centroids are used for building the BVH, we use an AABB over those centroids instead of AABB over the whole objects. This is mainly to avoid problem described on Fig. 2.

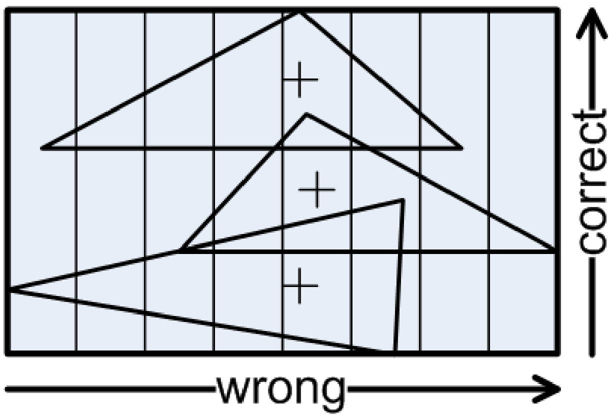

Figure 2: Object vs. centroid AABB

The equidistant split planes divide the node into several bins. We then iterate over all triangles in the node and add it to each respective bin. Each bin knows its number of triangles and AABB over all triangles contained in it. After all triangles have been assigned into their bin we perform two sweeps over the bins. First we go from left to right, computing cost to the left of each respective split plane. The second sweep goes in the opposite direction, computing cost to the right of each split plane and evaluating the cost function. The cost function evaluates is a simplified version of the aforementioned function. The simplification consists of removing terms common for all split planes i.e., the traversal cost, the intersection cost and the node surface area, resulting in:

$$Cost = SA(V_L)N_L + SA(V_R)N_R \quad (2)$$

Three common termination criteria are used to determine whether a node should be split any further. Minimal number of triangles in a node, maximal depth of the BVH and comparing cost of not splitting with the best cost of splitting to determine whether the split can yield any improvement. We added a fourth termination criterion, minimal size of centroid AABB, to avoid problems with float precision.

3 Packet traversal

Before we can start using Vertex Culling to efficiently compute intersection between ray packet and triangles we first need to traverse the packet through BVH into a leaf. The ray packets have to fulfill one condition imposed by the Vertex Culling algorithm. The rays have to have a common direction sign in at least one direction. If the packet does not fulfill this condition it is split into individual rays and the rays are traced separately.

Without a loss of generality we can assume that the rays have the common direction sign in axis x. Further, when speaking about *near* and *far* AABB planes, we will means planes perpendicular to axis x and we will consider plane to be near when it has a lower x coordinate than the other, far, plane.

First we compute rays' intersections with near and far planes of the root scene AABB. We use those intersection points to compute axis-aligned rectangles set by the ray packet on the near and far planes. Then we connect corresponding corners of those rectangles by corner rays that strictly define the packet. These corner rays are used to quickly discard nodes that are completely missed by the ray packet.

To avoid testing rays that have already missed higher nodes in the traversal, we remember index of the first ray that did intersect this node and ignore all rays with lower indexes. Whenever a single ray from a packet hits a node, the whole packet is taken down to the node and no further tests are therefore necessary.

We perform several tests to determine whether a packet intersects a node. First we test the first active ray against a node. If the ray hits we immediately return with positive result. If the first active ray misses, we use corner rays to compute axis-aligned rectangles on the node's near and far plane and determine whether both are strictly separated by one of the AABB's other planes or not. If they are strictly to one side of the node's AABB, the node is missed by all the packet's rays and no further tests are necessary (Fig. 3). The last test is to test all rays against the AABB to determine whether any of the rays hits the node. If any ray hits the intersection procedure returns immediately with positive result. The index of first active ray is updated throughout the whole process. For closer description of the algorithms and tests used see [3].

