Musschoot Adriaan

Implementing acceleration structures to optimize a sphere tracing ray marcher

Supervisor: Samyn Koen

Coach: Geeroms Kasper

Graduation Work 2024-2025

Digital Arts and Entertainment

Howest.be

CONTENTS

1 ABSTRACT

Sphere tracing, presented in (Hart, 1996), is a sub-branch of ray marching, a real-time rendering technique. It has many similarities to ray tracing and is often considered its little brother. One critical aspect for these rendering techniques is calculating where objects are in the scene and determining if said objects are hit by the rays cast by the rendering algorithm. Objects in a sphere tracing context are represented by signed distance fields (SDFs).

In ray tracing there are many acceleration structures to speed up this process. In this paper the Axis Aligned Bounding Box (AABB) and Bounding Volume Hierarchy Tree (BVHTree), commonly used in ray tracing, are considered and implemented for SDFs in a sphere tracing context. For each acceleration structure two types of shapes were considered and implemented, a sphere and a box.

In the tests ran performance of each implementation was compared when either no acceleration structure, the sphere implementation or the box implementation was active. In most cases the experiment yielded improved performance when applying the acceleration structures.

All tests were executed in static scenes and construction of the structures happened at the start of the application, in future work this should be tested for dynamic scenes and reconstruction should be tested at run time.

2 ABBREVIATONS

AABB	Axis Aligned Bounding Box
ALU	Arithmetic-Logic Unit
API	Application Programming Interface
BVHTree	Bounding Volume Hierarchy Tree
CPU	Central Processing Unit
GPU	Graphics Processing Unit
SAH	Surface Area Heuristic
SDF	Signed Distance Field

3 LIST OF FIGURES

4 INTRODUCTION

Rendering 3D scenes out and presenting them to 2D screens has been a challenge in the digital world since the dawn of computing. Multiple approaches exist, new ones get invented and old ones get brought back to life. This paper revisits sphere tracing, a sub-branch of ray marching.

The technique has been around for a while (Wyvill & Wyvill, 1989) in different shapes. It was mostly used as baked volumetric data. Recently, GPUs were evolving their computational/ALUs abilities faster than their memory band width. This meant purely mathematical SDFs became competitive against their 3D-texture based SDFs counterpart. (Quilez, n.d.)

When compared to rasterization, which consists of the vertex shader, rasterization and the fragment shader, the ray marching algorithm can be fully implemented in the fragment shader, allowing for integration with existing pipelines. Its use cases are cheap ambient occlusion, screen space reflections, rendering volumetrics, rendering clouds. (Papaioannou et al., 2010; “Ray Marching,” 2024; Schneider, 2024; Tomczak, 2012).

Some other places where ray marching can be found include:

1. In Unreal Engine 5 ray marching is used for ambient occlusion and distance field shadows. (Unreal Engine 5, n.d.-b, n.d.-a)
2. There is a game called “Claybook” (*Claybook*, n.d.) which handles the entire rendering pipeline using ray marching and fully exploiting the benefits and quirks of raymarching.
3. The website “Shadertoy” (*Shadertoy*, n.d.) developed by Inigo Quilez and Pol Jeremios is a tool used to teach and create demo scenes. It purely uses the ray marching algorithm.
4. There is also the “Demoscene” community, they specialize in creating non-interactive audio-visual presentations executed in real time on computers. (“Demoscene,” 2019) Which has embraced “Shadertoy” and SDFs in general for their scenes.

When looking into these applications, mainly “Shadertoy” creations and creations by the “Demoscene” community, most approaches didn’t really consider the performance. The priority was always to get beautiful scenes on the screen, but higher complexity and bigger scenes require more performance.

This paper discusses the axis aligned bounding box and the bounding volume hierarchy tree, which are acceleration structures used in ray tracing, and apply those same structures to sphere tracing.

The goal is to compare performance for each acceleration structure.

The encompassing research question is the following: How will applying mainstream ray tracing optimization techniques to sphere tracing affect performance for the sphere tracing rendering method? The following sub questions were also considered.

1. How does applying AABB to SDFs in sphere tracing affect performance?
2. How does applying a BVH-Tree to SDFs in sphere tracing affect performance?
3. How do sphere-shaped and box-shaped implementations compare in performance for AABB?
4. How do sphere-shaped and box-shaped implementations compare in performance for BVH-Tree?

The encompassing hypothesis is the following: The sphere tracing rendering method should benefit in performance from mainstream optimization techniques used in ray tracing. The following sub hypothesis were considered:

1. Applying the AABB will improve performance positively in most cases, except when the underlying SDF is not complex enough.
2. Applying the BVH-Tree will improve performance positively in most cases, except when the scene complexity and SDF complexity are too simple.
3. Sphere-shaped AABBs will improve performance more than their box-shaped counterparts.
4. Sphere-shaped BVH-trees will improve performance more than their box-shaped counterparts.

5 THEORETICAL FRAMEWORK

5.1 RAY TRACING

5.1.1 FUNDAMENTALS

5.1.1.1 TRIANGLE MESHES

At the core of todays computer graphics, 3D objects need to be visualized. The most popular approach is triangle meshes. These consist of triangles (Figure 1), each triangle consist of 3 vertices and 3 edges connecting these vertices (“Triangle Mesh,” 2024). There are multiple algorithms to convert triangle meshes from 3D space to a 2D screen.

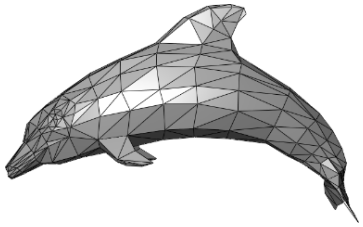


Figure 1: Example of a triangle mesh

5.1.1.2 RAY TRACING ALGORITHM

The ray tracing algorithm uses these meshes to fill the pixels on the screen with meaningful colour.

For every pixel on the screen the algorithm casts a ray from the origin through each pixel in an image plane into the scene (the origin is often referred to as the camera or eye point). Each ray tries to intersect with every triangle in the triangle mesh in the scene to determine its intersection point. If a ray passing through a pixel going into the 3D scene intersects with a mesh, the colour data from the mesh contributes to the final colour of the pixel (Figure 2). The process of executing this for one pixel is called ray casting.

The ray can bounce and repeat this process with different origins and directions to incorporate light, reflections and shadows. (*Ray Tracing | NVIDIA Developer*, n.d.)

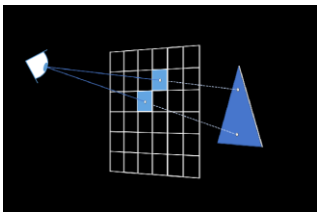


Figure 2: The process of ray casting and shading pixels

5.1.2 ACCELERATION STRUCTURES

The main purpose of acceleration structures is to speed up the process of ray casting. They can help quickly determine which rays are likely to hit which objects or parts of an object. The following two scenarios are presented as bottlenecks for performance.

- Scenario 1:

  When using a scene with a large quantity of triangle meshes. For every ray cast the algorithm needs to calculate the intersection point for every object in the scene. If the ray doesn't hit anything we still had to traverse all meshes and try to intersect with all their triangles. We could create a box around each triangle mesh with which we intersect first, to quickly be able to determine which meshes the ray is most likely to intersect with.

- Scenario 2:

  The triangle meshes in the scene are very complex, meaning they consist of a high number of triangles. Every triangle is than intersected with the ray, only for the algorithm to return one triangle it intersected with. This could be improved by grouping triangles and estimating which are most likely for the ray to intersect with reducing the number of triangle intersections.

