Optimizing Raytracing Intersection Using Bounding Volume Hierarchy Tree

Fachriza Ahmad Setiyono - 13523162
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
rizacal.mamen@gmail.com, 13523162@std.stei.itb.ac.id

***Abstract*—Ray tracing is a fundamental technique in computer graphics for rendering realistic images by simulating the behavior of light rays. However, the computational cost of determining ray-object intersections poses a significant challenge, particularly for complex scenes with numerous objects. This paper explores the optimization of ray tracing intersections through the implementation of a Bounding Volume Hierarchy (BVH) tree. BVH is a spatial data structure that accelerates ray traversal by organizing objects into hierarchical bounding volumes. We discuss the construction algorithms for BVH trees and examine traversal strategies to minimize intersection tests. Furthermore, performance benchmarks demonstrate the substantial improvements achieved in rendering speed and memory usage compared to brute-force intersection techniques. This work highlights the BVH tree's role in optimizing ray tracing, making it a practical and scalable solution for rendering complex scenes in real time.**

***Keywords*—Raytracing, Acceleration structures, Bounding Volume Hierarchy, Tree**

I. Introduction

Ray tracing has emerged as a powerful technique in computer graphics, enabling the rendering of highly realistic images by simulating the physical behavior of light. It works by tracing the paths of rays from the camera into a 3D scene and calculating the interactions with objects to determine color, reflection, refraction, and shadows. This process closely mirrors real-world light behavior, making ray tracing the foundation for visual effects in movies, architectural visualization, and modern gaming engines.

Due to the nature of ray tracing, any rendering effects that use ray tracing is noticeably slower. Computing millions of rays per frame is not an easy task even for modern GPUs. For complex scenes containing thousands or even millions of objects, the number of ray-object intersection tests required can grow exponentially, leading to performance bottlenecks. In naïve implementations, every ray must be tested against all objects in the scene, making real-time rendering impractical.

In recent years, hardware-accelerated ray tracing is getting more attention to the public. Though this does not mean that we can rely solely on the hardware power to do ray tracing, it is still a requirement to implement other software level optimizations. Most common type of ray tracing optimizations are spatial partitioning schemed. One of which is the Bounding Volume Hierarchy (BVH) tree.

Fig. 1. Example of raytracing used in the video game "Minecraft" to render realistic graphics.

(Source: https://www.ign.com/articles/what-is-ray-tracing)

This paper focuses on optimizing ray tracing performance using a Bounding Volume Hierarchy (BVH) tree, a widely used spatial data structure. BVH organizes objects into a hierarchy of bounding volumes, allowing rays to skip large sections of the scene that are guaranteed not to intersect, thereby minimizing computational overhead. By leveraging hierarchical pruning, BVH significantly improves efficiency without sacrificing visual accuracy.

The primary objective of this work is to investigate the construction and traversal techniques of BVH trees, analyze their impact on rendering performance, and compare their efficiency against brute-force approaches.

We will focus on the BVH tree construction and the traversal method of BVH structure. Through this investigation, we aim to demonstrate that BVH offers a scalable and practical solution for improving ray tracing efficiency, enabling its application in real-time rendering systems.

II. Theoretical Basis

A. Ray Tracing

Ray tracing is a computer graphics method to render or visualize objects from the scene to the screen. As opposed to the standard rasterization method, ray tracing works by sending rays per pixel and trace it to the scene until it hit an object in the scene.

