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

English:

Sphere tracing, presented by Hart (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 2 rendering techniques is calculating the distance to objects in the scene to determine 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.

To test if the acceleration structures improved performance, 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.

Testing was limited to static scenes and used structures constructed at the start of the application. Testing for dynamic scenes and reconstructions is out of the scope of this paper.

Dutch:

Sphere tracing, voorgesteld door Hart (Hart, 1996), is een subtak van ray marching, een realtime renderingtechniek. Het heeft veel overeenkomsten met ray tracing en wordt vaak beschouwd als zijn kleine broertje. Een kritisch aspect voor deze 2 renderingtechnieken is het berekenen van de afstand tot objecten in de scène om te bepalen of deze objecten worden geraakt door de stralen die door het renderingalgoritme worden uitgeworpen. Objecten in een sphere tracing context worden voorgesteld door signed distance fields (SDF's).

In ray tracing zijn er veel versnellingsstructuren om dit proces te versnellen. In deze paper worden de Axis-Aligned Bounding Box (AABB) en Bounding Volume Hierarchy Tree (BVHTree), die vaak worden gebruikt in ray tracing, beschouwd en geïmplementeerd voor SDF's in een sphere tracing context. Voor elke versnellingsstructuur werden twee soorten vormen overwogen en geïmplementeerd, een balk en een doos.

Om te testen of de versnellingsstructuren de prestaties verbeterden, werd elke implementatie vergeleken wanneer ofwel geen versnellingsstructuur, de bolimplementatie of de balkimplementatie actief was. In de meeste gevallen leverde het experiment betere prestaties op wanneer de versnellingsstructuren werden toegepast.

Het testen was beperkt tot statische scènes en gebruikte structuren die aan het begin van de toepassing werden geconstrueerd. Testen voor dynamische scènes en reconstructies vallen buiten het bestek van deze paper.

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 have been evolving their computational/ALUs abilities faster than their memory bandwidth. This meant purely mathematical SDFs became competitive against their 3D-texture-based SDF counterpart. (Quilez, n.d.-b)

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 the following:

1. Unreal Engine 5 uses ray marching for ambient occlusion and distance field shadows. (Unreal Engine 5, n.d.-b, n.d.-a)
2. The game “Claybook” (*Claybook*, n.d.) 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 for teaching and creating demo scenes and exclusively uses the ray marching algorithm.
4. The “Demoscene” community specializes in creating non-interactive audio-visual presentations executed in real time on computers. (“Demoscene,” 2019) They have adopted “Shadertoy” and SDFs in general to create demoscenes.

When looking into “Shadertoy” creations and creations by the “Demoscene” community, performance doesn’t seem to have been a central consideration. Priority was always given to aesthetically beautiful scenes on the screen. Higher complexity and larger scenes require more performance to keep performing in a real-time environment.

This paper discusses the axis-aligned bounding box and the bounding volume hierarchy tree. Both acceleration structures are used in ray tracing. In this paper, those same structures for sphere tracing are studied and implemented.

The goal is to compare the performance of each acceleration structure.

The encompassing research question is 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 taken into consideration.

1. How does applying AABB to SDFs in sphere tracing affect performance?
2. How does applying a BVHTree to SDFs in sphere tracing affect performance?
3. How does AABB performance compare with sphere-shaped or box-shaped implementations?
4. How does BVHTree performance compare with sphere-shaped or box-shaped implementations?

The encompassing hypothesis is that the sphere tracing rendering method should benefit in performance from mainstream optimization techniques used in ray tracing. The following sub-hypotheses 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 BVHTree 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 BVHTrees 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 today's computer graphics, 3D objects need to be visualized. The most popular approach is triangle meshes. These consist of triangles (Figure 1), each triangle consists of 3 vertices and 3 edges connecting these vertices ("Triangle Mesh," 2024). Multiple algorithms exist to convert triangle meshes from 3D space to a 2D screen.

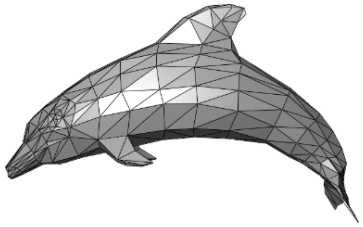


Figure 1: Example of a triangle mesh

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.)

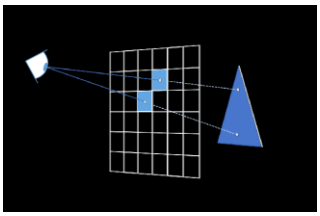


Figure 2: The process of ray casting and shading pixels

5.1.2 ACCELERATION STRUCTURES

The main purpose of acceleration structures is to speed up the process of ray casting. The acceleration structures 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:

  Performance drops 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 it still has to traverse all meshes trying to intersect with all their triangles. Creating a box around each triangle mesh with which the ray intersects, determining which meshes the ray is most likely to intersect with, this preserves performance.

- Scenario 2:

  Performance is also threatened when the triangle meshes in the scene are very complex, consisting of a high number of triangles. Every triangle is then intersected with the ray, only for the algorithm to return one triangle it intersected with. Grouping triangles and estimating which are most likely for the ray to intersect with will reduce the number of triangle intersections, thus improving performance.

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.

A simple box encloses the region that contains a triangle mesh. The edges of the box align 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 first calculates ray intersections using the AABB of a mesh. Only if an intersection with the AABB is found, it calculates further intersections with the triangles from the mesh itself. The method speeds up ray intersection calculations because it reduces the number of calculations that is executed (early out when the ray doesn't intersect with the AABB).

