Acceleration data structures for Virtual Ray Tracer

Bachelor's Project, Computing Science
June 30, 2022
Student: Bora Yilmaz
Primary supervisor: prof. dr. Jiří Kosinka
Secondary supervisor: dr. Steffen D. Frey

Abstract

Ray tracing is at the core of simulating light effects in computer graphics. Due to ray tracing's geometrical and computational complexity, students can find it hard to understand. Interactive and visual demonstrations of ray tracing can aid students in understanding it.

This project extends upon Virtual Ray Tracer (VRT), developed to help with ray tracing education. Acceleration data structures play a crucial role in ray tracing by reducing the number of intersection tests between rays and objects. Two of these data structures will be added on top of VRT; axis-aligned bounding boxes and octrees. Through these, the ray tracer will be accelerated and tutorial levels teaching them will be developed. Based on the design and implementation, the thesis aims to answer the following research question: What is the best way to visualize ray tracing acceleration data structures and their use cases in VRT?

Contents

1 Introduction

A must for learners in the field of computer graphics, ray tracing is known for its mathematical complexity. What ray tracing achieves is visually accurate rendered images of a described scene through the technique of casting rays from the viewpoint (camera or eye) into the scene through pixels and then tracing their paths to decide what colour that ray should return to the pixels they were cast through. So when a ray is traced from the viewpoint through each pixel on a screen, the end result is the complete image with the correct colours.

Working through calculations is not enough to completely grasp the core idea behind ray tracing. There is a multitude of visual and mathematical concepts that must be understood for this. Algebra is used for solving intersection points of rays and objects. Geometry and linear algebra with vectors comes into play to deal with the physical properties of light through vectors. It is safe to assume that the difficulty only increases with the addition of concepts like shadows, reflection and refraction or acceleration data structures.

What is worth noticing is that all of these specific concepts around ray tracing are visual at their core. The end goal of ray tracing is of course, an accurate visual image. Based on this, a visual tool to learn and interact with ray tracing concepts sounds natural to both the learner and the teacher. Virtual Ray Tracer was developed by Chris van Wezel and Willard Verschoore exactly for this purpose: to aid students in their learning and experimentation of ray tracing concepts [9]. It was developed in Unity using the coding language C# and it is an interactive demonstration of ray tracing concepts with tutorials and information on them. In their evaluation of the developed application, the positive feedback from students showed that it indeed was beneficial for how ray tracing is learnt. The application is still under active development.

In this thesis we propose to extend upon Virtual Ray Tracer with acceleration data structures. Axis aligned bounding boxes and octrees are the two data structures that are considered for this extension. They reduce the number of intersection tests between rays and objects when ray tracing. Based on the evaluation of the end results, the thesis concludes the following question: What is the best way to visualize acceleration data structures and their use cases in Virtual Ray Tracer?

After going over background information and how acceleration data structures work in Section 2, the requirements for the implementation aimed at are argued for and set in place in Section 3. In Section 4, the realization of the requirements within the game engine Unity using C# is explained in detail. In Section 5, based on feedback from 3 peers, we evaluate how the extensions perform as a learning tool for the acceleration data structures and explore what is the best way to visualize them. The speed-up of rendering is calculated as percentages and the visual result of the implementation is discussed in Section 6. Finally, every section comes together in Section 7 to conclude the project. In Section 8, potential future work that could follow this project is mentioned.

2 Background Information

Before heading into further sections, it is essential to introduce the information that lays the foundation of what is going to be implemented. In this section, the concept of ray tracing is summarized first in Subsection 2.1. After this, VRT is explained in detail on how it came to be the application it is today in Subsection 2.2. Finally, the background information is concluded with Subsection 2.3 which explains axis-aligned bounding boxes and octrees.

2.1 Ray Tracing

Every day, the relevance of ray tracing increases within computer graphics since it is commonly used in animated films, video games and design visualizations. Imagine a scene where we have a decorated room, with multiple light sources. How do we make the objects within this scene come to life with computer graphics accurately with respect to the lighting? Ray tracing achieves realistic results by simulating the physical behaviour of light [2].

Within ray tracing, the most prevalent technique is ray casting. As seen in Figure 1 below, a ray is shot from the eye, in this case a camera, and traced through each pixel in our screen, and based on the rays interactions with objects in the scene, the final colour of each pixel is calculated [3]. It is also clear from the figure that many factors must be taken into consideration when ray tracing, ranging from position of the camera, pixels, direction of rays, colour of the light source and the object, physical light (ray) behaviour, position and direction of light sources and many similar factors. When all of these concepts combine, the technique of ray tracing becomes high in computational cost and harder to understand as a whole. That is what we are concerned with in this thesis, making ray tracing easier to learn and study.

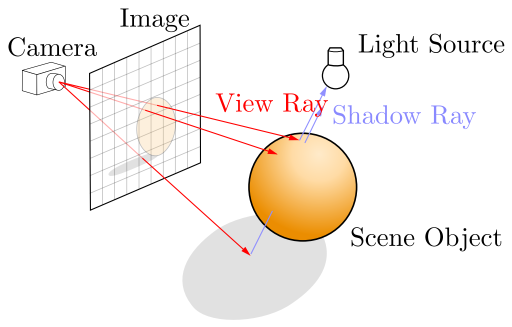
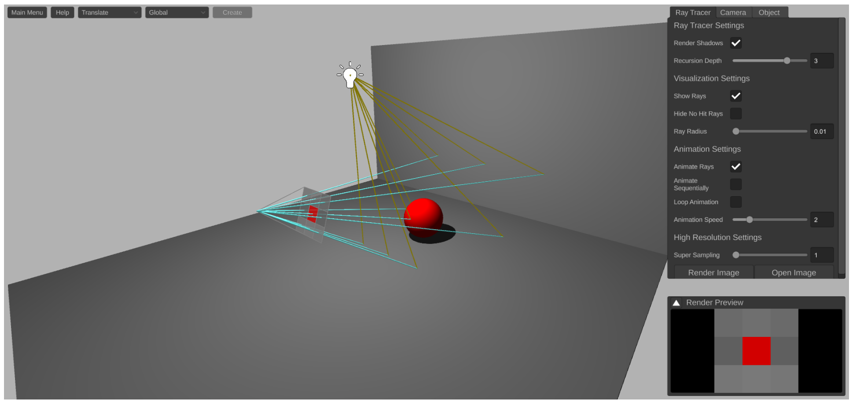


Figure 1: An illustration of ray casting.
Credit: Henrik, https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg

Even though it gets high in computational cost, the technique of ray tracing is not in vain and it can be sped up. The process can be accelerated in many ways. One way to accelerate ray tracing is using spatial information to ignore trivial cases and check for less ray-object intersections. In this project, the data structures that allow this acceleration technique is what we are focusing on in teaching to learners.

