Bounding Volume Hierarchy Construction

Niklas Sandén*

Lund University
Sweden

Abstract

This report discusses a few different ways of constructing a *bounding volume hierarchy* (BVH) used for accelerating ray tracing. It evaluates the performance of a few different *split methods* and discusses how to visualise the resulting BVH. One split method, referred to as the *median split method* in this report, created significantly lower quality results than the others on the high-poly meshes that were used for the experiments. Using the *Surface Area Heuristic* created higher quality results but took longer to construct the BVH.

1 Introduction

Ray tracing has multiple advantages over rasterisation when it comes to rendering certain effects. Effects such as reflections, refractions, shadows and indirect lighting more naturally map to the ray tracing paradigm. However, ray tracing is computationally expensive. One part of the algorithm that takes a significant amount of time is calculating the intersection points for the rays with a scene. One important part of modern ray tracers is *acceleration structures* that speed up this process, and is a necessity for larger scenes. This report will focus on one specific type of acceleration structure known as *bounding volume hierarchy* (BVH). BVH is based on subdividing the primitives of the scene. An alternative method is to use a *binary space partitioning tree* which uses spatial subdivision instead.

All of the BVHs considered in this report will work like the following: The BVH is a binary tree where the leaves are the primitives of the scene. Each node in the tree has a *bounding volume* (BV) that encloses all of the primitives in its subtree. The BV will always be an *axis-aligned bounding box* (AABB) in this report. The BVH has the property that if a ray does not intersect the BV in a node, then it will not intersect any of the primitives in its subtree either. This allows the ray tracing algorithm to avoid running certain intersection tests because it already knows that they will fail. It is also worth noting that ray-AABB intersection tests are computationally cheap. Figure 1 shows a simple example of a BVH for two primitives.

The project this report is about was a real time Vulkan application written in Rust. The BVH construction was done on the CPU while all of the ray tracing was done on the GPU leveraging compute shaders. While the application renders and builds the BVH in real time, the primitives are assumed to be static. Changing something about the scene requires rebuilding the entire BVH, which causes the program to freeze until it is completed. There exists techniques for efficiently refitting and rebuilding only parts of the BVH (see [Bikker]), but these are not considered in the report.

2 Algorithms

2.1 Ray Tracing

The ray tracing algorithm used to test the BVH was based on the one presented in Assignment 1 (2022) of the course EDAN35 at the Department of Computer Science, Lund University. This is a Whitted-style ray tracer where the materials support both having a reflective and a transparent factor. This means that a ray hitting a surface with a material might have to spawn two new rays recursively, one in the reflected direction and one in the refracted direction. Apart from those two, you also spawn one *shadow ray* per light source. However, *shadow rays* do not spawn new rays themselves and can therefore be handled separately without recursion. The only sources of light used in this project were a movable point light source and an atmosphere. The atmosphere was only used to give a colour to rays that did not hit anything. Therefore it did not contribute to surfaces without a reflective or transparent material.

One change in the ray tracing algorithm from the assignment was that the shadow ray does not just say whether the light is blocked or not, but rather how much light is blocked. This is done by multiplying together the transparency values of the surfaces along the path to the light source. The difference this makes is that shadows cast from objects with a material with a transparency value > 0 are lighter. This does not take refraction into account.

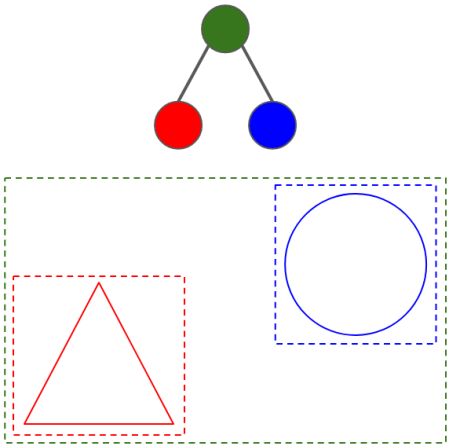


Figure 1: The top shows the BVH tree of a 2D scene containing a triangle and a circle. The bottom shows the scene with the BVs.

