A picture containing tool

Description automatically generatedMusschoot Adriaan

Implementing acceleration structures to optimize a sphere tracing ray marcher

Supervisor: Samyn Koen

Coach: Geeroms Kasper

Graduation Work 2023-2024

Digital Arts and Entertainment

Howest.be

A close up of a card

Description automatically generated

Contents

[Contents 1](#_Toc187309522)

[1 Abstract & Key words 3](#_Toc187309523)

[2 Preface 4](#_Toc187309524)

[3 List of Figures 5](#_Toc187309525)

[4 Introduction 6](#_Toc187309526)

[5 Literature Study / Theoretical Framework 7](#_Toc187309527)

[5.1 Ray Tracing 7](#_Toc187309528)

[5.1.1 Fundamentals 7](#_Toc187309529)

[5.1.2 Acceleration structures 7](#_Toc187309530)

[5.2 sphere tracing (ray marching) 9](#_Toc187309531)

[5.2.1 Fundamentals 9](#_Toc187309532)

[5.2.2 Approaches considered 12](#_Toc187309533)

[6 Case study 13](#_Toc187309534)

[6.1 aabbs for sdfs 13](#_Toc187309535)

[6.1.1 aabbs for sdfs are early outs 13](#_Toc187309536)

[6.1.2 Box early out 14](#_Toc187309537)

[6.1.3 Sphere early out 15](#_Toc187309538)

[6.1.4 Testing 16](#_Toc187309539)

[6.1.5 Results 17](#_Toc187309540)

[6.2 BVH-Tree for sdfs 19](#_Toc187309541)

[6.2.1 bvh-tree for sdfs are scene wide 19](#_Toc187309542)

[6.2.2 constructing the bvh-tree 19](#_Toc187309543)

[6.2.3 Using the bvh-tree 21](#_Toc187309544)

[6.2.4 testing 21](#_Toc187309545)

[6.2.5 Results 22](#_Toc187309546)

[7 Discussion 25](#_Toc187309547)

[8 Conclusion 26](#_Toc187309548)

[9 Future work 27](#_Toc187309549)

[10 Critical Reflection 28](#_Toc187309550)

[11 References 29](#_Toc187309551)

[12 Acknowledgements 31](#_Toc187309552)

[13 Appendices 32](#_Toc187309553)

# Abstract & Key words

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 SDFs (signed distance fields).

In ray tracing there are many acceleration structures to speed up this process. In this paper the Axis Aligned Bounding Box and Bounding Volume Hierarchy Tree, commonly used in ray tracing, are considered and implemented to 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.

# Preface

In recent times my interest grew towards ray marching, particularly sphere tracing. At first it didn’t really seem to be present anywhere, but as I kept digging, I found quite a few use cases. I mainly found it used in particle systems and such, but what I enjoyed most was seeing all these mathematical shapes creating these seemingly impossible worlds. It’s just like a wow-moment every time I find a new demo-scene.

# List of Figures

[Figure 1: Example of a triangle mesh 6](#_Toc187250884)

[Figure 2: The process of ray casting and shading pixels 6](#_Toc187250885)

[Figure 3: AABB for a single triangle 7](#_Toc187250886)

[Figure 4: Complex mesh with BVH applied (Sebastian Lague, 2024) 8](#_Toc187250887)

[Figure 5: Tree structure for BVH (Goren, 2021) 8](#_Toc187250888)

[Figure 6: Example of SDF output (*Figure 3*, n.d.) 9](#_Toc187250889)

[Figure 7: Ray marching missing a surface (Donnelly, 2005) 10](#_Toc187250890)

[Figure 8: Example of both scenarios when raymarching (Hart, 1996) 10](#_Toc187250891)

[Figure 9: Link SDF in experiment 17](#_Toc187250892)

[Figure 10: Pyramid SDF in experiment 17](#_Toc187250893)

[Figure 11: Mandel Bulb SDF in experiment 18](#_Toc187250894)