2.2 Virtual Ray Tracer

Seeing the visualization of a ray being traced and how ultimately a pixel screen is coloured would help a learner see the technique in action. Virtual Ray Tracer, referred to by its abbreviation VRT, aims to provide exactly this, an educative tool aimed at teaching ray tracing concepts to learners through visualization.



Ray Tracer Settings
Render Shadows ✓
Recursion Depth 3
Visualization Settings
Show Rays ✓
Hide No Hit Rays
Ray Radius 0.01
Animation Settings
Animate Rays ✓
Animate Sequentially
Loop Animation
Animation Speed 2
High Resolution Settings
Super Sampling 1

Figure 2: Screenshot of the Basic Ray Tracing Level in VRT.

VRT was developed by Chris van Wezel and Willard Verschoore at the University of Groningen with the purpose of aiding students of Computer Graphics through an interactive application covering the principles of ray tracing [9]. There are seperate levels about the concepts within ray tracing and each level in VRT focuses on explaining them seperately. For example, as seen in Figure 2, the level introduces the basics of ray tracing, while also allowing interactions to the user where they can transform all objects in the scene and use the toggles and sliders on the right side of the user interface. Ultimately, the user reaches the sandbox level in which they create their own scenes however they like with no constraints. We aim to extend upon VRT by adding an implementation of acceleration data structures and their respective tutorial levels to help the learner study them.

2.3 Acceleration Data Structures

The acceleration data structures in this project calculate spatial information of objects, which are then used to ignore rays or filter out unnecessary ray intersection calculations. Such information is used in combination with ray tracing techniques in order to speed up computations. First the axis-aligned bounding box is explained, followed by the octree.

2.3.1 Axis-Aligned Bounding Boxes

The axis-aligned bounding box (AABB) of a 2D object is the box which has two of it's opposite corners at the points Q and P where

$$Q = (x_{min}, y_{min})$$
$$P = (x_{max}, y_{max})$$

In Figure 3, we can see the points P and Q of an exemplary 2D object, a primitive triangle.

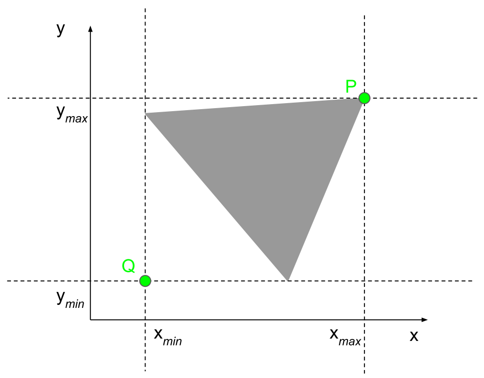


Figure 3: An illustration of points Q and P of a 2D object

When a box is formed with Q and P as opposite corners, the following AABB is formed as seen in Figure 4.

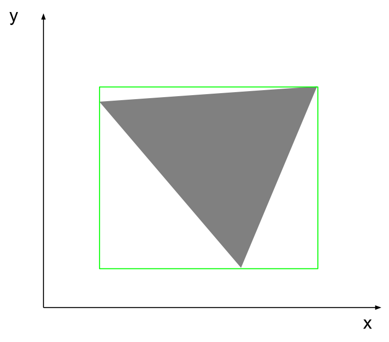


Figure 4: An illustration of AABB formed from points Q and P in Figure 3

AABB in 3D are formed under the same idea but with the extra dimension of the z coordinate. The Axis-Aligned Bounding Box (AABB) of a 3D object is the box which has two of it's opposite corners at the points Q and P where

$$Q = (x_{min}, y_{min}, z_{min})$$
$$P = (x_{max}, y_{max}, z_{max})$$

Such example of a 3D object and its AABB with points Q and P is seen in Figure 5.

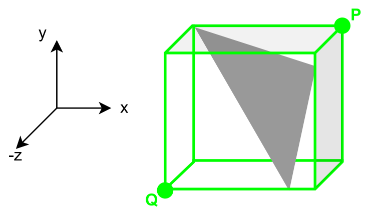


Figure 5: An illustration of AABB of a 3D triangle

AABBs spatially encapsulate the object within them. Computing whether a ray intersects a bounding box or not before checking for intersections with the object is key to accelerating intersection tests between rays and objects. If the ray does not intersect the AABB, then we can trivially ignore its intersection calculations for the object within the AABB. This is because for a ray that misses a bounding box, it can not intersect anything encapsulated by that same bounding box.

2.3.2 Octrees

An octree is a tree data structure for spatial partitioning of a 3D object. Each node in the tree can have 0 or 8 children. It is an axis-aligned bounding box of a 3D object where we keep splitting the nodes (boxes) into 8 inner children. Looking at Figure 6, the 3D visualization on the left shows the process of splitting an octree into deeper partitions. Note that not all boxes are split further because this splitting is decided over other factors, which is discussed below.

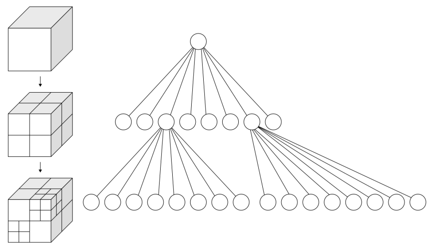


Figure 6: An octree data structure with it's 3D visualization on the left and tree visualization on the right. Credit: WhiteTimberwolf, https://commons.wikimedia.org/wiki/File:Octree2.svg

For any given node (including the root node), the decision to whether split into 8 inner nodes or not at any given point is dependent on two criteria:

- The octree's depth limit;
- Node (box) contains the object or not.

Splitting the octree into its children based on these two criteria will yield as a result a spatial partitioning of the object, with empty nodes and primitive containing nodes. Take for example Figure 7, the 3D bunny object's octree contains empty nodes to the sides and between its ears, while the other nodes contain some part of the 3D mesh and are not empty. This way, just like with AABB acceleration, we check for ray and octree nodes (boxes) intersections first. Here are the following cases that must be handled to accelerate ray intersection tests with a 3D object using octrees:

- Ray misses the octree (root node) completely. The ray is trivially ignored like in the AABB acceleration;
- Ray hits an octree node. The ray must be checked for intersection with the children of that node. If the node contains primitives (part of the 3D object), those must be checked for intersection. This accelerates the process because parts of the object that were not contained within intersected nodes are ignored. If the ray did not hit any node containing a part of the object, the ray is ignored, accelerating the process.

