Vrije Universiteit Amsterdam

Bachelor Thesis

# A Comparison of Acceleration Structures For Ray Tracing

**Author:** Basel Sammor (2608511)

*1st supervisor:* Atze van der Ploeg

*A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science*

July 12, 2020

# Table of Contents

## Contents

**Abstract**

To determine the advantages of different acceleration structures, a comparison of the performances of 3 of the most popular acceleration structures, Bounding Volume Hierarchies, KD-trees, and Octrees was carried out. The main benefit of the acceleration structures is to increase the rendering speed of a slow naive ray tracer. An explanation is given of the acceleration structures, how they are constructed, and traversed. An experiment consisting of 7 scenes of varying triangle counts and complexities was performed to measure the performance of the structures. The results indicate that Bounding Volume Hierarchies perform the best with low to high poly count scenes non-complex scenes and have the fastest build time, KD-trees perform the best with low to high poly count complex scenes and have the slowest build time, and Octrees perform worse than KD-trees & Bounding Volume hierarchies but have the second-highest build time.

# 1 Introduction

Ray tracing is a rendering technique in computer graphics where images or frames are produced from a scene of objects (often made up of triangles) with realistic lighting effects (e.g: reflection, refraction, global illumination) by simulating and tracing light rays in the scene. The resulting images can be incredibly realistic and hard to distinguish from real-life if done well. The algorithm itself is not a recent discovery by any means, as it was first described by Albrecht Dürer in the 16th century [1].

Despite the fact that ray tracing existed for so long and can perform such a feat, why has ray tracing historically not been utilized as much in mainstream rendering applications, specifically real-time and in video games, a multi-billion dollar market? The main issue with ray tracing is the time it takes to render images or frames. The accuracy of a ray tracer comes at a price, it is much slower than less accurate, real-time rendering techniques. This has led to ray tracing being utilized mostly in "offline rendering farms" and movie CGI where animation studios can take as much time as they please to take advantage of ray tracing and render realistic frames for their goal. Over the recent years, ray tracing has seen a significant increase in popularity mainstream real-time rendering because of newer, more powerful GPUs. One big example is Nvidia and its Turing GPUs [2].

The basic idea is simple, to determine a color of a pixel, shoot a ray through a pixel grid to the scene from a camera and determine if it intersects with an object. The ray-object intersection tests account for most of the time a ray tracer spends on rendering a scene. Hence, over the years, a large amount of research has gone into optimizing ray tracing and making it more viable for real-time rendering. One way to accomplish that is to find ways to minimize the intersection tests with the objects composed of possibly millions of triangles in a scene. This has resulted in the invention of acceleration structures that vastly reduce the number of intersections that a naive ray tracer would perform.

Following the trend, ray tracing is expected to gradually become more ubiquitous and popular in many graphical industries, and the need for efficient & optimized ray tracers will surely follow. The main focus of this paper is to compare the performance of the more popular acceleration structures, Bounding Volume Hierarchies, Octrees, and KD-trees [3], on a Ray Tracer implemented in C++ and on a variety of scenes to explore their

acceleration effects and to have a general idea of which acceleration structure performs best.

For a general outline of the paper, Section Two explains the Background Information where ray-tracing, acceleration structures, Bounding Volume Hierarchies, KD-trees, Octrees, and Surface Area Heuristics will be explained in detail to have a better understanding of the paper moving forward. Section Three will be the setup of the ray tracer which will include an overview of how the ray tracer was implemented in C++ to assess the correctness of the implementation for answering the research question. Section Four is the Results section which explains the experiments run to compare performances, their measurements & results, and graphs to visualize the results. In Section Five, the Conclusion, the results will be analyzed, discussed, and an answer to the question stated will be provided. Moreover, there will be a summary and discussion of the overall paper and mentions of any future recommendations and shortcomings encountered in the research.

# 2 Background Information

## 2.1 Intersection tests

One of the most important parts of a ray tracer which could sometimes take up to 95% of the time a ray tracer spends on rendering is the ray-intersection tests [4]. The first way of utilizing the ray-intersection tests occurs when a ray tracer wants to determine what is visible to the camera by shooting rays through a 2D image grid and performing ray-intersection tests with each primitive in the scene. This can get quite big quite fast due to the double nested for loops required. The first for loop is for all the pixels in the image grid, while the second for loop is for all the primitives in the scene.

Most sophisticated scenes in 3D graphics are represented by a large number of triangle primitives to make up Triangle meshes because having only primitives such as spheres, cubes, or single triangles does not allow us to represent anything useful for media or games. One example of rendering such a triangle mesh would be a scene containing a mesh composed of 1,000,000 triangles and we want to render an image with a resolution of 1920x1080. This results in $2.0736*10^{12}$ ray-intersection tests that need to be performed, or two trillion, seventy-three billion, and six hundred million. This is just for the primary utilization of the ray-intersection tests to determine visibility and it is already quite unfeasible for rendering high polygon count scenes. Keep in mind that other ray-intersection tests must also be used to determine shading such as shadows and reflections which adds fuel to the fire. This is not applicable for anything non-trivial or real-time based and ways to reduce unnecessary ray-intersections tests are required. The different acceleration structures discussed below achieve that by only performing a small number of ray-intersection tests for each ray by restricting the primitives to the ones the ray will be most likely passing through.