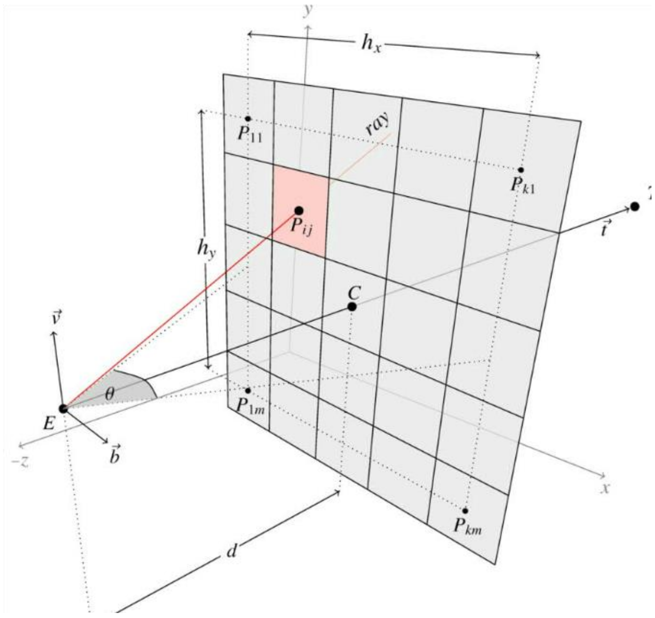


Fig. 2. Illustration of how rays are sent to the scene.

(Source: https://en.wikipedia.org/wiki/Ray_tracing_(graphics))

Ray tracing is the basis of many other advanced techniques such as ray casting, photon mapping, and path tracing. These techniques are often used in the context of photorealistic or physically based rendering. However, this does not mean that ray-tracing-based techniques are only used for such matters. It is certainly possible to use ray tracing for non-physical based rendering effects.

Ray tracing can simulate a variety of optical effects, such as reflection, refraction, soft shadows, scattering, depth of field, motion blur, caustics, ambient occlusion and dispersion phenomena (such as chromatic aberration). It can also be used to trace the path of sound waves in a similar fashion to light waves, making it a viable option for more immersive sound design in video games by rendering realistic reverberation and echoes. In fact, any physical wave or particle phenomenon with approximately linear motion can be simulated with ray tracing.

Ray tracing-based rendering techniques that involve sampling light over a domain generate image noise artifacts that can be addressed by tracing a very large number of rays, using denoising techniques, or spread the ray over time.

Formally, we can model the ray as:

$$P(t) = \vec{O} + t\hat{D} \qquad (1)$$

Where:

$\vec{O}$:	Ray origin point
$\hat{D}$:	Ray normalized direction from A to B
t:	Ray Euclidean distance

This equation is commonly referred to as the parametric ray equation (or parametric line equation). We must solve for the Euclidean distance *t* to “trace” or move the ray starting from the viewport (the observer) to the object in the scene. The method to solve this is to use ray-object intersection test for each object in the scene. Note that a ray can intersect with more than one object if the objects are stacked relative to the observer. To work around that, we need to store the distance of each intersection and keep the minimum distance. The intersection with the minimum distance is the closest one to the observer, which makes sense because objects with larger distance would be blocked by the closest object from the observer.

B. Acceleration Structure

Acceleration structures as the name implies are spatial data structures that speed up ray traversal, specifically, the part of ray traversal where we need to find the object to intersect.

There are basically two distinct “class” of spatial partitioning acceleration structure schemes. One is space subdivision methods such as BSP (Binary Space Partitioning), k-dimensional tree (kD-Tree), and octree. The other one is considered as object subdivision methods, such as Bounding Volume Hierarchy (BVH) tree.

Each method has its own advantages and disadvantages. In the case of space subdivision methods, since the subspaces do not overlap, it is usually possible to traverse the structure in front-to-back, or back-to-front, order more easily. When a ray is traversing such structure, as soon as it hits a surface, we can stop the traversal. This usually leads to faster traversal schemes. Several software renderers take advantage of the traversal efficiency given by the space subdivision schemes. On the other hand, space subdivision schemes may be more intricate to implement and may lead to deeper trees. Also, they do not like very much dynamic geometry [3]. If the geometry encoded into a space subdivision acceleration structure change, we usually must rebuild the acceleration structure from scratch.

