School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Signed Distance Field Generator

Zechen Geng

Submitted in accordance with the requirements for the degree of High-Performance Graphics and Games Engineering MSc

2021/2022

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Project Report	Report	SSO (12/08/22)
Source Code	GitHub Repository	Supervisor, Assessor (12/08/22)

Type of project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student)

Zechen Geng

Summary

In recent years, signed distance field (SDF) technology has developed in the computer graphic industry. The value of this technology has been proven in many fields, including ray-marching, soft shadow and collision detection. Besides, SDF has been applied to successful commercial game engines like Unreal and film special effects software like Houdini.

This project aims to implement an application which generates the SDF for 3D models through modern Graphic API. The understanding of SDF will be acquired and enhanced during the implementation process. This report covers a brief introduction of SDF, the reason for chosen technologies and the details of implementation and evaluation, which is vital for explaining existing functionality and possible future software refactoring.

Acknowledgements

First and foremost, I would like to thank Dr Hamish Carr, who provided valuable knowledge of geometry processing in the past year. During the process of this project, his suggestions helped me to achieve the targets correctly and quickly. I also want to thank Dr He Wang and Professor Markus Billeter for supporting and providing straightforward and valuable advice to me on academic writing.

Contents

Chapter 1

Introduction

1.1 Context

The signed distance field (SDF) is a solution for solving boundary problems. The research of SDF in Computer Graphics, especially for 3D shapes representation, has developed [47]. For example, collision detection [43] [25], liquid simulation [23] and ray-marching [44]. In recent years, related research results have been applied in the industry, for example, the Unreal Engine [31].

1.2 Project Aim

The main aim of this project is to build a tool to generate a signed distance field (SDF) for three-dimensional objects and implement a suitable SDF generate algorithm. The tool is expected to visualize the SDF calculate result as well.

1.3 Objectives

- Build a application that can read in and visualize given triangular meshes.
- Implement a SDF generating algorithm.
- Build a application to visualize the SDF compute result
- Comment the code properly for potential reuse
- Test the tool on multiple devices

1.4 Deliverables

- A application that generate Signed Distance Field.
- A GitHub repository that contains the source code of the tool.
- The MSc project report

1.5 Project Plan

The project will be implemented through the waterfall model as it has precise requirements and is easy to implement [4]. Every new feature will be implemented after the last phase is completely finished. The first phase is research and design. It takes four months to ensure that this project's requirement is rational and realistic. The programming work starts from the middle of June to early August. Most time is assigned to algorithm implementation, and then most time will be used to evaluate and write the report.

The Gannt chart of time management for this project is shown in figure 1.1.

Task	Start	Finish
Scope and Plan	**'22 Feb 22**	**'22 Mar 25**
Choose Research Direction	'22 Feb 22	'22 Mar 7
Write S&P Draft	'22 Mar 16	'22 Mar 21
Finalise S&P Document	'22 Mar 23	'22 Mar 25
Background Research	**'22 Mar 26**	**'22 Apr 25**
Background Research	'22 Mar 26	'22 Apr 13
Research Summarise	'22 Apr 14	'22 Apr 25
Software Development	**'22 Apr 26**	**'22 Aug 1**
Software Design	'22 Apr 26	'22 Apr 30
User Interface	'22 Jun 16	'22 Jun 23
Triangular Meshes Reader	'22 Jun 24	'22 Jun 30
SDF Algorithm Implementation	'22 Jul 1	'22 Jul 18
Bug Fix	'22 Jul 16	'22 Jul 19
Evaluation	'22 Jul 20	'22 Jul 26
Bug Fix	'22 Jul 26	'22 Aug 1
Write Report	**'22 Jun 7**	**'22 Aug 10**
Prepare Progress Report	'22 Jul 4	'22 Jul 11
Write Dissertation	'22 Jun 7	'22 Aug 10