[Figure 12: Example of BVH-tree (*Bounding Volume Hierarchies*, n.d.) 19](#_Toc187250895)

[Figure 13: Low complexity scene 22](#_Toc187250896)

[Figure 14: Medium complexity scene 22](#_Toc187250897)

[Figure 15: High complexity scene 23](#_Toc187250898)

# Introduction

**In the introduction, you write the background of your topic and discuss the observation that spurred you on to do this research project. Explain the purpose of the paper and present your research question(s) and the hypothesis at the end of this section. This section is typically a couple of pages long.**

Rendering 3D scenes out and presenting them to 2D screens has been a challenge in the digital world since the dawn of time. Multiple approaches exist, new ones get invented and old ones get brought back to life. In this paper we would like to revisit 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)

In Unreal Engine 5 ray marching is used for ambient occlusion and distance field shadows. (Unreal Engine 5, n.d.; Ureal Engine 5, n.d.)

There is a game called “Claybook” (<https://claybookgame.com/>) which handles the entire rendering pipeline using ray marching and fully exploiting the benefits and quirks of raymarching.

The website “Shadertoy” (<https://www.shadertoy.com/>) developed by Inigo Quilez and Pol Jeremios is a tool used to teach and create demo scenes. It purely uses the ray marching algorithm.

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, I noticed how most approaches didn’t really consider the performance. The priority was always to get beautiful scenes on the screen. And as we all know, if you want higher complexity and bigger scenes you need more performance.

In this paper we will discuss the axis aligned bounding box (AABB) and the bounding volume hierarchy tree (BVH-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.

# Theoretical Framework

## Ray Tracing

### Fundamentals

#### Triangle meshes

In computer graphics, we need to visualize 3D objects. One of the approaches is a triangle mesh. These consist of a bunch of triangles (Figure 1), each triangle consist of 3 vertices and 3 edges connecting these vertices. (“Triangle Mesh,” 2024)



Figure : Example of a triangle mesh

#### 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). If the ray intersects with one or multiple meshes, it calculates the shading at that point using the material properties of the closest mesh. The result is used to fill in the pixel the ray was cast for (Figure 2). This process can be repeated with different origins and directions to incorporate light, reflections and shadows. (*Ray Tracing | NVIDIA Developer*, n.d.)



Figure : The process of ray casting and shading pixels

### Acceleration structures

#### WHY?

If you have very complex geometries and many objects in your scene, then the algorithm executes a lot of unnecessary calculations.

Scenario 1: You have a lot of objects in your scene. The algorithm needs to calculate the intersection for every object just to conclude the ray didn’t hit anything.

Scenario 2: You have a super detailed object with 100 000 primitives. If the algorithm must go over every triangle in that object to find the one it is intersecting with. Then again you have a lot of unnecessary calculations.

To avoid this there are the following acceleration structures.

#### aabb

AABB or Axis Aligned Bounding Box is a common acceleration structure used when ray tracing primitives.

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 you must find the minimum and maximum points along each axis that defines the mesh. Then you can construct the AABB, using the minimum and maximum as two opposite corners of the box. (“Bounding Volume,” 2024; *CSSE451 Advanced Computer Graphics*, n.d.; *What Is AABB in Computing?*, 2024)

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 any of the triangles from the mesh. This method of early out speeds up ray intersection calculations in the case the ray misses an object.

A computer screen shot of a colorful pyramid

Description automatically generated

Figure : AABB for a single triangle

#### bvh

BVH or Bounding Volume Hierarchy is another common approach to speed up the ray intersection process.

When you have complex mesh such as in Figure 4 calculating intersections with all triangles is very costly. To avoid this, you can subdivide the initial AABB into smaller boxes and intersect with those first. This process can be repeated until one smaller box contains the desired number of triangles. (Sebastian Lague, 2024)

A white dragon with a green laser pointing at it

Description automatically generated

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

You can think of this structure as many smaller AABBs parented to each other. First you check if you intersect with the root node (which is also the AABB). If that is the case, you evaluate its child nodes. If the child node is a leaf node the algorithm can start calculating intersections with the primitives. This approach allows you to determine the level of detail for the boxes and the minimum number of triangles per box.