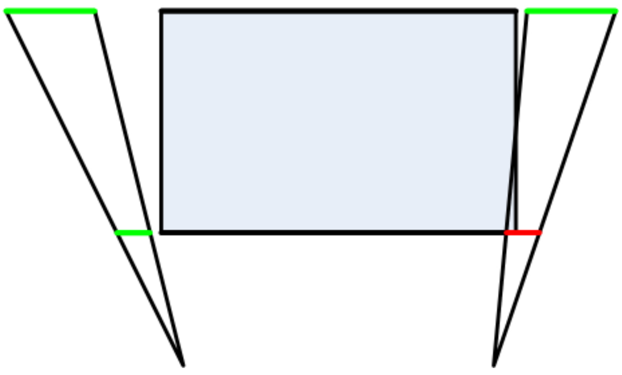

Figure 3: Two of the possible results of packet-box separation. Green - frustrum misses; Red - frustum intersects.

4 Vertex culling

When the packet is traced through BVH down to a leaf, it is necessary to perform a fast packet-triangle intersection test. We implement the methods described in [2], plus add a test to eliminate rays that completely miss the leaf.

First we create a transient frustum in a manner similar to the one used for packet traversal, but this time only from the rays that actually do intersect the leaf. There are three basic tests performed.

First, we test whether all three vertices creating the triangle are strictly separated from the packet by one of its boundary planes. If they are then no ray in-between those planes can intersect the triangle and we can terminate the result with a negative result (Fig. 4, left). Second, we intersect the packet corner rays with the triangle; computing barycentric coordinates u, v of the hitpoints. A ray misses a triangle if it violates any of these three conditions:

$$0 \leq u; 0 \leq v; u+v \leq 1 \tag{3}$$

If all four rays miss because of violating the same condition it means that the rays are strictly separated from the triangle by one of the triangle's edges (4, right).

On the other hand, if all four rays do intersect the triangle, we can skip barycentric coordinate check for all the rays in a packet. We also remember the distances t to the triangle and if another triangle is encountered that is also intersected by all four corner rays, but lies behind the first such triangle, it is skipped as no ray in the packet can reach it through the first triangle.

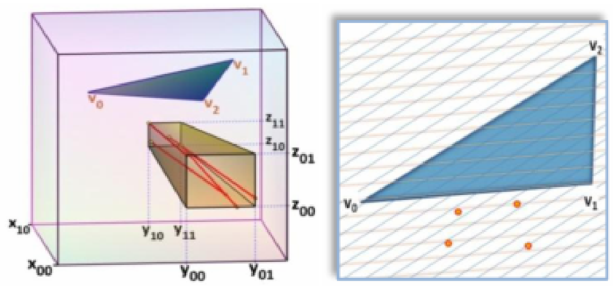

Figure 4: Vertex separation (left) and Triangle edge separation (right). Courtesy of [2]

The very last test, when all attempts to separate triangle from a frustum fail, is to intersect all rays with the triangle. We evaluate their distance to its plane and interpolate their texture coordinates and shading normal for all rays, but store them only for those that hit the triangle.

Here we have added another test. During the creation of transient frustum we flag all rays as hitting/missing the leaf completely. When in the SSE2 implementation of the intersection routine we encounter a pack of four such rays that all four completely miss the leaf, we skip any tests for them.

5 Shading via ray packets

So far we have described only methods used to accelerate primary rays. One option is to split the primary rays and shade each individually. We however consider this approach to be inferior, as it ignores the fact that coherent rays in a packet will often hit either the same triangle, or triangles with same material. We therefore decided to exploit this coherence and employed the following three step shading approach. We use a *StoreInfoPacket* structure that for each ray in a packet contains the following information:

- Hit point position
- Pointer to point's material
- Normalised shading normal
- Direction to camera
- Specular coefficient
- Diffuse coefficient
- Shininess

In the first step we iterate through all rays and for all rays that hit any object we compute hitpoint, direction to viewer, normalize shading normal we have from packet-triangle intersection method and set pointer to the material. If the ray did not hit any objects we set the material pointer to *NULL*. We will call such rays "*missing* rays", as opposed to "*hitting* rays." We also initialize an array of valid flags that marks as valid those rays that have yet to be processed. This array is initially set to false for *missing* rays, true for all others.