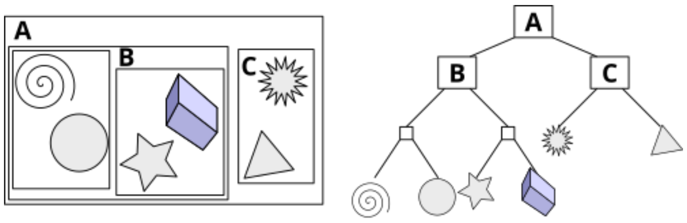


Fig. 3. Illustration of BVH acceleration structure.

(Source: https://en.wikipedia.org/wiki/Bounding_volume_hierarchy)

Object subdivision methods have a different characteristic. Since the object is subdivided with volumes, and these volumes may overlap, the traversal is traditionally slower. We cannot, for instance, stop traversing a BVH as soon as a ray finds an intersection with a surface. Since the volumes overlap, we may need to check for potential intersections with nearby primitives before quitting the traversal. On the other hand, it may be easier to implement a BVH because we do not have to split the object parts with planes. Also, BVHs usually generate shallower structures (which may eventually compensate for the slower traversal). However, one of the most interesting aspects of the BVHs is that they are dynamic geometry friendly. If geometry changes (but not much, actually), we can simply locally adjust the size and position of the corresponding bounding volume (by refitting). These adjustments may cause the need to adjust the parent volumes, a procedure that may culminate in a chain reaction that may reach the root node of the BVH.

When choosing between the two different methods, such as octree and BVH, pick the one that suits the most. Octree is superior when the scene is dense and can be represented as a uniform voxel[1] grid. Because the space are divided into regular grids, some traversal methods like DDA-tracing can be used to improve ray tracing or marching performance. However, if the grid is sparse, BVH might be faster, though another method like SVO (Sparse Voxel Octree) exists. BVH on the other hand is superior if the scene consists of various sizes of geometry. Typically, it's the small geometries that are hard to represent with voxels.

C. Graph

A graph is a structure in discrete mathematics, particularly in graph theory, which consists of set of objects that are related to each other in a sense. These objects are called nodes or vertices or points, whereas the line that represents the relation between the nodes are called edges. Also, another part of graph that is not commonly used are faces, which represents the areas that are enclosed between the edges. Note that the area "outside" the graph is also considered as a face. In mathematical notation, we can say that graph G is defined as $G = (V, E)$, where V are the vertex set, and E is the edge set. Generally, the vertex count should be above 0. This means that a collection of vertices without edges is also considered as a graph (usually called as null graph or empty graph).

There are various types of graphs in graph theory, but some of the common ones are:

1) Simple Graph

A regular graph or a simple graph is a graph that does not consist of ring nor doubly linked vertices.

2) Multi Graph

A multi graph is the opposite of simple graph. This graph has a doubly linked vertex in it.

3) Directed Graph (*Digraph*)

A directed graph is a graph with direction set for its edges. This means that a connection from vertices V_1 to V_2 does not mean the same vice versa. We can say that other graphs that does not have orientation on their edges are called undirected graphs.

4) Null Graph or Empty Graph

As stated before, a graph with only vertices and no edges are called a null graph or an empty graph.

Graph can also contain a path and/or a circuit [1]. A path is a sequence of edges which joins vertices. A directed graph contains a special type of path called directed path (or *dipath*). A circuit is type of path where the start and end of the path is the same.

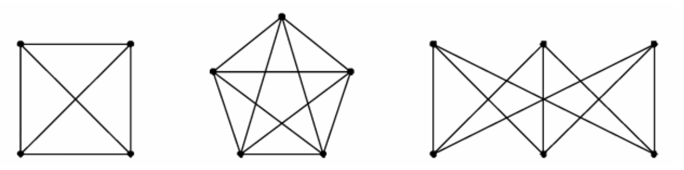


Fig. 4. Example of various graph structures.

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf)

D. Tree

Tree is an undirected graph that is connected and does not have a circuit [2]. Formally, let graph $G = (V, E)$ be an undirected simple graph with n vertices, then all these properties are equivalent:

1. G is a tree,
2. Every pair of vertices in G is connected with only a single path,
3. G contains $m = n - 1$ total number of edges,
4. G does not have a circuit,
5. G is a connected graph,
6. G does not have a circuit, and an additional edge will only result in one circuit,
7. G is a connected graph, and all the edges are "bridges".