2.2 BVH Tree Traversal

To find the primitives that a ray intersects with, we traverse the BVH tree starting from the root. This was done in *depth first search* (DFS) order in this project. When visiting a node, we first check if the ray intersects the AABB of the node. If it does not, it means that the ray does not intersect anything in the subtree of this node and the subtree is skipped. Otherwise, we recursively call the child nodes if it is an internal node, or test for intersection against the primitive if it is a leaf node.

2.3 BVH construction

The BVH construction algorithms explored in this project all create the resulting tree top-down recursively. The main operation is: Given a set of primitives and their BVs, create a node with a BV containing all of the BVs and split the set in two. This is applied recursively on the new sets (which become child nodes) until we only have one primitive left, which becomes a leaf node. The way

*e-mail: ni5552sa-s@student.lu.se

the algorithms presented differ from each other is in the way the set of primitives is split. However, they all do split along a specified axis.

Choosing an axis to split on can be done in a few different ways. The way presented in [Shirley 2020] chooses the axis to split on randomly. Another way presented in [Pharr et al. 2016] is to look at the distance between the minimum and maximum coordinate of the centroids of the BVs along each axis, and pick the axis with the largest distance. Both methods were implemented in the project, but the results shown in this report are only from using the latter method. A final way (that was not implemented in the project) is to try each axis and pick the one which produced the best results [Pharr et al. 2016]. This is most relevant when using the *surface area heuristic* presented in Section 2.3.2 since that gives us a way to define which result was *better*.

2.3.1 Median/mean split method

Once the split axis has been decided we just need to decide a point to split on. Two simple methods are to split on the median or the mean. These are called "EqualCounts" and "Middle" in [Pharr et al. 2016] respectively. In both cases, the median or mean refers to their position along the specified axis. Splitting on the median could be done by sorting all of the primitives along the axis and then splitting at the middle element. This is how it is done in [Shirley 2020]. However, it is sufficient to just reorder the elements such that all elements in the first half of the array have a smaller coordinate than those in the other half. This can be done in $O(n)$ time instead of $O(n \log n)$ and is how it is handled in [Pharr et al. 2016]. Rust's standard library has a function on slices called `select_nth_unstable` which implements this. As for the mean split method, the mean of the centroids along the axis is calculated. Then the array is partitioned based on whether the coordinate is less than the mean or not.

2.3.2 Surface Area Heuristic

The previous two split methods only need $O(primitives)$ time to find a split point, which is relatively fast for a top-down approach. Although efficient, these simple algorithms for deciding split points can sometimes make poor decisions, leading to low quality trees that take longer on average to traverse than is necessary. One example for the median split method is demonstrated in Figure 3. A more complex and time consuming algorithm that produces better quality trees using a *surface area heuristic* is presented here. Whether or not a higher quality tree is worth the extra construction time depends on how many rays are traced against it compared to how often it is partially or entirely rebuilt.

The SAH is a cost function that we want to minimise. It is done greedily for each internal node in a top-down approach. Calculating the SAH requires that the set of primitives is already split. Let N_L and N_R be the number of primitives in the left and right child nodes respectively. Further, let A_P, A_L and A_R be the surface area of the AABB of the node and its children. Then the cost function is as follows:

$$c(P, L, R) = \frac{N_L A_L + N_R A_R}{A_P} \tag{1}$$

The motivation behind the usage of surface areas relates to the probability that a ray from a random direction that intersects the AABB of the parent node also intersects the inner AABB. This is described in [Pharr et al. 2016]. To find the split point yielding the minimum SAH, one can test all of the different split points and calculate the SAH for each one. This could be done by first sorting the primitives and then testing splitting after each primitive in order. In order to avoid the sorting and reduce the number of split points to check, one could use *binnned SAH*. This involves splitting the axis into B equally sized buckets (starting from the centroid with the lowest to the highest coordinate). We can find the bucket each primitive is in in linear time. Then we can try to split after each bucket. The implementation used in this project is based on the one in [Pharr et al. 2016] which is $O(primitives + B^2)$, but the authors state that it could be reduced to $O(primitives + B)$ by calculating the surface areas and BVs iteratively scanning from the left and right. The more buckets used the higher quality the resulting tree will be, but the longer it takes to construct.