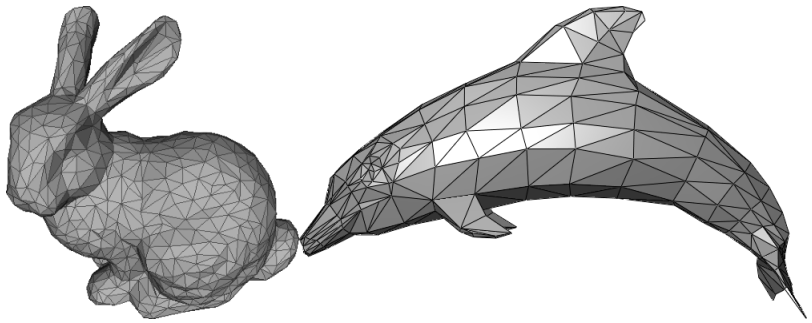


Figure 1: Triangle Meshes [5][6]

### 2.2 Bounding Volumes

to encompass every object in the scene, whether it's a primitive or a mesh, by an axis-aligned bounding volume, usually a box. Before a ray performs the intersection tests with an object, possibly consisting of millions of triangles, it will test if it intersects the box that surrounds the entire object. If the ray does intersect the box, we test everything inside. If not, we skip everything inside the box. This simple method can save a lot of computational power but still has the drawbacks that we need to check everything inside the box once an intersection occurs. The box for a triangle mesh or can be simply created by looping over all of the triangles in the mesh and finding the minimum point and the maximum point (x, y, and z values) [7].

### 2.3 Acceleration Structures

The three Acceleration Structures are ray tracing optimizations where tree data structures are used to minimize the number of intersection tests by going down a specific branch of the tree and performing intersection tests on the small numbers of primitives in the leaves [8]. Every acceleration structure has two main components: 1) Constructing the tree out of a scene, and 2) traversing that tree to find what reduced number of primitives to test for intersections.

The three structures are separated into two categories, **Spatial subdivision** and **Object subdivision/Hierarchies**. In Spatial subdivision, the scene's space is divided into a hierarchy of disjoint sets, while in Object subdivision the primitives themselves are divided into a hierarchy of disjoint sets. This means that in Spatial subdivision, the chances that a primitive is in more than one leaf node is greater since the focus is more on the space and not the primitives themselves. Figure 2 gives a visualization of the difference [8].

A traversing algorithm can either be **Top-Down** or **Bottom-Up**. Top-down algorithms work by starting from the root node in the tree and recursively go down the tree until they reach a leaf node, while bottom-up algorithms work by starting from the leaf nodes [9].

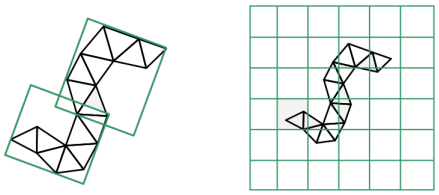


Figure 2: Object vs Spatial subdivision

### 2.3.1 Bounding Volume Hierarchies

A Bounding Volume Hierarchy is an Object subdivision structure that extends the idea of bounding volumes. As the name implies, it revolves around the idea of creating child bounding boxes inside a parent bounding box in the form of a binary tree. A parent node is split into two child nodes according to a chosen cost function which determines at what place to split the parent and partition the primitives inside. After the split is done, new bounding boxes are calculated for the children nodes. The splitting continues recursively until a single primitive's bounding box remains or a split would cost more than not splitting. This does mean that on occasion a leaf could contain more than one primitive.

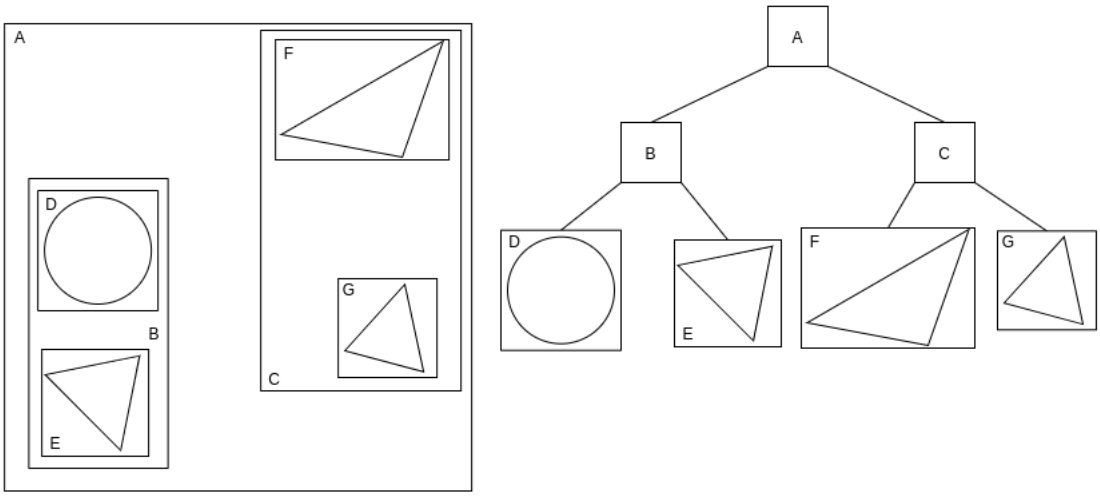



Figure 3: Bounding Volume Hierarchy

**Construction** First, two parameters, intersection cost, and traversal cost are chosen and will be used later to determine if it is worth to split a node and where. Next, the bounding box of the entire scene, which will act as the root node and is called the world bound, is calculated along with the primitive bounds. The construction then begins by figuring out which axis (x, y, or z) to use when it comes to splitting the parent node because focusing on one axis is more efficient than testing splits in all three. The utilized method in this paper is to project the center of every bounding box of the primitives

inside the current node to each axis and choose the one that has the largest range between any two centers. Figure 4 shows an example in 2D where the y-axis is chosen because projecting the centers to it gives us the biggest range possible. This is done because an axis that has this property will allow for the least amount of overlap of bounding boxes when splitting the node and in practice gives a good enough result but does not always result in the best axis [10].