5.1.2.1 AXIS ALIGNED BOUNDING BOX

For scenario 1 (5.1.2) the AABB is used to speed up the ray casting process.

It is a simple box which represents the closed region that contains a triangle mesh. The edges of the box are always aligned with the world axes, meaning that they are parallel to the x, y, and z axes in three-dimensional space.

To define the AABB for a triangle mesh the minimum and maximum points along each axis that defines the mesh must be found. Then the AABB can be constructed, using the minimum and maximum as two opposite corners of the box. ("Bounding Volume," 2024; *CSSE451 Advanced Computer Graphics*, n.d.)

The ray tracing algorithm can then calculate ray intersections using the AABB of a mesh, if and only if that is the case, does it calculate further intersections with the triangles from the mesh. This method of early out speeds up ray intersection calculations in the case the ray misses an object.

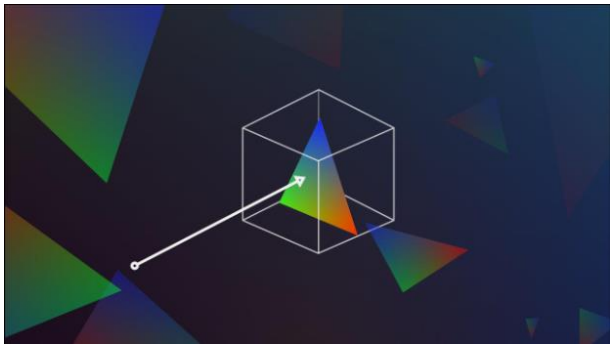


Figure 3: AABB for a single triangle

5.1.2.2 BOUNDING VOLUME HIERARCHY TREE

BVHTree is an improved approach to AABB to speed up the ray intersection process.

Calculating intersections with all triangles of a complex mesh such as in Figure 4 is costly. To avoid this, the BVHTree subdivides the initial AABB into smaller AABBs. This process repeats until one smaller box contains the desired number of triangles. (*Introduction to Acceleration Structures*, n.d.; Kay & Kajiya, 1986; Sebastian Lague, 2024) This allows you to determine the level of detail for the boxes and the minimum number of triangles per smaller bounding box.

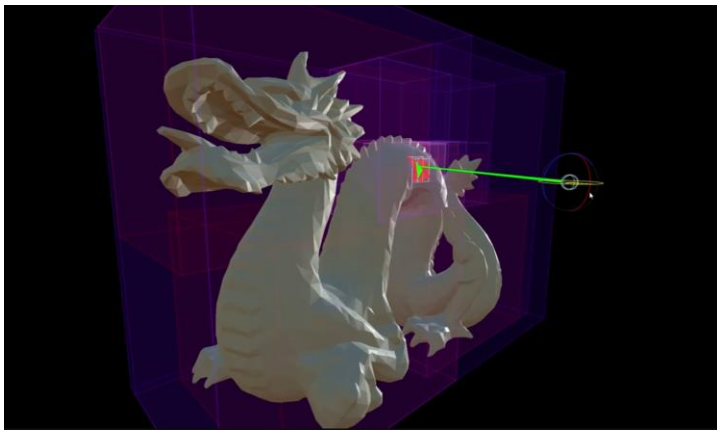


Figure 4: Complex mesh with BVH applied (Sebastian Lague, 2024)

The ray intersection process then follows these steps:

1. The ray tries to intersect with the root node. This is the initial AABB.
    a. If it doesn't intersect, the triangles of that mesh don't need to be intersected with.
    b. If it does intersect, the ray proceeds to step 2.
2. The ray tries to intersect with one of the bounding boxes of its child nodes.
    a. If it doesn't intersect with the child node, the ray can discard the entire sub-tree (Figure 5).
    b. If it does intersect with the bounding box of a child node and the node has child nodes, it repeats step 2 for those child nodes.
    c. If it does intersect with the bounding box of a child node and it is a leaf node, meaning no more children (Figure 5), the ray evaluates its intersection point with the triangles contained within the nodes bounding box.

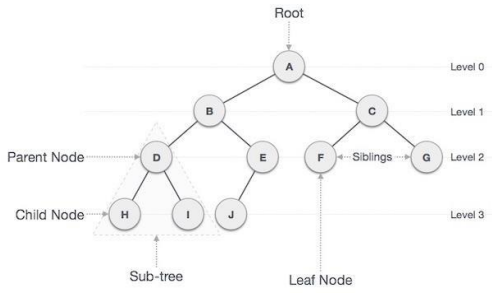


Figure 5: Tree structure for BVH (Goren, 2021)

This structure can also be applied on scene level. Instead of subdividing the triangles of triangle meshes the entire scene is divided. These steps are followed to construct a BVHTree on scene level.

1. All objects in the scene are grouped and form the root node and the initial bounding.
2. The group's bounding box is formed by finding the minimum and maximum extent out of all triangle meshes.
3. Then the objects are split into two groups using the SAH (5.1.2.2.1).
4. Each group is then presented to its left and right child node respectively and they repeat step 2, 3 and until there is only one object left in the group.

5.1.2.2.1 SURFACE AREA HEURISTIC (SAH)

The surface area heuristic (MacDonald & Booth, 1990) is a core metric to determine the optimal axis and split index to divide the objects in two groups. It estimates the likelihood a ray intersects with an object, based on the observation that the number of rays likely to intersect with an object is roughly proportional to its surface area.

This metric (leftCost line 12 and rightCost line 13 in **Error! Reference source not found.**The pseudo code

is calculated by multiplying the number of objects with the surface area of the bounding volume containing those objects. When the addition of both metrics (cost in the code below) is the lowest, that axis and split index is used to separate the group into two.

Pseudo code SAH evaluation

```
 1. vector3 bestAxis
 2. float bestCost = FLT_MAX
 3. int bestSplitIndex
 4. for every axis
 5.    sortedObjects = sort objects along currentAxis;
 6.
 7.    for every index in the sortedObjects
 8.      leftObjects = objects from start of sortedObjects until currentIndex
 9.      rightObjects = objects from currentIndex until end of sortedObjects
10.      leftArea = bounding box area around leftObjects
11.      rightArea = bounding box area around rightObjects
12.      leftCost = leftArea * nrOfLeftObjects
13.      rightCost = rightArea * nrOfRightObjects
14.      cost = leftCost + rightCost
15.
16.    if cost < bestCost
17.       bestCost = cost
18.       bestAxis = currentAxis
19.       bestSplitIndex = currentIndex
20.
21. sortedObjects = sort objects along bestAxis
22. leftObjects = objects from start of sortedObjects until bestSplitIndex
23. rightObjects = objects from bestSplitIndex until end of sortedObjects
```

5.2 SPHERE TRACING

5.2.1 FUNDAMENTALS

5.2.1.1 SIGNED DISTANCE FIELDS (SDFS)

Another way to visualize 3D objects is SDFs.

There are 2 ways of storing SDF data. One being baked volumetric data and the other being a mathematical function. The latter is explained hereafter and is the one referred to when writing about SDFs.

SDFs describe a shape in terms of distance of any point in space to the shapes boundary. By providing any point to the SDF, it will return the shortest distance to the surface of the shape it represents even if the point is inside the shape.