3 Implementation Details

3.1 Recursion in Compute Shaders

All of the ray tracing is done in compute shaders written in GLSL which does not support recursive function calls. This poses some challenges when implementing some of the aforementioned algorithms since they are most easily described recursively. The ray tracing algorithm used allows each ray to recursively spawn **two** new ones. When traversing the BVH-tree, each internal node has **two** child nodes that we need to recursively check. Both of these recursions can be handled by having a *stack* that stores the information about work that needs to be done. In the case of ray tracing, it stores the ray, the light contribution factor, and the depth. As for the BVH-traversal, the stack only needs to store a pointer to an unevaluated node. Since each node can have at most **two** children in both algorithms, the amount of elements that needs to be stored in the stack at the same time is bounded by $O(depth)$. The ray tracing is already capped at a certain depth value, but the largest depth of the BVH is not. The characteristics of the resulting trees, including how close they are to being balanced, are briefly discussed in Section 4, but this quality greatly depends on the split method used during BVH construction. One could imagine building a BVH where the height is $O(primitives)$. The current GLSL implementation uses a stack-allocated array for the stack, which would not be possible if the size of it could grow into the millions. However, all of the split methods empirically construct trees with a much smaller height even for millions of primitives, so this is not an issue in practice.

It is worth mentioning that as for the BVH, it is possible to traverse it without using the stack based approach. One way is to do the DFS traversal ahead of time when constructing the BVH and for each node save the node it traverses to right after its subtree has been evaluated. Then the GLSL implementation only needs a pointer to the current node, and it is still able to perform the full DFS traversal (while skipping subtrees early if the ray does not intersect the BV).

3.2 Pipelines

The project uses three different pipelines in order to render one frame. These can be seen in Figure 2. Both the *BVH Rendering* and the *Ray Tracing* pipelines write their results to the same *Output Image*. Then there is a third pipeline that just renders the *Output Image* as a texture onto a fullscreen quad, the result of which is stored in a *Swapchain Image* ready to be presented to the screen. The reason why the rendering algorithms do not output their results directly to a swapchain image is because it is not guaranteed that swapchain images can be written to directly from a compute shader, which is done in the *Ray Tracing* pipeline. The idea of using a graphics pipeline to just render the results of the ray tracer to a quad came from the *computeraytracing* example by Sascha Willems [Willems].

3.3 BVH Rendering

Rendering the BVH is not done through ray tracing but rather through rasterisation for efficiency. The visualisation of the BVH is not supposed to be shaded, but it should be in 3D space and should be able to block and be blocked by the ray-traced primitives.

The first step is to use instanced rendering to render all of the BVs at once. Since they are all axis aligned, the only per instance data needed is a scale and an offset. Each instance is rendered using

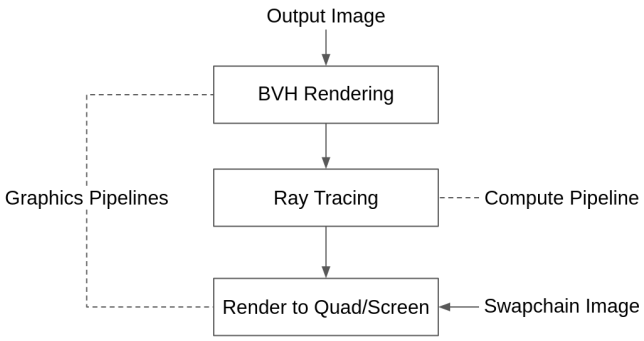


Figure 2: The three different pipelines used to render a frame.

the line primitive with each edge of a cube. In the fragment shader, the line colour is stored in the RGB-components in the output image. In the alpha component, we instead store the distance from the camera to the fragment. This distance is essential for the ray tracing step. This pipeline also uses a depth buffer, meaning the value in the alpha channel will be at the front.

When running the ray tracing algorithm in the compute shader, we first check if the corresponding pixel was written to by the BVH Rendering. If it were, we discard any intersection points for the first ray where the distance to the point is greater than the alpha value. If no intersection point was found, then we use the colour stored in the output image. When using this approach the lines will not be visible in reflections or refractions.