The second step sets the specular and diffuse coefficients and shininess for all rays. Using SSE we iterate through all rays in the packet by packs of four rays. For each valid ray in the packet we call its material's shader. The shader starts to operate on packet's rays from the first valid ray and for each valid ray that uses the same material it computes coefficients, shininess and sets the valid flag to false. The shader also operates on four rays at a time, with the only exception being texture fetch which can only done for individual rays. We therefore fetch textures only for rays that explicitly do use this material, unlike the other values that are computed for all rays and stored for those using the material. If none of the four rays is valid and uses the material we immediately skip to next four rays. In this way we call each shader only once as it automatically computes all required data for all rays that use the same material. The shader can also compute any color that is not dependent on direct lighting e.g., ambient color, indirect illumination etc.

We now have all the data required to compute shading in a single structure, independent on the number and type materials used. The last step is to iterate through all lights compute direct lighting using for all rays. For the sake of simplicity we always compute direct lighting for all rays, but add the result to pixels color only for *hitting* rays. It is obvious that this method requires all objects in the scene to use the same shading model e.g., Phong shading in our implementation.

It also is very straightforward to implement shadow rays via packet tracing at this point. We set shadow ray origin to the light's position and each shadow ray points at its respective primary ray's hitpoint. However, care has to be taken when there are *missing* primary rays. Those can have any arbitrary hitpoint which could break coherence of the shadow ray's packet. To avoid this we set hitpoint of all *missing* rays to the hitpoint of first *hitting* ray. This will help us preserve coherence without any impact on correctness of the result.

The described method of computing direct light presented up with an opportunity implement a simple algorithm for moving lights with just minimal modifications. We can store the aforementioned *StoreHitPacket* structure for each packet of primary rays and use it to repeatedly recompute direct light affecting each pixel. We use this to subtract light's contribution from the picture, move the light and add the contribution from the light's new position. We further accelerate this process by keeping two color buffers. The base buffer stores the picture without the moved light's contribution, the color buffer then stores the picture including this light. This way we subtract light's contribution only when it is first selected as the active moving light, increasing the frame rate by a factor of two.

6 Results

Our system consists of three separate algorithms and it is virtually impossible to evaluate all combinations of all settings in them. We have therefore decided to focus on settings of each algorithm separately, using the best or most neutral settings for the other two. In Chapter 6.1 we examine of approximate-SAH BVH algorithm, in Chapter 6.2 we evaluate effect of various Vertex Culling tests and in chapter 6.3 we present results brought by our packet shading algorithm. For our tests we used Happy Buddha (1M triangles), A10 (218k) and Sibenik's Cathedral (80k) models. All measurements have been performed on Core 2 Duo @ 2.16GHz, 2GB RAM and GT6600 graphics card.

6.1 BVH results

In evaluating BVH settings we have focused on two main parameters. The ratio of intersection and traversal cost influences size of leafs and number of triangles contained in each leaf, the larger the ratio the smaller the leaves. For single ray tracing this is usually beneficial, but in combination with vertex culling it should be possible to quickly cull away triangles that are not intersected by the packet.

We first measured dependence of average number of triangles on the ratio (Fig. 5) and also fps performance of pure ray traversal (Fig. 6). It is obvious that with ratio greater than 10 the cost termination is not used anymore as the minimal number of triangles takes precedence. We therefore assume that there will be no impact on performance over the ratio 10. The traversal performance was measured with packets of 16×16 with all vertex culling tests enabled and gives us a basic insight into system's performance for static scenes. We see that the peak performance is around ratio 1.0 with average 3.5 triangles per leaf.