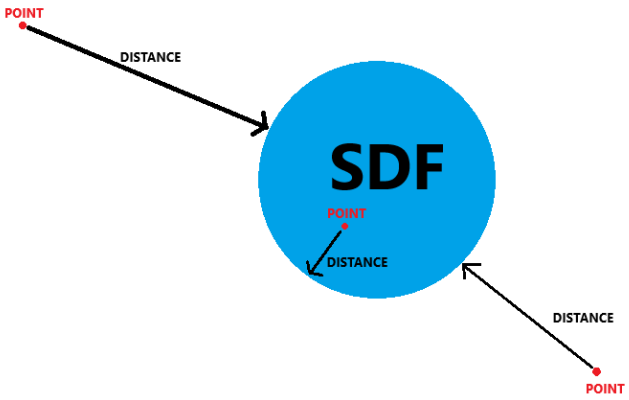


Figure 6: SDF Visualization

Let's look at the simplest shape to calculate: a sphere.

Pseudo code sphere SDF

```
1. float GetDistance(vector3 point)
2. {
3.         return length(point) - radius;
4. }
```

It takes in a point, calculates the length to the point and subtracts the radius. If this result is negative, the point is inside the sphere, if it is positive the point is outside the sphere. When the returned value is exactly 0, the point lays on the surface of the sphere.

This is assuming the sphere is at position (0, 0, 0). To have translations, rotations or scaling the point must be manipulated before calculating the final value. (Quilez, n.d.)

A benefit of SDFs is that they are fully implemented in the fragment shader. So they require less memory than triangle meshes which are stored in a file format (.obj or .fbx) and need to be passed to the shader.

A single triangle takes up 36 bytes, 4 for each component of its 3 vertices. The sphere only takes up 16 bytes, 4 for each component of its origin and 4 for the radius. When comparing these two in size, greater detail is achieved with the latter for a smaller memory footprint.

5.2.1.2 RAY MARCHING ALGORITHM

A different algorithm to ray tracing is required to visualize these SDFs, but with some similar principles. A camera is still used to cast rays through an image plane. However, rays cannot calculate any intersection points due to the nature of the SDFs. The most default approach is to step forward along a ray at a fixed increment. At every point the distance to the SDFs is calculated. If at any step the returned value of the SDF is negative, the algorithm considers the ray to have hit an object.

In Figure 7 one can see that if the increment size is too big, the algorithm can easily miss an object. On the contrary, if the step size is too small then it causes reduced performance. (Donnelly, 2005)

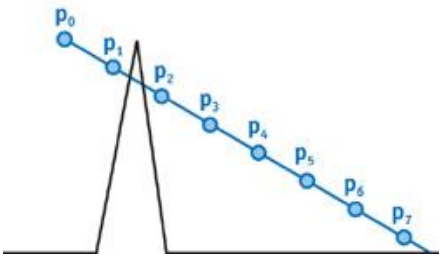


Figure 7: Ray marching missing a surface (Donnelly, 2005)

There is however a variation that is more efficient and precise: sphere tracing (Hart, 1996). The process goes as follows:

1. The distance to SDFs in the scene is evaluated using the origin of the ray.
2. Then the origin of the ray is displaced with the distance to the closest SDF in the scene, along the direction of the ray.
3. Steps 1 and 2 are repeated until one of 2 possible outcomes:
    a. The possible travel distance is smaller than an arbitrary value (0.001). Meaning the point is sufficiently close to an object it is considered a hit.
    b. The total distance travelled by the ray exceeds the scene boundaries.

Pseudo code sphere tracing algorithm

```
distance travelled = 0
origin = camera point
while (true)
      point = origin + distance travelled * direction
      possible travel distance = distance to scene based on point //Pseudo code distance to scene
      distance travelled = distance travelled + possible travel distance
      if distance able to travel < 0.001
                break //hit object
      if distance travelled > scene boundaries
                break //no objects hit
```

In Figure 8 both rays are being marched towards the triangle from left to right. The top ray represents the possible outcome (a) where the SDF result approaches zero. As a result, the pixel for which this ray was cast should be colored with the SDFs color properties. The bottom ray represents the possible outcome (b) where the distance the ray travelled exceeds the scene bounds (in this case the image bounds). (Devred, 2022)Pseudo code distance to scene

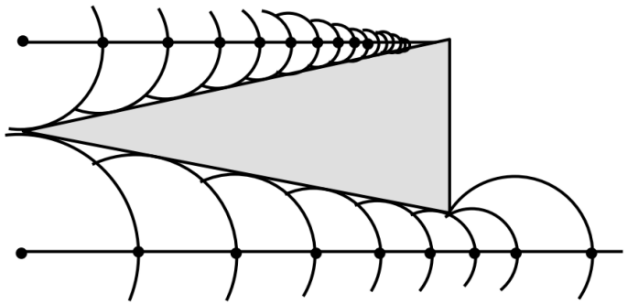


Figure 8: Example of both scenarios when raymarching (Hart, 1996)

While this approach is more memory friendly than using triangle meshes, it does mean the entire scene needs to be reconstructed every frame. As SDFs calculations become more complex they require more computation time.

The main bottleneck of performance is acquiring the distance to the scene for every step of the ray.

Pseudo code distance to scene

```
1. minimum distance = infinity
2.
3. for object in scene
4.        distance = distance to object based on point
5.        if distance < minimum distance
6.                  minimum distance = distance
7.
8. return minimum distance
```

This is a heavy computation happening every frame for every ray at each step it takes. If the application has a window of 100 by 100 pixels with an average ray step of 10 to traverse the scene. The ray marching algorithm would evaluate this function 100 000 times each frame. This linearly increases with the number of objects in the scene.

6 CASE STUDY

6.1 APPROACH TO SETTING UP THE EXPERIMENT

To determine the effect of applying the same acceleration structures commonly used for ray tracing, a framework is required. Creating a sphere tracing ray marcher from scratch is a hefty task and will take a considerable amount of time. The main goal is to compare the performance with and without acceleration structures.

I chose to implement this purely in C++. The main factor being development speed.

1. It is the language I am most familiar with.
2. Debugging the CPU is easier than when working with the GPU.
3. It has some nice functionality such as pointers and recursion which is not available in shader code.
4. When compared to GPU development it requires considerably less set up.
5. Easily measure performance and track other statistics as everything is available on the CPU. This is harder when working with the GPU and not possible in “Shadertoy”.

Some drawbacks are:

1. No use of the computational power of graphics cards. As this algorithm benefits hugely from the parallelism of GPUs. CPUs can make use of multithreading but have considerably less cores available than GPUs
2. Full implementation is required, when compared to implementing the experiment on “Shadertoy”.

Before I explored this topic, I had created my own implementation of the ray-tracing algorithm. As discussed earlier, the ray marching algorithm has some of the same traits as the ray tracing algorithm. So, this is a good building block to set up the experiment, I am also familiar with this framework. Unlike setting up something entirely from scratch using a graphics API.

6.2 AABBS FOR SDFS

6.2.1 AABBS FOR SDFS ARE EARLY OUTS

When working with AABB for triangle meshes (5.1.2.1), every point is known in world space, allowing for easy construction of the axis aligned bounding box.

However, the only data available from SDFs is the distance from the surface to a point. Given this property, the maximum boundaries that contain this SDF can be estimated. Two approaches for constructing the shape of the early out SDF were implemented, and their performance is compared in 6.2.5.

This property can be used to determine the furthest distance for any axis and use it to create a sphere (Figure 9) around the objects or the furthest distance for each axis and use it to create a box (Figure 10) around the object. Their construction is explained in 6.2.2 and 6.2.3 respectively.