In computer graphics (and informatics in general), a tree is usually a rooted tree. A rooted tree is a tree with one of the vertices considered as the root and its edges are given an orientation or direction.

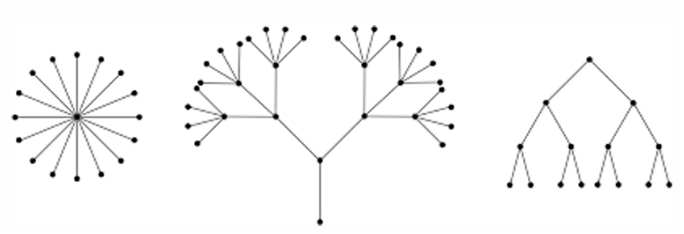


Fig. 5. Example of various tree structures.

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf)

A rooted tree has its own terminologies that are commonly referred to, such as:

1) Children and parents

Given vertex V_1, V_2, and V_3 with edges coming from V_1 to V_2 and V_3, then V_1 is the parent of V_2 and V_3, while V_2 and V_3 are the children of V_1.

2) Siblings

Given children V_2 and V_3 from parent V_1, then we can say V_2, and V_3 are siblings. Children from different parents are not siblings.

3) Subtree

[1] Volume pixel, a three-dimensional version of pixel.

A subtree is a tree within a tree. The largest subtree would be the trees where the root is the direct child of the original tree.

4) Degree

The degree of a vertex is the number of subgraphs of that vertex. The degree of a tree is usually the degree of the root.

5) Leaf

A leaf is a vertex of a tree with no children. The degree of a leaf is zero because it does not contain a subtree.

6) Internal node

An internal node is a vertex of a tree with children except for the root, hence the name *internal* node.

7) Level

The level or depth of a vertex is the length of the path taken from the root to said vertex. This means that the root has depth of zero.

Given tree T with the degree of the root n, we can say that T is an n-ary tree. In computer graphics, binary ($n = 2$) tree is one the most used tree. With each vertices containing only a maximum of two children, we can say that a vertex in binary tree only has left and right children. Because the order of the children matters, a binary tree is considered an ordered tree.

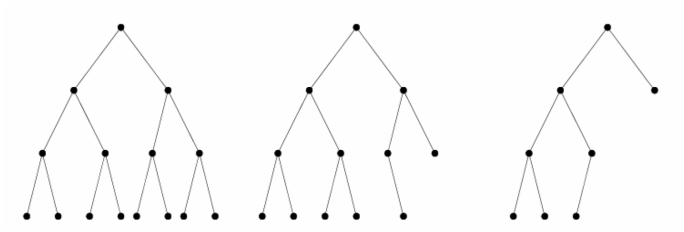


Fig. 7. Example of binary tree.

(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/24-Pohon-Bag2-2024.pdf)

E. Bounding Volume Hierarchy Tree

BVH is binary tree structure on a set of geometric objects. These geometric objects are in the form of leaf nodes of the tree, which are wrapped in bounding volumes. In computer graphics, a bounding volume for a set of objects is a closed region that completely encloses the union of the objects in the set. These wrapped leaf nodes are also wrapped in another, larger, bounding volumes. So, the resulting BVH tree is a single bounding volume that contains another bounding volume recursively.

The way BVH groups object in the scene is by splitting the object by the BVH axis relative to the world. This grouping is done recursively until ideally the smallest bounding box is only covering a single object. To construct the BVH, there are three primary methods:

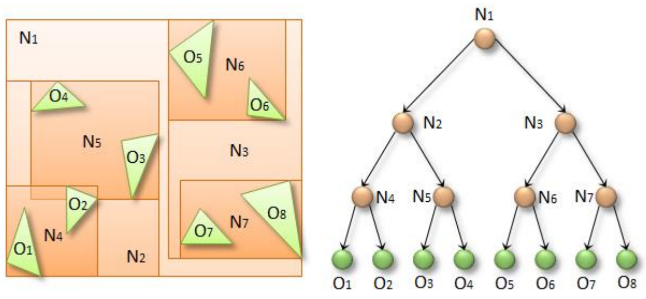


Fig. 6. How BVH groups objects.