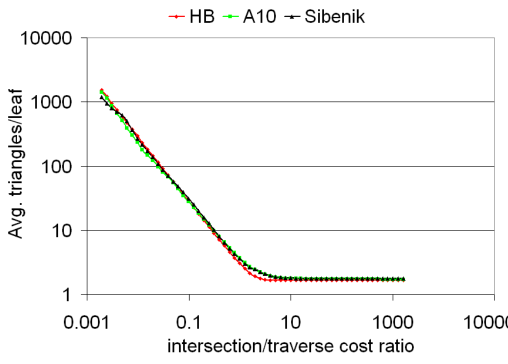

Figure 5: Average number of triangles per leaf

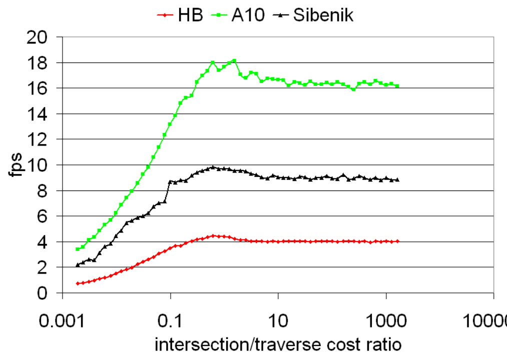

Figure 6: Frame rate of traversal only

Next we measured how the ratio affects performance of raytracing when the BVH is rebuilt each frame. We also included three different packet sizes. Too large packets should in theory have negative impact on performance, as the vertex culling is most effective when the packet is smaller than the leaf.

However, as is apparent from Figs. 7, 8 and 9 this is not the case. The largest tested packets also give the best performance. We address this unexpected result in Chapter 6.2. We can also see that the best performance shifted to ratio of 0.1 and average of 30 triangles per leaf. This confirms our assumption that while smaller leaves are better for pure raytracing performance, the lower build time for larger leaves more than compensates for slightly slower traversal. Because the ratio 0.1 is gives near to optimal performance for all three diverse scenes we assume that it is most suitable to be used as the universal ratio for our system and use it for further measurements.

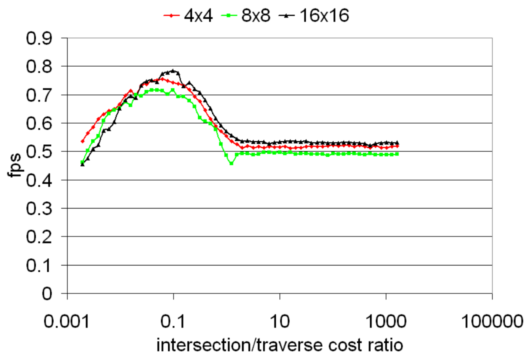

Figure 7: Happy Buddha build + trace frame rate (cost ratio)

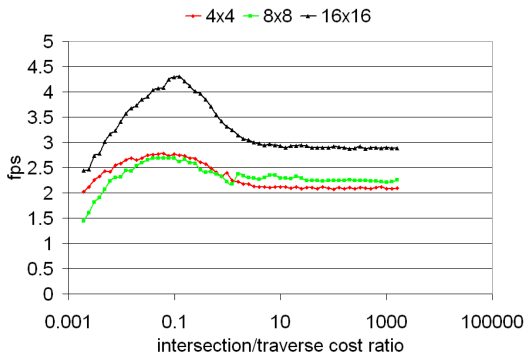

Figure 8: A10 build + trace frame rate (cost ratio)

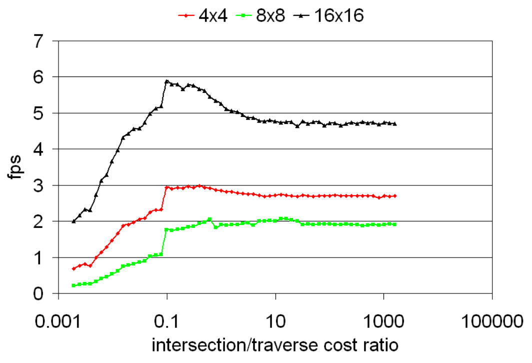

Figure 9: Sibenik's Cathedral build + trace frame rate (cost ratio)

The second evaluated parameter is the number of split planes necessary to produce BVH of adequate quality. We use $2^1 - 1$ to $2^{10} - 1$ split planes, meaning 2^1 to 2^{10} bins evaluated in each split. Two bins is BVH equal to spatial median split with no cost function and the approximation of SAH should improve with increase of number of bins giving us BVH with higher build but lower traversal time.

