

# Generating Signed Distance Fields from Triangle Meshes using Vulkan Compute Shaders

**William Whitehouse**

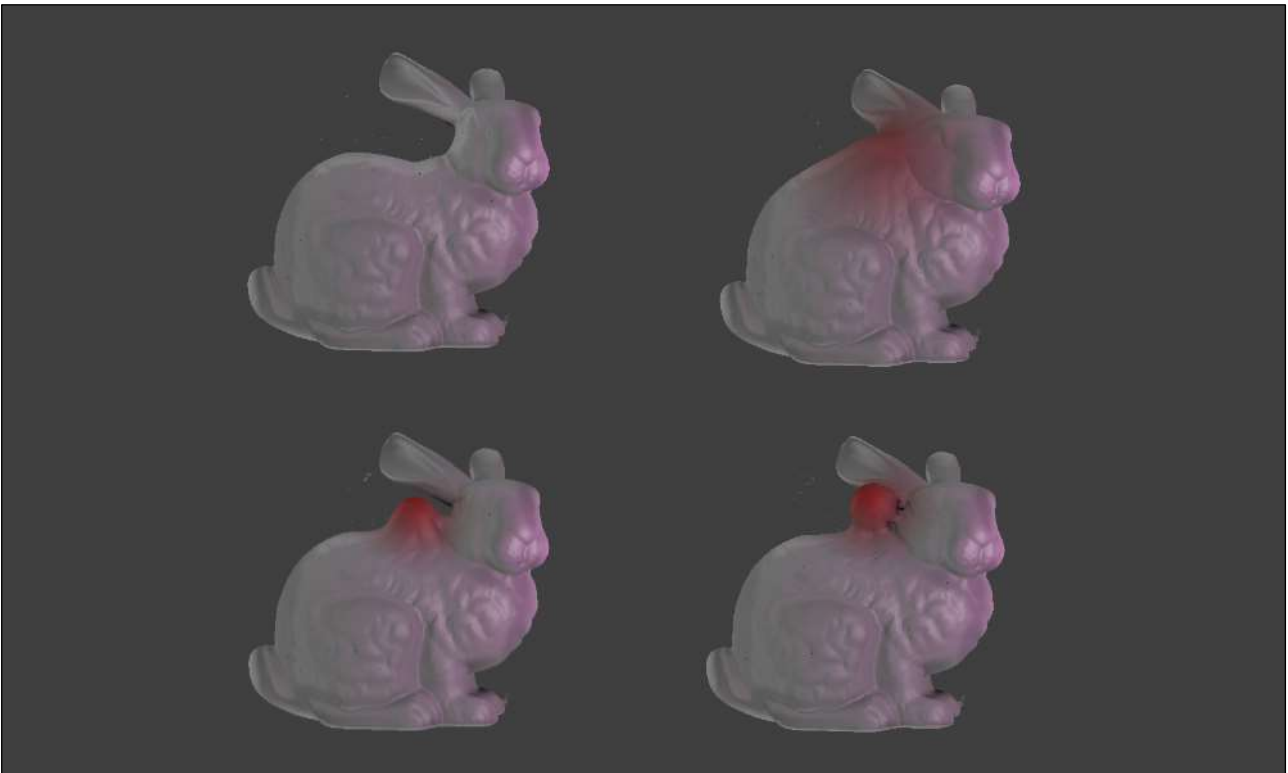
*Supervisor: Dr. Thomas Bashford-Rogers*

**Department of Computer Science and Creative Technology**

University of the West of England

Coldharbour Lane

Bristol BS16 1QY



## Abstract

This project aims to explore various techniques at computing signed distance fields from triangle meshes in C++ and using the Vulkan graphics and compute API. Voxel grids are chosen as a method where each cell stores its distance to the mesh, which paired with the jump flooding algorithm allows for rapid generation of under a second on meshes that contain over 14K triangles. Binary Space Partitioning is explored, with the use of different types of partitioning planes and ways to render the tree using raymarching. This approach was found to be very poor regarding generation and runtime performance, but offered perfect accuracy compared to the voxel grid.

**Keywords:** signed distance field, triangle mesh, raymarching, vulkan, c++

## Brief Biography

I have a strong interest in low-level engine programming and graphics. This project explores representing meshes using distance fields and their rendering within a custom C++ engine. I've gained knowledge about frequently overlooked yet crucial engine systems required for real-time applications, such as games, and have become proficient in using the Vulkan graphics API. There's much more to learn in this area, and even after this project ends, I'm eager to continue exploring new techniques and methods.

Portfolio: <https://williamwhitehouse.dev/>

## How To Access The Project

The project can be accessed via this repository: <https://github.com/WSWhitehouse/CCTP-Project>

Please follow the getting started guide in the README to ensure the project is cloned properly and correct versions of the required software is installed – download links are provided in the README file. The project contains multiple branches to showcase the different approaches that were explored during development.

The *vendor* folder contains code from third parties, the *src* folder contains all the C++ source files for this project. Any code that is borrowed from other authors is marked accordingly.

## Showcase Video

<https://www.youtube.com/watch?v=vs0w3iPgb40>

---

## 1. Introduction

This project aims to explore various techniques of representing a triangular mesh as a *Signed Distance Field* (SDF). These techniques will be compared in terms of efficiency, performance, and memory usage.

Although meshes are the primary way games visually represent objects, they may not be suitable for various technical situations, particularly those where an SDF would be more appropriate. Having a performant and memory efficient SDF representation of the mesh could result in greater usage within games.

Distance fields and implicit surfaces are intriguing because they can undergo countless mathematical operations to manipulate them in space, which includes twisting, blending, and applying n-dimensional transformations. So far, these operations are primarily applied to primitives. However, representing a mesh as a distance field may enable these interesting operations to be performed on a complex shape.

To render the generated distance fields, the raymarching rendering process will be implemented using the Vulkan API. Optimisations to raymarching will be employed to ensure high framerates and real-time rendering of the distance fields.

### 1.1 Key Deliverables

- Program to generate SDFs from triangle meshes.
- Renderer written in C++ using the Vulkan API to showcase the generated distance fields.
- A report outlining the research, implementation processes, key findings, evaluation, and future work.

### 1.2 Project Objectives

#### Must:

- Volumetric raymarching renderer written using the Vulkan graphics API.
- Triangle mesh to SDF generation application.

#### Should:

- Explore different ways of representing the distance field.
- Implement, and compare different SDF to mesh generation methods.

#### Could:

- Support different types of meshes (i.e. open and self-intersecting).

## 2. Literature Review

### 2.1 Vulkan Graphics API

Vulkan is an API designed for “*explicit control of low-level graphics and compute functionality*” (The Khronos Group, 2022) allowing for efficient access to GPU resources and rendering capabilities with low overhead.

The official specification is a guide written by many contributors that outlines best practices and correct usage of the Vulkan API (The Khronos Group, 2022). It is updated regularly and is highly relevant when developing an application with Vulkan. Updates to the specification are managed through pull-requests and issues on the Vulkan-Docs GitHub (The Khronos Group, 2022), meaning the information must be officially approved and accurate.

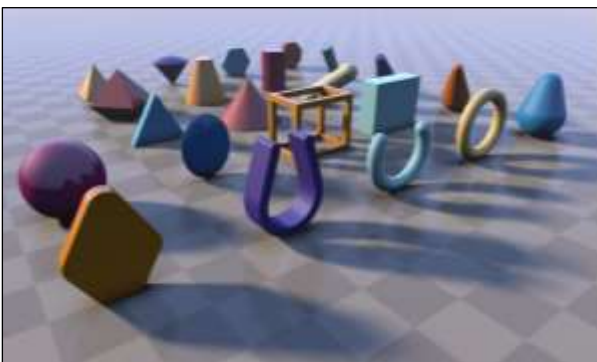
Online guides such as the “*Vulkan Tutorial*” (Overvoorde, 2016) and “*VkGuide*” (Blanco, n.d.) give a general overview and example C++ code for getting started. The “*Vulkan Programming Guide*” (Sellers & Kessenich, 2016) is a book written by a Vulkan specification lead developer, which gives in-depth insight into the more complicated aspects of Vulkan, in contrast with the online guides.