Figure 9: Example of Sphere Early Out

Figure 10: Example of Box Early Out

Unlike in ray tracing rays cannot intersect with a box first using the ray marching algorithm. Instead, the ray marches towards the early out shape (the black part in Figure 9 and Figure 10) along the ray, until the origin of the ray is inside the early out SDF then the distance to the underlying SDF is computed (the red part in Figure 9 and Figure 10). This means an additional SDF as early out is necessary, for the underlying more complex SDF.

The pseudo code

```
1. float sdf::Object::GetDistance(glm::vec3 const& point)
2. {
3.       float const earlyOutDistance{ EarlyOutDistance(point) };
4.       if (earlyOutDistance >= 0.001f)
5.       {
6.                return earlyOutDistance;
7.       }
8.       return GetEnclosedDistance(point);
9. }
```

It should only return the distance of the early out test when it is greater than the threshold value (0.001). Otherwise, it returns the distance to the underlying SDF

This speeds up the ray marching process greatly because rays that miss the object must only compute the early out distance, which is computationally less expensive than the underlying shape. However, it does mean that once the ray is marching inside the early out SDF, the early out distance is evaluated as well as the distance to the underlying shape. However, the benefits of an early out hugely outweigh those extra calculations when approaching and hitting an object.

6.2.2 SPHERE EARLY OUT

6.2.2.1 CONSTRUCTING THE BOUNDARY

To construct the boundary or radius of the sphere early out, for every intersection of a spheres 360 longitude lines and its 360 latitude lines, a point is created and retrieves its distance value to the SDF. After executing this for the initial sphere radius, which encompasses the maximum scene boundary, the smallest possible distance from any point to the SDF is used as the radius of the next sphere. This process is then repeated until the surface of the SDF is reached.

```
1. float const nextRadius{ length(closestPoint) - minimumDistance };
```

Through this method the sphere can march towards the origin by decreasing its radius every time. Once one of the points on the sphere has a distance value smaller than a certain threshold (0.001), the point is considered to have reached the surface. The distance to that point is considered the furthest possible distance to any part of the surface of the SDF from the origin (0,0,0) in the cartesian coordinate system. Then the length of the vector representing this point is used as radius it to construct a sphere SDF acting as a form of early out.

The pseudo code

```
 1. while surface point has not been found
 2.       generate sphere points with radius
 3.
 4.       for each sphere point
 5.                distance = distance to SDF
 6.                if (distance < 0.001)
 7.                         store surface point
 8.                if (distance < closest distance)
 9.                         store distance and closest point
10.
11.       calculate and store the nextRadius value
```

Once this point is retrieved, we can determine the radius of the sphere. And use it as the early out SDF.

```
1. radius = glm::length(surfacePoint);
```

6.2.2.2 FORMULA

Pseudo code for the distance to a sphere (at the origin (0,0,0)).

```
1. float sdf::Sphere::EarlyOutTest(glm::vec3 const& point)
2. {
3.         return length(point) - radius;
4. }
```

This shape is the cheapest possible calculation for any shape in ray marching.

If the SDF enclosed by the early out sphere has a peak distance on the y-axis and is very slim in the xz-plane it, the sphere enclosing it will have the value of the y-axis as its radius. This creates a lot of empty space underneath the sphere surface.

6.2.3 BOX EARLY OUT

6.2.3.1 CONSTRUCTING THE BOUNDARIES

The following approach was used to approximate the boundaries of a box around any underlying SDF.

For each direction on every axis in the cartesian coordinate space (x, -x, y, -y, z, -z) a “wall” of points is constructed in a rectangular grid perpendicular to its direction. The initial distance at which the walls are constructed is the maximum scene boundary in the direction for which the wall was constructed. Each wall then calculates its minimum distance to the SDF and stores the point for which it was calculated.

The pseudo code

```
1. float const newDistanceValue{ glm::length(closestPoint * direction) - minDistance };
```

When calculating the length of the point we only want to incorporate the component of the point which is also present in the direction. Otherwise, if the point has a value for all 3 components, the new distance value might be greater than the previous one. And we would never approach the surface of the SDF.

Then each wall marches forward opposite its initial direction (so towards the origin) using the new distance value. Each wall repeats this process until one of the points on the wall has a distance value smaller than a certain threshold (0.001). Then that point is considered reaching the surface and the maximum distance along that axis and direction.

The pseudo code

```
 1. while not all walls have reached the surface
 2.
 3. for each direction
 4.
 5.    if wall has reached the surface //in previous iteration
 6.       do nothing for this direction
 7.    else
 8.       generate a wall of points using direction and new distance value //from last iteration
 9.       for each point on the wall
10.          distance = distance to SDF
11.
12.          if distance < 0.001
13.                store that point //wall has reached surface
14.          if distance < closest distance
15.                store distance and point //new minimum was found
16.
17.       calculate and store the new distance value for that wall
```

Once all 6 walls have evaluated their distance to the scene, the absolute values of the x, y and z components of each point are compared to determine the box extent of the SDF box. Which will be used to compute the early out box enclosing the SDF.

```
1. for each minimum point of each wall
2.        if boxExtent.x < abs(point.x)
3.                boxExtent.x = abs(point.x);
4.        if boxExtent.y < abs(point.y)
5.                boxExtent.y = abs(point.y);
6.        if boxExtent.z < abs(point.z)
7.                boxExtent.z = abs(point.z);
```

6.2.3.2 FORMULA

Below you can find the formula used for the distance of a box (at origin (0,0,0)).

```
1. float sdf::Box::EarlyOutTest(glm::vec3 const& point)
2. {
3.         vec3 const q{ abs(point) - boxExtent };
4.         return length(max(q, 0.0f)) + min(max(q.x, max(q.y, q.z)), 0.0f);
5. }
```

Due to the nature of the box SDF it is always a mirrored box. Meaning that even if the underlying SDF only has values in the positive y-axis the box would have the same length in the positive as in the negative y-axis.

A box SDF is still quite cheap to evaluate but slightly more expensive than a sphere SDF.

6.2.4 TESTING

The test compared 6 different objects in separate scenes, meaning 6 different SDFs each with different complexity.

Every object was tested without, with a sphere and with a box early out.

For object and each usage of early out, the camera position was also changed. Two camera positions were used. A close-up and a normal shot. The window size for the test was 600x600, in total there are 360 000 pixels total.

1. The close-up shot has a ray hit ratio of approximately 25% of the rays hitting an object and 75% misses an object. For the pixel count in the experiment this means 90 000 rays did hit and 270 000 rays that didn't hit.
2. The normal shot has a ray hit ratio of approximately 3% of the rays hitting an object and 97% missing an object. For the pixel count in the experiment this means 10800 rays did hit and 349200 rays didn't hit.

The field of view of the camera is set at 90 degrees for all shots.

For each possible combination the frame time was captured for all frames rendered within an 11 second window. The results were sorted and the top 5% and bottom 5% of all the fame times were removed. Resulting in approximately 10 seconds of frame times captured.

6.2.5 RESULTS

3 cases will be presented. More can be found in the appendices.