Figure : Tree structure for BVH (Goren, 2021)

## sphere tracing (ray marching)

### Fundamentals

#### Signed distance fields (sdf)

Another way to visualize 3D objects is by using signed distance fields (SDF).

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

SDFs take in a point and calculate the smallest distance from the surface of the SDF to that point considering the SDFs origin is at (0,0,0) in cartesian coordinate system. If a point is outside the surface of the SDF, the function will return a positive distance value. If it is inside the surface, it will return a negative distance value. When a point is perfectly aligned with the surface the returned value will be 0. Figure 6 is an example of what values such a function returns.



Figure : Example of SDF output (*Figure 3*, n.d.)

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

1. float GetDistance(vector3 point)

2. {

3. return length(point) – radius;

4. }

It takes in a point (the current position along the ray, this is explained later. It 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. This is assuming the sphere is at position (0, 0, 0).

If you want to have translations, rotations or scaling you must manipulate the point before calculating the final value. (Quilez, n.d.)

The biggest benefit of SDFs is that they are usually fully implemented in code in the fragment shader. Meaning this requires way less memory than triangle meshes. As there is no need to store all the vertices that make up those meshes. Only one line of code in the case of a sphere.

#### Ray marching algorithm

##### The marching loop

We need a slightly different algorithm to ray tracing, but with some similar principles. We still use a camera to cast rays through an image plane. However, we 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. If at any point the returned value of the SDF is negative, we can consider that we have hit an object. In Figure 7 one can see that if the increment size is too big the algorithm can easily miss a surface. However, if the step size is too small then it can cause reduced performance. (Donnelly, 2005)

A diagram of a graph

Description automatically generated

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

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

1. We can calculate the distance to the SDFs in the scene by giving passing in the origin of the ray.
2. Then we can displace the origin of the ray along its direction, using the distance value returned by the SDF.
3. Repeat steps 1 and 2 until one of 2 possible outcomes:
   1. The distance we can travel is smaller than an arbitrary value (0.001), meaning we hit an object
   2. The value of the total distance travelled by the ray has exceeded the scene boundaries.

The pseudo code

1. distance travelled = 0

2. origin = camera point

3. while (true)

4. point = origin + distance travelled \* direction

5. distance able to travel = distance to scene based on point //loops over all SDFs

6. distance travelled = total distance + distance able to travel

7. if distance able to travel < 0.001

8. break //hit object

9. if total distance > scene boundaries

10. break //no objects hit

In Figure 8 both rays are being marched towards the triangle from left to right. The top ray represents outcome ‘a’ where the SDF result approaches 0. As a result, the pixel for which this ray was cast should be colored with the triangle’s color properties. The bottom ray represents outcome ‘b’ where the distance the ray travelled exceeded the scene bounds (in this case the image bounds). (Devred, 2022)

A black and white drawing of a triangle

Description automatically generated

Figure : 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. And as SDFs 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. In its most primitive form of execution the pseudo code for it looks something like this.

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 you have a screen of 100 by 100 pixels with an average ray step of 10 to traverse the scene. You would evaluate this function 100 000 times each frame. This linearly increases with the number of objects in the scene. To speed this up can we apply the same acceleration structures commonly used for ray tracing?

### Approaches considered

Creating a 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 optimization techniques. The following approaches were considered.

#### Custom framework / own implementation

The framework can completely be run on the CPU, meaning that debugging would be easy.

Running completely on the CPU is of course also a drawback. There would be a considerable drop in performance, as we can’t use the computational power of the GPU.

Writing regular C++ code is a huge benefit. There is more functionality than when writing shader code. In C++ you can also use recursion, classes, pointers, containers, etc. In C++ there is no built-in support for vector math and matrix operations.