The octree has to be derived for a 3D object by calculation. For still scenes where objects do not transform, this is perfect for acceleration as the octree is calculated once at the start of ray tracing. For real time ray tracing in dynamic scenes, octrees have to be handled as well as objects in the scene transform.

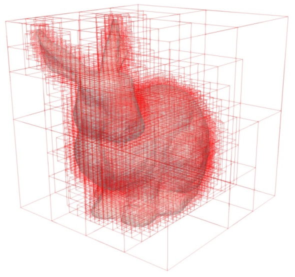


Figure 7: An octree of a 3D object. Credit: Martin Cavarga, https://mshgrid.com/2021/01/19/octree-for-voxel-outlines-of-a-triangular-mesh/

3 Requirements

There are key design requirements to assert before implementing anything in VRT to ensure a beneficial end result. Both the AABB and the octree data structures should:

- Accelerate the ray tracing rendering so that it is not only a visualization, but a functional component in VRT;
- Be interactive;
- Be drawn as a wire frame box (only their edges drawn) on top of the respective object. This is required because AABBs and octrees are bounding volumes over objects. Having the boxes drawn solid (filled in) would obstruct the visual on the object within them.

After implementing the data structures themselves, their respective tutorial levels are to be implemented. We aim to design 2 levels for each data structure.

Level 1 should:

- Have the learning goal of introducing the data structure by explaining what is it is and how it is calculated;
- Focus on only a single object and the data structure with no ray tracing involved;
- Visualize how the data structure is re-calculated as the object transforms.

Level 2 on the other hand should:

- Have the learning goal of teaching the acceleration process when ray tracing under the respective data structure;
- Focus on visualizing the acceleration process between a single object and a single ray;
- Display statistics of the acceleration process (e.g. number of rays accelerated);
- Display the acceleration status of the ray at any given moment (e.g. display through text that the ray has been ignored).

4 Implementation

In this section, how the designed features were implemented within Unity using C# is explained. For both data structures, the layout of the implementation is the same. First, the calculation subsections 4.2.1 and 4.3.1 explain how the data structures are calculated. Visualization subsections 4.2.2 and 4.3.2 explain how the data structures are drawn within the scene for the user to see. Utilization subsections 4.2.3 and 4.3.3 explain how the data structures are used to accelerate the ray tracer. Tutorial Level subsections 4.2.4 and 4.3.4 for level 1, subsections 4.2.5 and 4.3.5 for level 2 explain how the levels were built in order to convey the concepts to the user. Before all of these sections, Subsection 4.1 explains an essential package that is used when drawing the acceleration data structures within Unity builds.

4.1 Popcron Gizmos Package

In Unity, Gizmos are used for visual debugging [5]. One of the methods for debugging is `DrawWireCube` which draws a wire frame box given a Bounds object. The wire frame box is the requirement that was set previously in Section 3. The Bounds object is what is used in both the AABB and the Octree components, representing a bounding box in Unity. Thus, `DrawWireCube` is the perfect method to use for visualizing these data structures in Unity.

However, Gizmos are only rendered in the Scene view of Unity which is only available during development and not at runtime. So Gizmos are only usable as debuggers during development. To tackle this issue, the Popcron Gizmos package is used under the MIT License. This is a package which allows the runtime drawing of Gizmos, so that the build version of the application can display them [4].

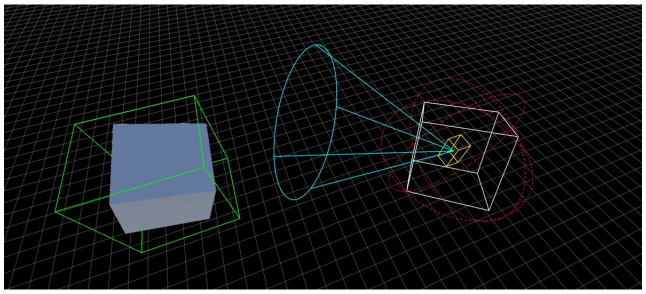


Figure 8: Popcron Gizmos package examples. Credit: popcron, https://github.com/popcron/gizmos/blob/master/README.md

The equivalent of `DrawWireCube` in Popcron Gizmos is called `Popcron.Gizmos.Bounds` and a visual example is the green wire frame box seen on the left in Figure 8. In all coding sections where a Bounds objects box has to be drawn, `Popcron.Gizmos.Bounds` is used.

4.2 Axis-Aligned Bounding Box

4.2.1 Calculation

In VRT, the ray traced objects within the scene are RTMesh objects. It is asserted that these objects have a MeshRenderer component. This components functionality is to render a mesh given through another component called MeshFilter. Of course, the RTMesh objects also have a MeshFilter.

From the MeshRenderer, its bounds property is accessible. This is a Bounds class object and it is defined as the "axis-aligned bounding box fully enclosing the object in world space" [6].

4.2.2 Visualization

From the calculation, we have the Bounds object which is directly the AABB of the object. `Popcron.Gizmos.Bounds` is used with the Bounds as a parameter and the color as green as follows

```
Popcron.Gizmos.Bounds(bounds, Color.green);
```

When included in the update behaviour of the object, this line of code draws the AABB over the object as intended, giving the result in Figure 9.

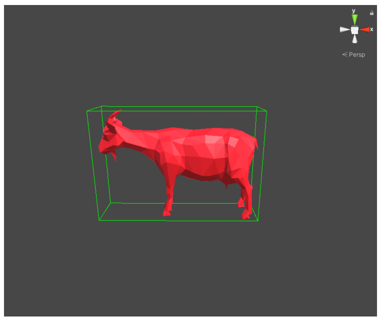


Figure 9: A 3D object and its AABB drawn in green lines.

4.2.3 Utilization

In order to utilize the AABB and accelerate ray tracing, we need to check whether a ray that is cast intersects the AABB or not before checking for intersection with the object itself and proceeding with the calculations. The ray intersection test with the AABB can be done as

```
if (!bounds.IntersectRay(raycast, out distance)) {
     savedAmount++;
     return BackgroundColor;
 }
```

This conditional statement is checked for each ray that is cast on the object with the AABB. The increment on the second line is used to keep track of rays that were ignored, so that it can be displayed to the user in Tutorial Level 2.

4.2.4 Tutorial Level 1

In this level, the AABB is introduced to the user. In order to ensure the user only focuses on the AABB and its object, ray casting is stopped. At the start of the level, the tutorial text goes over how the AABB is derived, how it is updated as an object moves and how it is used to accelerate ray object hit tests. The text gives a hint towards how the user should transform the object around and see how the AABB updates. The object starts off spinning, so the AABB is clearly seen being updated as the object spins. The user can toggle this spin motion on or off by the side panel. The user can also hide the AABB if desired. The end result of the level is seen in Figure 10.