6.2.5.1 LINK

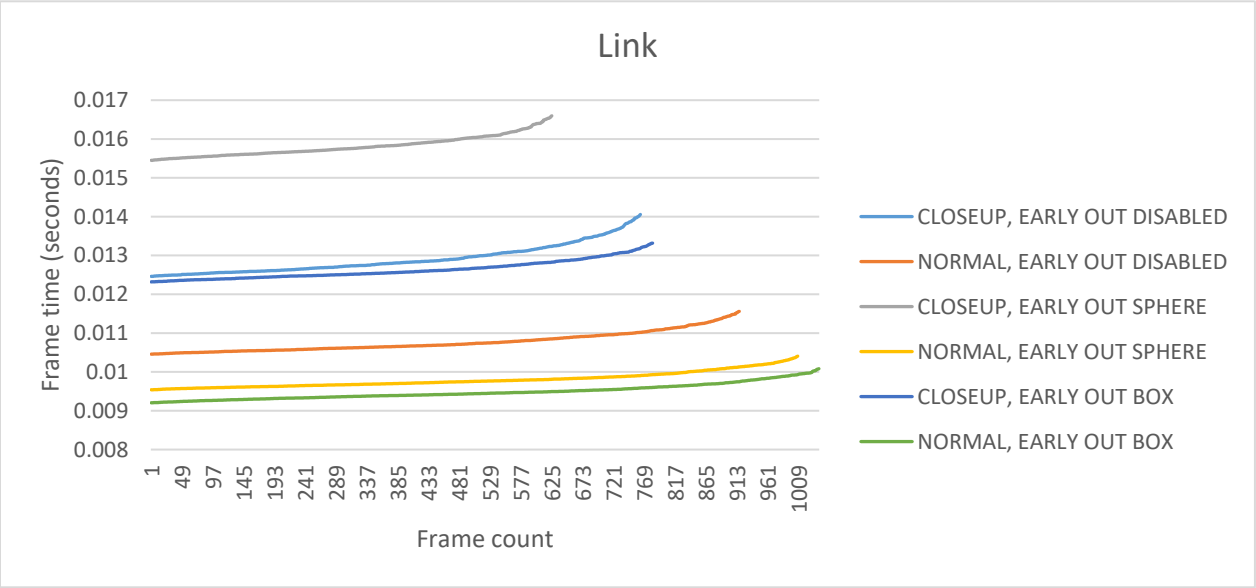
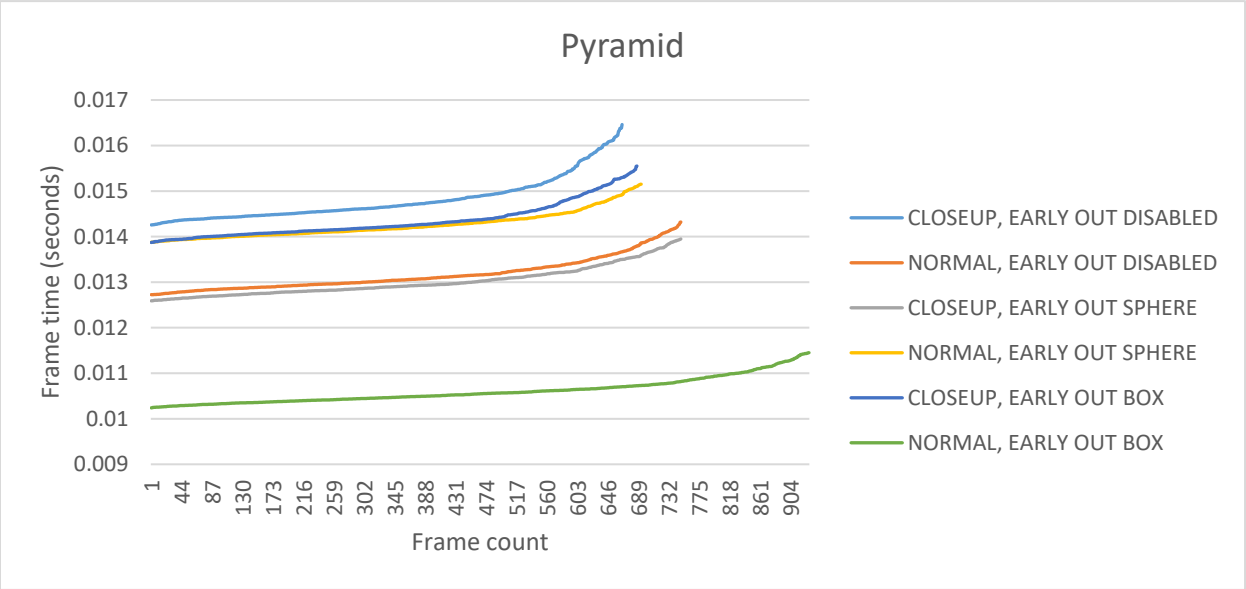


Figure 11: Normal shot link SDF in experiment

As can be expected, enabling early out when the camera is at the normal shot distance always decreases the frame time.
When the camera is at close-up distance enabling the sphere early out increases the frame time. This is due to the early out sphere taking up a considerable part of the screen when in close-up, and the sphere enclosing the pyramid has too much empty white space.
When early out is enabled, the frame time of the box is faster than the sphere.
Performance for the link benefits from a box early out in any scenario, but not from the sphere in close-up.

6.2.5.2 PYRAMID



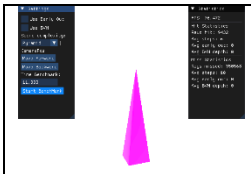
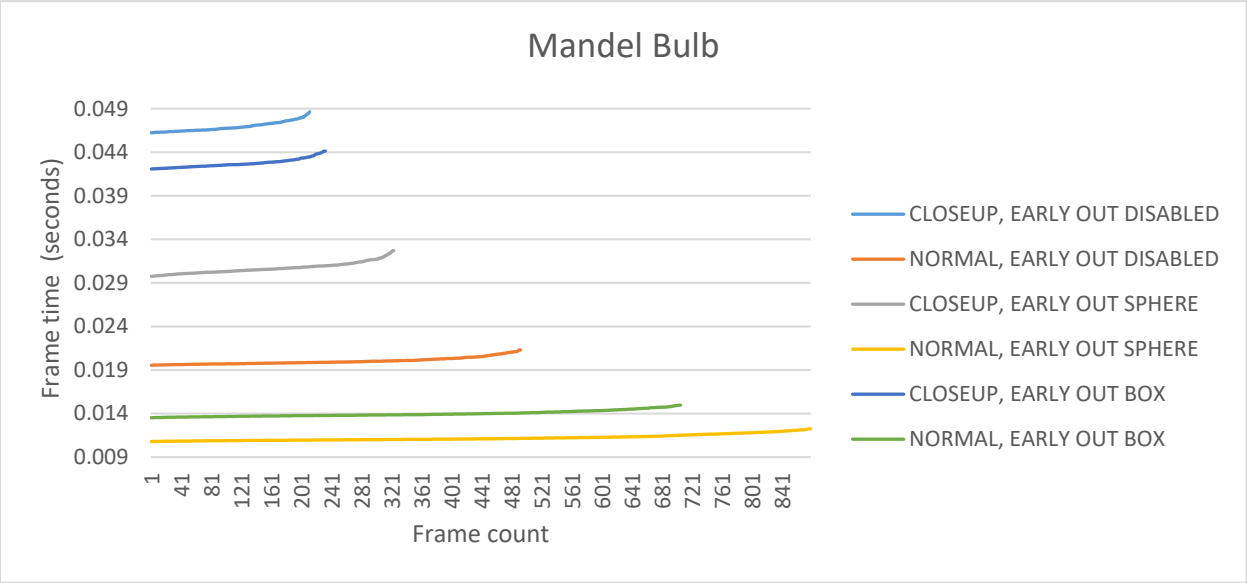


Figure 12: Normal shot pyramid SDF in experiment

The frame time for when the early out sphere is enabled in a close-up is faster than when the same is executed in the normal shot. This is due to the early out sphere taking up a considerable part of the screen when in close-up, and the sphere enclosing the pyramid has too much empty white space. The box reduces this slightly.
Early out disabled in a normal shot is faster than the sphere early out in the same shot. Again, the bounding sphere leaves too much empty space, forcing both the sphere and the pyramid SDF to be computed too often for missing rays.
When early out is enabled, the frame time of the box is always faster than the sphere. Performance for the pyramid benefits most from a box early out and actually suffers from a sphere early out.