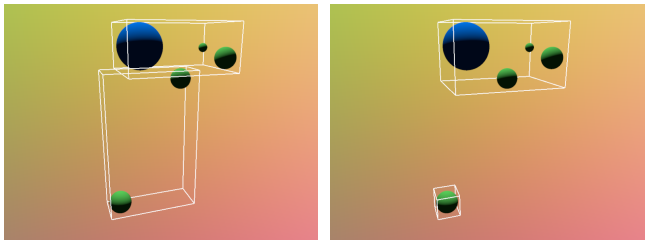
3.3.1 Rendering a subset of BVs

The program allows dynamically specifying a filter of depth values that should be rendered. Any BV with a depth value (in the BVH tree) not in the range will not be rendered. The part of the `VkBuffer` containing the per instance data was sorted according to these depth values when it was filled. With this memory layout, it suffices to change the offset and size values sent to the `vkCmdDrawIndexed` function to render any specific range of depth values.

4 Results

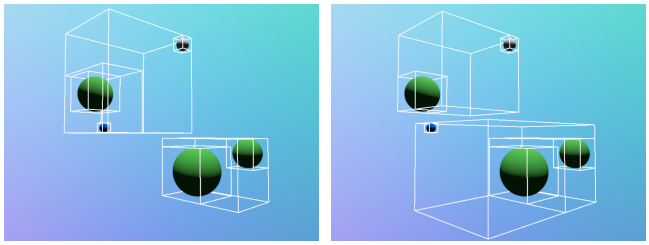
4.1 BVH Showcase

Figure 3 shows the BVs after splitting the set of five spheres once. SAH and the mean split method chose to split at the same point while the median method chose a different one. Splitting on the median ensures that the counts of primitives in the two resulting sets differ by at most one, which results in picking a seemingly bad split point. Other times, SAH and the median split method are the ones who split at the same point. Figure 4 shows all of the BVs except for the one containing all five spheres.



(a) Median (b) SAH & Mean

Figure 3: Shows the BVs after splitting the spheres once using the different split methods. SAH and Mean produced the same results.



(a) Median & SAH (b) Mean

Figure 4: Shows the BVs at all depth values (except for the root) using the different split methods. Median and SAH produced the same results.

Figure 5 shows the BVs at depth eight and nine on a high-poly model. Interestingly, this frame is rendered faster than it would if the BVH was not rendered at all (around 12 ms instead of 17 ms). This is because every pixel that is covered by a BV outline in the final image would only have sent a single ray into the scene (more if super sampling is enabled), as described in Section 3.3. Rendering the BVH is very efficient with instanced rendering, so the time saved in the compute shader ends up being more than the time spent rendering the BVH. The line width was doubled when taking the screenshots for this report to make them easier to see, which further increases the amount of time saved.

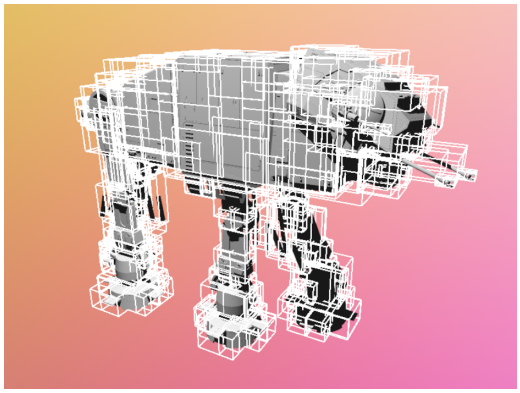


Figure 5: Only the BVs with depth eight and nine in the tree are rendered. This model contains over 700 000 triangles.

4.2 Performance

The performance varies greatly depending on the scene geometry and the camera's position and direction. Nevertheless, Table 1 is provided with some performance metrics when the different split methods were used to render the image seen in Figure 6. SAH outperformed the others but took longer to construct the BVH. Increasing the bin size from 12 (which was the default value in [Pharr et al. 2016]) did not make a comparatively large difference in frame time, but did drastically increase the build time. The most notable performance difference is how much better the mean split method performed compared to the median one. Although it performs worse, the focus on splitting the set into two of equal count by the median split method results in a more balanced binary tree with a smaller height. The mean split method produces the most unbalanced trees, but this ends up not being an issue because the tree height is still small.

Although the performance varies with different models and camera setups, the relative ranking of the split methods remains the same as in this specific experiment. Another scene that was tested

was one made up of spheres placed uniformly at random in a cube volume. When the primitives are placed like this, all split methods produce much more similar trees, with similar frame time and tree height compared to the results in Table 1.