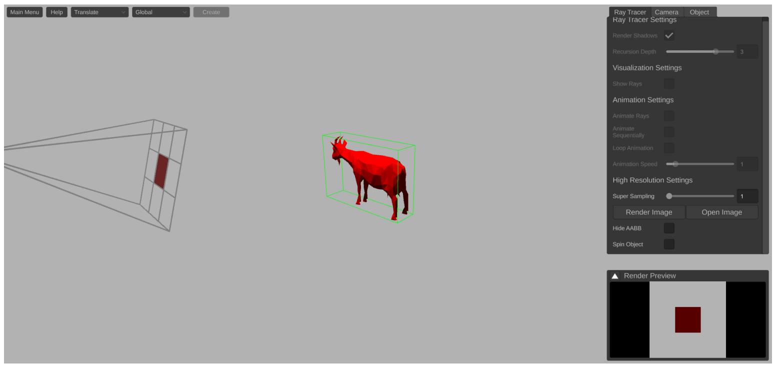


Figure 10: A screenshot of the AABB Tutorial Level 1. The AABB is visualized over the 3D object in the scene with a green color. The spin toggle button is seen on the right panel.

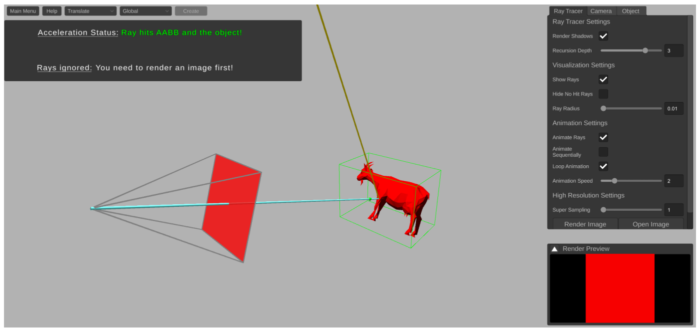
4.2.5 Tutorial Level 2

In this level, the user is already familiar with how the AABB is derived and what properties of it could be used. Now, ray casting is started. The tutorial text hints to the user on what will be in the scene, what the new visualization components are and that the user should experiment around by moving the object, experimenting with directing the ray towards the AABB and away from it. There are three visualization components implemented in order to visualize how the AABB accelerates the ray casting:

- Acceleration status section;
- Amount of rays ignored section;
- Hitpoint sphere on the AABB.

The acceleration status bar indicates to the user if the ray has been ignored, has hit the AABB or has hit the object or not. It updates as the object is moved around by the user. The amount of rays ignored bar is responsible for displaying the number of rays that were ignored while rendering an image. An image must be rendered first in order to see a number here, and the level hints for the user to do so. Finally, the hitpoint sphere is seen on the AABB as a green dot only when the ray intersects the AABB. This helps with showing that the ray intersects the AABB, even though the intersection point is not neccesarily important for acceleration.

When the level starts, the ray cast hits the object (and of course its AABB). This is seen in Figure 11. As the user interacts with the object through transformations, when the ray hits the AABB but misses the object, the level shows that through the acceleration status text on the top left, as seen in Figure 12. The other case is when the ray cast completely misses the AABB, and in that case it is made clear to the user that the ray is ignored and not considered for any intersection tests with the object and for any ray tracing calculations, seen in Figure 13.

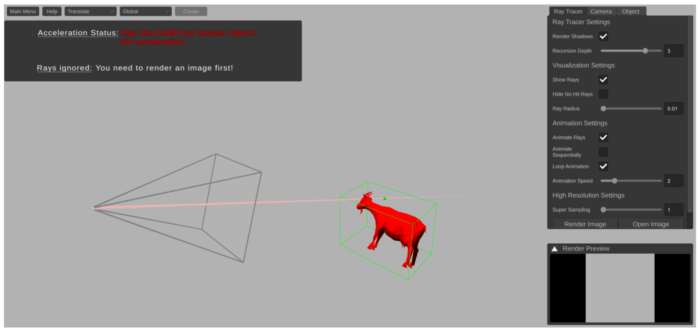


Acceleration Status: Ray hits AABB and the object!

Rays ignored: You need to render an image first!

Render Shadows ✓
Recursion Depth 3
Show Rays ✓
Hide No Hit Rays
Ray Radius 0.01
Animate Rays ✓
Animate Sequentially
Loop Animation ✓
Animation Speed 2
Super Sampling 1

Figure 11: A screenshot of the AABB Tutorial Level 2, where the ray hits the AABB and the 3D object.

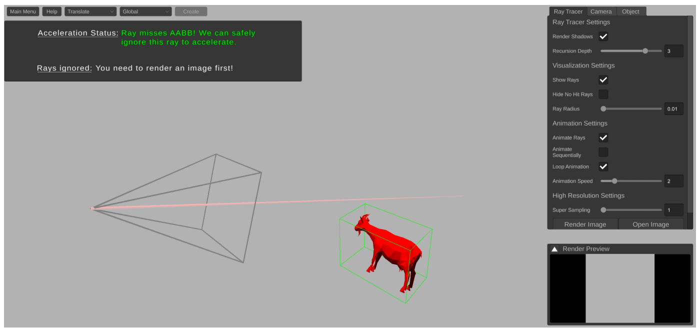


Acceleration Status: Ray hits AABB but misses object! No acceleration.

Rays ignored: You need to render an image first!

Render Shadows ✓
Recursion Depth 3
Show Rays ✓
Hide No Hit Rays
Ray Radius 0.01
Animate Rays ✓
Animate Sequentially
Loop Animation ✓
Animation Speed 2
Super Sampling 1

Figure 12: A screenshot of the AABB Tutorial Level 2, where the ray hits the AABB but misses the 3D object itself.


Acceleration Status: Ray misses AABB! We can safely ignore this ray to accelerate.

Rays ignored: You need to render an image first!

Render Shadows ✓
Recursion Depth 3
Show Rays ✓
Hide No Hit Rays
Ray Radius 0.01
Animate Rays ✓
Animate Sequentially
Loop Animation ✓
Animation Speed 2
Super Sampling 1

Figure 13: A screenshot of the AABB Tutorial Level 2, where the ray misses the AABB.

4.3 Octree

4.3.1 Calculation

Compared to the AABB, the calculation of an octree is more complex. To make the abstraction clearer, it is split into three different classes; Octree, OctreeRoot and OctreeNode.