6.2.5.3 MANDELBULB



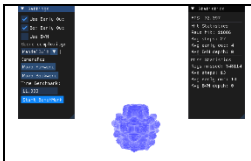


Figure 13: Normal shot Mandel Bulb SDF in experiment

As can be expected, the frame time of close-up shots is higher than normal shots.

In any case, applying any early out is faster than applying none. This is due to the super high complexity of the computation for this Mandel Bulb SDF.

The sphere as early out SDF performs better than the box early out, due to the shape of the Mandel Bulb itself approaching a more spherical shape.

6.3 BVHTREE FOR SDFS

6.3.1 BVHTREE FOR SDFS ARE SCENE WIDE

In the theoretical framework the construction for BVH-tree was discussed for triangle meshes. However, subdividing our SDF's is simply not possible. Instead, the BVH-tree must be created and represents the entire scene.

6.3.2 CONSTRUCTING THE BVHTREE

6.3.2.1 A BVH-NODE

The BVH-tree exists of nodes. Such a node has the following properties.

1. An origin
2. A radius or a box extent
3. A left child node
4. A right child node
5. An SDF
6.

There are 2 types of nodes:

1. Parent nodes: they act as early outs for multiple SDFs/objects, with an origin and radius/box extent around multiple objects. Only if the point is inside the parents' boundaries should it evaluate its children, which are either also parent nodes or leaf nodes. Parent nodes don't contain an SDF object. In Figure 14 they are visualized by the full outline.
2. Leaf nodes: they are the final nodes of the tree and only contain an SDF object. Their boundaries are first an early out and then the SDFs. In Figure 14 they are visualized by the dotted outline.

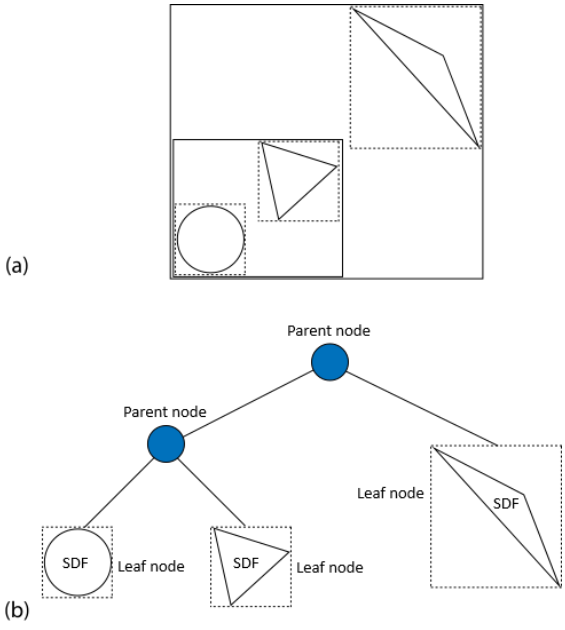


Figure 14: Example of BVHTree (*Bounding Volume Hierarchies*, n.d.)

6.3.2.2 DETERMINING THE NODES

To determine the nodes, the following steps are executed started at the root node. All objects in the scene form the initial group of objects presented.

1. All the origins from the group of objects are accumulated and their average is taken as origin for the bounding volume of that node, and it builds a sphere or box around them using the minimum and maximum extent that can be found for any object.
2. Then it splits the objects into two groups using the SAH (5.1.2.2.1).
3. Each split group is then presented to its left and right child node respectively and they repeat step 1, 2 and 3 until only one SDF/object remains in the object group.

When only one SDF remains, it will execute step one, to construct its early out and store the SDF.

The pseudo code

```
Create BVH node pass in the objects
      origin = sum of all objects
      radius = maximum of any object // or extent = minimum and maximum of any object

      if 1 object in objects // leaf node
                SDF = object
                return and end the recursion for this branch
      else
                Split objects into objects1 and objects2 //using the SAH (5.1.2.2.1)
                          Left node = Create BVH node pass in objects1
                          Right node = Create BVH node pass in objects2
```

6.3.3 USING THE BVHTREE

In the regular approach the ray marching algorithm loops over all objects for every ray at every step the ray takes. With the BVHTree in the scene, we just ask the root object for its distance to the point.

The pseudo code

```
float GetDistanceToScene(glm::vec3 const& point) const
{
      return BVHRoot->GetDistance(point);
}
```

The implementation for get distance is quite different from the SDFs get distance (5.2.1.1). And it does more than evaluate one distance method. If needed the method checks the distance to all its children.

The pseudo code

```
1. float BVHNode::GetDistance(glm::vec3 const& point) const
2. {
3.       if (ObjectSDF) // leaf node -> return SDF distance
4.       {
5.                return ObjectSDF->GetDistance(point - ObjectSDF->Origin());
6.       }
7.       float const distanceToBoundingVolume{ GetDistanceToBoundingVolume(point) };
8.       if (distanceToBoundingVolume > 0.001f) //outside the bounding volume SDF
9.       {
10.               return distanceToBoundingVolume;
11.      }
12.      //we are in the bounding volume SDF check the children
13.      float const leftResult{ LeftNode->GetDistance(point) };
14.      float const rightResult{ RightNode->GetDistance(point) };
15.      if (leftResult < rightResult)
16.      {
17.               return leftResult;
18.      }
19.      return rightResult;
20. }
```

This function will execute recursively until it has either reached the SDF / leaf node traversing one entire branch of the tree. Or it will return the distance to one of its child nodes bounding volumes as it is not necessary to evaluate the leaf node/SDF.

6.3.4 TESTING

The test compares 3 different scenes, in each scene the complexity of the objects increases, the displacement between objects also increases.

For every scene the test compared performance without, with a sphere and with a box BVHTree.

For each usage of BVHTree, the camera position was also changed. Two camera positions were used. A close-up and a normal shot. The window size for the test was 600x600, in total there are 360 000 pixels total.