### 2.2 Signed Distance Fields

An SDF returns the minimum distance to a surface given any point in space ( $P$ ), with the sign indicating whether the point is inside or outside the object (Jones, et al., 2006), as shown in **Equation 1**; where  $f(P)$  is the function that returns the distance to the SDF.

$$f(P) \begin{cases} < 0.0, & \text{if } P \text{ is inside} \\ = 0.0, & \text{if } P \text{ lies on the surface} \\ > 0.0, & \text{if } P \text{ is outside} \end{cases}$$

**Equation 1** – SDF function.



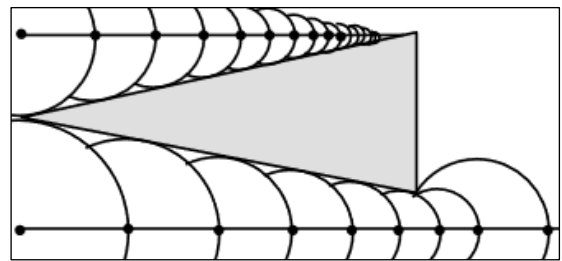
**Figure 1** - Raymarching Primitives on ShaderToy (Quilez, 2013)

Inigo Quilez’s website (Quilez, n.d.) has been the primary resource for understanding SDFs. His blog contains a large collection of implicit SDF primitives, with code examples available on ShaderToy (Quilez, 2013) (see Figure 1). While the blog may contain some potential bias, it is challenging to argue with the results as an open-source example is available. Additionally, due to Quilez’s impressive employment history, where he worked on the procedural sets at Pixar Animation Studios, there is a level of trust in the information he provides regarding SDFs.

### 2.3 Raymarching

Raymarching is a ray traversal “*technique for rendering implicit surfaces using geometric distance*” (Hart, 1995). Hart’s article describes a ray being projected from the camera position into the scene. The ray moves incrementally, based on the distance from all objects in the scene, ensuring it can safely move without intersecting a surface (Hart, et al., 1989). Once the distance approaches zero, it is considered a hit. Hart’s publications are peer-reviewed and heavily cited by other relevant sources.

Figure 2 shows the algorithm where two rays are fired from the left side of the image (Hart, 1995). Spheres represent the distance at each step along the ray. This is repeated until the ray hits a surface (top ray) or reaches a max iteration count / maximum distance (bottom ray).



**Figure 2** – Raymarching two rays (Hart, 1995). One hits the surface (top), one misses (bottom).

### 2.4 Computing Signed Distance Fields

Payne & Toga (1992) use a 3D voxel grid to store precomputed mesh distances. They outline a naïve implementation, where each cell iterates through all the triangles in a mesh and takes the minimum distance. The paper suggests some improvements, such as organising the mesh hierarchy before computing the distances (Payne & Toga, 1992). Wald, et al. suggests organising the voxel grid using a K-Dimensional tree which results in highly efficient ray traversal during rendering (Wald, et al., 2005).

A Binary Space Partitioning (BSP) tree can efficiently represent the distance field and “*adapts very well to the shape of the underlying surface*” (Wu & Kobbelt, 2003). The mesh is recursively split into smaller pieces, for each piece the distance is approximated and an error

is estimated. If the error is above a threshold the model is split again. Another paper found similar results, but it was noted that the algorithm takes a while to generate and is not suitable for larger data sets (Fryazinov, et al., 2011).

Ensuring the accuracy of the mesh distance field is important. Bærentzen, et al. discusses the use of a pseudonormal which is a “*weighted sum of the normals on incident faces*” (Bærentzen & Aanæs, 2005). When generating the sign of a distance field, the face normal of the triangle is used, but this normal may break down near an edge or vertex. The pseudonormal allows the normal to be calculated more accurately and the authors provide mathematical proof in another paper (Bærentzen & Aanæs, 2002).

In his thesis, Zeng proposes a different approach where he investigates the use of an *Artificial Neural Network* (ANN) and a *Generative Adversarial Network* (GAN) to generate an SDF, which he compares to voxel grid-based methods (Zeng, 2018). It concludes that the GAN was the best type of network to generate a distance field.

Further research has trained GANs on voxel grids and point clouds, successfully generating shapes that include intricate details such as “*thin chair legs*” and “*precise modelling of an aeroplane tail*” (Kleineberg, Fey, & Weichert, 2020).

DeepSDF employs a machine learning approach which predicts the SDF value at a given query position (Park, Florence, Straub, Newcombe, & Lovegrove, 2019). Their method also preserves oriented surface normals which provides more visually accurate results.

Wang et al. developed a *Signed Distance Map Neural Network* (SDMNN) that rapidly produced an SDF (Wang, et al.). They compared their approach to other machine learning methods and found it to be faster. However, they also found it did not scale well to larger data sets as the number of SDMs would grow quadratically with the number of shape parameters.

### 3. Research Questions

**Q1.** *How can SDFs be generated from triangle meshes, considering the trade-off between accuracy, performance, and memory usage?*

Investigate various techniques for creating SDF representations of a mesh. Assess the precision, runtime performance, and memory consumption of each method to determine the most optimal approach.

**Q2.** *How can the performance of raymarching be optimised?*

Explore and evaluate techniques for improving raymarching performance in real-time Vulkan applications.

**Q3.** *Can techniques such as parallelisation be used to speed up SDF computation?*

Utilise and examine the use of Vulkan compute shaders to improve generation times for the various SDF computation approaches. Additionally, explore CPU multi-threading techniques that could improve the computation speed.

### 4. Research Methods

The research for this project was secondary, quantitative research, which includes using papers, reports, and presentations from various academic sources. Google Scholar and the UWE library were the key resources that enabled this type of research. Primary research is not possible in the timeframe allocated as this project is of a technical nature.

The topic area for this project is extensively covered in academic literature, which was used throughout the project for guidance. Initially, research started with general search terms related to the topic area. As different methods were identified, the terms became more focused to concentrate on specific techniques.

This project also involves creating a renderer using the Vulkan API, in which primary sources of information include forums, blog posts and the official specification. However, potential biases in these sources require careful analysis to ensure the information's accuracy and reliability.

Online question and answer sites like Stack Overflow were used in some instances. These sites offer well-defined answers and insights from a range of people, often supported by credible sources. Experts can easily share their knowledge; however, these sites shouldn't be primary research sources due to potential biases and inaccuracies. In this project, they provided a basic understanding of a subject area, followed by more accurate research to back up claims.

### 5. Ethical & Professional Principles

This project generates no negative impacts, no human participants were involved during the research or development phase and no personal data is being held.

There are some ethical and legal factors that must be considered. Throughout development different meshes were acquired for testing, these are obtained legally, with appropriate credit given to the authors. Any third-party code and libraries are used in accordance with their respective licenses.

All assets used throughout the project can be seen in *Appendix: Assets & Third-Party Code*.

## 6. Research Findings

Games require maximum performance and reliable framerates, using Vulkan ensures low overhead from the graphics driver which allows the program to run at its fastest. Furthermore, some computations can take advantage of Vulkan's compute functionality which will further increase performance.

Research indicates that implementing a Vulkan renderer can be challenging. However, the online guides provide a decent foundation to build upon throughout the project's development.

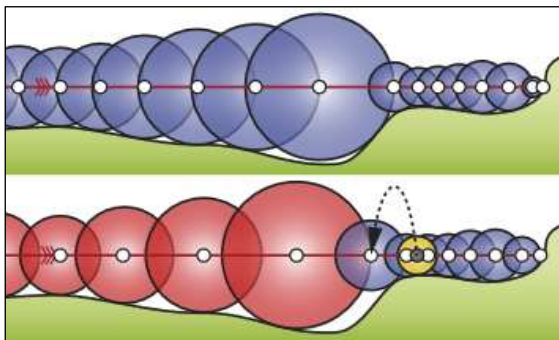
SDFs are widely used in games for collision detection, soft shadows, and ambient occlusion. Unreal Engine 5 employs distance fields for all the above; as they lead to faster computation during collision testing and more realistic and faster results with soft shadows (Epic Games, 2021); see figure 3 for a comparison.