#### Graphics api

Graphics APIs allow the user to communicate with the GPU. This will get us all the frames we could ever wish for. However, there are some drawbacks. Debugging support for graphics applications is very limited, not built-in and often not as straightforward and requiring external tools such as “RenderDoc”.

Feature support for languages such as GLSL or HLSL is also limited as it has a different execution environment. They do have built-in support for vector math and matrix operations.

As mentioned above, the ray marching algorithm only must be implemented in the fragment shader of a rasterization pipeline. This means all that is needed is a singular compute shader. So once that has been set up using the API, all that is left to write is in the shader. This is of course a nightmare for organizational purposes, if everything is in one file.

Vulkan and OpenGL were considered. OpenGL for its ease of use and its wide usage across the C++ community. Vulkan is where I have the most experience, and I already have an existing framework there.

#### shadertoy

The “shadertoy” website would require the least amount of work to execute this research in. It runs entirely on the ray marching algorithm already. It uses GLSL, which is simple to understand, and it has its own little compiler to tell you what is wrong.

However, there aren’t any methods of capturing frames and looking at the output, like with “RenderDoc”.

There is no possibility for me to track anything, the user only has access to the shader.

#### choice and motivation

Custom implementation is the approach of choice. It is the approach that will allow the easiest gathering of data while keeping the amount of work relatively small.

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 could potentially be a good building block to set up the experiment, I am also familiar with this framework. Unlike setting up something entirely from scratch using the graphics API.

As mentioned before there is more added functionality when writing C++ instead of shader languages. Debugging will also be easier on the CPU, definitely when it comes to acceleration structures as they don’t provide any visual output.

The first drawback of this approach is that it uses a custom math library which is slower and incomplete. It still needs to be swapped for OpenGL Mathematics (GLM).

Secondly, nothing could ever run as fast on the CPU as on the GPU. The application will most definitely have to be multi-threaded to execute the research at more than 1 FPS.

# Case study

## aabbs for sdfs

### aabbs for sdfs are early outs

In the theoretical framework creating the AABB for triangle meshes was discussed. However, when working with those, every point is known in world space, allowing for easy construction of the axis aligned bounding box.

The only data we have from SDFs is that we can get a distance from the surface to a point. Given this property we can approach the maximum boundaries that contain this SDF. Two approaches were implemented and compared for constructing the shape of the early out SDF.

What does the axis aligned bounding box look like in ray marching? Unlike ray tracing we cannot intersect with a box first using the ray marching algorithm. We must march towards the early out shape along the ray. This means we need an additional SDF as early out. Two SDF shapes were considered a box, and a sphere.

To use this additional SDF, first the distance to the additional SDF is calculated. If the distance to the additional SDF is smaller than a certain threshold (0.001), the distance to the object enclosed by the additional SDF must be computed and returned. If the distance to it is bigger than the threshold that distance is returned and acts as a form of early out, and the need to use the enclosed object is redundant.

1. float sdf::Object::GetDistance(glm::vec3 const& point)

2. {

3. float const earlyOutDistance{ EarlyOutTest(point) };

4. if (earlyOutDistance >= 0.001f)

5. {

6. return earlyOutDistance;

7. }

8. return GetDistanceUnoptimized(point);

9. }

Only if the result of the early out test is greater than the threshold value, should it return the early out value. This speeds up the ray marching process greatly. However, it does mean that once the ray is marching inside the early out-SDF, extra and unnecessary calculations will happen. This is to be expected and no different than what happens in ray tracing. However, the benefits of an early out hugely outweigh those extra calculations when actually approaching and hitting an object.

### Box early out

#### 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. Each wall then calculates its minimum distance to the SDF and stores the point for which it was calculated.

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, retrieved by subtracting the minimum distance from the wall to the SDF from the length of the vector representing the point for which the minimum distance was calculated. Each wall repeats this process until one of the points on the wall has a distance value smaller than a certain threshold (0.001). This is considered as reaching the surface.

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 values are retrieved, we compare the absolute values of the x, y and z components of each point to determine the box extent of the SDF box. Which will be used to compute the 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);

#### 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. }

#### why this shape

This shape directly compares to its ray tracing implementation. Logically, it should be of similar value here.