We know the OctreeRoot will always have a root node that is calculated by first getting the Bounds of the object, and then transforming that Bounds box into a cube with equal sides. This results in an AABB that is scaled up to its largest side on each side. This ensures that when splitting the Octree into its 8 children, the children nodes are calculated by evenly splitting the cube into the octants with respect to the x, y and z axes. With only the root node established, the octree in Figure 14 is reached, which is considered an octree with a depth of 0.

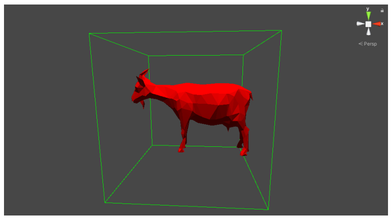


Figure 14: Octree with depth 0

After the root node is settled, now it is time to subdivide the Octree into its children nodes. For this, selective subdivision must be done. In other words at each point where a node can be subdivided or not, we check if the depth limit has been reached or if there any triangles are present within the node, while iterating over all triangles of the 3D object's mesh. The conditions of reaching depth is handled by the following code which stops an ongoing subdivision and adds the triangle to the list of contained triangles in the respective node.

```
// If max depth reached add the triangle and stop subdivision
if (currentDepth == 0) {
    containedTriangles.Add(new Triangle(t.v1, t.v2, t.v3));
    return;
}
```

If the max depth has not been reached, then whether or not the triangle is contained by any children of the current node is checked and handled in the following for loop when subdividing. The depth is handled in the `OctreeNode` initilization call, by passing `currentDepth - 1` as the depth parameter because this new node will be 1 level closer to the max depth criteria.

```
for(int i = 0; i < 8; i++) {
    // If children do not already exist, instantiate node
    if (children[i] == null)
        children[i] = new OctreeNode(childBounds[i],
                                     currentDepth - 1);

    // Check if triangle is included in child
    if (childBounds[i].Contains(t.v1) ||
        childBounds[i].Contains(t.v2) ||
        childBounds[i].Contains(t.v3)) {
        // If so, keep dividing and
        // call subdivision with this triangle on the child
        dividing = true;

        // Call subdivision on this child that contains the triangle
        children[i].DivideAndAdd(t);
    }
}
```

If there are no triangles contained within a nodes bounds, there is no reason to subdivide in that node, because the object is not contained there anymore. Such subdivision would lead to unnecessary children nodes with empty list of triangles, not containing any spatial information.

The subdivision is started off by calling it on the root node, which then recursively calls subdivide on it's children. An octree with depth limit 2 is visualized in Figure 15. As the depth goes from 2 to 3, the octrees dense sections over the 3D object appear, seen in Figure 16.

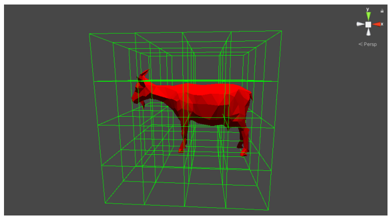


Figure 15: Octree with a depth of 2

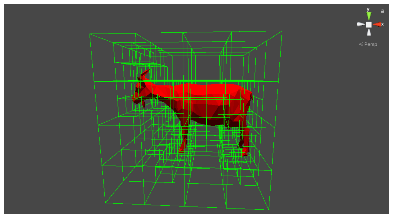


Figure 16: Octree with a depth of 3

4.3.2 Visualization

To visualize any given octree node, the function `Draw(OctreeNode node)` is used. The parameter node is the current node being drawn passed onto the function. We use the same draw call from the Popcron Gizmos package from the AABB visualization as

```
// node: current node
// nodeBounds: Bounds of the node being drawn
Popcron.Gizmos.Bounds(node.nodeBounds, Color.green);
```

In order to completely visualize and draw all nodes of an octree, recursion is used. After drawing itself, all nodes call the `Draw(OctreeNode node)` function on their children by recursion.

```
if (node.children != null) {
    for (int i = 0; i < 8; i++) {
        if (node.children[i] != null)
            Draw(node.children[i]);
    }
}
```

When this recursion is started by calling it on the root node of the octree, the whole octree in question will be drawn because all children will be visited by the recursion. This is achieved in the Octree components `Update()` function by the following line of code

```
Draw(octreeRoot.rootNode);
```

4.3.3 Utilization

Due to the underlying core function behind the ray traced rendering of VRT, the iteration over primitives, in this case triangles, is abstracted away. The `Physics.Raycast` function is given a ray and it checks for a single rays intersection between all objects in a specified layer. Thus, it is not possible to modify this ray intersection testing process such that it only checks for intersection between a set of triangles. The intersection testing calculations of the `Physics.Raycast` function is abstracted away from the developer. Render time acceleration with the octree is thus not achieved and the resulting rendering times are not expected to be faster than the default or AABB accelerated renderings. A workaround for this is discussed in the section 8, which includes run-time mesh editing or run-time layer masking with clone objects. Even though the rendering process itself is not accelerated, the triangle references in all octree nodes and the acceleration status displayed in the octree levels are correct and there is no issue in this aspect. With these correct references and values, we can still analyze the number of intersection tests the octree would have saved.

In order to utilize the Octree to accelerate the ray tracing, the ray cast is intersected with it before the object. The utilization starts at the root node much like the visualization process. If the root node is not intersected, the Octree and the 3D object can be ignored for any given ray. Then the background color is returned for the pixel rendered.

```
if (!ot.nodeBounds.IntersectRay(raycast)) {
    return BackgroundColor;
}
```

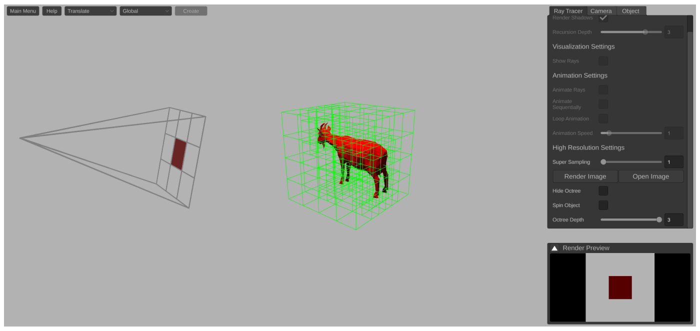
If the ray does intersect the Octree root, then the ray is checked for intersection with nodes through the tree structure. If an intersected node contains triangles, they are passed on to be ray traced. We should also check for intersection with the children of any intersected node. Such triangles in intersected nodes are kept track of in order to visualize to the user how many triangles should be utilized (and consecutively ignored) in the end.

```
private void IntersectOctreeNode(OctreeNode node, Ray ray) {
    if (node.children != null) {
        for (int i = 0; i < 8; i++) {
            if (node.children[i] != null &&
            node.children[i].nodeBounds.IntersectRay(ray)) {
                if (node.children[i].containedTriangles.Count != 0) {
                    octreeStatusFlag = true;
                    trianglesNotIgnored +=
                    node.children[i].containedTriangles.Count;
                }
                // Proceed with the children of this intersected node
                IntersectOctreeNode(node.children[i], ray);
            }
        }
    }
}
```

In the function above, the ray proceeds with checking for intersections with the nodes of an already intersected node if they do exist and are not null. Eventually there will be no more recursive calls to this function and thus the intersection process will end.