**Figure 3** – Shadows using a traditional shadow map (left), and distance fields (right) (Epic Games, 2021).

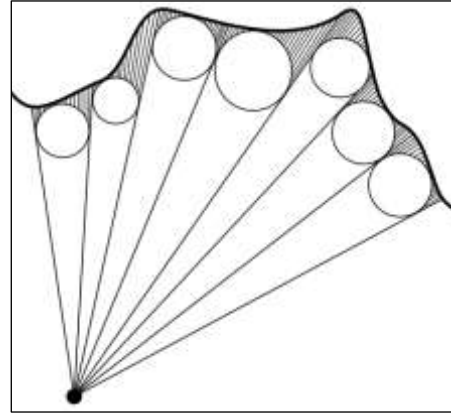
Keinert et al.'s "Enhanced Sphere Tracing" (2014) presents an improvement to the raymarching algorithm originally proposed by Hart, called Over-Relaxation. This technique overshoots the distance when marching the ray. If the distance overlaps with the previous iteration's distance, it is safe to assume the ray can safely pass through without intersecting a surface. If it does not overlap, the algorithm returns to Hart's standard implementation.

Figure 4 provides an illustration of the proposed technique (in red), where the yellow sphere represents where over-relaxation fails and returns to the standard algorithm (in blue).



**Figure 4** – Comparison between the standard raymarching algorithm (top) and over-relaxation (bottom) (Keinert, Schafer, Korndorfer, Ganse, & Stamminger, 2014)

Coarse Cone Tracing is used to approximate the surface of the SDF and speed up raymarching, which was used in the game Claybook (Aaltonen, 2018). Cones, which act as "growing spheres" are marched over multiple pixels in a pre-pass; each pixel's ray starts from the end of the cone rather than the camera's position (see figure 5). This reduces the number of steps each ray must make through empty space.



**Figure 5** – Coarse Cone Tracing. Cones are marched at a lower resolution, with individual pixel rays beginning where they end; this reduces the empty space the individual rays must march through.

SDFs are typically defined as implicit surfaces, meaning an equation defines the object rather than any physical representation, such as a mesh (Hart, 1995). Real time representation of a triangular mesh as a distance field poses an interesting problem; the minimum distance to all the triangles must be known from any point in space. Doing this calculation for every triangle on each ray step, pixel and frame is extremely expensive.




It is clear from the research that alongside raymarching optimisations either precomputing the mesh distances or using a hierarchical data structure is vital for maintaining high frame rates.

The voxel grid approach contains a variety of techniques and data structures to improve the accuracy, efficiency, and memory usage. The naïve generation approach outlined by Payne & Toga (1992) is highly parallelisable, making it an ideal candidate for a compute shader.

Wald, et al.'s use of a K-Dimensional tree could increase voxel traversal speed. However, it was aimed at the CPU and used SIMD to speed up the algorithm, but they note that it "should similarly benefit GPU-based ray tracing approaches" (Wald, et al., 2005).

Zeng's thesis findings show that a neural network can successfully approximate an SDF with only minor errors. Furthermore, the GAN takes up less than 0.1% memory than the voxel grid, and even less memory than the triangle mesh itself shown in figure 6 (Zeng, 2018).

Neural Networks also rapidly generate an SDF, with the SDMNN approach taking 0.2 seconds to generate compared to the 10.7 seconds the grid-based approach took (Wang, et al.). However, it is unclear what optimisations, if any, were made to the grid-based method.

#	size	rendering result
triangle mesh	2.28MB	
grid-based SDF	33.9MB	
generative model	47KB	

**Figure 6** – Size comparison of a mesh, voxel grid SDF and a GAN approximate of an SDF (Zeng, 2018).

## 7. Practice

This project was written in C++ and used the C bindings to interface with the Vulkan API. To begin with, a small rendering engine was created with a raymarching pipeline that renders SDFs. The pipeline consists of a vertex shader that draws a fullscreen quad and a fragment shader that performs the raymarching algorithm; all shaders are written in GLSL.

### 7.1 Raymarching Pipeline

The vertex stage is optimised by using a single triangle to cover the screen rather than the typical two-triangle quad. This saves one vertex shader invocation and reduces the number of fragment shader calls. In a two-triangle quad, the shared edge can invoke the fragment shader multiple times for the same pixels without performing any additional work, as GPUs invoke the stage in 2x2 pixel size blocks. This is known as *helper invocations* or *helper pixels* (Giesen, 2011). To match the screen rather than the triangle, the UVs are created using Equation 2; where  $\vec{V}$  represents each triangle vertex in clip space.

$$\vec{u}\vec{v} = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \begin{bmatrix} \vec{V}_x \\ \vec{V}_y \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

**Equation 2** – Creating screen space UVs from a triangle vertex position in clip space.

A ray per-pixel is created, and the raymarching algorithm is performed, as shown in Code Listing 1. Note that the `RaymarchMap` function (line 10) is defined elsewhere and returns the minimum distance to all objects in the scene from the current ray position (`pos`). This function is different for each SDF generation approach and will be discussed later alongside each implementation.

```

1 bool Raymarch(vec3 rayDir)
2 {
3     float dist = 0.0;
4
5     for (int i = 0; i < MAX_ITERATIONS; ++i)
6     {
7         vec3 pos = camPos + rayDir * dist;
8
9         // get the min distance to
10        // all objects in the map...
11        float mapDist = RaymarchMap(pos);
12
13        dist += mapDist;
14
15        // passed render distance
16        if (dist > MAX_DISTANCE)
17        {
18            return false;
19        }
20
21        // hit something
22        if (mapDist < 0.001)
23        {
24            return true;
25        }
26    }
27
28    // passed max iterations
29    return false;
30 }

```

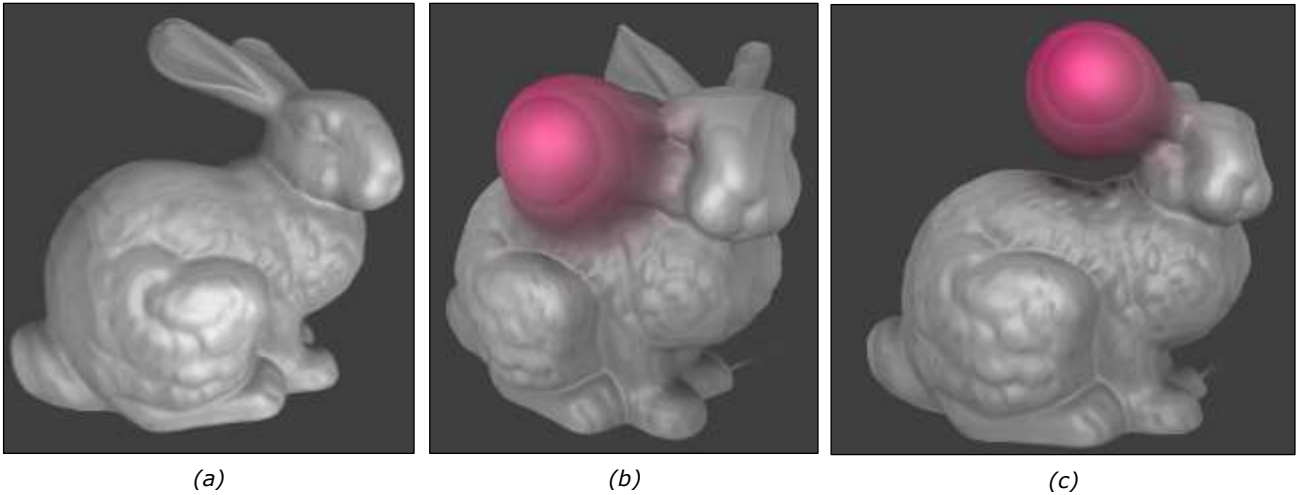
**Code Listing 1** – Raymarching algorithm implemented in the fragment shader (GLSL).

The raymarch algorithm was optimised using the *over-relaxation* method described by Keinert et al (2014). This approach makes some changes so that the ray oversteps by a relaxation value ( $\omega$ ) where  $\omega \in [1; 2)$ . To ensure the ray does not pass through or overstep a surface, the bounding sphere on the  $i^{\text{th}}$  iteration must overlap the previous step's ( $i - 1$ ) sphere; as shown in Equation 3, where  $\delta$  is the sphere step length from an iteration. When the spheres don't overlap, it falls back to the algorithm described in Code Listing 1.

$$\delta_i = f(P_i) \cdot \omega$$

$$|f(P_{i-1})| + |f(P_i)| \begin{cases} \text{spheres do not overlap,} & \text{if } < \delta_{i-1} \\ \text{spheres do overlap,} & \text{otherwise} \end{cases}$$

**Equation 3** – Check for overlapping bounding spheres in the Over-Relaxation raymarching optimisation.



**Figure 7** – Stanford Bunny mesh represented as an SDF using a voxel grid ( $256^3$  resolution). Subfigures (b) and (c) have a smooth minimum operation applied between the SDF voxel grid and a red sphere.