(Source: https://github.com/boonemiller/Ray-Tracer)

1) Top-down method

In top-down method, the input object set is divided into two or more subsets which then bound in the selected bounding volume and continue dividing (and bounding) recursively until each subset contains only one primitive (leaf nodes are reached). Top-down approaches are by far the most common, simple to use, and quick to build, but they don't always provide the greatest trees.

2) Top-down method

Starting with the input set as the tree's leaves, bottom-up methods group two or more of them to create a new (internal) node. This process is repeated until all the input is gathered under a single node, that is, the tree's root. Although it is more challenging to execute, bottom-up approaches are probably going to result in better trees overall. According to several recent studies [3], sorting items using a space-filling curve and applying approximate clustering based on this sequential order can significantly increase construction speed in low-dimensional space, matching or surpassing top-down approaches.

3) Insertion method

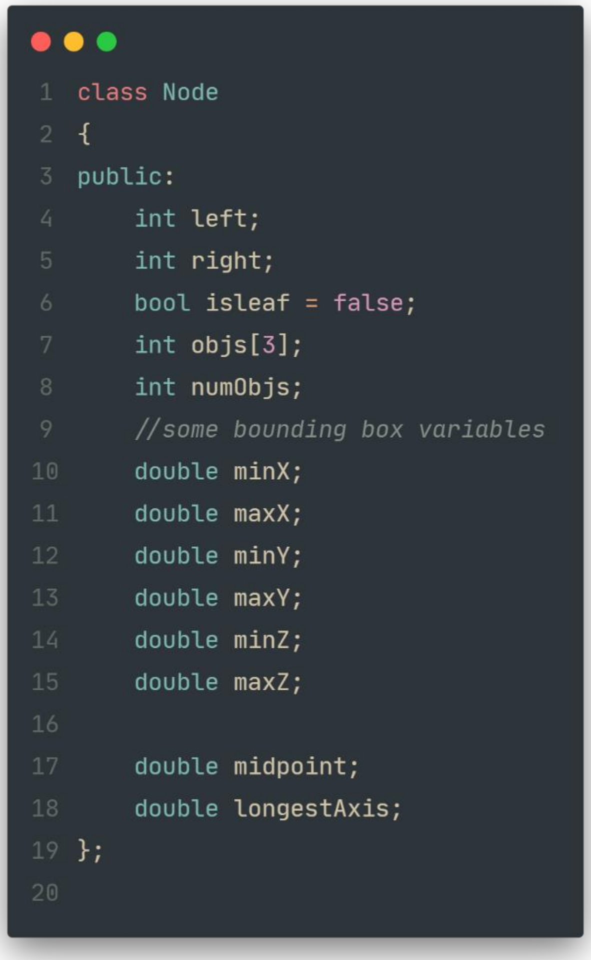
Insertion methods build the tree by inserting one object at a time, starting from an empty tree. The insertion location should be chosen that causes the tree to grow as little as possible. Insertion methods are considered on-line methods since they do not require all primitives to be available before construction starts and thus allow updates to be performed at runtime.

III. Implementation

For the sake of simplicity, we will use and analyse an already made ray tracer with BVH built in it. This ray tracer is made by boonemiller and is publicly available on their GitHub. This ray tracer is made in C++ language.

A. The BVH Structure

The structure of the BVH tree's vertex is implemented in the file `bvh.hpp`



```cpp
class Node
{
public:
    int left;
    int right;
    bool isleaf = false;
    int objs[3];
    int numObjs;
    //some bounding box variables
    double minX;
    double maxX;
    double minY;
    double maxY;
    double minZ;
    double maxZ;

    double midpoint;
    double longestAxis;
};

```

Fig. 8. BVH vertex structure.