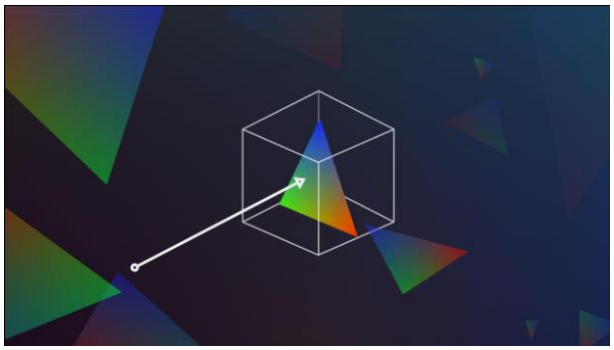


Figure 3: AABB for a single triangle

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 uses time and computation power. To reduce this cost, the BVHTree divides the initial AABB into smaller AABBs. This process repeats until one smaller box contains the preset 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.

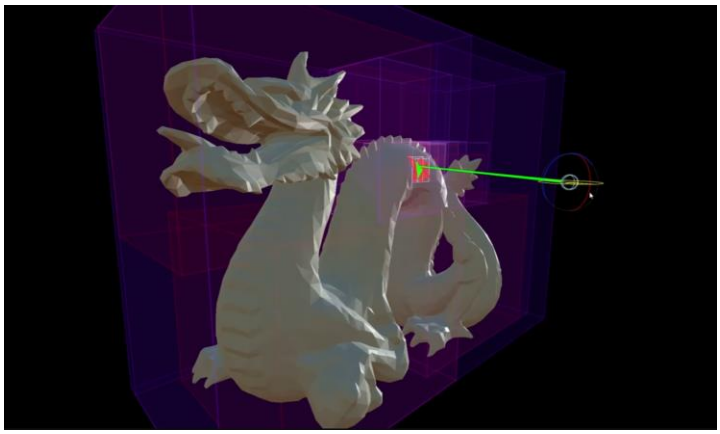


Figure 4: Complex mesh with BVH applied (Sebastian Lague, 2024)

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.

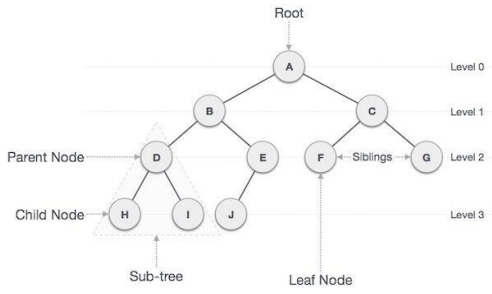


Figure 5: Tree structure for BVH (Goren, 2021)

This structure can be applied on scene level. Instead of subdividing the triangles in one triangle mesh 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 steps 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 Pseudo code SAH evaluation)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.

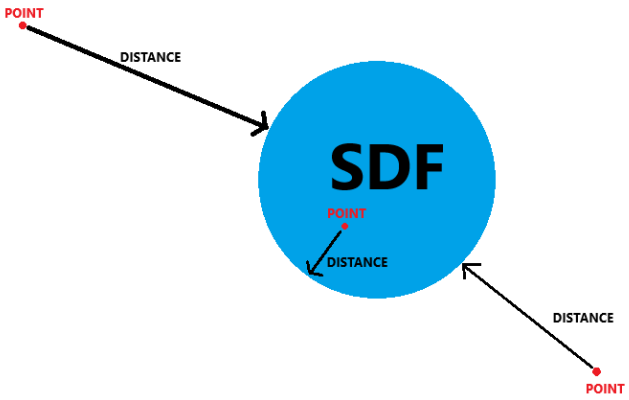


Figure 6: SDF Visualization

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 - origin) – 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. If the returned value is exactly 0, the point lays on the surface of the sphere.

Without an origin the code assumes the sphere is at position (0, 0, 0). To have translations, rotations or scaling, the point must be manipulated before calculating the distance (Quilez, n.d.-b).

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)

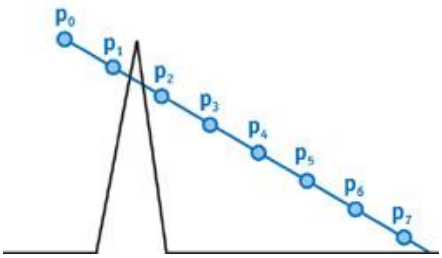


Figure 7: Ray marching missing a surface (Donnelly, 2005)

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)

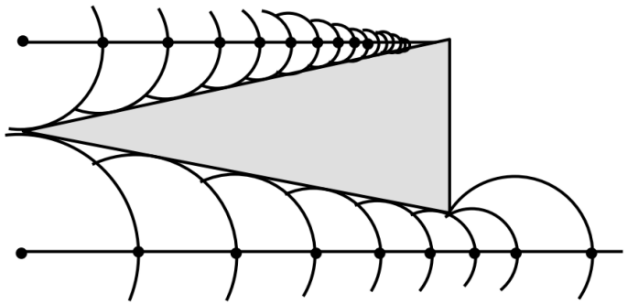


Figure 8: Example of both scenarios when raymarching (Hart, 1996)

While this approach is more memory friendly than using triangle meshes, it does mean the entire scene needs to be reconstructed at every step of the ray. Similarly to triangle meshes, as SDFs become more complex they will require more computation time.