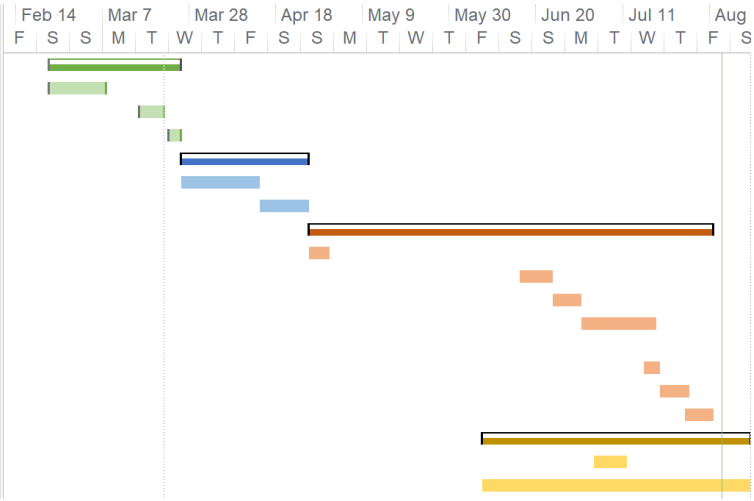


Figure 1.1: The time management Gannt chart

1.6 Report Structure

This report is divided into six chapters.

The first chapter introduces the project aims and objectives and discusses potential ethical, legal and social issues. The second one presents the background research for the project, including a survey of past SDF-related research and the methodology and technologies used in the project. The third chapter documents the design of the software product, while the fourth one is about detailed implementation. The contents of chapter five illustrate the test and evaluation results. Finally, the conclusion and potential improvements can be found in the sixth chapter.

1.7 Ethical, legal, and social issues

The SDF generate software product of this project only reads in and processes triangular meshes; it does not collect personal data at any time. Therefore, there are no significant ethical or social issues.

Several open source third-party libraries have been contained and invoked in the source code to save development time and use test data sets quickly. All usage of the libraries followed its licenses to avoid potential plagiarism and legal issues.

Chapter 2

Background Research

2.1 Literature Survey

2.1.1 2D Distance Field

The term "Distance Field" or "Distance Function" was first introduced by Rosenfeld and Pfaltz [65] in 1966, which was applied to solve digital image processing problems. Therefore, the initial research on Distance Field is related to two-dimensional image processing.

After two years of the first announcement, the distance function algorithm of Rosenfeld and Pfaltz was announced[64]. Maurer [55]introduced a linear algorithm for Euclidean Distance computing, which is used to generate the Voronoi diagram. More recently, in 2007, Green [32] from Valve shared a method of rendering anti-aliased fonts in their game product Team Fortress 2 through alpha-testing techniques. The distance function technique played a vital role in many subdivisions of image processing in the past years.

2.1.2 3D Signed Distance Field

With the development of computer graphics, the significance of three-dimensional objects increased quickly, and the distance field technique has been applied to deal with three-dimensional problems [12] [47]. Jones et al. [47] have done a comprehensive review in 2006, contributing to understanding the early research of the three-dimensional distance field. In recent years, more researchers have noticed the value of SDF and explored the technology from different aspects. Most of the researches centre on finding better SDF generate methods, while some other research explored the potential enrichment of SDF for other technologies.

Sanchez et al. [67] compared the efficiency of SDF generation by using different optimizing strategies to build Bounding Volume Hierarchy (BVH) data structure. Besides, they also explored the acceleration performance of GPU implementation.

The research by Koschier et al. [52] focuses on the generation aspect. Their algorithm is able to generate more accurate and continuous SDF with lower memory costs. Besides, to prove the practicability, the authors' teams applied their method's result to complex collision detection simulation scenarios and showed satisfying performance.

Bán and Valasek [9] have published an SDF generation algorithm called First Order in 2020, which used first degree Taylor approximation to generate a signed distance field. As they proved, their method can reduce storage consumption and improve rendering efficiency while maintaining accuracy. Besides, the approximation solution may be able to be used on other functions.

Meshed reconstruction is also a vital application of SDF, and several research papers discussed this direction.

The research of Barill et al. [5] discussed the significance of winding number for solving the fundamental problem of whether a point is the interior or exterior of a three-dimensional object. They explored a fast algorithm to test points through the winding numbers for triangle soups or point clouds. As they proved, generating signed distance field for soups and point clouds became possible by using their method, while the same problem is recognized as complicated to implement in the past.

Combining the deep learning technique with 3D shape reconstruction is a new potential research orientation for computer graphics. Park et al. [58] introduced a learning reconstruction method named DeepSDF. The key to their solution is using a deep neural network to complete the learning of signed distance function from different representations like point clouds and reconstruct continuous three-dimensional surfaces through DeepSDF.