We can see from Figs. 10, 11 and 12 that we get almost constant tracing time with number of bins between 8 and 64. The most apparent improvement from the spatial median is in the Sibenik's Cathedral scene. We contribute this to the fact that this is the only scene in our test set where the camera is actually inside the scene and a good BVH can better eliminate geometry outside camera's field of view. The drop in build plus trace frame rate between 8 and 64 bins confirms our first impression that high number of bins brings no improvement to raytracing and only increases the build cost. The most noticeable feature is the very sharp drop of performance for 512 bins that is apparent in all three scenes. The number of split planes grows exponentially and each test always includes all split planes used in the previous tests. It is therefore not possible that a particularly good split plane position was omitted because the split was considered in test with 256 bins and not in the 512 bin test. We suspect two possible reasons for this behavior. The first is that is only a heuristic estimation of the cost function, not the actual cost function and it is possible that at 512 bins this estimation is particularly different from the actual cost function. The second reason would be an implementation error causing a serious glitch at this number of bins.

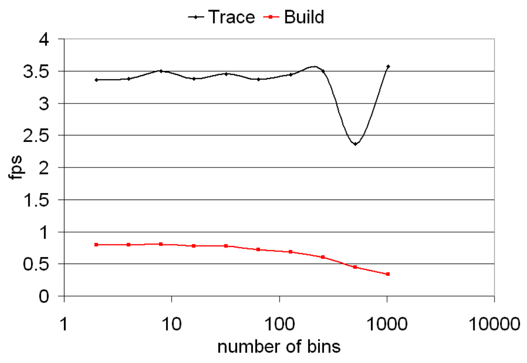

Figure 10: Happy Buddha build + trace frame rate (bins)

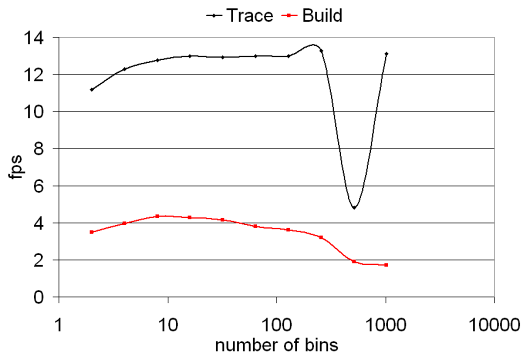

Figure 11: A10 build + trace frame rate (bins)

We have therefore performed a more detailed measurement (Fig. 13) for number of bins between 400 and 600. Should the behaviour be caused by an implementation error, we would expect extremely sharp drop in performance for some numbers of bins. However, it is apparent that the performance degradation is gradual as the number of bins approaches 512. We therefore assume that the reason behind this is a particularly bad estimation of cost function for all three models. We conclude that the optimal universal settings for further tests are:

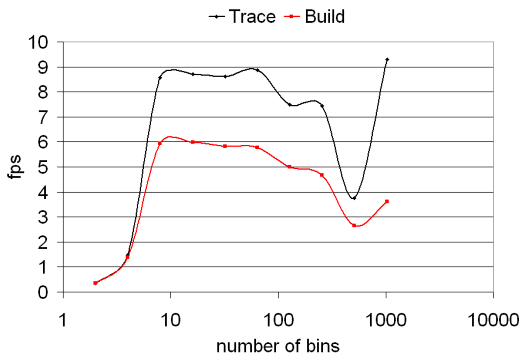

Figure 12: Sibenik's Cathedral build + trace frame rate (bins)

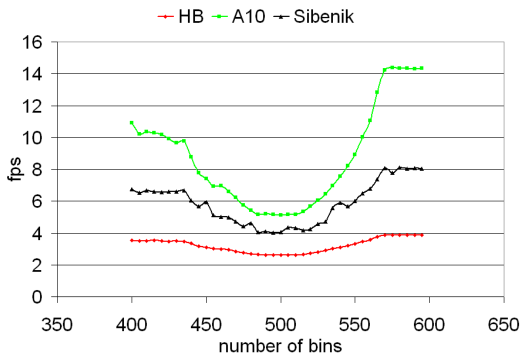

Figure 13: Frame rates for number of bins between 400 and 600

- Intersection/traversal cost ratio: 0.1
- Packet size: 16×16
- Number of bins: 8

6.2 Vertex culling

Once we determined good settings for BVH build we shifted our focus on effect of each particular vertex culling test. Because the use BVH parameters are the same for all measurements in this chapter we decided to measure only the tracing performance. We also included two sets for packet sizes 8×8 and 16×16 to determine effect of the tests on large and medium sized packets. The Figs. 14 and 15 show the results. Please note that the results are normalized, with vertex culling off used as 100% performance. For comparative performance between packet sizes see Figs. 7, 8 and 9.