(Source: https://github.com/boonemiller/Ray-Tracer)

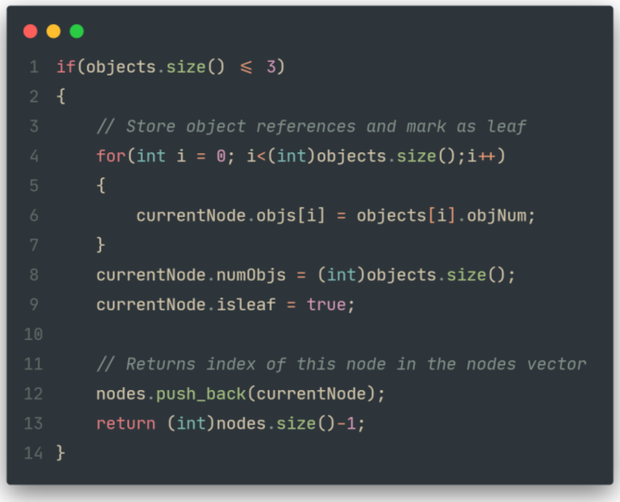
Since BVH is a binary tree, each vertex hold references to the left and right children. To hold the bounding volume data for the inner vertices, it stores the minimum and maximum 3-dimension position of the bounding volume.

B. The BVH Construction

The BVH construction is inside the file `bvh.cpp` within the function `constructTree`. Let us break down the main algorithm inside this recursive function.

1) Base Case

The base case of this function is when the object that is enclosed by the bounding volume is less than 3. In another word, when a node only has less than or equal to 3 objects, it will activate the base case. The node will then become a leaf node that stores direct reference to the objects.



```cpp
if(objects.size() <= 3)
{
    // Store object references and mark as leaf
    for(int i = 0; i<(int)objects.size();i++)
    {
        currentNode.objs[i] = objects[i].objNum;
    }
    currentNode.numObjs = (int)objects.size();
    currentNode.isleaf = true;

    // Returns index of this node in the nodes vector
    nodes.push_back(currentNode);
    return (int)nodes.size()-1;
}
```

Fig. 9. BVH construction base case.

(Source: https://github.com/boonemiller/Ray-Tracer)

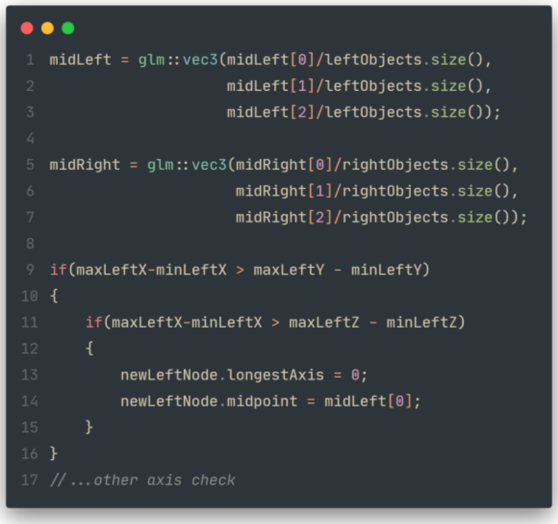
2) Axis Splitting

We know that BVH works by grouping set of objects into two subsets by splitting based on the axis. The code splits objects into two groups based on their position along the "longest axis" of the current node. Objects are also sorted into left/right groups based on whether they're below or above the midpoint. For each group/vertex, it calculates:

- The bounding box (min/max coordinates in x, y, z),
- The centroid (average position) of objects in that group,
- The new longest axis for that group's vertex.

3) Axis Selection

After creating new left and right children, we also want to set their longest axis and centroid. This code determines the longest axis by comparing the extent of the bounding box in each dimension. This helps create more efficient splits.