The main bottleneck for 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, resulting in the ray marching algorithm evaluating this function 100 000 times for 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. This is the language I have familiarized myself with the most.
2. Debugging a CPU is easier than a GPU.
3. C++ has some nice functionality such as pointers and recursion that are not available in shader code.
4. When compared to GPU development it requires considerably less set up.
5. Allows measuring performance and tracking other statistics easily 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, while “Shadertoy” would allow for partial implementation.

Before exploring this topic, I already created my own implementation of the ray-tracing algorithm in a custom framework. As discussed earlier, the ray marching algorithm has some traits similar to the ray tracing algorithm. So, my previous work provided 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 USING AABBS FOR SDFS

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 has been 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

In sphere tracing rays cannot intersect with a box or sphere. 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 early out in the form of an SDF is necessary, for the underlying more complex SDF.

The pseudo code for getting the distance with an additional early out

```
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 only calculate the early out distance, which is computationally less expensive than calculating the distance to the underlying shape. However, it also means that once the ray is marching inside the early out SDF, alongside the early out distance, the algorithm also calculates the distance to the underlying shape. But, the benefits of an early out hugely outweigh the cost of the 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), that point is considered to have reached the surface. It 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 to construct a sphere SDF.

Pseudo code for approaching the SDF boundaries

```
 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.

Pseudo code for calculating the radius of the sphere

```
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, 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

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.

Pseudo code for computing wall distance

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

When calculating the distance to the point, the component of the point which is also present in the axis should the only one taken into account. Otherwise, if the point has a value for all 3 components, the new distance value might be greater than the previous one. And the walls 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). When the distance to a point reaches threshold value, that point is considered to have reached the surface in that axis and direction.

Pseudo code for approaching SDF

```
while not all walls have reached the surface

for each direction

   if wall has reached the surface //in previous iteration
      do nothing for this direction
   else
      generate a wall of points using direction and new distance value //from last iteration
      for each point on the wall
         distance = distance to SDF

         if distance < 0.001
               store that point //wall has reached surface
         if distance < closest distance
               store distance and point //new minimum was found

      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 SDFs box extent. Which will be used to compute the early out box enclosing the SDF.

Pseudo code for calculating the box extent

```
for each minimum point of each wall
      if boxExtent.x < abs(point.x)
               boxExtent.x = abs(point.x);
      if boxExtent.y < abs(point.y)
               boxExtent.y = abs(point.y);
      if boxExtent.z < abs(point.z)
               boxExtent.z = abs(point.z);
```