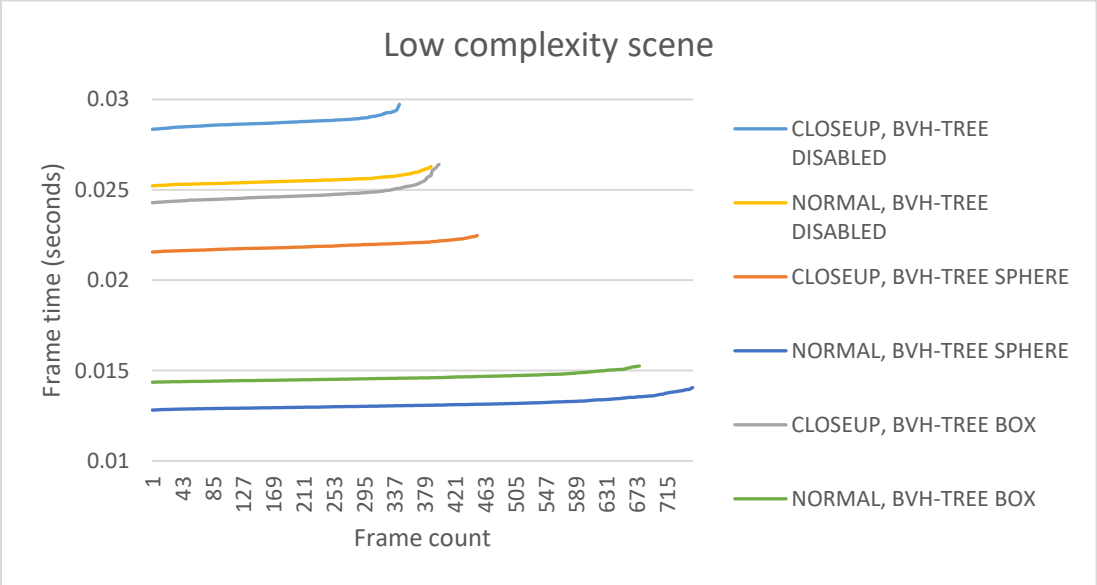
1. The close-up shot has a hit ratio of approximately 16% of the rays hitting an object and 84% misses all objects. For the pixel count in the experiment this means 60 000 rays did hit and 360 000 rays that didn't hit.
2. The normal shot has a hit ratio of approximately 8% of the rays hitting an object and 92% missing all objects. For the pixel count in the experiment this means 30 000 rays did hit and 330 000 rays didn't hit.

The field of view of the camera is set at 90 degrees for all shots.

For each possible combination the frame time was captured for all the frames rendered within an 11 second window. The results were sorted and the top 5% and bottom 5% of all the fame times were removed. Resulting in approximately 10 seconds of frame times captured.

6.3.5 RESULTS

6.3.5.1 LOW COMPLEXITY SCENE



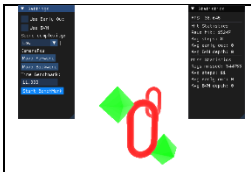
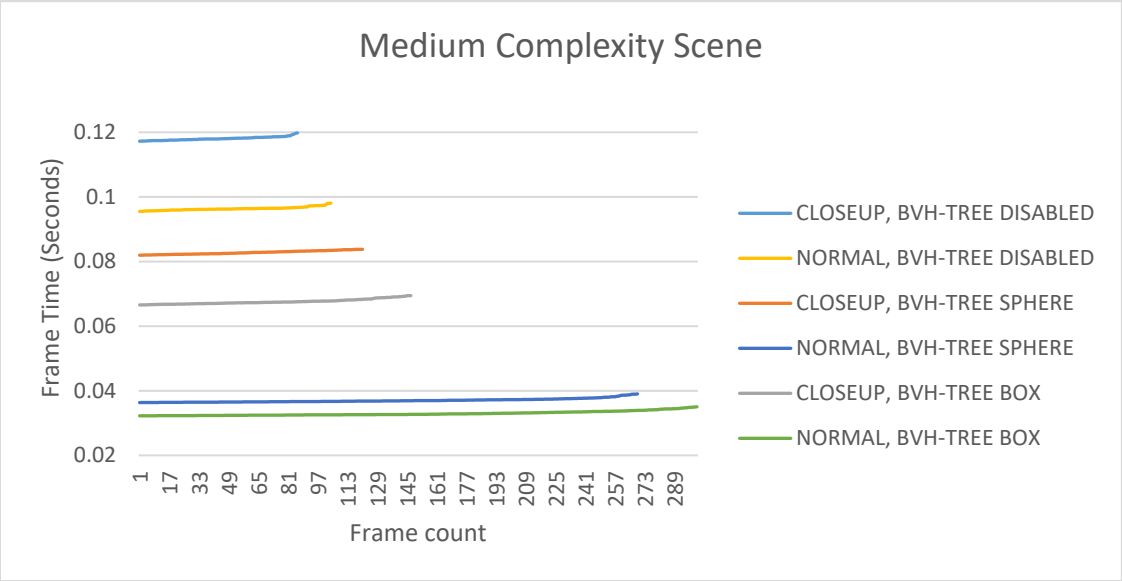


Figure 15: Low complexity scene

All close-up shot frame rates are slower than their respective normal frame rate.
In this case the sphere shaped BVH outperforms the box shaped BVH. This is due to the SDFs being quite close together.
There isn't that much empty space inside each node sphere bounding volume.

The SDFs themselves have low computation times. So, even if the sphere bounding volume has a considerable amount of extra space, the extra calculation underneath is still cheap to compute.

6.3.5.2 MEDIUM COMPLEXITY SCENE



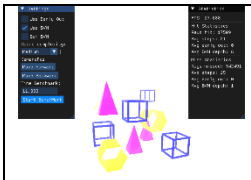
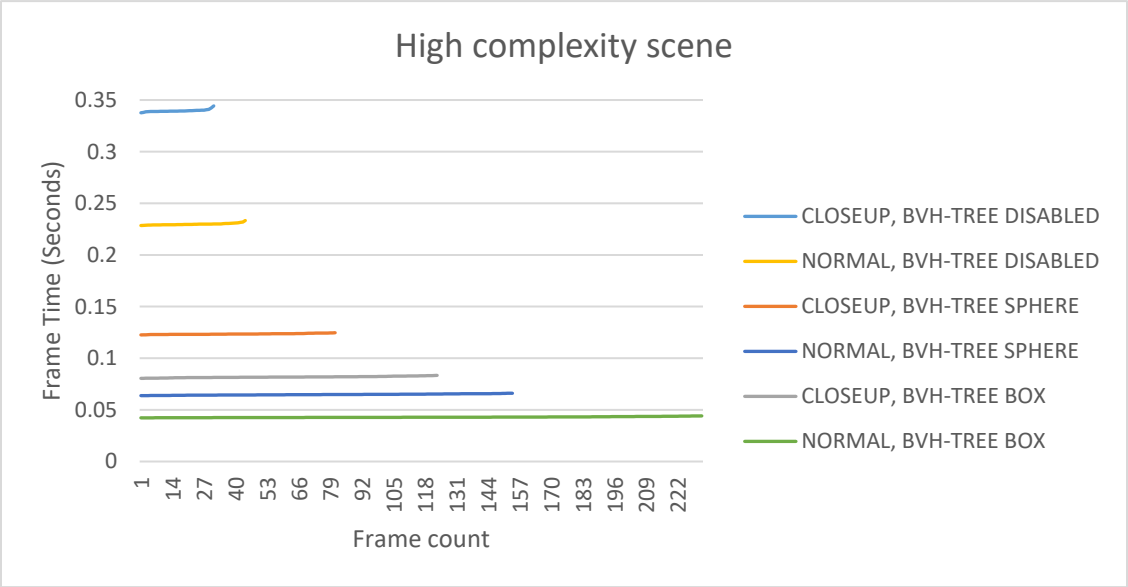


Figure 16: Medium complexity scene

All close-up frame rates are slower than their respective normal frame rate.
In both cases the box outperforms the sphere shaped BVHTree.
The accuracy of the box outweighs the lower computation time of the sphere.
SDFs are spaced more apart meaning parent nodes (using sphere boundaries) would overcompensate and encompass too much area.
SDFs are heavier to compute in this scene, so having more accuracy to not have to compute them is more efficient.