4.3.4 Tutorial Level 1

In the first tutorial level, the octree is introduced. The interactivity consists of transforming the object around and allowing the user to spin the object around and experiment with different octree depths using a slider in the panel. This level is completely the same as with the first AABB level but with the data structure as an octree instead. There is a slider for the depth value of the octree on the user panel. The screenshot of the level is shown in Figure 17.

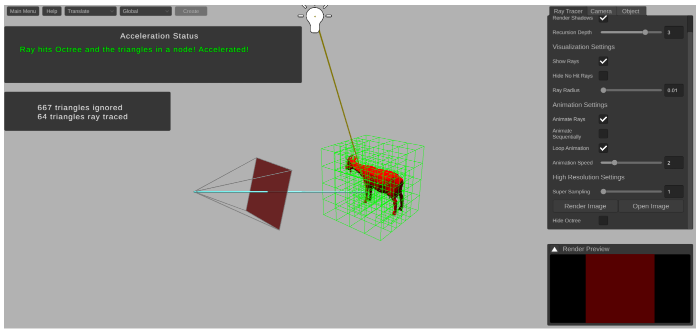


Render Shadows ✓
Recursion Depth 3
Show Rays
Animate Rays
Animate Sequentially
Loop Animation
Animation Speed 1
Super Sampling 1
Hide Octree
Spin Object
Octree Depth 3

Figure 17: A screenshot of the Octree Tutorial Level 1. The Octree is visualized over the 3D object in the scene with a green color. The spin toggle button and depth slider is seen on the right panel.

4.3.5 Tutorial Level 2

In the second tutorial level, a single ray is cast towards the object and its octree. There are specific cases for the ray intersection and acceleration process, and they are all shown below in seperate figures from Figure 18 to 21. The status display on the top left is crucial for the user to look at as they are experimenting with the level to see what is happening with respect to the Octree utilization and ray hit status.

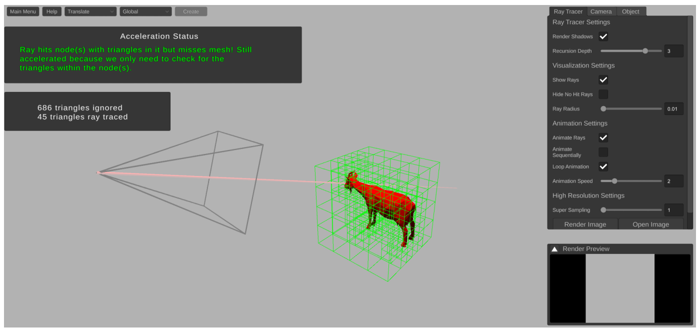


Acceleration Status
Ray hits Octree and the triangles in a node! Accelerated!

667 triangles ignored
64 triangles ray traced

Render Shadows ✓
Recursion Depth 3
Show Rays ✓
Hide No Hit Rays
Ray Radius 0.01
Animate Rays ✓
Animate Sequentially
Loop Animation ✓
Animation Speed 2
Super Sampling 1
Hide Octree

Figure 18: A screenshot of the Octree Tutorial Level 2, where the ray hits both the octree and the object.



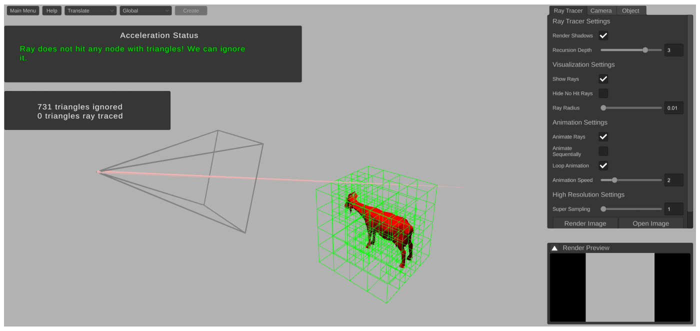
Acceleration Status

Ray hits node(s) with triangles in it but misses mesh! Still accelerated because we only need to check for the triangles within the node(s).

686 triangles ignored
45 triangles ray traced

Ray Tracer Settings
Render Shadows ✓
Recursion Depth 3
Visualization Settings
Show Rays ✓
Hide No Hit Rays
Ray Radius 0.01
Animation Settings
Animate Rays ✓
Animate Sequentially
Loop Animation ✓
Animation Speed 2
High Resolution Settings
Super Sampling 1

Figure 19: A screenshot of the Octree Tutorial Level 2, where the ray hits octree node(s) which contain primitive triangles in it.



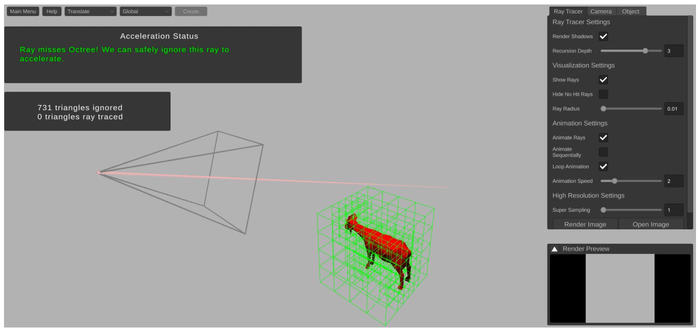
Acceleration Status

Ray does not hit any node with triangles! We can ignore it.

731 triangles ignored
0 triangles ray traced

Ray Tracer Settings
Render Shadows ✓
Recursion Depth 3
Visualization Settings
Show Rays ✓
Hide No Hit Rays
Ray Radius 0.01
Animation Settings
Animate Rays ✓
Animate Sequentially
Loop Animation ✓
Animation Speed 2
High Resolution Settings
Super Sampling 1

Figure 20: A screenshot of the Octree Tutorial Level 2, where the ray does not hit any octree node(s) which contain primitive triangles in it.



Acceleration Status

Ray misses Octree! We can safely ignore this ray to accelerate.