We briefly sum up each test measured, for more detailed descriptions please see Chapter 4:

N/A - No vertex culling, only SSE2 intersection

TE - Separating packet by triangle edges

BP - Separating triangle by packets planes

NF - TriEdge plus consider when all corner rays hit the same triangle

AB - Flag and skip those rays that completely miss the tested leaf.

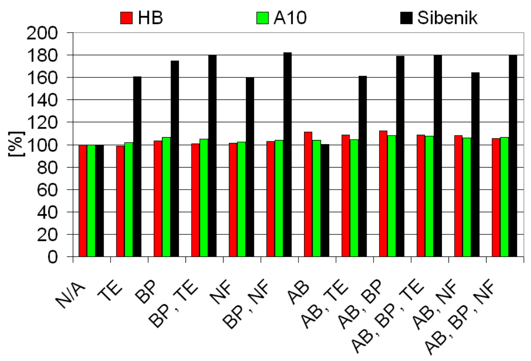

Figure 14: Relative speedup of different vertex culling tests, packets 8×8

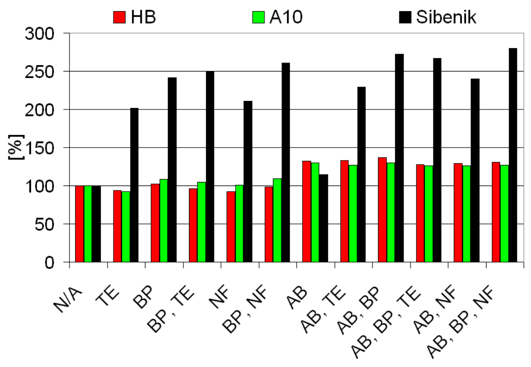

Figure 15: Relative speedup of different vertex culling tests, packets 16×16

We can see that the effect for both Happy Buddha and A10 models is negligible. AABB test brings 5-10% frame rate improvement for 8×8 packets and about 25% for 16×16 packets. The other tests have no or negative impact on performance. We assume that even with 30 triangles per leaf the triangles and subsequently the leaves are too small compared to packet's size to benefit from the culling tests.

The AABB tests does improve performance, because it does not try to cull triangles from packet but rather culls rays that would be tested for intersection even when they completely miss the leaf. This explains why the performance increase is higher for larger packets as they contain higher percentage of rays that benefit from skipping costly ray-intersection test.

The situation is exact opposite for the Sibenik's Cathedral scene, showing that the efficiency of the whole method is dependent on the character of displayed scene. The Sibenik scene contains relatively few (76k) large triangles and therefore most rays in a packet hit the same leaf even for 16×16 packets. The AABB test offers no improvement for 8×8 packets and gives only small performance boost (15%) for 16×16 packets, confirming this assumption.

However, the actual culling tests bring a significant improvement of the frame rate. When taken individually, the culling by packets planes is the most effective test, improving speed by 240% for large packets. Adding triangle edge tests brings only 10% more, even though the test by itself speeds raytracing by a factor of two. We can see that a large number of triangles can be culled by either of the tests. Adding NearFar test gives us yet another 10%, bringing the total speedup to 260% for large packets. The benefit for smaller packets is similar in composition, but the improvement over no culling is always smaller than for large packets. This is probably caused by initialization overhead that is not sufficiently amortized in this scene. We also see that the effect of AABB test can be almost directly added to speedup caused by culling tests, as it brings another 10-15% for large packets and no improvement for the small packets.

6.3 Packet shading

To evaluate the effect of our packet shading algorithm we decided to measure tracing and shading performance without BVH build time. The reasoning is the same as for vertex culling measurement. All tested versions use the same BVH setup, so the total build time is the same for all test runs and can therefore be omitted. We also decided to measure both 8×8 and 16×16 packet sizes. The results are summed up on Fig. 16.