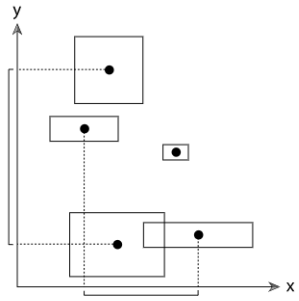



Figure 4: Axis Projection [10]

Next, the Surface Area Heuristic algorithm or SAH is used to determine where to split the root node in the previously chosen axis. SAH is an algorithm that estimates the cost for a ray to traverse the nodes and perform ray intersections tests for a specific partitioning involving all of the edges of the primitives' bounds. This means that whenever a node is to be split, the different possible partitions can be treated as leaves to test for intersections and the one that results in the least cost is chosen. Once the split plane is chosen, the construction recursively continues with the children until the tree is complete [11].

$$c(A, B) = t_{trav} + P_A \sum_{i=1}^{N_A} t_{isect}(a_i) + P_B \sum_{i=1}^{N_B} t_{isect}(b_i)$$

The SAH can be seen above. c(A, B) is the cost of partitioning a node into children A and B. $t_{trav}$ is the time of traversing an interior node. $P_A$ and $P_B$ are the probabilities of each intersecting the two children nodes. They can be calculated by dividing the surface area of the child node by the parent node. $N_A$ and $N_B$ are the numbers of primitives in each region. $a_i$ and $a_b$ are the ith primitive in each child node. Lastly, $t_{isect}$ is the cost of intersecting a single primitive. $t_{trav}$ and $t_{isect}$ are constants that are assumed before executing the algorithm [11].

To calculate the cost for a minimal number of possible splits, the implemented BVH algorithm calculates the SAH cost of the edges of "buckets". The algorithm splits the axis into buckets of equal extents as seen in Figure 5. The buckets are the areas between two adjacent non-dotted vertical lines. A primitive is placed in the bucket depending on where the center of its bounds resides. The costs of the blue lines shown in the Figure are then calculated and the one that produces the least cost is selected [11].

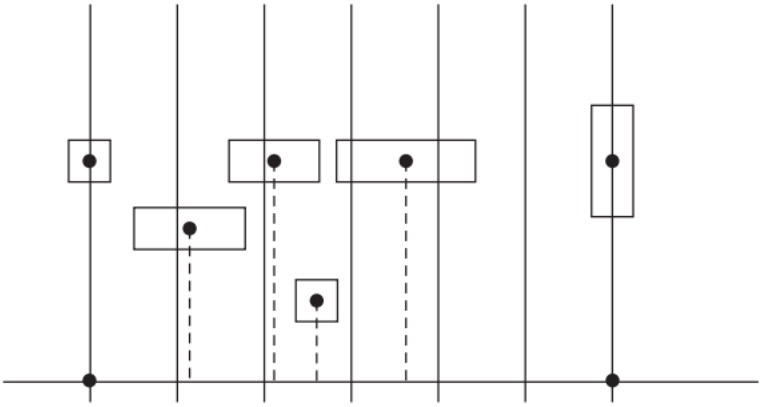


Figure 5: SAH Buckets [11]

**Traversal** The traversal of a bounding volume hierarchy is quite simple. We start by testing the world bound for an intersection. If one occurs, we recursively traverse the nearest child to the ray origin and testing its bounding box. This way if there is a primitive in the nearer child, we will not need to traverse the farther child and save computation power since that primitive will always be in front. The recursive procedure continues until we reach a leaf node and test the primitive inside for intersection with the ray [12].

### 2.3.2 Octrees

An Octree is a Spatial subdivision structure that revolves around the idea of each non-leaf node in the tree having 8 children, hence the name Octree. This can be extended to accelerating a ray tracer by thinking of each node as a bounding box and its children are 8 bounding boxes produced by splitting the parent using 3 constant splitting planes that go through the center of the bounding box [13]. This results in the structure shown in Figure 6.

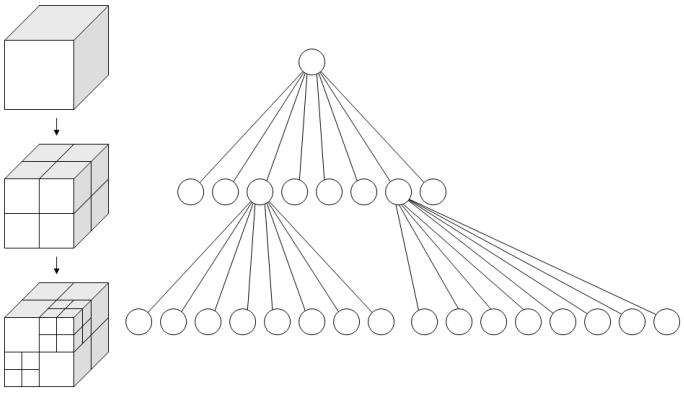


Figure 6: Octree [14]

**Construction** Before beginning the construction, a tree depth and a minimum number of primitives in a non-leaf node must be chosen. This ensures that the Octree has a stop

condition that checks if the maximum depth has been reached or if a node has fewer primitives than the minimum number to stop splitting. This is done to avoid the Octree growing too large and requiring an unfeasible amount of memory since the tree can grow quite large due to the nature of having 8 children. These two parameters affect the results significantly and need to be chosen carefully depending on the scene [15].

Once those two values have been chosen, The construction begins by calculating the root node's world bound. The stop conditions are then tested to check if the maximum depth has been reached or the world bound contains enough primitives to perform a split. if either is true, the root node is marked as a leaf and the construction stops. Otherwise, the center of the world bound center box is calculated along with the implicit corners to create the 8 children bounding boxes by finding the minimum and maximum between each of those corners and the center.

After creating the 8 new children, the primitives inside the parent node need to be distributed along with them by finding which out primitives overlap with which of the children. This can be accomplished by finding out if the bounding box of a primitive overlaps with the child by checking if the extents of both boxes overlap in all axis. The construction algorithm continues recursively for each node until a stop condition evaluates to true and the node is marked as a leaf.