6.2.3.2 FORMULA

Pseudo code 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. }
```

The box SDF is always a mirrored box. 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 uses 6 different objects of increasing complexity (higher computation time) in separate scenes. (Formula's from (Quilez, n.d.-a)):

- Link
- Octahedron
- Box Frame
- Hexagonal Prism
- Pyramid
- Mandel Bulb

Each object is compared in 3 test scenarios:

- Without an early out
- With a sphere early out
- With a box early out

For each test scenario two camera positions are used:

- 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.
- 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.

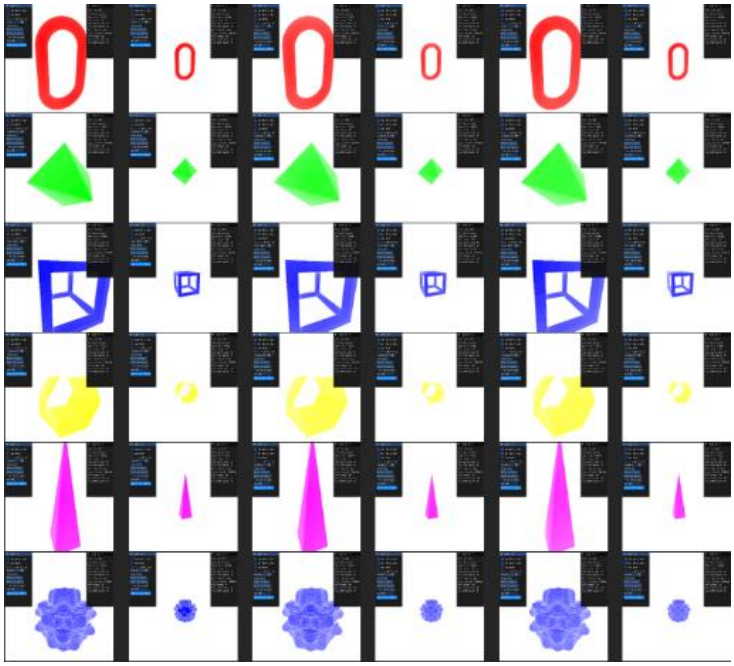


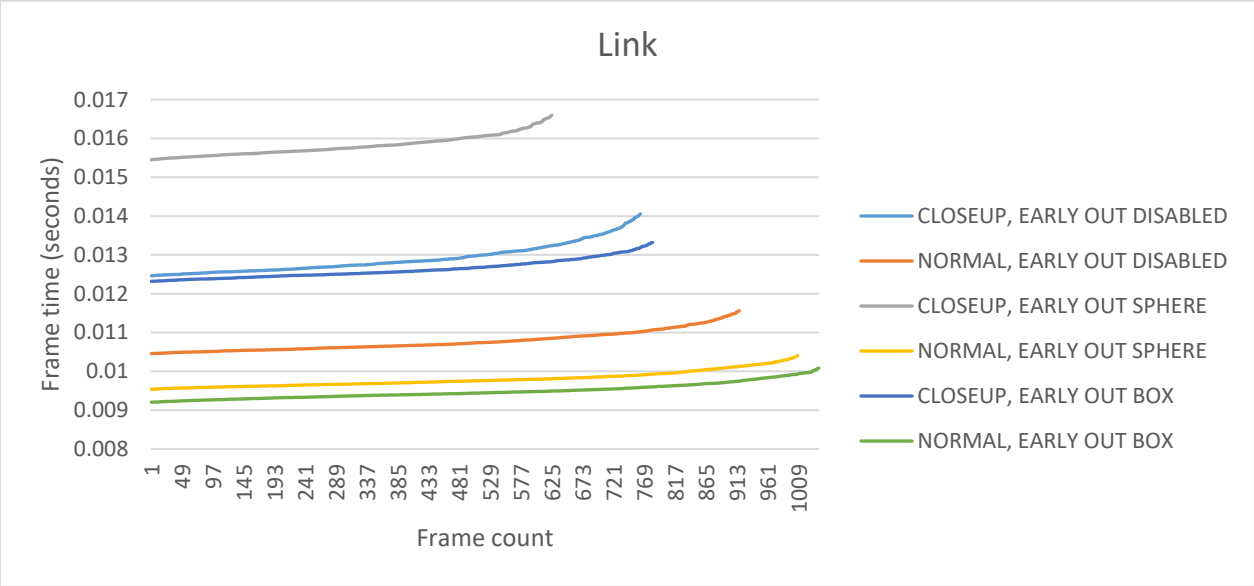
Figure 11: Every Scenario Evaluated in Testing

For each of the 36 resulting use cases, the computer captured the time it took to render a frame during an 11 second window. The frame time of each rendered frame was sorted. The top 5% and bottom 5% of outliers were removed. The approximately 10 seconds of remaining data was plotted in a line graph.

6.2.5 RESULTS

The test was conducted for 6 shapes. The link, the pyramid and the Mandel Bulb will be presented. All results can be found in the appendices.

6.2.5.1 LINK



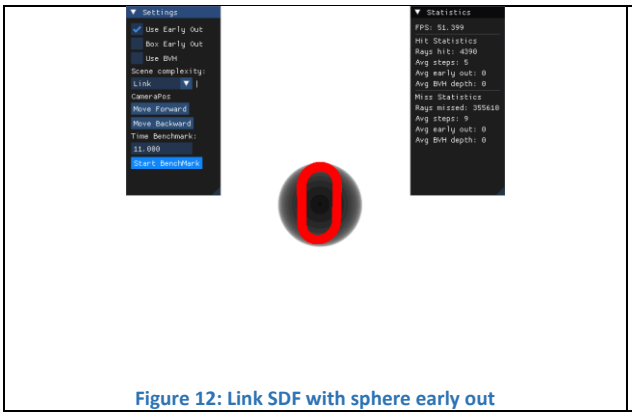


Figure 12: Link SDF with sphere early out

Figure 13: Link SDF with box early out

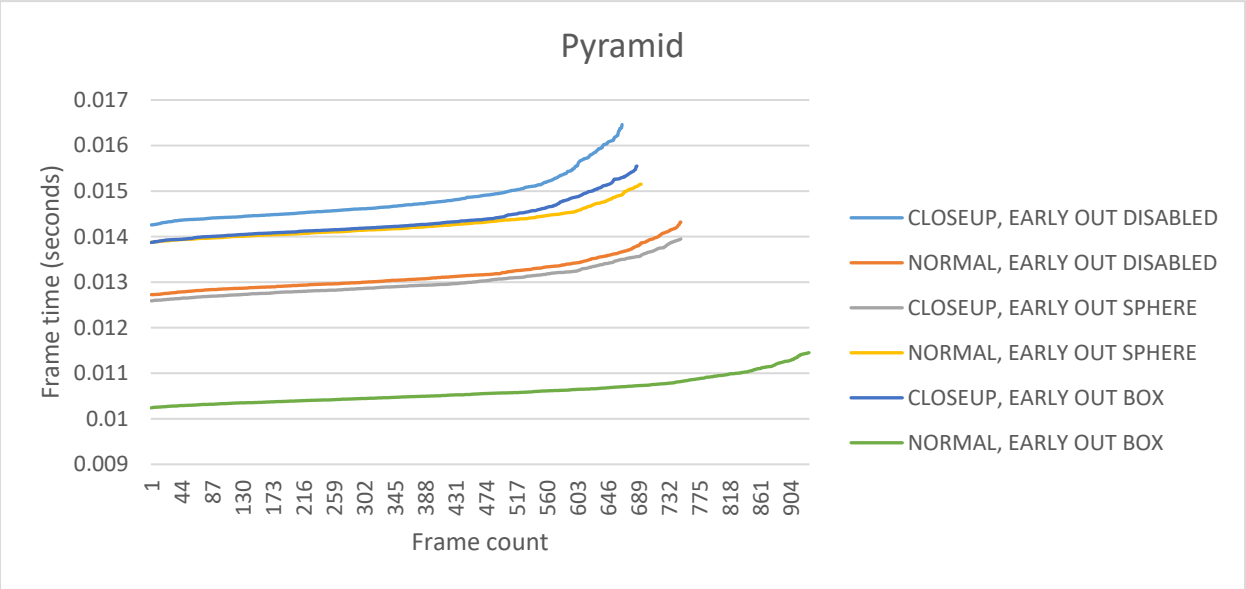
The hypothesis being that enabling early out would decrease frame time, is not confirmed in every scenario.

When the camera is at the normal shot distance, the frame time decreases as can be expected. The decrease is more significant with the box scenario as compared to the sphere scenario.

When the camera is at a close range, enabling the early out box only decreases frame time only slightly. And when enabling the early out sphere, a sharp rise in frame time is observed. The rise can be explained by the sphere taking up a considerable part of the screen and it having too much area not filled with the underlying shape (black area in Figure 12).