Calculating the distance to a box is quite cheap in ray marching, however it is not the cheapest thing in ray marching.

The box can with great precision enclose the boundaries of the underlying shape.

However, due to the nature of the box SDF it always is a mirrored box. Meaning that even if the underlying SDF only has values in the positive y-axis (such is the case for the pyramid) the box would have the same length in the positive as in the negative y-axis.

### Sphere early out

#### constructing the boundary

At the core of calculating the maximum boundary for your SDF you sample points on a sphere and request the distance for each point to the SDF.

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 having executed this for the initial sphere radius, which should encompass the maximum scene boundary, the smallest possible distance from any point to the SDF is used to set up the next sphere radius like so.

1. float const nextRadius{ glm::length(closestPoint) – minimumDistance };

Through this method we can march towards the origin by decreasing the radius every time. Once one of the points on the sphere has a distance value smaller than a certain threshold (0.001), we consider the point to have reached the surface and consider it the furthest possible distance from the origin (0,0,0) in the cartesian coordinate system to the surface area. We can then calculate the length of the vector representing this point and use it to construct a sphere SDF acting as a form of early out.

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);

#### Formula

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

1. float sdf::Sphere::EarlyOutTest(glm::vec3 const& point)

2. {

3. return length(point) - radius;

4. }

#### Why this shape

This shape is the cheapest possible calculation in ray marching.

Due to the nature of the sphere if the SDF it encloses has a peak distance on the y-axis and is very slim in the xz-plane it, the sphere will have the value of the y-axis as its radius. Meaning that it creates a lot of empty space underneath its surface, whereas a box prevents this more (also not fully).

### Testing

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

For every object we 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 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 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 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. Meaning afterwards we had approximately 10 seconds of frame times captured.

### Results

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

#### Link

|  |  |
| --- | --- |
| Figure : Link SDF in experiment | 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.  When early out is enabled, the frame time of the box is always faster than the sphere. |

#### Pyramid

|  |  |
| --- | --- |
| Figure : 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.  When early out is enabled, the frame time of the box is always faster than the sphere.  Early out disabled in a normal shot is faster than the sphere early out. |

#### Mandelbulb

|  |  |
| --- | --- |
| Figure : Mandel Bulb SDF in experiment | All close-up shots are faster than normal shots.  Applying any early out is faster than applying none.  The sphere as early out performs better than the box early out. |

## BVH-Tree for sdfs

### bvh-tree 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. This means that the BVH-tree must be created and represents the entire scene instead of a single object.

### constructing the bvh-tree

#### 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

There are 2 types of nodes:

1. Parent nodes: they act as early outs for multiple shapes, with an origin and radius / box extent. 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.
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.

A black background with white triangles and blue dots

Description automatically generated

Figure : Example of BVH-tree (*Bounding Volume Hierarchies*, n.d.)

#### determining the nodes

Starting with the root node it executes the following steps. The initial group of objects is everything in the scene.

1. The objects are presented. It accumulates all the origins and takes their average as origin, 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 surface area heuristic (SAH).
3. Each group is then presented to its left and right child node respectively and they repeat step 1, 2 and 3 until they reach an SDF.

When an SDF is reached, it will not execute those steps again instead it will just store the SDF.

1. Create BVH node pass in the objects

2. origin = sum of all objects

3. radius = maximum of any object // or extent = minimum and maximum of any object

4.

5. if 1 object in objects // leaf node

6. SDF = object

7. return and end the recursion for this branch

8. else

9. Split objects into objects1 and objects2 //using the SAH

10. Left node = Create BVH node pass in objects1

11. Right node = Create BVH node pass in objects2

Splitting the objects happens according to the surface area heuristic. This evaluates the best axis along which to separate the objects.