**Traversal** The Octree traversal algorithm chosen is a top-down algorithm created by Revelles et al [9]. The algorithm begins by testing the world bound for intersection and if one occurs, it finds out the first child node that the ray interests. This starts by figuring which plane the ray enters the node from. This plane can either be the XY plane, the YZ plane, or the XZ plane as shown in Figure 7. Once the plane is found, the algorithm will consider the four nodes that are closest to that entry plane. Only four nodes are considered for the first child node because when entering a rectangular shaped object, the ray will always enter through one of four nodes. Depending on how the ray intersects the node, one of the four nodes is chosen as the first child-node. Since the first child-node is found, we can delve recursively into that node and check for intersections once we reach the leaves.

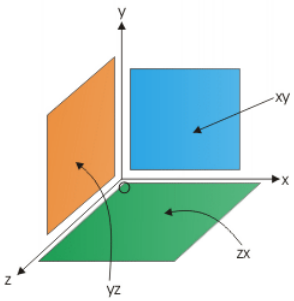



Figure 7: XY, YZ, XZ Planes

Even though the first child-node was found, the next intersected child-node is important to figure out as well because if no intersections occur in the first child-node, delving into

| Current child-node | Exit Plane YZ | Exit Plane XZ | Exit Plane XY |
|---|---|---|---|
| 0 | 4 | 2 | 1 |
| 1 | 5 | 3 | END |
| 2 | 6 | END | 3 |
| 3 | 7 | END | END |
| 4 | END | 6 | 5 |
| 5 | END | 7 | END |
| 6 | END | END | 7 |
| 7 | END | END | END |

Table 1: Next child-node automation

that next child-node will be necessary. The Revelles et. al algorithm automatically finds out which node is next by considering the state (child-node) we are currently in and the exit plane of the ray. To construct such automation, the numbering of child-nodes must be constant from 0 to 7. The number and the axis system is represented in Figure 8.

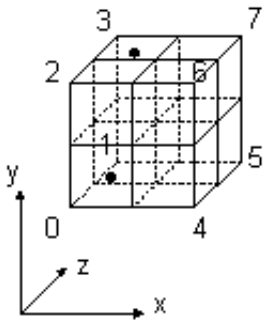



Figure 8: Indexing child-nodes

Starting from the current sub-child, there are only three possible next nodes. For example if we are in sub-child 0 in Figure 8. the possible next child-nodes are 1, 2, and 4. To figure out which one is next, we determine what the exit plane of the ray is. If the exit plane is the XZ plane, the next child-node will be 2, if it is the YZ plane, the next child-node will be 4, and if it is the XY plane, the next child-node must be 1. Generalizing this to every possible state, Table 1 shows the entirety of the automation. End represents the ray exiting the node since there are no next child-nodes.

### 2.3.3 KD-trees

KD-tree is a Spatial subdivision structure where each parent node is split into two child nodes using a plane, resulting in a binary tree. This plane is adapted according to the configuration of the primitives inside the node. The only restriction on the plane is that it must be perpendicular to at least one of the axis.

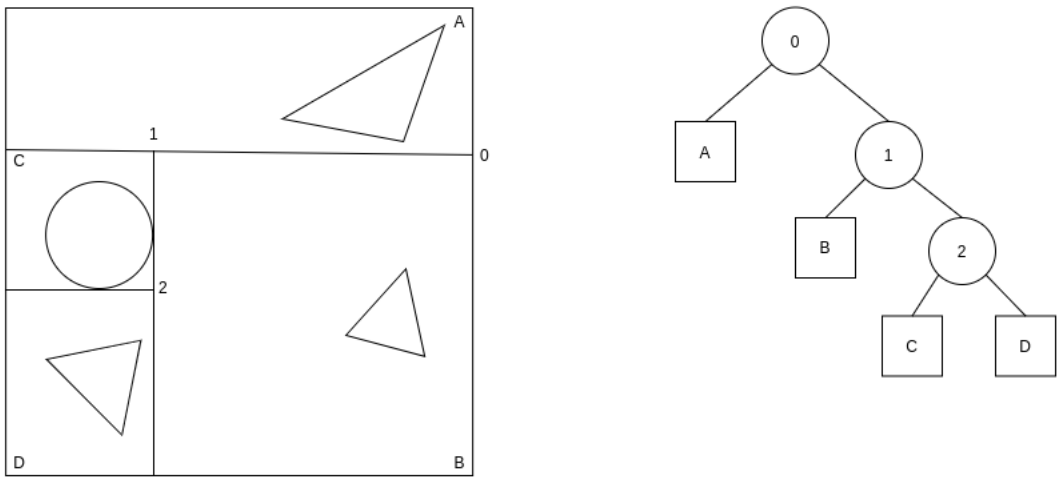



Figure 9: KD-tree

**Construction** Before the construction begins, a value for the maximum depth, intersection cost, traversal cost, and minimum numbers of primitives in a non-leaf node must be chosen. These values play a role in deciding when to stop the recursive procedure to avoid memory overflow similar to Octrees and using the Surface Area Heuristic to determine if it is worth to split and the split plane position. Again, these parameters are important as they have a great effect on the performance of the structure [16].

Similar to the other structures, the algorithm begins by calculating the world bound of the scene. The maximum depth condition and a minimum number of primitives in a leaf are then checked to determine if the node is a leaf or to continue the recursion. If the node is a leaf, the algorithm inserts the primitives overlapping that region of space into the node and stops working down that tree branch. If the node is an interior node, the algorithm determines the split with the least cost and recursively continues the construction on the resulting child nodes. [16].

