In this document I will explain my implementation of BVH, what BVH is, why I use it. In this document I will use pseudo code and big-o notations to clear things up.

**What is BVH?**

BVH stands for Bounding Volume Hierarchy, it is a simple to understand algorithm to accelerate for example in my case a ray tracer.

**What is a ray tracer – short explanation:**

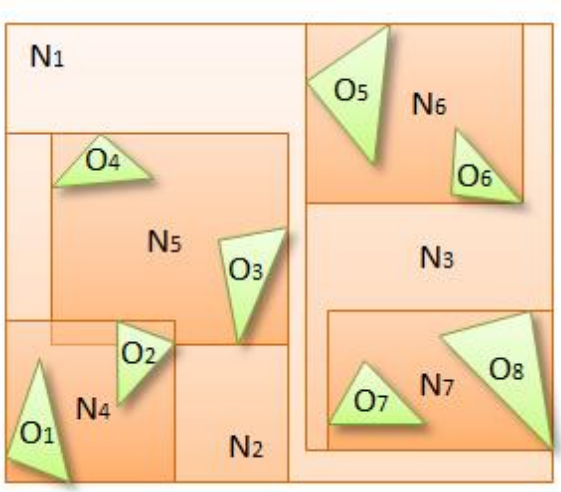
A ray tracer is an alternative and much more straight forward way of rasterization. Basically you have a 3D scene that holds a bunch of shapes that need to be displayed on the screen. So for every pixel on the screen we shoot a ray in to the 3D scene and we check if we hit a shape. If there is a hit then we retrieve the color data of that shape and display it.

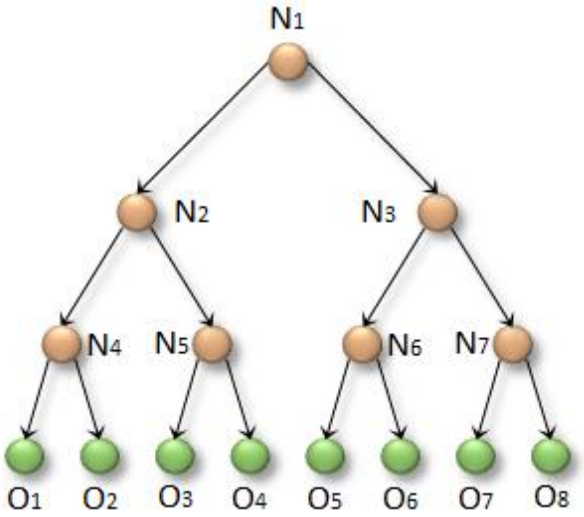
**How the BVH works:**

To check if a shape got hit by a ray can be a expensive call. Especially when you have many shapes in your scene that needs to be checked if they got hit by a ray. A BVH can help accelerate that process. A BVH exists out of a tree of nodes. Every node has a bounding box. I will refer to the first node of this tree as the root node, the root node has a bounding box that exactly fits all of the shapes. So the idea is too split all of the shapes into Nodes to cut down the amount of intersection tests, as seen in the image below.

```
// Basic setup of a Node
Class Node {
        shared_ptr<Node> leftNode
        shared_ptr<Node> rightNode
        bool leaf
        AABB bBox;
        unsigned int index
        unsigned int n_objs
}
```

So we shoot a Ray and we check if we hit the first Node and if that node is a leaf then we check if we hit a shape inside that Node. If the Node is a interior Node meaning that it has 2 child nodes then we check for an intersection with those and so on.