731 triangles ignored
0 triangles ray traced

Ray Tracer Settings
Render Shadows ✓
Recursion Depth 3
Visualization Settings
Show Rays ✓
Hide No Hit Rays
Ray Radius 0.01
Animation Settings
Animate Rays ✓
Animate Sequentially
Loop Animation ✓
Animation Speed 2
High Resolution Settings
Super Sampling 1

Figure 21: A screenshot of the Octree Tutorial Level 2, where the ray does not hit any octree node, as it misses the root node.

5 Evaluation

There is the assumption and expectancy that the acceleration data structure levels in VRT are explored after users have already went through the previous levels and are familiar with those concepts. This is because in order to be able to introduced to them, the learner must know ray casting and the presence of hit tests. Thus, we focus on a smaller form of feedback, namely from 3 peers within University of Groningen where we are certain about their knowledge on the previous, beginner concepts of ray tracing. All 3 of them have passed the Computer Graphics course at the university where ray tracing is studied in detail.

The users were asked to experiment with moving the camera and object around the scene to explore different acceleration cases. The following questions were asked in order to guide the users on what they can experiment with and evaluate;

- Can you clearly see the acceleration data structures and understand how they are calculated over a 3D object?
- Is it easy to experiment with a single ray and single object?
- Is the information displayed about the acceleration status understandable and does it help in understanding when acceleration occurs or not?
- How is the learning experience overall?

For the positive part of the feedback, it was noted that

- The acceleration data structures are cleanly visualized.
- Focusing on a single ray and single object helps in applying the concepts in a smaller scope.
- The acceleration status display in the second levels work well in showing the cases that can occur when accelerating ray tracing.
- No bugs were encountered.

As for the neutral, constructive part of the feedback, it was noted that

- The ability to change the 3D object into other meshes would be nice to see.
- Making the number of rays or objects plural might have a positive effect for further experimentation.

Finally, for the negative part of the feedback, it was noted that

- At certain angles, the octree is not clearly seen. The lines of wire cubes drawn in the denser areas of the octree get mixed into each other.
- There are no diagrams in the tutorial text.

These feedback points are discussed in Section 6 and used in Section 7 of this thesis in order to conclude the end result.

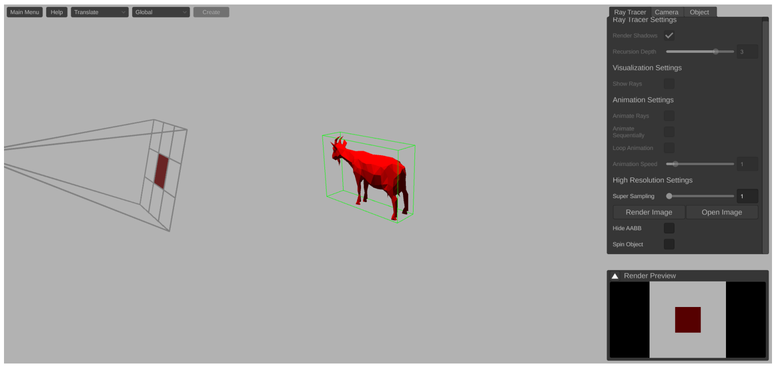
6 Discussion

In this section, we discuss the results that were achieved in the implementation with respect to the acceleration and visualization aspect of the project.

6.1 Acceleration

In this subsection, the resulting time and number of ray - triangle hit tests are analyzed. It is crucial to keep in mind that the data for the Octree are true theoretically, but as mentioned before, in the time period of this project, it was not possible to utilize the Octree acceleration. Thus, those values would be reached when it is utilized to accelerate rendering. It is also crucial to always keep in mind when discussing these results that such ray tracing on a scene is not really practical, as a scene will have many more objects, but we can still get an indication on the acceleration statistically.

All of the tests were done in the simple level with a single 3D object in the scene, which is a mesh resembling a goat, positioned right in front of the camera. This scene is seen in Figure 22



Ray Tracer Settings
Render Shadows ✓
Recursion Depth 3
Visualization Settings
Show Rays
Animation Settings
Animate Rays
Animate Sequentially
Loop Animation
Animation Speed 1
High Resolution Settings
Super Sampling 1
Hide AABB
Spin Object

Figure 22: A screenshot of the scene in which the rendering trials were done. In this particular screenshot, the AABB is being utilized.

Super Sampling 1	No Acceleration	AABB	Octree
Trial 1	0.1805	0.1542	0.2703
Trial 2	0.1692	0.1451	0.2830
Trial 3	0.1912	0.1448	0.2820
Average	0.1803	0.1480	0.2784

Table 1: Rendering times in seconds, under Super Sampling 1

Super Sampling 2	No Acceleration	AABB	Octree
Trial 1	0.5870	0.4703	1.0562
Trial 2	0.5487	0.4661	0.5879
Trial 3	0.5666	0.4761	1.0265
Average	0.5674	0.4708	0.8902

Table 2: Rendering times in seconds, under Super Sampling 2

Super Sampling 3	No Acceleration	AABB	Octree
Trial 1	1.2646	1.001	2.2801
Trial 2	1.2697	0.9977	2.1577
Trial 3	1.2232	0.9767	1.2862
Average	1.2525	0.9915	1.9080

Table 3: Rendering times in seconds, under Super Sampling 3

The AABB accelerates the rendering process by an average percentage of 17.91% for Super Sampling 1, 17.03% for Super Sampling 2 and 20.83% for Super Sampling 3 compared to the no accelerated default rendering.

For the octree, it is seen how the rendering is not accelerated as expected but slows down due to the extra intersection tests between the rays and the octree nodes. If the octree nodes containing triangles were to be utilized to ignore the rest of the triangles of the mesh, we would see the decrease in number of ray - triangle hit tests listed in Table 4.

	No Acceleration	AABB	Octree
Super Sampling 1	116,960,000	22,994,336	1,173,277
Super Sampling 2	467,840,000	92,273,399	4,700,384
Super Sampling 3	1,052,630,000	207,390,548	10,557,042

Table 4: Number of ray - triangle intersection tests

From Table 4, the number of ray and triangle intersection tests were calculated. The amount of intersection tests decreases with both acceleration data structures.

The AABB results in an average decrease in ray and triangle intersection tests by 80.34%, 80.27% for Super Sampling 2 and 80.3% for Super Sampling 3 compared to the no accelerated rendering. The octree results in an average decrease in hit tests by 98.99% for all Super sampling levels, compared to the no accelerated rendering.

6.2 Visualization

For the visualization aspect of the implementation, it can be said that there are no key design elements missing. Even though the octree does not accelerate rendering, it stays as an essential component which can keep track of the triangles of the mesh and their indices, thus allowing proper visualization and education on octrees.