Like when choosing a split plane for a BVH in Figure 4, the axis that gives the largest extent is chosen to begin with when considering the planes. If no good split was found in that axis, other axes are checked afterward. Additionally, only a specific set of planes are considered when calculating the SAH cost instead of considering each possible split in the chosen axis. These planes are the same planes that represent the edges of the bounding boxes. $a_0, b_0, a_1, b_1, c_0, c_1$ in Figure 10 are examples of split planes considered for the x-axis. Since each bounding box has two edges for a chosen axis, the maximum number of edges to test is $2 * numPrimitives$. [16].

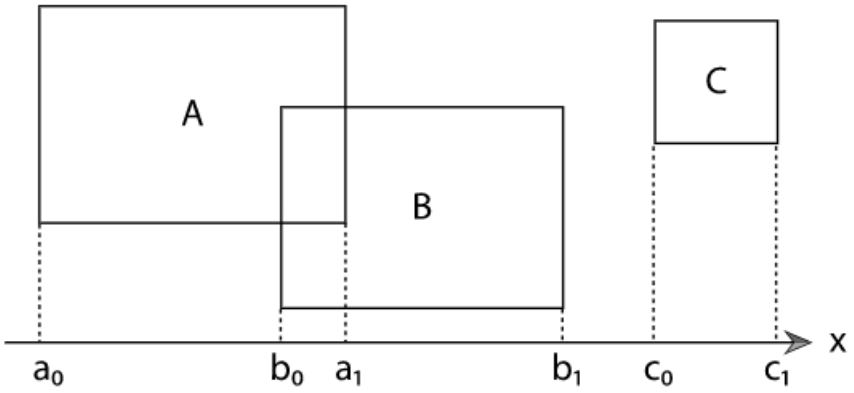



Figure 10: Edges as Split planes positions [16]

Once the edges have been found, they are stored and sorted from the lowest to the highest in the chosen axis. The cost is then calculated for the edges and the best one is chosen according to a slightly different SAH shown below. This is because since KD-trees are spatial, it is better to choose a split where one of the two child nodes is empty of primitives to avoid going down that branch completely [16].

$$c(A,B) = t_{trav} + P_A \sum_{i=1}^{N_A}(t_{isect} * (1 - b_e))(a_i) + P_B \sum_{i=1}^{N_B}(t_{isect} * (1 - b_e))(b_i)$$

The variables are the same as the previous SAH except for the addition of $b_e$. $b_e$ is a value that is 0 unless one of the child nodes is empty of primitives. Otherwise, it takes on a value between 0 and 1 that is chosen as a parameter before the algorithm starts. In the updated SAH, $1 - b_e$ would result in a lower cost if a child node was empty (i.e, $b_e$ would be greater than 0) [16].

Depending on how the primitives and their bounding boxes are positioned, the algorithm might sometimes be unable to find a position that would be considered good and instead assign the node as a leaf. There are two main scenarios where this occurs. The first involves the case where all splits result in a cost higher than not splitting. This can be dealt with some leniency where a certain number of bad splits is allowed until it reaches the maximum number specified. The second case is when many bounding boxes overlap and all splits would result in the primitives being in both child nodes [16].

**Traversal** The traversal begins by checking if the ray intersects with the world bound. If it does not, the traversing exits immediately. If the node is a leaf, ray-intersection tests are performed to find out which primitive is visible. If the node is an interior node, the algorithm finds out which of the two children was intersected first and delves into that child. The children are called near or far depending on their proximity to the ray origin. It is not always necessary to traverse both children as the ray sometimes enters a node in such a way that results in entering one child node only. Looking at Figure 15, (a) shows

the ray only intersecting the near child. (b) shows the ray only intersecting the far child [17].

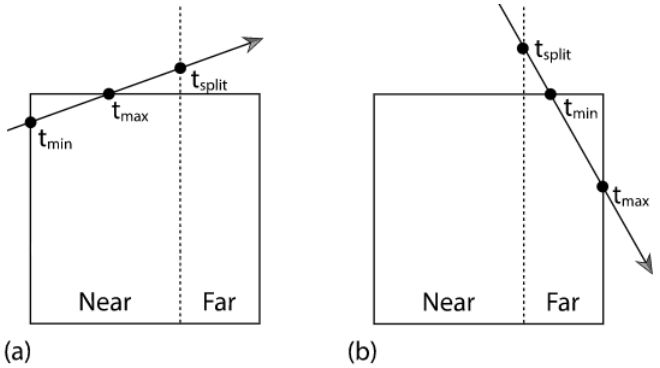



Figure 11: Near and Far child [17]

# 3 Ray Tracer Setup

The Ray Tracer was implemented in C++ with an emphasis on Object-Oriented Programming. OOP has been crucial in certain parts where Inheritance and Polymorphism were essential. The project consists of 1928 lines of code excluding any third-party libraries and is separated into appropriate translation units. The units can be divided into the categories shown below.

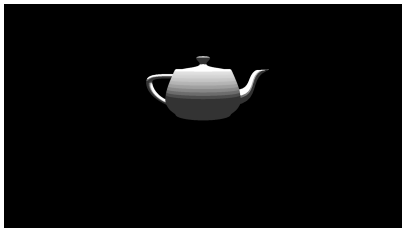
The Core, Math, Images, Geometry units were inspired by material on **Scratchapixel.com** [18], while BVH & KD-tree were heavily influenced by the online copy of **Physically Based Rendering: From Theory To Implementation** [19]. Lastly, the Octree was implemented using the material from **Revelles et. al paper** [9] and the website flipcode [15]. The code-base for the project and detailed instructions on how to build and run the ray tracer can be found at GitHub Repository.

- Core Ray Tracer: contains the main C++ components that make up the Ray Tracer.
  - Main, Base_Tracer, Camera, Ray, Light,
- Math: contains the C++ units representing all of the mathematics and constants needed for a functional ray tracer.
  - Vec3, Globals
- Images: contains the C++ units representing the handling and storing of images and textures.
  - Color, Texture, Utilities
- Geometry: contains the C++ units representing all of the different primitives and objects.
  - Object, Sphere, Plane, Triangle, BBOX, tiny_obj_loader library (.OBJ).