**In code:**

The BVHAccelerator class has 4 important functions as seen in the code below.

```
Class BVHAccelerator {
public:
bool Hit(Ray const& ray, HitInfo& hitInfo);
void Build(std::vector<Shape*> const& shapes, unsigned int const& maxDepth, unsigned int const& treshold);
private:
bool HitRecursive(Ray const& ray, HitInfo& hitInfo, std::shared_ptr<const Node> parentNode);
void BuildRecursive(unsigned int const& leftIndex, unsigned int const& rightIndex, std::shared_ptr<Node> parentNode, unsigned int const&
depth, unsigned int const& maxDepth, unsigned int const& treshold);
std::vector<Shape*> m_shapes;
std::shared_ptr<Node> root;
}
```

To build the BVHAccelerator you call the function Build and give it a vector of shapes, maxDepth and a threshold. We copy the values from the vector of shapes into a other vector of shapes so that we can always access the vector and change the order of the elements inside the vector.

```
// Inside the Build function
m_shapes.reserve(shapes.size());
copy(shapes.begin(), shapes.end(), back_inserter(m_shapes));
root->CalculateBbox(0, m_shapes.size(), m_shapes);
BuildRecursive(0, m_shapes.size(), root, 0, maxDepth, treshold);
```

The Build function will call the function BuildRecursive and this will setup the entire tree of nodes. The first thing we do is check if the amount of shapes left is less then or equal to the threshold or if the maxDepth has been reached then we create a leaf out of that Node.

```
// Inside the BuildRecursive function
if (rightIndex - leftIndex <= treshold || depth >= maxDepth)
parentNode->MakeLeaf(leftIndex, rightIndex - leftIndex);
```

If that's not the case then we search for the splitIndex and if there is one then we make two new child nodes and call Build Recursive again for those two child nodes. The splitIndex is the index to the shape that divides the shapes in two groups. For example when the x axis is the greatest axis we then split the shapes either to the left group or the right group and if the y axis is the greatest axis we then split the shapes into the bottom group or the upper group. If there is no splitIndex then we make the current Node a leaf. If for example all the shapes are in the same group then there is no splitIndex. We also change the order of the shapes. If for example the x axis is the greatest axis then we order the shapes from left to right and if the y axis is the greatest we then order from bottom to top.

```
//Inside the BuildRecursive function
int splitIndex = FindSplitIndex (leftIndex, rightIndex, parentNode);
// If there is a splitIndex
BuildRecursive(leftIndex, splitIndex, leftChildNode, depth+1, maxDepth, treshold);
BuildRecursive(splitIndex, rightIndex, rightChildNode, depth+1, maxDepth, treshold);
// Otherwise make a leaf of the current Node
```

If we want to check if a Ray hits a shape inside of the BVHAccelerator we call the Hit function and the Hit function will call the HitRecursive function.

```
// Inside the Hit function
if (root->GetBBox().Hit(ray)) {
return HitRecursive(ray, hitInfo, root);
}
return false;
```

Inside of the HitRecursive function we first check if the currentNode is a leaf and if we so then we look for the closest intersection if there is one.

```
//Inside the HitRecursive function
HitInfo hitInfoTmp;
if (parentNode->IsLeaf()) {
for (unsigned int i = parentNode->GetIndex(); i < parentNode->GetIndex() + parentNode->GetNObjs(); i++) {
if (m_shapes[i]->Hit(ray, hitInfoTmp) && hitInfoTmp.m_distance < hitInfo.m_distance) {
hitInfo = hitInfoTmp;
}}
if (hitInfo.m_shape != NULL) {
return true;
}}
```

If the node is not a leaf then we call HitRecursive again for the childNodes and it is possible that in both of the child nodes we hit a shape but we only take the shape that is closest to the ray.

```
if (leftNode->GetBBox().Hit(ray, distance)) {
HitRecursive(ray, hitInfoTmp, leftNode);
}
if (rightNode->GetBBox().Hit(ray, distance2) && HitRecursive(ray, hitInfoTmp2, rightNode) && hitInfoTmp2.m_distance <
hitInfoTmp.m_distance) {
hitInfo = hitInfoTmp2;
return true;
}
if (hitInfoTmp.m_shape != NULL) {
hitInfo = hitInfoTmp;
return true;
}
```

**Why BVH is faster than checking for a hit with every possible shape:**

To show why BVH is faster when it comes to finding a hit with closest possible shape compared to checking every possible shape for the closest hit I am going to use the big-o notation. Big-o notation is used in Computer Science to describe the performance or complexity of an algorithm.