Conclusion: for a simple object, leaving a high amount of empty space in its enclosing SDF the early out box is the most cost-effective scenario.

6.2.5.2 PYRAMID



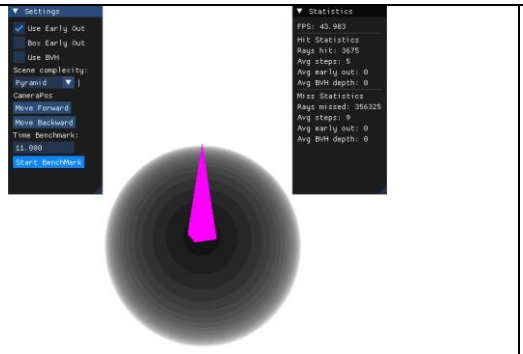


Figure 14: Pyramid SDF with sphere early out

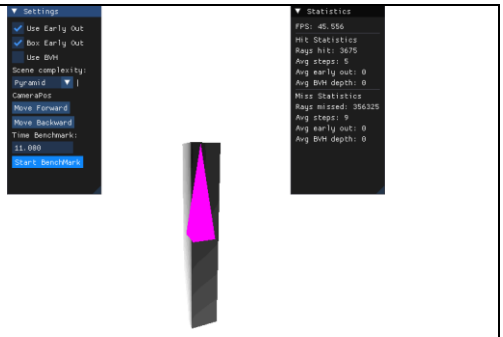


Figure 15: Pyramid SDF with box early out

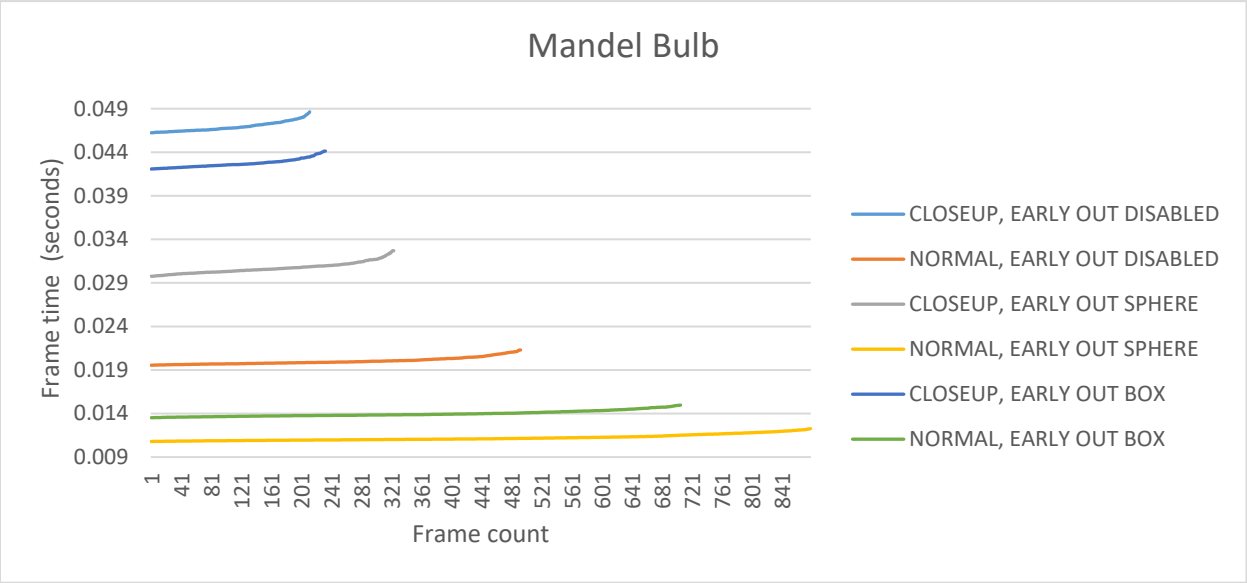
Also for this shape, the hypothesis that early out always reduces frame times is not always confirmed.

With the camera in close-up, enabling early out reduces frame time. The reduction is more significant in the sphere scenario as compared to the box-scenario.

When the camera is at a normal distance, the early out box reduces frame times and the early out sphere increases frame time.

Conclusion: the early out box scenario delivers the highest cost reduction for a normal camera standpoint. When working with close-ups, the largest cost reduction is linked with the sphere scenario.

6.2.5.3 MANDELBULB



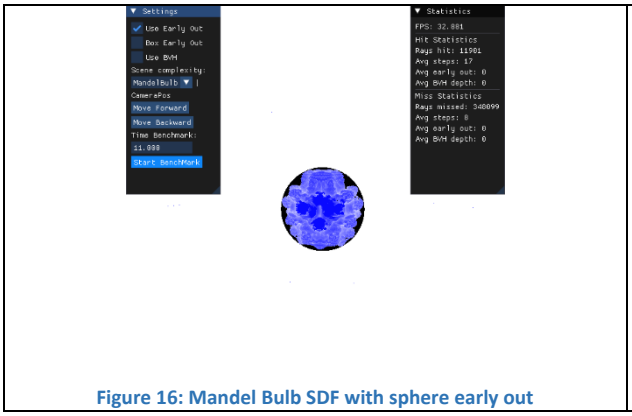


Figure 16: Mandel Bulb SDF with sphere early out

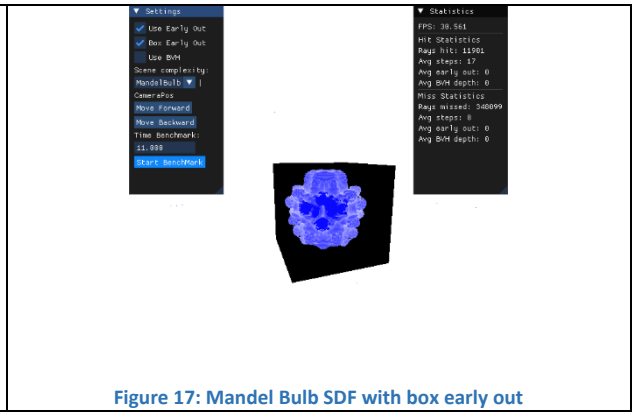


Figure 17: Mandel Bulb SDF with box early out

The number of frames captured decreases by two thirds when early out is disabled. This is expected, as the number of frames captured depends directly on the calculation time for one frame.

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

With both camera standpoints, applying an early out is faster than none. This is due to the super high complexity of the computation for this Mandel Bulb SDF.

Conclusion: The early out sphere scenario outperforms the early out box scenario. This is probably linked to the overall shape of the Mandel Bulb itself, that resembles the sphere early out shape.

6.3 BVHTREE FOR SDFS