## 7.2 SDF Voxel Grid

Following the implementation of raymarching, the next step is to explore SDF generation. This approach is where each cell in a 3D voxel grid ( $V$ ) holds the minimum distance to the surface of an object.

The voxel grid is a 3D image allocated on the GPU from a device local memory heap for maximum read/write performance in shaders and initialised to `F32_MAX` (the maximum number a floating-point value can represent). The image uses the `VK_FORMAT_R32_SFLOAT` format, allowing each cell to hold the signed distance at normal floating-point precision. Half precision is preferable as it would significantly reduce the size of voxel grid memory without a visually perceivable difference, but the required SPIR-V and Vulkan extensions for 16-bit floating point atomics are not supported on the author's graphics drivers so was not further investigated.

To compute the distance, each cell ( $c \in V$ ) iterates through the triangles in the mesh ( $M$ ) to find its nearest triangle and stores the distance to this triangle as its value:

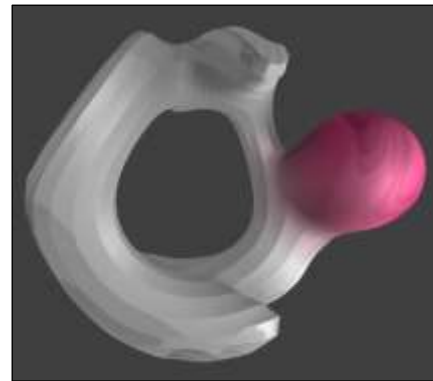
$$c = \min_{t \in M} (\text{dist}(t, c))$$

**Equation 4** – Calculating the minimum distance to a collection of triangles in a mesh for a single voxel.

When comparing the distance to each triangle, the absolute value of the distance is used. This is because the nearest distance to the triangle, regardless of whether it is in front or behind it, is needed. However, the signed value is assigned to the cell so that the inside of the object is maintained as negative.

Figure 7 showcases the generated SDF voxel grids obtained using the Stanford Bunny as the input mesh. The voxel grid method also enables

the use of distance field operations, which allow for smooth blending between surfaces as shown in figures 7(b) and 7(c). Figure 8 demonstrates the twist operation.



**Figure 8** – Low-poly torus mesh as an SDF voxel grid ( $256^3$  resolution). A twist operation has been applied to the torus and a smooth minimum blend has been used between the torus and a red sphere.

This brute force method has a quadratic time complexity. Meaning increasing the number of voxels or triangles in the mesh will result in a significantly longer generation time.

Initially, the computation of the voxel grid was performed on the CPU with each cell computing its distance consecutively. However, since each cell is computed independently, it is an ideal algorithm for parallel execution in a compute shader, which is officially known as an *embarrassingly parallel problem*.

Compute shaders are programs that take advantage of the hundreds to thousands of cores that are on a GPU. They operate on the *Single Instruction Multiple Data* (SIMD) memory model, where the same code is executed on different data. In this case, the same distance computations are performed on the mesh but on different cells from within the voxel grid.

A Vulkan compute pipeline is created that takes three read only input buffers (bindings 0, 1, & 2), and a read/write `image3D` uniform for the voxel grid (binding 3). As seen in **Code Listing 2**, the buffer at binding 0 contains information about the voxel grid and the mesh. Binding 1 and 2 contain the vertex and index arrays respectively, which must be in their own separate buffers as their length is not known at compile time. With the uploaded data, the compute shader is run on each cell in parallel and performs the same distance computation as seen in Equation 4.

```

1 layout(set = 0, binding = 0, std140)
2 readonly buffer UB OVoxelGridInput
3 {
4     mat4 transform; // model transform
5
6     // Voxel Grid Data
7     uvec4 cellCount; // w = X * Y * Z
8     vec3 gridOffset;
9     vec3 gridExtents;
10    vec3 cellSize;
11    float gridScale;
12
13    // Mesh Geometry Data
14    uint indexCount;
15    uint indexFormat;
16
17 } voxelGridInput;

```

**Code Listing 2** – Voxel grid compute shader generation input buffer (GLSL).

The mesh indices could be in either a 32 or 16-bit format. To make the compute shader inputs simpler, it only accepts 32-bit indices. But two 16-bit indices can be encoded in a single 32-bit integer and later unpacked in the shader; as shown in Code Listing 3. The branch checking for index format (line 3) will not affect performance as all threads will be following the same branch; ensuring the lock-step parallelism stays in sync.

```

1 uint GetIndex(uint i)
2 {
3     if (voxelGridInput.indexFormat == INDEX_FORMAT_16)
4     {
5         // unpack 16 bit index from 32 bit int...
6
7         const uint uint32Index = i / 2;
8         const uint uint16Offset = i % 2;
9
10        const uint encodedIndex = indices[uint32Index];
11
12        return (encodedIndex >>
13                (16U * uint16Offset)) & 0xFFFFU;
14    }
15
16    return indices[i];
17 }

```

**Code Listing 3** – Get an index, possibly unpacking a 32-bit integer into two 16-bit indices (GLSL).

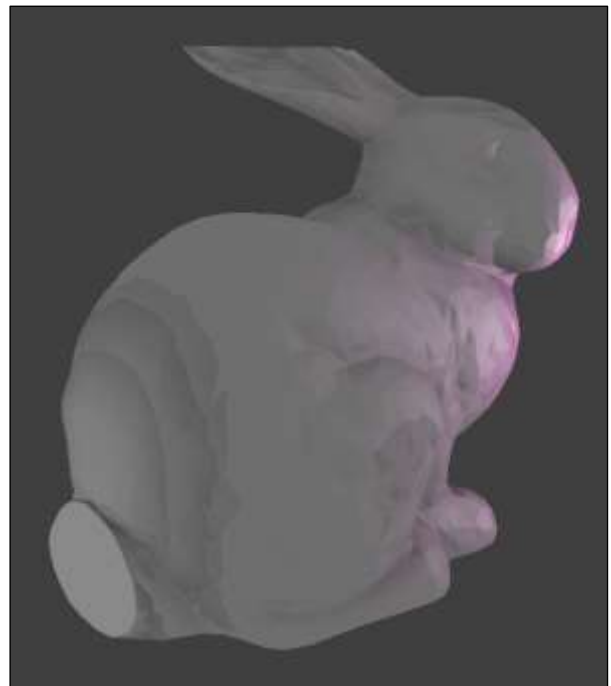
During the distance computation, the triangles are transformed by a scaling factor and are offset within the voxel grid, as seen in Code Listing 4. This is to ensure that the SDF grid represents the same size as the original mesh, and to centre it to fully utilise the cells in the grid. Without these transformations, the mesh won't fill the grid to its maximum extents, or the mesh could extend out of the grid as seen in Figure 9.

```

1 const vec3 voxelPos = /* Index of this cell */;
2 const vec3 sdfPos = voxelGridInput.cellCount * 0.5;
3
4 for (int i = 0; i < indexCount; i += 3)
5 {
6     // Get indices
7     uint index0 = GetIndex(i + 0);
8     uint index1 = GetIndex(i + 1);
9     uint index2 = GetIndex(i + 2);
10
11    // Get vertices
12    vec4 vert0 = voxelGridInput.transform *
13                vec4(vertices[index0].position, 1.0);
14    vec4 vert1 = voxelGridInput.transform *
15                vec4(vertices[index1].position, 1.0);
16    vec4 vert2 = voxelGridInput.transform *
17                vec4(vertices[index2].position, 1.0);
18
19    // Transform vertices into grid position
20    vec3 pos0 = voxelGridInput.gridScale *
21                (vert0.xyz - voxelGridInput.gridOffset);
22    vec3 pos1 = voxelGridInput.gridScale *
23                (vert1.xyz - voxelGridInput.gridOffset);
24    vec3 pos2 = voxelGridInput.gridScale *
25                (vert2.xyz - voxelGridInput.gridOffset);
26
27    // Create Triangle
28    vec3 a = sdfPos + pos0;
29    vec3 b = sdfPos + pos1;
30    vec3 c = sdfPos + pos2;
31    vec3 normal = normalize(cross((b - a), (a - c)));
32
33    // Compute Triangle Distance from this cell
34    float thisDist = sdfTriangle(
35                voxelPos,
36                a, b, c,
37                normal
38            );
39
40
41    // compare thisDist with other triangles and
42    // find absolute minimum...
43 }

```

**Code Listing 4** – Calculating distance to triangle on each voxel cell. Transforming each triangle to ensure it is inside the grid bounds.