6.3.5.3 HIGH COMPLEXITY SCENE



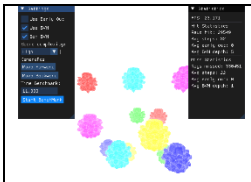


Figure 17: High complexity scene

All close-up frame rates are slower than their respective normal frame rate.
SDFs are spaced apart the most in this scene, thus the scene benefits from the box approach most as this does not use a singular value for all node boundaries, as is the case for the sphere).
The SDFs themselves are highly complex. Therefore, minimizing the need for these calculations through the box BVHTree for improved precision proves to be significantly more advantageous.

7 DISCUSSION

A few observations can be made from the measurements for the early out tests. In almost all cases, the performance has improved. In some cases, however, not implementing any early out was the most effective. Deciding if an early out should be used and what the best shape is for the early out depends heavily on the SDF it encloses and the distance to the camera.

As for the BVHTree, the sphere-shaped implementation proved more performant in simpler scenes. As the scenes become more complex and larger, the sphere-shaped implementation resulted in an excessive amount of empty space, reducing its efficiency. Using the box-shaped for higher complexity scenes implementation provides better detail and encloses the underlying shapes tighter.

Considering the frame time of the application exceeds the minimum value 0.0416s (24FPS) in the simplest of scenes, the experiment is not a real time application. While the CPU approach was helpful for quick development, a GPU-based approach should be tested.

8 CONCLUSION & FUTURE WORK

Although the experiment was run in a custom framework and the implementations are not perfect, the results show a decrease in frame time in most scenarios using the acceleration structures.

Overall, the experiment yielded the expected results and the encompassing research question: “How will applying mainstream ray tracing optimization techniques to sphere tracing affect performance for the sphere tracing rendering method?”, was answered by the hypothesis: “The sphere tracing rendering method should benefit in performance from mainstream optimization techniques used in ray tracing.”.

For future work it needs to be considered that this application / experiment ran purely on the CPU. To test if this is feasible for real-time applications this should be implemented on the GPU. All construction of the acceleration structures happened at the start of the application / experiment. It still must be tested if they are reusable at run-time.

9 CRITICAL REFLECTION

This project has been a huge learning experience. I've never really dug into something this deeply before, and I had a lot of fun doing so, especially when it came to the programming side. I really enjoyed getting my hands dirty with code and figuring things out. It felt good to see the theory work in practice.

One thing I got better at was using academic papers as resources. I learned how to pull out useful info from other studies and apply it to my own work. Connecting the theory to real-world stuff became quite enjoyable.

What I'm taking away from this is that you can't just rely on your strengths, you've got to keep pushing through the hard parts too. I'm proud of my work, and I feel like I've learned plenty that I can take with me moving forward in my career.

10 REFERENCES

Bounding volume. (2024). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Bounding_volume&oldid=1226824951

Bounding Volume Hierarchies. (n.d.). Retrieved January 7, 2025, from https://pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies

Claybook. (n.d.). Retrieved January 12, 2025, from https://claybookgame.com/

CSSE451 Advanced Computer Graphics. (n.d.). Retrieved December 29, 2024, from https://www.rose-hulman.edu/class/cs/csse451/AABB/

Demoscene. (2019). In *Wikipedia*. https://nl.wikipedia.org/w/index.php?title=Demoscene&oldid=53157976

Devred, M. (2022). *Rendering 3D cross-sections of 4D Fractals*. https://allpurposem.at/paper/2022/fractals.pdf

Donnelly, W. (2005, April). *Per-Pixel Displacement Mapping with Distance Functions*. NVIDIA Developer. https://developer.nvidia.com/gpugems/gpugems2/part-i-geometric-complexity/chapter-8-pixel-displacement-mapping-distance-functions

Goren, O. (2021, December 7). TREES DATA STRUCTURE. *Medium*. https://medium.com/@omergrn25/trees-data-structure-1c566db3b726

Hart, J. C. (1996). *Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces*. http://link.springer.com/10.1007/s003710050084

Introduction to Acceleration Structures. (n.d.). Retrieved January 10, 2025, from https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part1.html

Kay, T. L., & Kajiya, J. T. (1986). Ray tracing complex scenes. *ACM SIGGRAPH Computer Graphics*, *20*(4), 269–278. https://doi.org/10.1145/15886.15916

MacDonald, J. D., & Booth, K. S. (1990). Heuristics for ray tracing using space subdivision. *The Visual Computer*, *6*(3), 153–166. https://doi.org/10.1007/BF01911006

Papaioannou, G., Menexi, M., & Papadopoulos, C. (2010). *Real-Time Volume-Based Ambient Occlusion*. https://www.researchgate.net/publication/45113929_Real-Time_Volume-Based_Ambient_Occlusion

Quilez, I. (n.d.). *Inigo Quilez*. Retrieved December 29, 2024, from https://iquilezles.org

Ray marching. (2024). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Ray_marching&oldid=1265162208

Ray Tracing | NVIDIA Developer. (n.d.). Retrieved December 29, 2024, from https://developer.nvidia.com/discover/ray-tracing

Schneider, A. (2024, July 25). *Synthesizing Realistic Clouds for Video Games*. https://www.guerrilla-games.com/read/synthesizing-realistic-clouds-for-video-games

Sebastian Lague (Director). (2024, June 12). *Coding Adventure: Optimizing a Ray Tracer (by building a BVH)* [Video recording]. https://www.youtube.com/watch?v=C1H4zIiCOaI

Shadertoy. (n.d.). Retrieved January 12, 2025, from https://www.shadertoy.com/

Tomczak, L. J. (2012). *GPU Ray Marching of Distance Fields* [Technical University of Denmark]. https://www2.imm.dtu.dk/pubdb/edoc/imm6392.pdf

Triangle mesh. (2024). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Triangle_mesh&oldid=1234853041

Unreal Engine 5. (n.d.-a). *Distance Field Ambient Occlusion in Unreal Engine | Unreal Engine 5.5 Documentation*. Epic Games Developer. Retrieved October 19, 2024, from https://dev.epicgames.com/documentation/en-us/unreal-engine/distance-field-ambient-occlusion-in-unreal-engine

Unreal Engine 5. (n.d.-b). *Mesh Distance Fields in Unreal Engine | Unreal Engine 5.5 Documentation*. Epic Games Developer. Retrieved October 19, 2024, from https://dev.epicgames.com/documentation/en-us/unreal-engine/mesh-distance-fields-in-unreal-engine

Wyvill, B., & Wyvill, G. (1989). *The Visual Computer*. SpringerLink. https://link.springer.com/journal/371

11 ACKNOWLEDGEMENTS

I would like to express my gratitude to certain individuals whose contribution was invaluable to writing this paper.

Firstly, I would like to thank my supervisor Koen Samyn, who helped me conduct the research for this paper and provide feedback along the process. I would also like to thank my coach Kasper Geeroms, for providing feedback on the reflection report and making sure I am on track.

My heartfelt thanks to my friends and family for their unwavering support during the process of writing this paper. In particular my dad for being my “rubber duck” at the dinner table, when I was implementing the experiment and ran into roadblocks. A big thanks to Stan Dirix as well for helping me out with the technical details of formatting the paper and the results of the experiment.

Additionally, a huge thanks to Stan Dirix, Arne Olemans and my mom giving me insights by proofreading the paper.

My deepest gratitude to everyone who contributed, either directly or indirectly, to the completion of this paper. Your support and encouragement have been amazing.

12 APPENDICES

To consult any additional resources used to conduct the research for this paper please refer to https://github.com/AdriaanMusschoot/GraduationWork it includes the framework, executables, source code, images, results, etc.