When you check a ray with all of the shapes the big-o will look like this:

N

'N' Means the numbers of elements

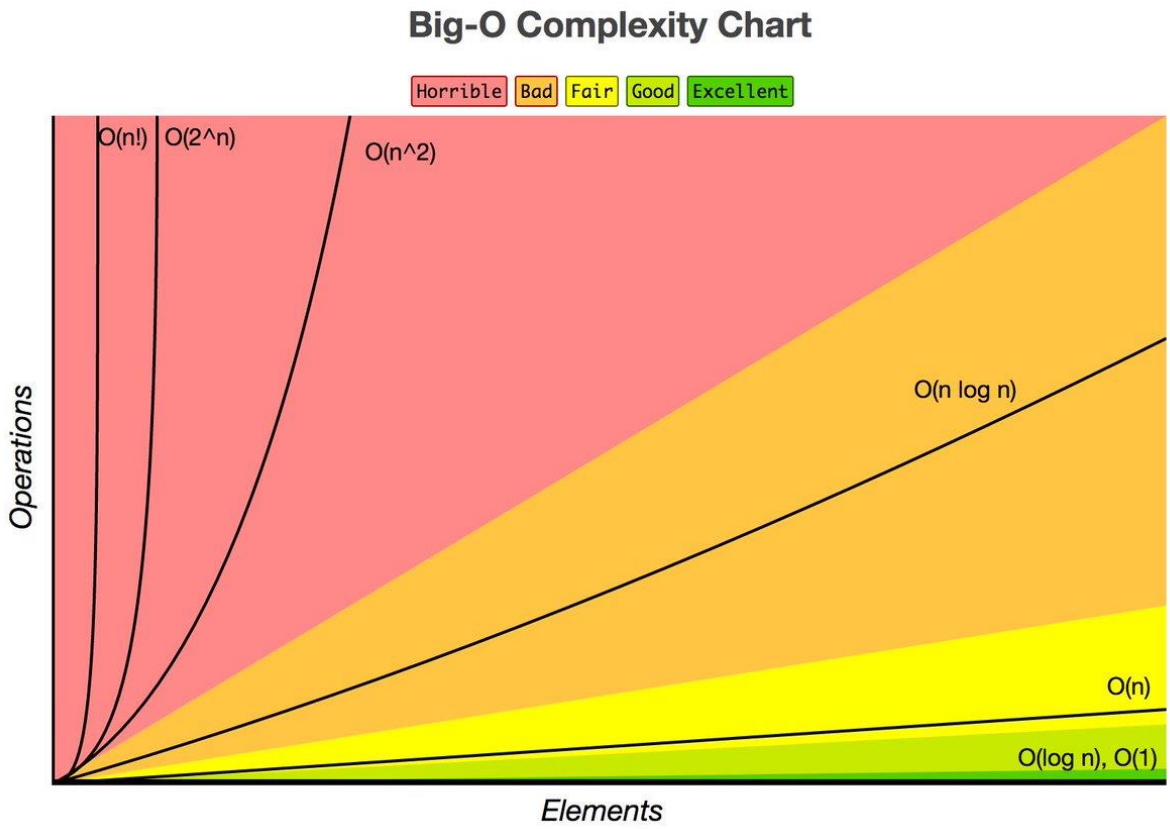
And with BVH the big-o would look like this on average:

Log(N)

This is because we split the shapes into a tree of nodes dividing the amount of checks.

But in the worst case scenario the BVH will be:

N




As you can see in this graph there is a huge difference between N and Log(N)

**Why I chose for BVH and not k-d tree, octree or something else**

Because BVH is the one that was for me the easiest to understand and implement, also I had a very good source explaining everything step by step on how to make one. Also from what I understand from octrees is that there is a chance that we check for a hit on the same shape multiple times which can be very expensive. This is because octrees/k-d trees and other space subdivision, divide the space recursively meaning that a shape can be in multiple different nodes.

**My resources:**

**1.** http://fileadmin.cs.lth.se/cs/education/edan30/lectures/s2-bvh.pdf

**2.** http://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part1