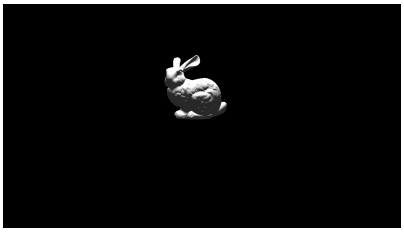
- Acceleration Structures: contains the C++ units representing the three acceleration structures.
  - Tree, Octree, KD-tree, BVH
- Libraries: contains the C++ units representing third-party libraries.
  - stb_image, tiny_obj_loader

# 4 Experiments & Results

To gauge the different construction time and rendering time of the three acceleration structures, 7 different scenes will be utilized. The scenes differ in the number of primitives they contain and the complexity. The first 4 scenes are simple scenes that have a varying amount of triangles starting from 6,300 triangles to over a 1,000,000 triangle. The last 3 scenes are more complex scenes with a triangle count starting from 260,000 all the way to around a 1,000,000.

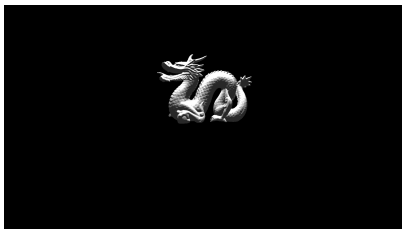


(a) 6.3k Triangles

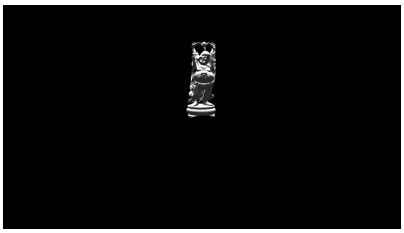


(b) 144k Triangles

Figure 12: Utah Teapot & Stanford Bunny



(a) 871k Triangles



(b) 1087k Triangles

Figure 13: Stanford Dragon & Buddha

(a) 262k Triangles

(b) 533k Triangles

Figure 14: Sponza & Room

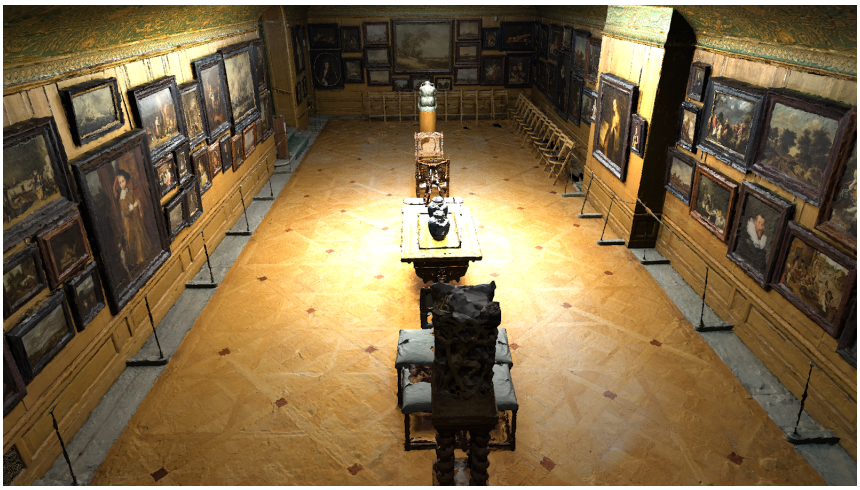


Figure 15: Gallery 999k Triangles

## 4.1 Experimental Setup

Resolution: 1920x1080
Light Source: 1 Basic Light source
CPU: AMD Ryzen 5 2600
Threads: 12
Memory: Dual Channel 8GB DDR3 (4GB x 2, 2667 MHz)
Operating System: Ubuntu 18.04

## 4.2 Parameter Tuning

Since the acceleration structures have their own parameters that affect the results, tuning them is essential to obtain the best performance.

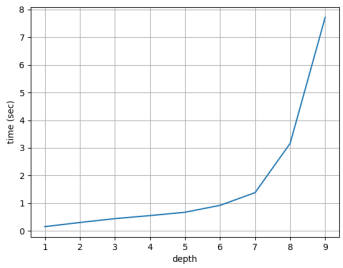
### 4.2.1 BVH Parameters

For the intersection and the traversal costs, the ratio between the two is what matters. Traversing involves performing ray-box intersections while intersection involves performing ray-triangle intersections. They are both close in cost. After testing different ratios, the

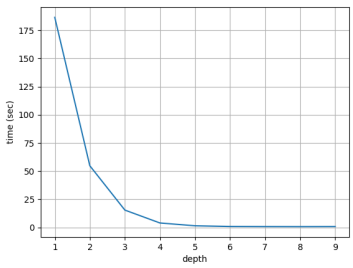
parameters picked were 1/4 for traversal and 1 for intersection, meaning that it is predicted that intersections cost 4 times as much as traversing.

### 4.2.2 Octree Parameters

For the maximum depth of the Octree, increasing it resulted in slower construction times but faster rendering times up to a diminishing return for some of the scenes. This is because at some point traversing the extra depth required more time than testing for the intersections with the overlapping primitives, Figure 16 shows the results of rendering the Stanford Bunny as an example. Due to the nature of Octrees, using a maximum depth larger than 9 was infeasible on the utilized hardware as the construction ended up taking quite a long time. Hence, only depths from 1 to 9 were tested for each scene and the best results were selected. The selected maximum depth was 5 for the Utah Teapot, 7 for the Sponza scene, and 8 for the rest of the scenes.