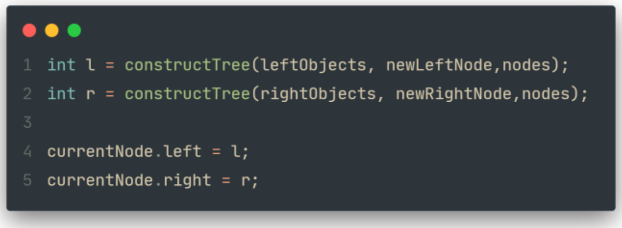
```cpp
midLeft = glm::vec3(midLeft[0]/leftObjects.size(),
                    midLeft[1]/leftObjects.size(),
                    midLeft[2]/leftObjects.size());

midRight = glm::vec3(midRight[0]/rightObjects.size(),
                     midRight[1]/rightObjects.size(),
                     midRight[2]/rightObjects.size());

if(maxLeftX-minLeftX > maxLeftY - minLeftY)
{
    if(maxLeftX-minLeftX > maxLeftZ - minLeftZ)
    {
        newLeftNode.longestAxis = 0;
        newLeftNode.midpoint = midLeft[0];
    }
}
// ...other axis check
```

Fig. 10. BVH axis splitting.

(Source: https://github.com/boonemiller/Ray-Tracer)

4) Recursion

After processing the current vertex, the function recursively builds subtrees for the left and right groups.



```cpp
int l = constructTree(leftObjects, newLeftNode,nodes);
int r = constructTree(rightObjects, newRightNode,nodes);

currentNode.left = l;
currentNode.right = r;
```

Fig. 11. Recursion in BVH construction.

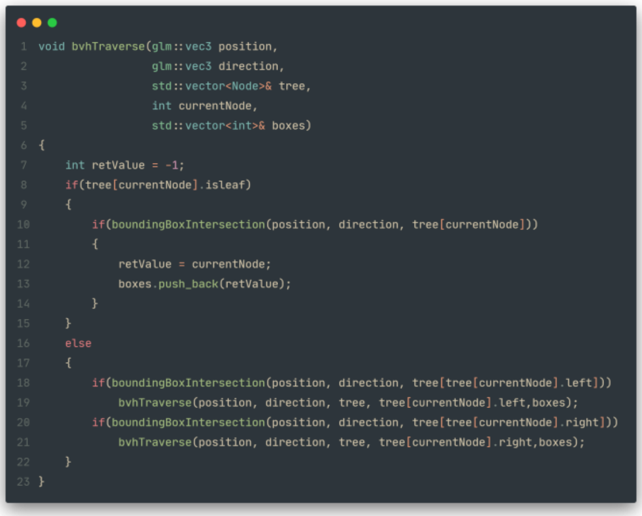
(Source: https://github.com/boonemiller/Ray-Tracer)

C. The BVH Traversal

To traverse the BVH structure, each ray must start from the root, and recursively find the subtree that collides with the object inside the bounding volume. This means that in each iteration, a ray-bounding-volume intersection test must be done.

Because the traversal in done in a recursive manner, then a base case must be set up. The base case for this traversal is when the current node is a leaf node, which means that there are no deeper or smaller bounding volumes. The ray will either hit or miss the scene based on this intersection test.

This also shows how BVH traversal can speed up a regular ray tracing. If the ray happens to intersect with a leaf node early, then an early return is done. Compared to regular ray tracing, this is surely a huge speed up than intersecting the whole object in the scene. The only way a BVH can be slower than regular ray traversal is if the BVH construction performance overhead is worse than ray tracing the whole scene.



```cpp
void bvhTraverse(glm::vec3 position,
                 glm::vec3 direction,
                 std::vector<Node>& tree,
                 int currentNode,
                 std::vector<int>& boxes)
{
    int retValue = -1;
    if(tree[currentNode].isleaf)
    {
        if(boundingBoxIntersection(position, direction, tree[currentNode]))
        {
            retValue = currentNode;
            boxes.push_back(retValue);
        }
    }
    else
    {
        if(boundingBoxIntersection(position, direction, tree[tree[currentNode].left]))
            bvhTraverse(position, direction, tree, tree[currentNode].left,boxes);
        if(boundingBoxIntersection(position, direction, tree[tree[currentNode].right]))
            bvhTraverse(position, direction, tree, tree[currentNode].right,boxes);
    }
}
```

Fig. 12. Function to traverse BVH structure.