6.3.1 BVHTREE FOR SDFS ARE SCENE WIDE

In the theoretical framework the construction for the BVHTree was discussed for triangle meshes (5.1.2.2). However, subdividing SDFs is not possible. Instead, the BVHTree represents the entire scene.

6.3.2 CONSTRUCTING THE BVHTREE

6.3.2.1 A BVH-NODE

The BVHTree 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

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. A parent node does not contain an SDF object. Only if the point is inside the parent nodes boundaries should it evaluate its child nodes. Child nodes are either also a parent node or a leaf node. Figure 18 visualizes the parent nodes with a full outline.
2. Leaf nodes: they are the final nodes of the tree and contain only one SDF object. Their boundaries are first an early out and then represent the SDF itself. Figure 18 visualizes leave nodes with a dotted outline.

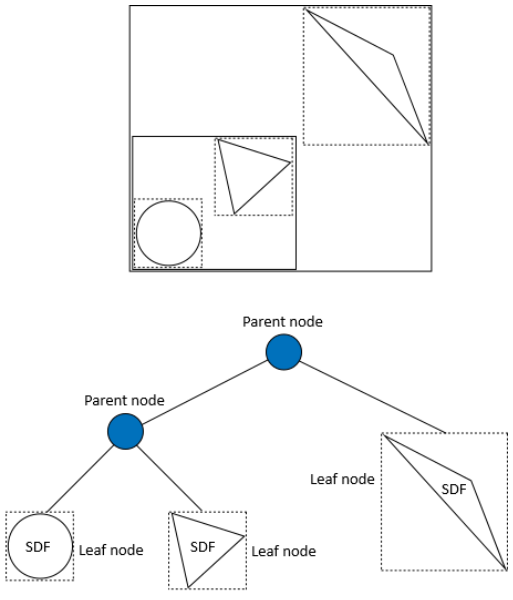


Figure 18: Example of BVHTree (*Bounding Volume Hierarchies*, n.d.)

6.3.2.2 DETERMINING THE NODES

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

1. The origin of every SDF in the group is accumulated. Their average is taken as the origin for the bounding volume of the node. The code builds a sphere or box around the group of objects using the minimum and maximum extent that can be found for any object.
2. Only when more than one SDF remains in the group of objects, it splits the objects into two groups using the SAH (5.1.2.2.1). Otherwise it jumps to step 4.
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.
4. When only one SDF remains, the early out is constructed and the SDF is stored.

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 a regular approach the ray marching algorithm will loop over all objects for every ray at every step the ray takes. When getting the distance to the scene with a BVHTree, the root node evaluates its distance to the point.

Pseudo code for getting the distance to the scene