(a) Construction Times




(b) Rendering Times

Figure 16: Bunny Depth effects

For the minimum number of primitives, increasing it resulted in faster construction times but slower rendering times as is expected. This occurred because leaves ended up containing more primitives. After testing, all scenes performed best on a minimum number of primitives of 1.

### 4.2.3 KD-tree Parameters

Similar to the Octree, a larger maximum depth results in slower construction times but faster rendering times. Although since a KD-tree is a binary tree, it does not have the same issue as an Octree and a larger number of depths can be tested. Instead of testing many possible values, this KD-tree used a dynamic maximum depth calculated using $8 + 1.3 * Log2(numPrimitives)$. In general, this resulted in excellent performance without the hassle of testing a large number of possibilities.

For the intersection and traversal costs, again the ratio is what matters. For KD-trees, traversing nodes is much cheaper than performing intersection tests. After testing, 80 was picked for intersection cost and 1 for traversal cost, meaning that intersection cost is 80 times as expensive as traversing.

Moreover, A lower number of primitives results in faster rendering times but slower construction times and a minimum number of 1 was selected as all scenes performed best with it

### 4.3 Results

The results are divided into two parts, the construction time compares the performance of the algorithms with regards to how long it took to build the trees. The rendering time deals with traversing the tree and finding out which primitives are visible. The metrics that will be compared are the number of triangles, construction time in seconds, and the rendering speed in million rays per second. Table 2 shows the overall results of the experiments While Figure 17 and figure 18 show a visualization of the performance comparison.

| Scene | Triangle Count | Build Time | | | Render Time | | | Million Rays/Sec | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | ***BVH*** | ***KD*** | ***Octree*** | ***BVH*** | ***KD*** | ***Octree*** | ***BVH*** | ***KD*** | ***Octree*** |
| Teapot | 6.3k | 0.04 | 0.32 | 0.06 | 0.41 | 0.67 | 0.74 | 5.01 | 3.12 | 2.80 |
| Bunny | 144k | 1.22 | 4.52 | 3.16 | 0.32 | 0.48 | 0.65 | 6.40 | 4.41 | 3.21 |
| Sponza | 262k | 2.39 | 12.16 | 3.95 | 4.75 | 2.03 | 13.80 | 0.44 | 1.02 | 0.15 |
| Room | 533k | 4.97 | 21.25 | 8.20 | 3.27 | 2.21 | 7.77 | 0.63 | 0.94 | 0.27 |
| Dragon | 871k | 8.14 | 27.09 | 11.25 | 0.45 | 0.55 | 1.18 | 4.61 | 3.77 | 1.76 |
| Gallery | 999k | 9.80 | 45.47 | 12.94 | 5.51 | 3.44 | 5.44 | 0.38 | 0.60 | 0.38 |
| Buddha | 1087k | 10.55 | 34.64 | 13.35 | 0.31 | 0.41 | 0.69 | 6.78 | 5.01 | 3.01 |

Table 2: Overall Results

### 4.3.1 Construction Time

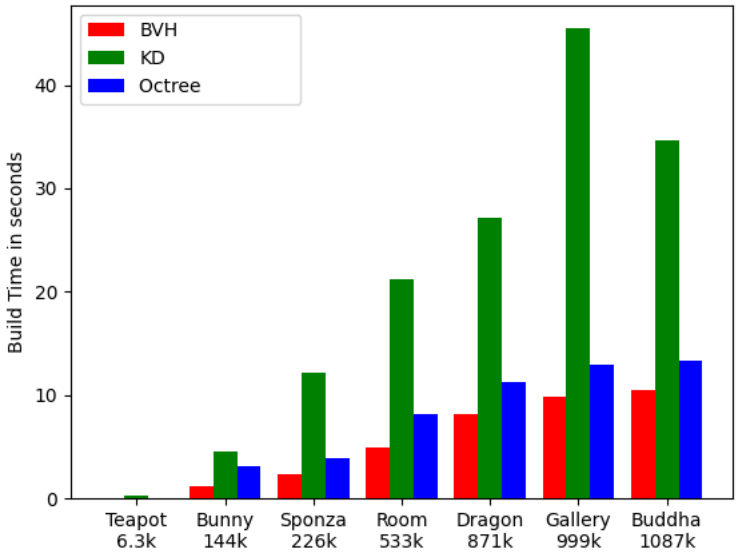



Figure 17: Tree Construction Time

Looking at Figure 17, The teapot's results are not significant due to the low number of triangles that compose the scene. Next, it is immediately clear that the KD-tree took quite substantially longer to construct than constructing the Bounding Volume hierarchy and the Octree, and in general, Bounding Volume hierarchies were the fastest to build, followed by Octrees. Averaging the performance of the 7 scenes, a BVH was 4 times as fast to build when compared to a KD-tree, while it was 1.5 times as fast when compared to an Octree.

KD-trees are much slower to build than a BVH because when choosing where to split a KD-tree node, a much bigger number of split positions are considered and that other axes are also considered if the axis picked first does not result in a good split. Compare it to a BVH where only a pre-determined number of buckets are created and the edges of the buckets are tested.

The reason for Octrees being slower than BVH is because the depth of the Octrees was between 7-8 and the fact that each non-leaf node has 8 children. In the worst case, this can result in $2^{d+1} - 1$ nodes, where d is depth. It gets expensive to split & test for overlapping triangles in each one of those nodes. Testing lower depths resulted in much faster Octree construction times.