The AABB accelerates the ray tracing as designed. The acceleration data structures are interactive (real-time updated and dynamic), drawn as a wire frame on top of the 3D object. From the feedback, the visualization is denoted as clear, except for the octree because of certain angles and the background color.

When it comes to the first levels, they focus on introducting the data structure with all ray tracing aspects of VRT disabled. By focusing on the data structures re-calculation and interaction within the scene, an introductory level is achieved.

As for the second levels, the aim was to focus on visualizing the acceleration process between a single object and a single ray. Through the feedback it is seen that by restricting the visualization to a single ray and object, the user can focus on interacting with the camera and object to see how the acceleration applies to a single ray and for any other that would have been cast. The display of the acceleration process on the top left corner helps in showing the learner when a change occurs in the acceleration process. The number of rays or triangles ignored also goes into statistical detail on the acceleration process. It is a feasible way to cover the cases that occur when acceleration data structures are used so the user can learn and experiment.

7 Conclusion

In this project, acceleration data structures were added to VRT with the aim of visualizing how acceleration is achieved and accelerating the ray traced rendering. Two acceleration data structures were included: axis-aligned bounding boxes and octrees. To conclude the best way to visualize acceleration data structures in VRT, we refer to the feedback evaluation in Section 5 and discussion in Section 6.

As a conclusion for accelerating VRT rendering, it was discussed through the timing of renderings and amount of ray triangle intersection tests that the axis-aligned bounding box accelerated the rendering time of VRT by 20.83% when the super-sampling was set to its maximum value of 3, casting more rays. That is a reasonable increase in speed, especially considering if it was to be used in real-time ray tracing. As for the octree acceleration, it was discussed in its utilization how the nature of the Unity Physics function used at the core of the ray tracer, Physics.Raycast was hiding the ray - triangle intersection calculations and process, making it not possible to filter out the triangles not included in intersected octree nodes in a conventional and feasible way. It is mentioned in further work how this could be worked around using Unity layer masking or run-time mesh editing. In Table 4 of Section 6, the decrease in the number of ray - triangle intersection tests does not mean there is a perfect speed-up of 80.3% for the AABB nor 98.99% for the octree. These percentages do not take into count the time spent on utilizing the data structures, they are solely based off of the number of intersection tests. So the decrease in time spent rendering is a better indicator then the amount of decrease in ray - triangle intersection tests when it comes to concluding increase in rendering speed.

On the other hand is the conclusion for the visualization aspect. The visualization of the acceleration data structures is critical for the learner to see how they are calculated and utilized when ray tracing. From the end result of the tutorial levels, it was seen that the acceleration data structures are displayed clearly over a mesh, and update accordingly to the transformations on the mesh. The acceleration process where the structures are utilized is also displayed to the user through the panel on the top left of the user interface, with correct updates and status. From feedback evaluation in Section 5, it was seen that focusing on a single ray and object helps in applying the concepts in a smaller scope and understanding how the acceleration works for any (or all) rays. No bugs were encountered and the acceleration cases were showing the process clearly. There are also missing points in terms of the visualization noted from the feedback. The ability to change the 3D mesh into other meshes is not possible, and it would have helped with seeing how the structures form on other meshes. Allowing the visualization on more rays or objects would have been nice to see as well, but it is reasonable that this would require different levels, because the acceleration status display would have to change and evolve in a different way. There was also negative feedback on some aspects of the visualization. The octree gets harder to see in certain angles, the denser parts of the octree mixes up the green lines into each other. Also, in the tutorial text before a level starts, there are no diagrams, which made it harder to go straight into the level on a 3D visualization of a structure.

Overall, 4 tutorial levels, educating the users on axis-aligned bounding boxes and octrees, were added to VRT. The ray traced rendering has been accelerated effectively with the axis-aligned bounding box, and is close to achieving even better acceleration with the octree, but the implementation currently does not utilize the mesh triangle references it holds due to Unity ray casting functionality hiding away the control over which primitives of a mesh are tested for intersection or not. The rendering time speed up has been recorded and discussed using the built-in Unity time system. The acceleration with the AABB proved a significant speed-up of 20.83% when super sampling is set to maximum. The decrease in number of ray - triangle intersection tests hint that when the octree utilization is fixed and implemented, there will be a much higher speed-up in rendering. The visualization approach taken proved to be successful in displaying acceleration data structures. There is room for improvement with certain aspects such as wire frame lines in octrees being harder to distinguish at certain angles. The project is looking forward to act as an extension of VRT, broadening its functionality and educative benefit for learners in the field of ray tracing.

8 Future Work

The future work of this project consists mainly of the following points:

- Implementing octree utilization using run-time mesh editing by filtering the triangles that should be ignored or not, or by using layer masking to ignore parts of 3D meshes in the scene;
- Working on feedback gained on the visualization aspect to improve visibility and interactivity;
- Add more statistical information on the acceleration process to the levels.

All of these points can be built on top of what has already been implemented.

9 Acknowledgements

I thank and appreciate my supervisors prof. dr. Jiří Kosinka and dr. Steffen Frey for their time and care in supervising me through the project from start to finish. I also give my thanks to Chris van Wezel for his support and guidance on the VRT ray tracer implementation and the Unity game engine.

10 References

[1] W.A. Verschoore de la Houssaije. A Virtual Ray Tracer, BSc Thesis, University of Groningen, 2021.

[2] NVIDIA Developer. NVIDIA RTX ray tracing. `https://developer.nvidia.com/rtx/ray-tracing`, 2022.

[3] NVIDIA Developer. Ray Tracing. `https://developer.nvidia.com/discover/ray-tracing`, 2022.

[4] popcron. gizmos README. `https://github.com/popcron/gizmos/blob/master/README.md`, December 2019.

[5] Unity Technologies. Scripting API: Gizmos. `https://docs.unity3d.com/ScriptReference/Gizmos.html`, 2022.

[6] Unity Technologies. Scripting API: MeshRenderer. `https://docs.unity3d.com/ScriptReference/MeshRenderer.html`, 2022.

[7] C.S. van Wezel. A Virtual Ray Tracer, BSc Thesis, University of Groningen, 2022.

[8] S. Frey W.A. Verschoore de la Houssaije, C.S. van Wezel and J. Kosinka. Virtual Ray Tracer, Eurographics 2022 Education Papers. `https://diglib.eg.org/handle/10.2312/eged20221045`, 2022.

[9] wezel. Virtual Ray Tracer README. `https://github.com/wezel/Virtual-Ray-Tracer/blob/8cc116180dd20c354c524ae7b94310f1bed71fac/README.md`, 2022.