```
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. Only when the ray hits a node object, the distance to all child nodes will be evaluated.

Pseudo code for evaluating the distance to the node

```
float BVHNode::GetDistance(glm::vec3 const& point) const
{
	if (ObjectSDF) // leaf node -> return SDF distance
	{
		return ObjectSDF->GetDistance(point – ObjectSDF->Origin());
	}
	float const distanceToBoundingVolume{ GetDistanceToBoundingVolume(point) };
	if (distanceToBoundingVolume > 0.001f) //outside the bounding volume SDF
	{
		return distanceToBoundingVolume;
	}
	//we are in the bounding volume SDF check the children
	float const leftResult{ LeftNode->GetDistance(point) };
	float const rightResult{ RightNode->GetDistance(point) };
	if (leftResult < rightResult)
	{
		return leftResult;
	}
	return rightResult;
}
```

This function will execute recursively. When it reaches the SDF / leaf node, traversing one entire branch of the tree, it will subsequently calculate the distance to the SDF. If the point is outside a certain node before reaching the SDF, it returns the distance to the bounding volume of that node.

6.3.4 TESTING

The test uses 3 different scenes of increasing complexity and displacement:

- Low
    a. 2 Links
    b. 2 Octahedrons
- Medium
    a. 4 Box Frames
    b. 3 Hexagonal Prisms
    c. 3 Pyramid
- High
    a. 12 Mandel Bulbs

Each scene is compared in 3 test scenarios:

- No BVHTree
- With a sphere BVHTree
- With a box BVHTree

For each test scenario two camera positions are used:

- The close-up shot has a hit ratio of approximately 16% of the rays hitting an object and 84% missing all objects. For the pixel count in the experiment this means 60 000 rays did hit and 360 000 rays that didn't hit.
- 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 330000 rays didn't hit.

Figure 19: Every Scenario Evaluated in Testing

For each of the 36 resulting use cases, the computer captured the time it took to render a frame during an 11 second window. The frame time of each rendered frame was sorted. The top 5% and bottom 5% of outliers were removed. The approximately 10 seconds of remaining data was plotted in a line graph.

6.3.5 RESULTS

6.3.5.1 LOW COMPLEXITY SCENE

Figure 20: Low complexity scene

As expected, all close-up shot frame rates take more time than normal shots.

With both camera standpoints, the sphere scenario outperforms the box scenario. The reason for this is the SDFs being quite close together, without much empty space inside each node sphere bounding volume.

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.

Conclusion: The sphere scenario results in the best performance.

6.3.5.2 MEDIUM COMPLEXITY SCENE

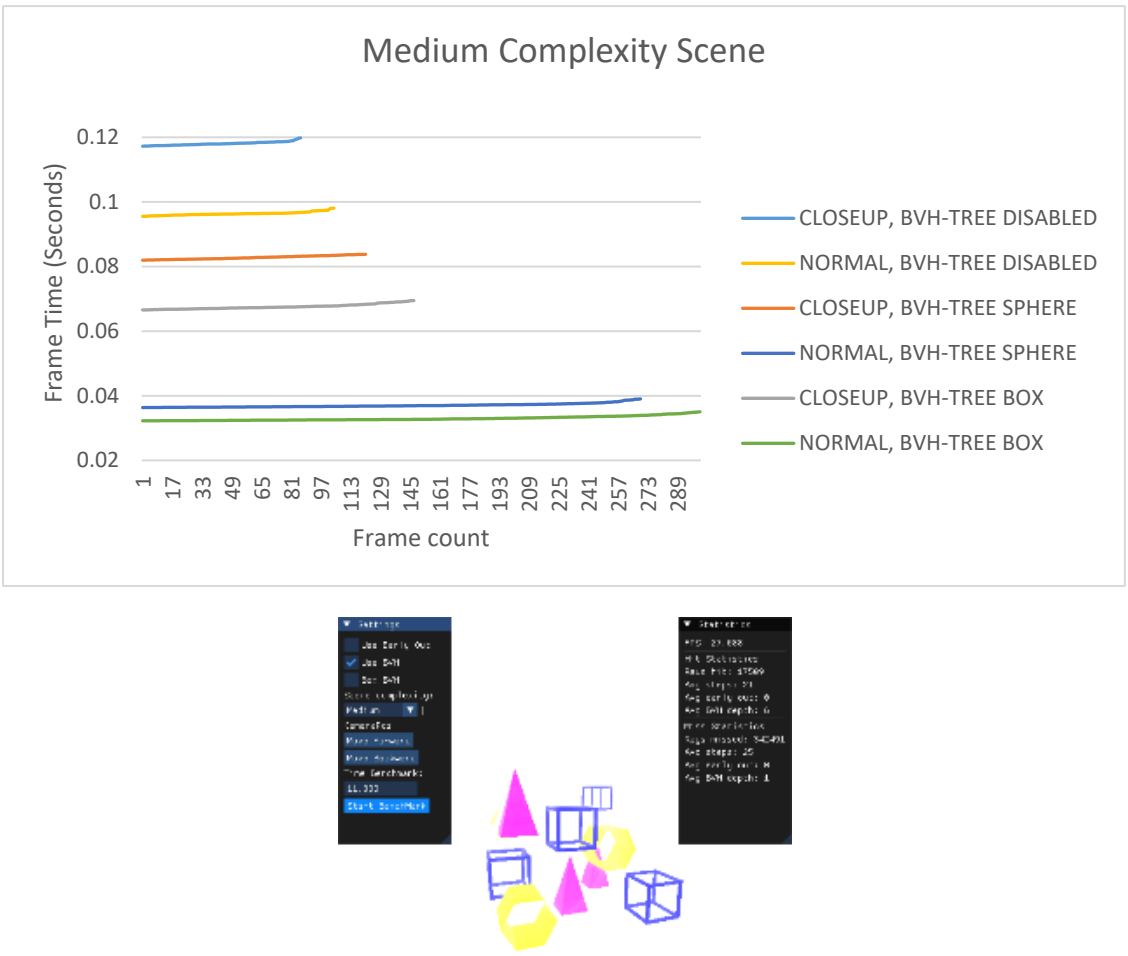


Figure 21: Medium complexity scene

All close-up frame rates are slower than the normal camera standpoint frame rates.

With both camera standpoints, the box scenario outperforms the sphere scenario in terms of frame-rate reduction. SDFs are spaced more apart leading to parent nodes (when using sphere boundaries) encompassing too much overlaying area.