### 4.3.2 Rendering Speed

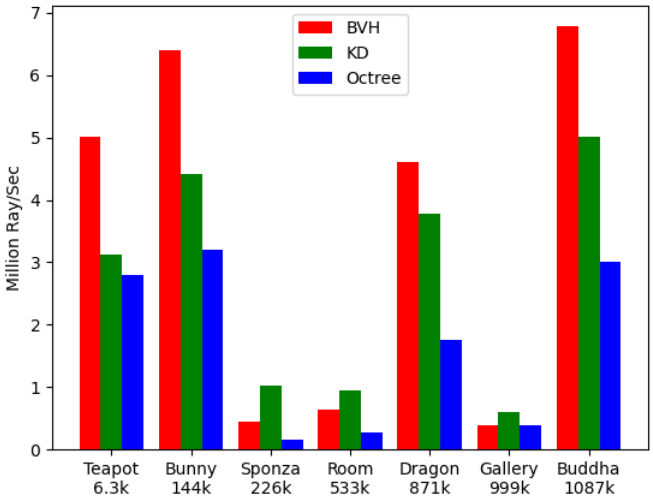



Figure 18: Rendering Speed

Looking at the simple scenes, Utah Teapot, Stanford Bunny, Stanford Dragon, and Buddha in Figure 18, It is noticeable that the BVH has the best performance on all 4. This is followed by the KD-tree and lastly by the Octree. This was expected because an Octree's splits are always constant when compared to the dynamic splits performed by the BVH and KD-tree. On the other hand when looking at the 3 complex scenes, Sponza, Room, and Gallery, the KD-tree performed better than the other two structures. The KD-tree was followed by the BVH and then the Octree.

## 5 Conclusion

In conclusion, the different structures have their pros and cons depending on the nature of the scene that someone is working with. First, If the scene is simple, consists of low to high triangle count, and is expected to be built repeatedly, a Bounding Volume Hierarchy would be the best option. Second, If the scene is more complex, consists of low to high triangle count, and is only required to be built once, then a KD-tree is the obvious choice. Lastly, the Octree in general had the worst rendering performance and the second-best construction performance. Due to those two results, The other two structures could be chosen over an Octree in most cases. Although one of the benefits of the Octree is the simplicity of its construction.

Some of the shortcomings of the paper involve the limited number of traversal algorithms and cost functions utilized as each structure only utilized one traversing algorithm and the SAH function. Traversal algorithms play a major part in the rendering performance and testing how they perform is desirable. Additionally, The heuristics used, such as the intersection cost & the traversal cost, could be adjusted to result in a better-built tree

for the KD-tree & BVHs. Furthermore, the comparison could be extended to include shading intersection tests. Future Work could involve working on the shortcomings with the addition of using non-static scenes (i.e, animated). Moreover, performing a comparison of a GPU-based Ray Tracer would be desirable as all of the graphical work nowadays is performed by a GPU.

## References


[1] Georg Rainer Hofmann. "Who invented ray tracing?" In: *The Visual Computer* 6.3 (1990), pp. 120–124. DOI: 10.1007/bf01911003.

[2] Brian Caulfield. *What's the Difference Between Ray Tracing, Rasterization?: NVIDIA Blog*. Mar. 2018. URL: https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/.

[3] Hector Villa-Martinez. "Accelerating Algorithms for Ray Tracing". In: (Dec. 2006).

[4] Akira Fujimoto and Kansei Iwata. "Accelerated Ray Tracing". In: *Computer Graphics* (1985), pp. 41–65. DOI: 10.1007/978-4-431-68030-7_4.

[5] *Triangle Mesh Processing*. URL: http://www.lix.polytechnique.fr/~maks/Verona_MPAM/TD/TD2/.

[6] *Polygon mesh*. URL: https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Dolphin_triangle_mesh.png.

[7] Scratchapixel. Oct. 2015. URL: https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume.

[8] Andrew S. Glassner. "Ray Tracing Acceleration Techniques". In: *An introduction to ray tracing*. Kaufmann, 2007.

[9] Jorge Revelles et al. "An Efficient Parametric Algorithm for Octree Traversal". In: (May 2000).

[10] Matt Pharr, Wenzel Jakob, and Greg Humphreys. "Primitives and Intersection Acceleration/Bounding Volume Hierarchies/BVH Construction". In: *Physically Based Rendering: From Theory to Implementation*. 3rd ed. Morgan Kaufmann Publishers/Elsevier, 2018, pp. 260–261.

[11] Matt Pharr, Wenzel Jakob, and Greg Humphreys. "Primitives and Intersection Acceleration/Bounding Volume Hierarchies/The Surface Area Heuristic". In: *Physically Based Rendering: From Theory to Implementation*. 3rd ed. Morgan Kaufmann Publishers/Elsevier, 2018, pp. 263–268.

[12] Matt Pharr, Wenzel Jakob, and Greg Humphreys. "Primitives and Intersection Acceleration/Bounding Volume Hierarchies/Traversal". In: *Physically Based Rendering: From Theory to Implementation*. 3rd ed. Morgan Kaufmann Publishers/Elsevier, 2018, pp. 282–284.

[13] Hanan Samet. "Implementing ray tracing with octrees and neighbor finding". In: *Computers & Graphics* 13.4 (1989), pp. 445–460. DOI: 10.1016/0097-8493(89)90006-x.

[14] *Octree*. Jan. 2020. URL: https://en.wikipedia.org/wiki/Octree.

[15] *Octree Implementation*. URL: https://www.flipcode.com/archives/Octree_Implementation.shtml.

[16] Matt Pharr, Wenzel Jakob, and Greg Humphreys. "Primitives and Intersection Acceleration/KD-tree Accelerator/Tree Construction". In: *Physically Based Rendering: From Theory to Implementation*. 3rd ed. Morgan Kaufmann Publishers/Elsevier, 2018, pp. 288–297.

[17] Matt Pharr, Wenzel Jakob, and Greg Humphreys. "Primitives and Intersection Acceleration/KD-tree Accelerator/Traversal". In: *Physically Based Rendering: From Theory to Implementation*. 3rd ed. Morgan Kaufmann Publishers/Elsevier, 2018, pp. 297–302.

[18] Scratchapixel. URL: https://www.scratchapixel.com/.

[19] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers/Elsevier, 2018.