Table 1: Performance data when using the different split methods to render Figure 6. It shows the average frame time, the construction time and the resulting BVH tree height.

Split method	Frame time *(ms)*	Build *(ms)*	Tree height
Median	41.8	105	20
Mean	30.1	124	34
SAH 12 bins	26.5	255	30
SAH 32 bins	26.1	604	30

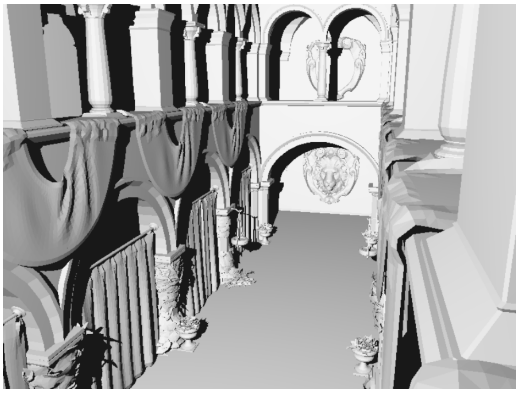


Figure 6: The Sponza scene rendered at 800x600 pixels. This model contains over 250 000 triangles.

5 Discussion

5.1 Optimisation attempts

There is actually another part to the SAH-based algorithm described in [Pharr et al. 2016] that is not implemented in the latest version of the project. If leaf nodes can contain multiple primitives, then one can calculate the cost of creating a leaf node right away with all of the primitives and compare it to the best cost found when splitting. When doing this, the cost function for splitting (1) contains an additional term (see [Pharr et al. 2016]). This was later removed from the project in favour of code simplicity since it did not have any noticeable impact on performance.

The program supports having both spheres and triangles as primitives in the same scene. These require different types of data during ray tracing. The current implementation has one storage buffer with all of the data for the spheres, and a separate one for the triangles. A BVH leaf node contains the index into one of these buffers along with a flag specifying whether it is a sphere or a triangle primitive. A different approach that was once tested was to store spheres and triangles in the same buffer (padding the smaller of them). This required reinterpreting the bits of some of the fields into a different type in the shader. This approach had very little positive impact on performance (if any) and made the compute shader more complex and was therefore not kept.

The last thing that was tested in the project was to add double buffering to see if it could increase the performance. Almost all of the frame time is taken up by the compute shader (ray tracer) alone. This means that the only real downtime is while the CPU is re-recording to the command buffer after it is no longer in use by the GPU. I was also thinking that maybe enqueueing the next frame's work would allow the scheduler to more efficiently distribute the workload. However, the performance difference was not noticeable and therefore removed in favour of simplicity.

5.2 Conclusions

The results turned out well, particularly the visualisation of the BVH trees. The most surprising revelation was that the mean split method consistently outperformed the median split method by a good margin on all of the high-poly models that I tested them on (which are more than those presented in the report). The mean split method also came respectably close to the performance of SAH. Having a BVH drastically increases the scene complexity that is feasible to ray trace, even when using the median split method which performed the worst. Before adding the BVH, the application had no chance of ray tracing those complex scenes with hundreds of thousands of primitives. Even after just 1000 primitives, the frame time was already worse than the frame time of rendering the Sponza scene using the median split method.

A potential next step for the project would be to look into refitting and partially rebuilding the BVH to allow rendering dynamic scenes in an efficient way. There is also *Linear BVH* (LBVH) which constructs the tree bottom-up instead and is easier to parallelize than the methods presented in this report. The goal of the project was originally to explore dynamic BVH construction, but ended up instead focusing on the SAH and visualising the BVH tree.

References

BIKKER, J. How to build a bvh. `https://jacco.ompf2.com/2022/04/13/how-to-build-a-bvh-part-1-basics`. Accessed: 2022-12-18.

PHARR, M., JAKOB, W., AND HUMPHREYS, G. 2016. *Physically Based Rendering: From Theory to Implementation*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Nov.

SHIRLEY, P., 2020. Ray tracing: The next week. `https://raytracing.github.io/books/RayTracingTheNextWeek.html`, December.

WILLEMS, S. Vulkan c++ examples and demos. `https://github.com/SaschaWillems/Vulkan`. Accessed: 2022-12-05.