(Source: https://github.com/boonemiller/Ray-Tracer)

IV. Result

Boonemiller states that BVH trees increases cache performance of primary rays. This happens as a side effect of casting rays in one area consecutively. For example, if a primary ray is cast in one pixel, and then cast a primary ray in the pixel next to it, it is likely that the two pixels will be accessing the same parts of the BVH tree and as a result the objects needed for the object intersection tests will likely already be in the cache. The graph shows the speed ups achieved by adding a BVH acceleration structure into the ray tracing code. This shows the percentage speed up of using a BVH acceleration structure over a naïve, no acceleration structure method of ray tracing, on a variety of scenes.

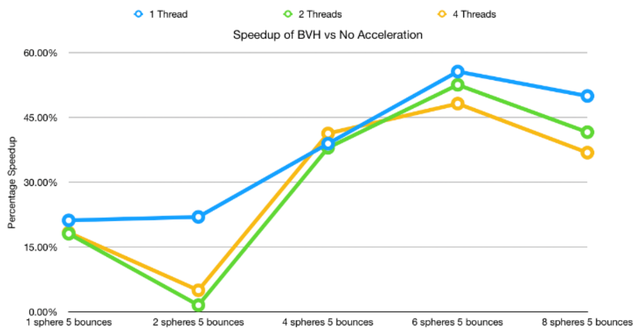


Fig. 13. Performance graph of BVH.

(Source: https://github.com/boonemiller/Ray-Tracer)

V. Conclusion

We have covered BVH construction technique. which balance the trade-off between preprocessing time and runtime efficiency. We also examined traversal algorithms that exploit BVH's hierarchical nature for early termination, further optimizing intersection tests. Experimental results demonstrated substantial performance improvements in terms of rendering speed, reduced intersection computations, and memory usage when compared to the regular naive methods.

Despite its strengths, BVH construction remains computationally expensive, particularly for dynamic or deformable scenes requiring frequent updates. Future work can address these limitations by incorporating parallel processing techniques, GPU acceleration, and hybrid approaches that combine BVH with other spatial partitioning methods. Additionally, adapting BVH to dynamic environments through incremental updates rather than full rebuilding offers an avenue for further optimization.

In conclusion, BVH proves to be a scalable and practical solution for accelerating ray tracing computations, making it highly suitable for modern graphics applications requiring real-time performance. This work not only highlights BVH's capabilities but also lays the groundwork for future innovations in rendering technologies.

VI. Appendix

The source code for the BVH and ray tracer for this paper is made by boonemiller, which is available at their GitHub here: https://github.com/boonemiller/Ray-Tracer

VII. Acknowledgment

The author would like to express his gratitude to God Almighty who has given his blessings, so that the author can complete the paper entitled "Optimizing Raytracing Intersection Using Bounding Volume Hierarchy Tree" which was completed on time. The author would also like to thank Arrival Dwi Sentosa, S.Kom., M.T. as the lecturer in charge of the IF2123 Discrete Mathematics Class 3 course for the guidance and teaching that has been carried out in this class. The author would also like to thank Dr. Ir. Rinaldi Munir, MT., as one of the lecturers in Discrete Mathematics for providing references and learning resources for Discrete Mathematics through his website. Finally, the author would like to thank his parents, family, and all parties who helped the author in completing this paper.

References

[1] Munir, Rinaldi. 2024. "Graf (bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf (accessed on 8 January 2025).
[2] Munir, Rinaldi. 2024. "Pohon (bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf (accessed on 8 January 2025).
[3] Ingo Wald, Solomon Boulos, and Peter Shirley. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. ACM Trans. Graph. 26, 1 (January 2007), 6–es. https://doi.org/10.1145/1189762.1206075 (accessed on 8 January 2025).

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Januari 2025

[illegible]

Fachriza Ahmad Setiyono 13523162