**Figure 9** – Low-poly Stanford bunny as an SDF voxel grid without triangle transformation. Ears and tail are cut off as the triangles are outside the grid bounds.



To speed up voxel grid computation, the *jump flooding* algorithm was implemented. This changes the compute shader where each invocation is at a triangle perspective rather than a cell. An *Axis-Aligned Bounding Box* (AABB) is created that encapsulates the triangle, then iterates through every cell that overlaps the box and calculates its distance, as seen in Code Listing 5. It's important that atomics are used to load and store distances to the `image3D` as multiple triangles may overlap cells leading to threads accessing the image simultaneously.

```

1 int triIndex = int(gl_GlobalInvocationID.x) * 3;
2
3 // Create Triangle - See Code Listing 4 for
4 // creating a triangle from three vertices.
5 vec3 a = sdfPos + pos0;
6 vec3 b = sdfPos + pos1;
7 vec3 c = sdfPos + pos2;
8 vec3 normal = normalize(cross((b - a), (a - c)));
9
10 // Create Triangle AABB
11 ivec3 aabbMin = ivec3(min(a, min(b, c)) -
12                     voxelGridInput.cellSize);
13 ivec3 aabbMax = ivec3(max(a, max(b, c)) +
14                     voxelGridInput.cellSize);
15
16 // Clamp AABB to Grid
17 ivec3 gridMin = max(ivec3(0, 0, 0), aabbMin);
18 ivec3 gridMax = min(voxelGridInput.cellCount,
19                   aabbMax);
20
21 for (int z = gridMin.z; z <= gridMax.z; ++z)
22 {
23     for (int y = gridMin.y; y <= gridMax.y; ++y)
24     {
25         for (int x = gridMin.x; x <= gridMax.x; ++x)
26         {
27             // Compute distance from triangle to cell
28             // using a,b,c & normal (lines 5 to 8).
29
30             // Load current distance stored in cell,
31             // ensure image load atomics are used.
32
33             // Compare current/computed distances, see
34             // Equation 4. Update voxel grid if necessary.
35         }
36     }
37 }

```

**Code Listing 5** – Create a triangle AABB, go through all overlapping voxel grid cells and compare distances (GLSL).

A second compute shader is then dispatched on every cell to check the neighbouring cells in the four cardinal directions and diagonally (multiplied by a step length) for their distances. The algorithm then selects the minimum distance among the neighbouring cells and adds the distance between the current cell and the neighbour to ensure an accurate distance from the current cell position.

The jump flooding algorithm runs iteratively and allows each cell to pass on its distance to other cells in the grid (Rong & Tan, 2006). Any cells that are within a certain distance to the original triangles are treated as seeds and therefore not updated. The number of iterations is calculated as the  $\log_2$  of the maximum dimension of the grid, as outlined by Rong and Tan (2006). The start of each iteration also defines the step length, which is simply the number of iterations minus  $i$  (being the current iteration).

Rendering the SDF voxel grid is simple, as the distance has been precomputed it involves looking up the value stored in the cell the ray is currently positioned in. However, the ray position must be transformed into the local space of the voxel grid to calculate the correct voxel index; as seen in Code Listing 6.

As the grid is stored as an `image3D`, the raymarching shader can sample the grid as a texture through an image sampler. The sampler has the `VK_FILTER_LINEAR` filter set, which is used to interpolate the distance values between voxels automatically, resulting in a smooth SDF.

```

1 float RaymarchMap(vec3 pos)
2 {
3     // Calculate grid values...
4     vec3 gridSize = voxelCellCount / voxelGridScale;
5     vec3 gridHalfSize = gridSize * 0.5f;
6     vec3 center = pos + gridHalfSize;
7
8     // Calculate voxel index...
9     vec3 voxelIndex = (center / gridSize);
10
11    // Texture lookup in voxel grid. Only taking red
12    // component as the image is R32 format.
13    float voxel = texture(voxelGrid, voxelIndex).r;
14
15    // Final computed distance value...
16    float dist = voxel / voxelGridScale;
17    return dist;
18 }

```

**Code Listing 6** – RaymarchMap function for rendering the SDF voxel grid (GLSL).



**Figure 10** – Stanford bunny SDF generated using the jump flooding method.

### 7.3 Binary Space Partitioning Tree

A *Binary Space Partitioning* (BSP) tree is a data structure that recursively subdivides space into two sets using dividing (or partitioning) hyperplanes (Ericson, 2005). During rendering the ray can traverse the tree and calculate the distance to the nearest point on the surface of the mesh. The KD-Tree used by Wald, et al. (2005) is a type of BSP tree, where all the dividing planes are aligned with the coordinate axes.

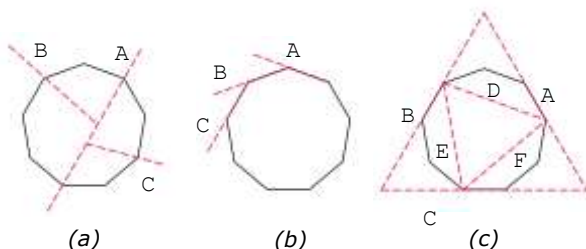
The BSP Tree must be constructed from the mesh on the CPU before it can be traversed during raymarching. The steps to construct a BSP tree are as follows:

1. Select a dividing plane.
2. Partition the geometry into two sets with respect to the dividing plane.
3. Form or add to the tree using the two newly created nodes.
4. Check the new nodes and recursively subdivide again (go back to step 1) or stop BSP tree generation based on some heuristic.

The first step involves selecting a dividing plane that splits the geometry into two sets. There are two options when choosing a dividing plane:

- *General Partitioning Planes* subdivide the geometry evenly and aim to produce a well-balanced tree. This method allows for even traversal through the tree no matter what path the ray takes.
- *Autopartitioning Planes* use a section of the geometry (i.e. a triangle) as the dividing plane. This aims to divide the object from empty space, which allows "early-outs" during ray traversal.

Figure 11 shows the various approaches. With subfigure 10(c) representing a hybrid approach, which attempts to create a well-balanced tree that still allows for "early-outs" during traversal (Ericson, 2005).



**Figure 11** – Various BSP tree partitioning plane approaches. Subfigure (a) represents General planes, (b) represents Autopartitioning planes, (c) represents a mixture of both approaches.

General partitioning planes are created by placing them at the centroid of all the vertices in the current set and orienting them in the direction of the largest variance. This ensures that the plane is positioned evenly between all vertices and divides the object equally. The normal of the plane is determined by using the eigenvector corresponding to the largest eigenvalue of the vertices covariance matrix.

The covariance between two sets of values is calculated using the following formula (Lanhenke, 2021):

$$\sigma(x, y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{n}$$

**Equation 5** – Calculate covariance between two sets of values.

where  $n$  is the number of vertices,  $x_i$  and  $y_i$  are the  $i^{\text{th}}$  component values from the sets being compared, and  $\bar{x}$  and  $\bar{y}$  are the means of the respective sets.

The 3x3 covariance matrix ( $C_{3x3}$ ) for the set of vertices ( $V$ ) is calculated as follows, using Equation 5 to calculate the covariance between each vertex component:

$$C_{3x3} = \begin{bmatrix} \sigma(V_x, V_x) & \sigma(V_x, V_y) & \sigma(V_x, V_z) \\ \sigma(V_y, V_x) & \sigma(V_y, V_y) & \sigma(V_y, V_z) \\ \sigma(V_z, V_x) & \sigma(V_z, V_y) & \sigma(V_z, V_z) \end{bmatrix}$$

**Equation 6** – Calculate the 3x3 covariance matrix for a set of vertices.

Once the covariance matrix is constructed, *eigendecomposition* is used to find the eigenvalues and eigenvectors of the matrix.