In the normal camera-standpoint, reduction in time frame leads in both scenarios to doubling of the number of renders returned during the test-period. Although the close-up standpoint equally leads to almost halving the calculation time, the increase in number of rendered frames returned is less impressive.

Conclusion: Due to the heavier computation cost of the SDFs in this scene, the box scenario reduces frame rates the most.

6.3.5.3 HIGH COMPLEXITY SCENE

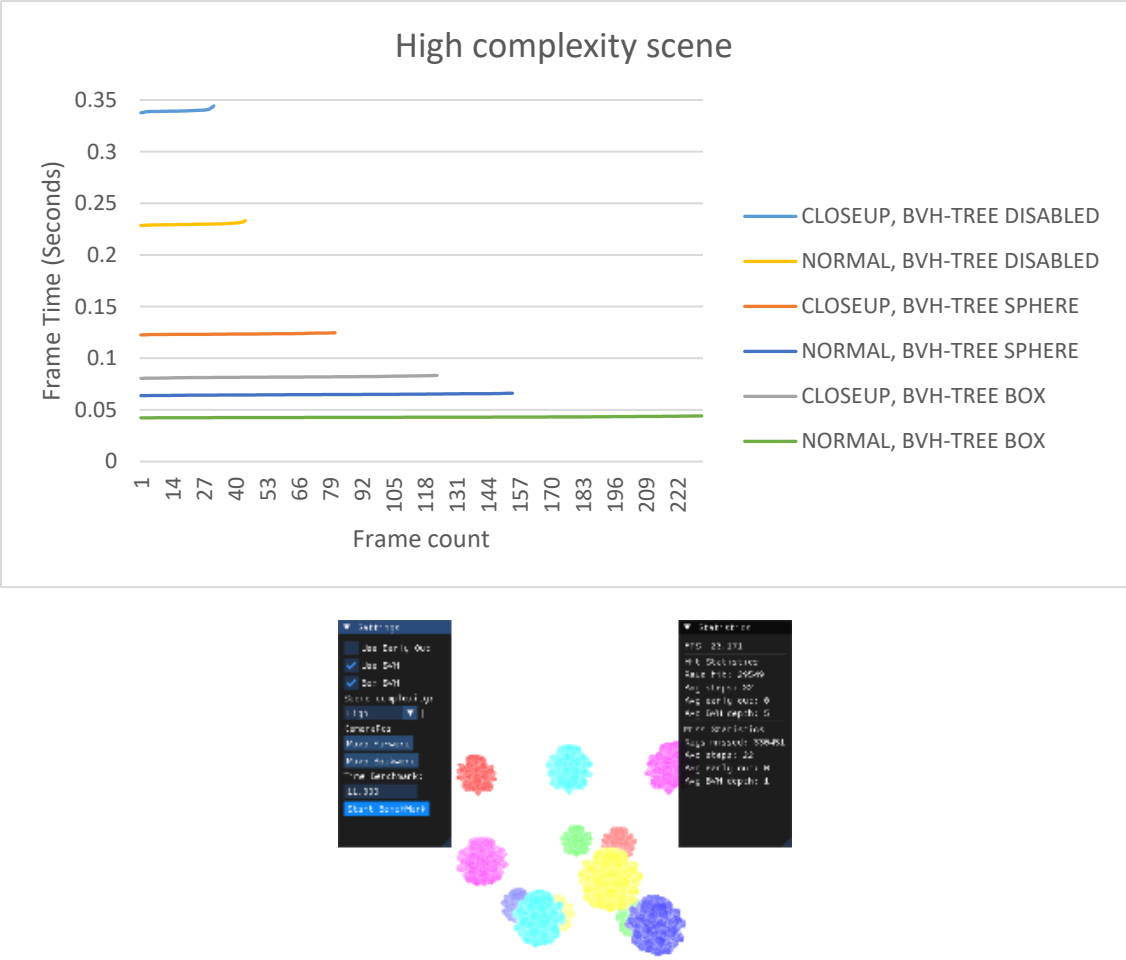


Figure 22: High complexity scene

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 markedly reduced more frame time.

Conclusion: With both camera standpoints the BVH-Tree box scenario reduces frame time more than the BVH-Tree Sphere scenario.

7 DISCUSSION

A few observations can be made from the measurements for the AABB tests (6.2.5). In almost all cases, the performance improves when implementing an early out. Exceptions are the close-up with early out sphere for the link and the normal early out sphere for the pyramid. 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.

When using a BVHTree (6.3.5), the sphere-shaped implementation proved more performant in simpler scenes. As scenes become larger and more complex, 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. Although I've never really dug into such a specific topic this deeply before, doing so was fun, especially when it came to the programming side. I really enjoyed experimenting 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 just relying on your strengths isn't enough, you have 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.-a). *Distance functions for primitives*. Retrieved October 19, 2024, from https://iquilezles.org/articles/distfunctions/

Quilez, I. (n.d.-b). *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

My gratitude goes to those whose contributions were invaluable to writing this paper.

Firstly, I would like to thank my supervisor Koen Samyn, for his help in conducting the research for this paper and his 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 stayed on track.

A heartfelt thank you goes to my close friends and family for their unwavering support during the process of writing this paper. In particular my father 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 big thank you to Arne Olemans and my mother for giving me insights by proofreading the early versions of this paper,to my brother for having my back when I needed it and to my sister for her London-based virtual support.

To everyone who contributed directly or indirectly to the completion of this paper. I am grateful for all the support and encouragement I received.

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.