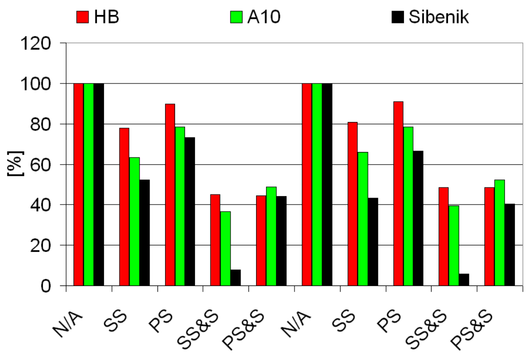

Figure 16: Comparison of Single Shading (SS), Packet Shading (PS), Single Shading & Shadows (SS&S) and Packet Shading & Shadows (PS&S) to frame rate of no shading (N/A) for packets 8×8 (left) and 16×16 (right)

We can see that shading single rays is inferior to our packet shading. Packet shading slows down the frame rate to 75-90% of performance without shading, while the single shading slows down to 50-75%. The lower values are for the Sibenik's Cathedral scene, where we use textures instead of just solid colors. The scene also occupies larger percentage of the resulting picture, meaning the ratio of shaded/non-shaded rays is higher than in the other two.

Once we take into account shadow rays it is obvious that our approach offers highly superior performance over shading and showing each hitpoint separately. We can see that the difference between single and packet approach is up to factor of 6 for Sibenik's scene and packets 8×8, the 16×16 packets give us a factor of 8. The improvement for Happy Buddha and A10 is quite minimal and we assume that this because of the character of the scenes. There are virtually no occluders between a hitpoint and a light for Happy Buddha, leading to very fast shadow ray traversal. There are few such occlusions on the A10 models and the benefit of tracing several shadow rays at once is small but apparent. The Sibenik's Cathedral scene contains a high number of occluders (e.g., columns and railing).

7 Conclusions

Our system combines three approaches suitable for ray-tracing of dynamic scenes. We confirmed the result published in [3] that using even only 8 bins gives us a BVH of good quality. Raising this number brings almost no speed up of ray tracing and slows down the build process. We also discovered that there is a drastic drop of performance when the number of bins approaches 512. We confirmed this by more fine measurement and concluded that most probable reason is a poor cost estimate when number of bins approaches this value from either side.

Another conclusion is that it is necessary to define desired result of BVH acceleration structure before setting its intersection/traversal cost ratio. For static scenes, when BVH is built only once and then traversed repeatedly, the best ratio is 1.0. However for dynamic scenes, when BVH is rebuilt every frame, the best ratio is 0.1 for all the scenes.

We used a packet traversal [4] of BVH and vertex culling [2] algorithm for packet-ray intersection. We added one more test to the vertex culling algorithm that allows us to skip intersection test for rays that miss the leaf. This yields us a 30% speedup for scenes with small leaves (e.g., Happy Buddha) and about 20% speedup for scenes with large leaves over the original vertex culling implementation. This improvement is however effective only for large packets (16×16), which proved to be the better choice for the overall speed of our system than the 8×8 packets suggested in [2].

Adding packet shading proved to be effective for all scenes. However the most apparent benefit is for fully textured Sibeniks Cathedral scene when not only shade rays in packets, but also evaluate shadow rays via packet shading.

We have not implemented any parallelization of either BVH build or the packet tracing. We believe there is an opportunity for another speed up for multicore machines. The packet shading algorithm currently suffers from per-ray access to textures. Ray coherence could be further exploited here, possibly reusing texture samples. Rays in packets are currently organizes row-wise. Considering we perform most operations using SSE2 intrinsics it could be beneficial to change this organization to raise the odds of four rays processed once behaving in a similar way e.g., hitting the same triangle or missing the same leaf.

References

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

[2] Alexander Reshetov. Faster ray packets - triangle intersection through vertex culling. In *SIGGRAPH '07: ACM SIGGRAPH 2007 posters*, page 171, New York, NY, USA, 2007. ACM.

[3] Ingo Wald. On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*.

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