An  $n \times n$  matrix  $\mathbf{A}$  is said to have an eigenvector  $\mathbf{x}$  and corresponding eigenvalue  $\lambda$  from an *eigensystem* in the following form (Van Verth & Bishop, 2016):

$$\mathbf{Ax} = \lambda\mathbf{x}$$

**Equation 7**

We can solve for eigenvalues by rearranging **Equation 7**; where  $\mathbf{I}$  is the identity matrix:

$$\mathbf{Ax} = \lambda\mathbf{Ix}$$

or

$$(\lambda\mathbf{I} - \mathbf{A})\mathbf{x} = 0$$

**Equation 8**

Zero vectors are not considered an eigenvector, so **Equation 7** only holds true if:

$$\det(\lambda\mathbf{I} - \mathbf{A}) = 0$$

**Equation 9**

When expanded out, Equation 9 is a polynomial in  $\lambda$  of the  $n^{\text{th}}$ -degree called the *characteristic polynomial* for  $\mathbf{A}$ , where its roots are the eigenvalues. This means there are  $n$  eigenvalues for an  $n \times n$  matrix (Press, Teukolsky, Vetterling, & Flannery, 2007). However, they may not be distinct from one another; the same eigenvalues from multiple roots are called *degenerate*.

Typically, eigenvalues are solved using iterative transformations, such as the Householder method, or the *QR/QL* method. However, these calculations are not numerically robust. Luckily, in this case, Eberly provides a numerically robust noniterative method for solving the characteristic polynomial of a  $3 \times 3$  matrix that is faster than the other methods. **Equation 10** is the expanded out characteristic polynomial for a  $3 \times 3$  matrix, where  $\text{tr}()$  is the trace of the matrix (Eberly, 2014).

$$\det(\lambda I - \mathbf{A}) = \lambda^3 - \text{tr}(\mathbf{A})\lambda^2 + \frac{(\text{tr}(\mathbf{A})^2 - \text{tr}(\mathbf{A}^2))}{2}\lambda - \det(\mathbf{A})$$

**Equation 10**

Using  $C_{3 \times 3}$  as the input matrix ( $\mathbf{A}$ ) and solving for the roots of this polynomial, as outlined in Eberly's paper (2014), gives the three eigenvalues which can each be substituted in Equation 8 to solve for eigenvector  $\mathbf{X}$ . Once the eigensystems have been solved, the eigenvector corresponding to the largest eigenvalue is chosen as the dividing plane normal.

Autopartitioning planes are much simpler to calculate. Firstly, the AABB that contains all the vertices in the current set is found, and the largest axis of the box is selected. Next, all triangles in the mesh are tested against this axis using a dot product. The triangle with a normal facing closest to the axis is chosen as the dividing plane. This approach finds a triangle which divides the object from empty space the most.

Once the plane has been chosen, the object must be split into two sets. There are four ways a polygon can be classified with respect to a plane (Ericson, 2005):

1. Infront of the dividing plane.
2. Behind the dividing plane.
3. Straddling the dividing plane.
4. Coincident with the dividing plane.

Its relatively trivial to classify the triangles with respect to the dividing plane. The signed distance to the plane for each triangle vertex is calculated (Quilez, n.d.); see Equation 11 where  $\vec{N}$  and  $P$  is the plane normal and position respectively,  $t$  is the triangle vertex.

$$d = \vec{N} \cdot (t - P)$$

**Equation 11** - Calculate the signed distance from a point to a plane.

If all three of the triangle vertices share the same sign, it's either in front (when distance is positive) or behind (when distance is negative) of the plane. If there is a mixture of positive and negative signs, the triangle is straddling the plane. The triangle is coincident if the distances are all 0, but in practice this is rare and can be safely ignored and act as if the triangle is either in front or behind.

To solve the issue of straddling triangles, there are two approaches. One approach splits the triangle into two, and they are added to the corresponding nodes with respect to the plane. Another approach adds the triangles to both sides, creating an overlap of the plane. Both approaches were tried during the implementation of the project. With the first approach being dropped as it made the mesh harder to split.

Finally, the tree is built up from the nodes recursively, with the leaf nodes containing a set of triangles from the mesh. Each node is checked against a heuristic to determine if the geometry should be further split, in this case a maximum triangle count.

Raymarching the BSP tree involves traversing the tree at each ray step, which can be a slow process and requires many tree traversals per ray per frame. The tree is traversed to find a leaf node, where the distance to each triangle in this node is compared. Which is done until no more leaf nodes are in the path of the ray.

To try and improve performance, an optimisation to initially raytrace the tree and cache the closest leaf node was added. An implementation outlined in Wald's thesis is used to traverse the tree using raytracing (Wald, 2004). During raymarching, the distance is computed from the singular leaf node and therefore does not traverse the tree again. However, this causes errors during rendering as the ray will never check the distances to other triangles in the mesh, which may be closer.



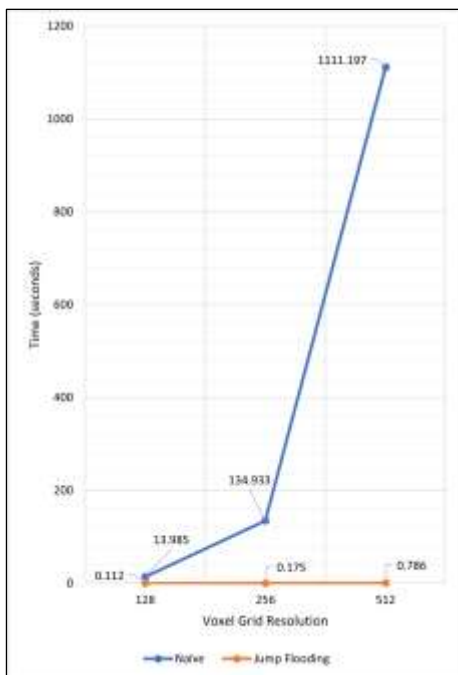
**Figure 12** - Low-poly sphere rendered using an SDF BSP tree.

## 8. Discussion of Outcomes

### 8.1 Evaluation

All performance results were captured on hardware equipped with an *Intel Core i9-9900K CPU* at 3.6GHz, 32GB RAM and an *Nvidia Titan X (Pascal) GPU* with 12GB VRAM.

Figure 13 shows the graphed time for generating an SDF voxel grid using the naïve and jump flooding methods at different grid resolutions from the Stanford bunny model (~14K triangles). The naïve implementation generation time rapidly increases as the resolution is doubled, while the jump flooding approach remained under one second. The jump flooding approach will support higher resolution grids and could theoretically perform this at runtime on dynamic meshes every couple of frames.



**Figure 13** – Time comparison between SDF voxel grid generation methods.

The accuracy of SDF voxel grids relies on two factors: the grid resolution and the size of the input mesh. Lower grid resolutions may result in “blocky” SDFs during rendering due to the fewer cells storing the distance, as seen in Figure 14.



**Figure 14** – Stanford bunny SDF grid at 32<sup>3</sup> resolution.

The grid is also scaled to the size of the mesh to maximize cell usage. Therefore, smaller meshes have smaller cell sizes than larger meshes, meaning the same grid resolution covers a smaller area and results in greater accuracy on small meshes. These parameters need to be adjusted for each mesh to achieve desirable results. For instance, the visual difference between the Stanford bunny at 256<sup>3</sup> and 512<sup>3</sup> resolutions are very minimal, as seen in Table 1.

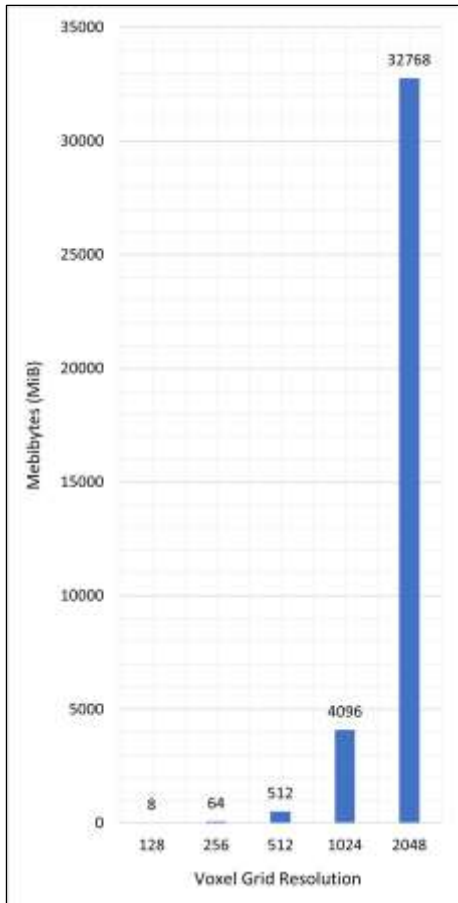
The brute-force approach generally provides more accurate distances compared to the jump flooding method, especially with objects that contain holes or are considered more complex.