1. glm::vec3 bestAxis

2. float bestCost = FLT\_MAX

3. int bestSplitIndex

4. for every axis

5. sortedObjects = sort objects along currentAxis;

6. for every index in the sortedObjects

7. leftObjects = objects from start of sortedObjects until currentIndex

8. rightObjects = objects from currentIndex until end of sortedObjects

9. leftArea = bounding box area around leftObjects

10. rightArea = bounding box area around rightObjects

11. cost = leftArea \* nrOfLeftObjects + rightArea \* nrOfRightObjects

12. if cost < bestCost

13. bestCost = cost

14. bestAxis = currentAxis

15. bestSplitIndex = currentIndex

16. sortedObjects = sort objects along bestAxis

17. leftObjects = objects from start of sortedObjects until bestSplitIndex

18. rightObjects = objects from bestSplitIndex until end of sortedObjects

By sorting the objects and calculating the index for which it is cheapest to split them, the objects can be optimally sorted.

### Using the bvh-tree

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

1. float GetDistanceToScene(glm::vec3 const& point) const

2. {

3. return m\_BVHRoot->GetDistance(point);

4. }

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

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 meaning it went through one entire “branch of the tree”. Or it will return its bounding volume distance as it is not necessary to evaluate the leaf node/SDF object.

### testing

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

For every scene we tested without, with a sphere and with a box BVH-tree.

For object and each usage of BVH-tree, 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 330000 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. Meaning afterwards we had approximately 10 seconds of frame times captured.

### Results

#### Low complexity scene

|  |  |
| --- | --- |
| Figure : Low complexity scene | All close-up 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 its boundaries.  The SDFs themselves have low computation times. So, even if the node (using sphere boundaries) has too much extra space, the extra calculation underneath is still cheap to compute. |

#### medium complexity scene

|  |  |
| --- | --- |
| Figure : 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 BVH-tree.  The accuracy of the box now 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 also are heavier to compute in this scene, so having more accuracy to not have to compute them is more efficient. |

#### High complexity scene

|  |  |
| --- | --- |
| Figure : 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, and thus benefit from the box approach most as they don’t use one value for all node boundaries.  The SDFs themselves are highly complex, so avoiding as many calculations as possible using the box BVH-tree for more precision is hugely beneficial. |

# Discussion

**In this section, you offer an interpretation of the results you obtained and try to relate them to the theoretical framework you presented. This is typically not a very long section, but obviously one of the most important ones.**

From the measurements for the early out tests we can make a few observations. While, in almost all cases the performance is improved massively

# Conclusion

**In this section, you ascertain the demonstrable outcomes of your study and outline the merits of the project for the academic field and the discourse community. This is typically not a very long section, but obviously also one of the more important ones.**

# Future work

**This section is sometimes standalone, sometimes incorporated in the conclusion. It looks at the shortcomings of the study, alternative strategies, and what could be the next course of action in the research field. This is typically not a very long section.**

# Critical Reflection

**This section is typically associated with a bachelor paper, not other forms of serious writing. It allows the student to reflect on the learning outcomes, both academically and in terms of personal growth.**

# 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

*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

*Figure 3: The signed distance-field of a given oriented surface S...* (n.d.). ResearchGate. Retrieved January 7, 2025, from https://www.researchgate.net/figure/The-signed-distance-field-of-a-given-oriented-surface-S-left-classifies-the-R-3-space\_fig1\_263593124

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

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

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.). *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

Ureal Engine 5. (n.d.). *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

*What is AABB in Computing? (Axis Aligned Bounding Box)*. (2024, April 16). https://60sec.site/buyer-guides/what-is-aabb-in-computing-axis-aligned-bounding-box

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

# Acknowledgements

**In this section, you can thank people who contributed to your work in a meaningful way.**

# Appendices

**In many cases, there are items that were developed for a research paper that can’t go into the actual paper in full. Things suc as code, art pieces, output of statistical analysis, questionnaires, … In this section, you can present these elements; use the first page to list and number the items, then paste them sequentially. If some items are too large, you can store them online, and link to them. Common practice is to keep those links active at least one year after the publication of the thesis.**