Grid Resolution	128 <sup>3</sup>	256 <sup>3</sup>	512 <sup>3</sup>
Naïve Method			
Jump Flooding Method			

**Table 1** – Visual comparison between naïve and jump flooding voxel grid methods with different grid resolutions. (Please zoom; images are high resolution)

Comparing the memory usage of different resolutions is shown in Figure 15. The graph indicates that smaller resolution grids ( $\leq 512^3$ ) have a negligible impact on memory usage, but it increases rapidly as the grid resolution is doubled. A resolution of 2048<sup>3</sup>, for example, results in GPU memory usage above 32GB, making it unsuitable for most consumer-grade graphics devices<sup>1</sup>.

<sup>1</sup> Please note that a resolution of 2048<sup>3</sup> was not tested due to its high memory usage. The value presented in Figure 15 represents the calculated memory usage for an image at that resolution, based on the other results.



**Figure 15** – Voxel Grid Memory Usage (in Megabytes) at different resolutions.

The BSP tree provides perfect accuracy for the distance field as it uses the original triangles from the mesh. This contrasts with the voxel grid method where its accuracy is depended on the grid resolution, as discussed earlier.

However, generation performance is very low. The tree must be prebuilt but cannot take advantage of compute shaders to speed up this process, meaning it's all done sequentially on the CPU. Although, CPU multithreading could be utilised after a split has been made – this is discussed further in the following section.

Dealing with individual triangles that straddle the dividing plane is not ideal. There isn't a "good" way to handle them; splitting the triangle into two creates new triangles to deal with later either during generation or to test against during traversal. Adding the triangle to both sides of the tree means higher memory usage as some triangles are doubled and leads to the distance computation being performed on the same triangle. Both methods make building the tree more complex, and in some situations means the tree may never complete – in one instance it was generating for a few hours and consumed over 97% of memory without completing.

Furthermore, larger meshes were practically impossible to generate useful trees for with the current system. As it's possible to get stuck in infinite loops when adding the same triangles to both nodes. Which lead to only simple meshes being generated with this method.

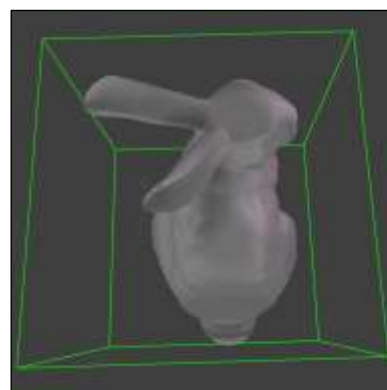
Runtime performance is particularly poor in general with an average of  $0.0498\text{ms}$  frame time, there are a few main reasons that could be behind this. Firstly, tree traversal for raymarching is very slow; the tree must be traversed for each point marched along the ray, for each pixel for each frame. Secondly, traversing a BSP tree causes a lot of branching, which is extremely unforgiving for GPU parallelism due to their lock-stepping thread architecture.

Memory consumption is very high for BSP trees, the triangle mesh must be maintained as well as the BSP tree nodes and each leaf node indices.

## 8.2 Future Work

The jump flooding method for the SDF voxel grid approach could be further improved. Currently, it's the fastest method for generating an SDF representation of a mesh but is the least accurate. Rong and Tan (2006) consider accuracy in their paper and provide some possible improvements that could be implemented.

Memory usage of the voxel grid is also a major topic of future work. Figure 16 shows the bounds of the voxel grid; there is a lot of empty space, and some cells store distances that aren't important. Employing a spatial data structure should be explored, examples include a *Sparse Voxel Oct-tree* or a *VDB Tree* which is highly memory efficient and allows for an average  $O(1)$  random access (Museth, 2013). Both data structures only store data where it's needed, which significantly drops memory usage. In hindsight, perhaps this should have been a major direction of the project after seeing the speed and accuracy of the voxel grid approach.



**Figure 16** – SDF Voxel Grid bounds.

Wald's thesis covers several optimisations for BSP trees, including cache alignment and SIMD traversal, which greatly improved rendering efficiency for meshes. Wald also offers solutions to keep threads in sync, which would help with GPU parallelism. These improvements should be applied to the current BSP tree implementation, which has not yet been heavily optimised.

To further improve the BSP tree, tweaks to the tree generation should be made. Currently, the generated trees are very large and don't evaluate multiple dividing planes to find the most efficient. During generation the process is only utilising an average of 6.7% of the CPU, implementing simple multithreading to split the work between threads could improve performance dramatically and utilise more CPU cores. Theoretically, putting the first split nodes onto their own thread would halve generation time.

Furthermore, dealing with triangles makes the tree awkward as some triangles overlap and belong to multiple nodes which makes future cuts almost impossible. A future direction could be to explore point clouds; in-fact this was started during project development.

Generating a point cloud from a mesh involves randomly sampling a set of points from each triangle using barycentric coordinates, as seen in the following equation:  $P$  being the computed point on the triangle, and  $r_{1,2}$  being the two randomly generated values ( $r_{1,2} \in [0, 1]$ ).

$$P = (1 - \sqrt{r_1})t_1 + (\sqrt{r_1}(1 - r_2))t_2 + (\sqrt{r_1} \times r_2)t_3$$

**Equation 12** - Randomly sample point on triangle.

Once a point cloud is generated, it can be built in a similar manner to the triangle BSP tree. However, since points are not connected by edges like triangles, there is no need to split them or place them on multiple nodes. This makes the construction of the tree simpler and potentially more efficient.

An alternate direction for this project would involve the use of neural networks. These were a key part of the project research, but never implemented within this project. This approach could result in minimal memory usage, which is a major area which is lacking in the current implementations, as the neural network would be the only thing that must be stored once it has been trained. It can also represent an SDF with minimal errors (Zeng, 2018).

### 8.3 Discussion & Reflection

This project investigates the representation of meshes as signed distance fields, through the exploration of three methods, each with their own pros and cons. SDFs are widely used in game engines for various applications such as

collision detection and ambient occlusion (Epic Games, 2021). While the methods explored in this project provide fast and accurate representations of meshes as SDFs, they also require high memory usage, making them impractical for real-world game engines in their current state. Therefore, future work is required to improve the memory efficiency of these methods (as mentioned in section 8.2) to make them more viable in game engines.

To date, SDFs are not commonly used as a replacement for meshes in games, with only a few recent games such as Claybook (Second Order, 2018) utilising them. Within this game only simple shapes are represented; see Figure 17. With new explorative techniques at representing complex objects as SDFs, they would become more popular in games. SDFs can also provide unique shapes with the mathematical operations, such as blending and twisting. Additionally, their capability to represent  $n$ -dimensional objects adds potential for unique game mechanics in the future.



**Figure 17** – Screenshot from Claybook (Second Order, 2018). A dynamic world sculpted entirely from SDFs.

Looking back at the originally proposed research questions, various techniques were explored to represent meshes as SDFs. Each of the methods supported different levels of accuracy, runtime, and generation performance, which addressed the key point from Q1. However, reducing memory consumption was not explored in any of the proposed techniques, making it a crucial area for future work.

Optimisations, specifically Over-Relaxation, was implemented to improve raymarching performance (Q2). Although, it's unclear if this technique had a big positive improvement on performance. Other techniques, such as coarse cone tracing, should be investigated to evaluate their potential benefits for raymarching.

The project fully explored parallelism using GPU compute shaders (Q3), which significantly improved the speed of voxel grid generation and enabled concurrent computation. Future work on parallelism could involve the use of CPU multithreading to speed up computations and the exploration of SIMD and hardware intrinsics to improve CPU performance.

## 9. Conclusion & Recommendations

The jump flooding voxel grid proved to be the strongest method among the implemented approaches. It is very quick at generating the SDF voxel grid and provides adequate accuracy for most use cases (for example, ambient occlusion). If higher accuracy is needed and the generation process is not taking place on client hardware, the brute force voxel grid approach is recommended as it provides instantaneous lookup speed at  $O(1)$ , unlike the BSP tree which requires traversal. The BSP tree is recommended when perfect accuracy is essential and when runtime performance is not a primary concern (for instance, in non-real-time applications). But a lot of time tweaking the parameters of tree generation is needed to ensure the tree builds properly. Ultimately, the choice between the approaches will depend on the specific needs and constraints of the application.

Although this project successfully implemented these techniques, there is still room for improvement and future work (as discussed in section 8.2). This could include modifications to the voxel grid data structure to improve memory usage, as well as exploring alternate approaches such as machine learning and neural networks.

The approaches explored allow for complex objects to be represented as signed distance fields and may allow more games in the future to use them to represent objects instead of meshes.

In conclusion, this project successfully explored various techniques for representing triangle meshes as signed distance fields. Three working methods were developed, each with a varying level of accuracy, generation speed, and runtime performance, that can be applied to different needs and requirements.

## 10. References

- Aaltonen, S. (2018, July 9). *YouTube*. Retrieved from Digital Dragons 2018: Sebastian Aaltonen - GPU based clay simulation and ray tracing tech in Claybook: <https://www.youtube.com/watch?v=Xpf7Ua3UqOA>
- Bærentzen, A., & Aanæs, H. (2002). Generating Signed Distance Fields from Triangle Meshes.
- Bærentzen, A., & Aanæs, H. (2005). Signed Distance Computation Using the Angle Weighted Pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11(3).
- Blanco, V. (n.d.). *VkGuide*. Retrieved 11 21, 2022, from <https://vkguide.dev/>
- Eberly, D. (2014). *A Robust Eigensolver for  $3 \times 3$  Symmetric Matrices*.
- Epic Games. (2021). *Mesh Distance Fields*. Retrieved 10 17, 2022, from <https://docs.unrealengine.com/5.0/en-US/mesh-distance-fields-in-unreal-engine/>
- Ericson, C. (2005). *Real Time Collision Detection*. Morgan Kaufmann.
- Fryazinov, O., Pasko, A., & Aszhiev, V. (2011). BSP-Fields: An exact representation of polygonal objects by differentiable scale fields based on binary space partitioning. *Computer-Aided Design*, 43, 265-277.
- Giesen, F. (2011, July 10). *The ryg blog*. Retrieved from A trip through the Graphics Pipeline 2001, part 8: <https://fgiesen.wordpress.com/2011/07/10/a-trip-through-the-graphics-pipeline-2011-part-8/>
- Hart, J. C. (1995). Sphere Tracing: A Geometric Method for the Anti-Aliased Ray Tracing of Implicit Surfaces. *The Visual Computer*, 12.
- Hart, J. C., Sandin, D. J., & Kauffman, L. H. (1989). Ray Tracing Deterministic 3-D Fractals. *Computer Graphics*, 23(3).
- Jones, M. W., Bærentzen, J. A., & Sramek, M. (2006). 3D Distance Fields: A Survey of Techniques and Applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4), 581-599.
- Keinert, B., Schafer, H., Korndorfer, J., Ganse, U., & Stamminger, M. (2014). Enhanced Sphere Tracing. *Smart Tools and Applications in Graphics*.
- Kleineberg, M., Fey, M., & Weichert, F. (2020). Adversarial Generation of Continuous Implicit Shape Representations.

- Lanhenke, M. (2021, December 27). *Implementing PCA From Scratch*. Retrieved from Towards Data Science: <https://towardsdatascience.com/implementing-pca-from-scratch-fb434f1acbaa>
- Museth, K. (2013). VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Trans. Graph.*
- Overvoorde, A. (2016). *Vulkan Tutorial*. Retrieved 11 19, 2022, from <https://vulkan-tutorial.com/>
- Park, J. J., Florence, P., Straub, J., Newcombe, R., & Lovegrove, S. (2019). DeepSDF: Learning Continuous Signed Distance Fields for Shape Representation.
- Payne, B. A., & Toga, A. W. (1992). Distance Field Manipulation of Surface Models.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical Recipes: The art of scientific computing*.
- Quilez, I. (2013). *ShaderToy - Raymarching Primitives*. Retrieved 10 16, 2022, from <https://www.shadertoy.com/view/Xds3zN>
- Quilez, I. (n.d.). *Distance Functions*. Retrieved 10 16, 2022, from <https://iquilezles.org/articles/distfunctions/>
- Rong, G., & Tan, T.-S. (2006). Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. *Interactive 3D graphics and games*, 109-116.
- Second Order. (2018, August 31). Claybook. Second Order.
- Sellers, G., & Kessenich, J. (2016). *Vulkan Programming Guide*. Pearson Education.
- The Khronos Group. (2022). *Vulkan Docs*. Retrieved 12 01, 2022, from GitHub: <https://github.com/KhronosGroup/Vulkan-Docs>
- The Khronos Group. (2022). *Vulkan Specification*. Retrieved 11 25, 2022, from <https://registry.khronos.org/vulkan/specs/1.2-extensions/html/vkspec.html>
- Van Verth, J. M., & Bishop, L. M. (2016). *Essential Mathematics for Games and Interactive Applications*.
- Wald, I. (2004). *Realtime Ray Tracing and Interactive Global Illumination*.
- Wald, I., Friedrich, H., Marmitt, G., Slusallek, P., & Seidel, H.-P. (2005). Faster Isosurface Ray Tracing Using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5).
- Wang, Z., Vandersteen, C., Demarcy, T., Gnansia, D., Raffaelli, C., Guevara, N., & Delingette, H. (n.d.). A Deep Learning based Fast Signed Distance Map Generation.
- Wu, J., & Kobbelt, L. (2003). Piecewise Linear Approximation of Signed Distance Fields.
- Zeng, Z. (2018). *Approximating Signed Distance Field to a Mesh by Artificial Neural Networks*.

## 11. Bibliography

- Eberly, D. (2007). *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics. Second Edition*. Morgan Kaufmann.
- Gregory, J. (2019). *Game Engine Architecture. Third Edition*. CRC Press.
- Korndorfer, J., Keinert, B., Ganse, U., Sanger, M., Ley, S., Burkhardt, K., . . . Heusipp, J. (2015). *HG\_SDF: A glsl library for Building signed distance functions*. Retrieved 10 17, 2022, from [https://mercury.sexy/hg\\_sdf/](https://mercury.sexy/hg_sdf/)
- Lanhenke, M. (2021, December 29). *Understanding the Covariance Matrix*. Retrieved from Towards Data Science: <https://towardsdatascience.com/understanding-the-covariance-matrix-92076554ea44>
- Lengyel, E. (2019). *Foundations of Game Engine Development. Volume 2: Rendering*. CRC Press.
- Strain, J. (1999). Fast Tree-Based Redistancing for Level Set Computations. *Journal of Computational Physics*, 152(2), 664-686.
- The Khronos Group. (2022). *Vulkan Learn*. Retrieved 12 01, 2022, from <https://vulkan.org/learn>
- Williams, A. (2019). *C++ Concurrency In Action. Second Edition*. CRC Press.



## Appendix: Assets & Third-Party Code

Project assets:

- **Stanford Bunny 3D Scan** - Stanford University, license: [Stanford Scan](#)
  - Rexported as a GLTF model using Blender.
  - A low-poly version created in Blender by the author.
- **Sponza 3D Scene** – Academy Software Foundation License (free for research & education)
  - Rexported as a GLTF model using Blender.
- **Low-Poly Sphere** – Created by the author in Blender.
- **Yokohama Skybox** – Emil Persson, license: [Creative Commons Attribution 3.0 Unported](#)

Third-Party code and libraries:

- **stb Image Library** – Sean Barrett, License: [public domain / MIT](#)  
<https://github.com/nothings/stb>
- **OpenGL Mathematics (GLM) Library** – License: [The Happy Bunny License / MIT](#)  
<https://github.com/g-truc/glm>
- **ImGui** – Omar Cornut, license: [MIT](#)  
<https://github.com/ocornut/imgui>
- **RapidJSON** – Tencent, license: [MIT](#)  
<https://github.com/Tencent/rapidjson>