Seyb et al. [69] mentioned the gap of implicit representation in deformation and animation. The SDF technology helped them explore a new non-linear sphere tracing solution technique for deform modelling, which improved the performance of existing deformation techniques. Besides, they hope their research makes the industry focus on the potential of implicit representation like SDF.

The signed distance field also inspires some research. For instance, the research of Sanchez [66] mentioned the problem that the signed distance field of polygon meshes is sometimes not C^1 continuous. They explored the convolution filtering of SDF as a new solution for modelling.

2.2 Methods and Techniques

2.2.1 Signed Distance Function

For an arbitrary point p and a three-dimensional mesh Σ in space, the unsigned distance function of p is defined as the shortest distance from p to the mesh object, to be specific, the closest point in Σ [47].

The formula is as 2.1:

$$\mathrm{dist}_{\Sigma}(\mathbf{p}) = \inf_{\mathbf{x}\in\Sigma} \|\mathbf{x} - \mathbf{p}\| \tag{2.1}$$

Generally, the signed distance function is used for solid meshes. Assuming that the mesh Σ is a solid mesh S, its boundary can be represented as ∂S [47], to represent whether the point p is inside or outside the mesh, the sign function is necessary, which is defined as 2.2:

$$\mathrm{sgn}(\mathbf{p}) = \begin{cases} -1 & \text{if } \mathbf{p} \in S \\ 1 & \text{otherwise} \end{cases} \tag{2.2}$$

Finally, the complete formula of the signed distance field is as 2.3:

$$\mathrm{d}_S(\mathbf{p}) = \mathrm{sgn}(\mathbf{p}) \inf_{\mathbf{x}\in\partial S} \|\mathbf{x} - \mathbf{p}\| \tag{2.3}$$

2.2.2 Computing Methods of Signed Distance Fields

Frequently, the most effective solution for three-dimensional representation is using triangular meshes. Therefore, generating a distance field for triangular meshes is a developed direction for the three-dimensional signed distance field research. For SDF generating, the mesh needs to be a closed and watertight manifold [47]. The following discussion concentrate on this type of object.

2.2.2.1 Distance Function Computation

To compute a whole distance field, first, we have to solve the problem of calculating the distance from a single point to a single triangle.

Jones [48] has introduced two approaches to solve the problem. One is to calculate the distance from the point to its projection on the triangle plane, which can be seen as 2.1, and another is to convert the problem to a two-dimensional problem by using affine transformations.

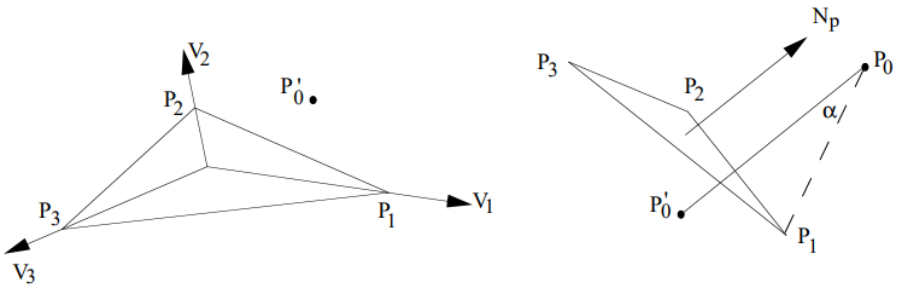


Figure 2.1: Calculating distance through projection by Jones[48]

Eberly [22] introduced a computation solution that uses the vector technique. He parameterized the triangles $\mathbf{T}(s,t)$ as the weighted sum of two edges and a point as formula 2.4:

$$\mathbf{T}(s,t) = \mathbf{B} + s\mathbf{E}_0 + t\mathbf{E}_1 \tag{2.4}$$

for $(s,t) \in D = \{(s,t) : s \geq 0, t \geq 0, s+t \leq 1\}$, then the shortest distance can be found through calculate the minimal value of the function 2.5:

$$\mathbf{Q}(s,t) = |\mathbf{T}(s,t) - \mathbf{P}|^2 \tag{2.5}$$

Ericson [24] applied the Barycentric coordinates of the triangles to compute the distance. This author's method computes the distance by calculating the barycentric coordinates of point P first and then testing which Voronoi region does the projection of p located to compute the distance.

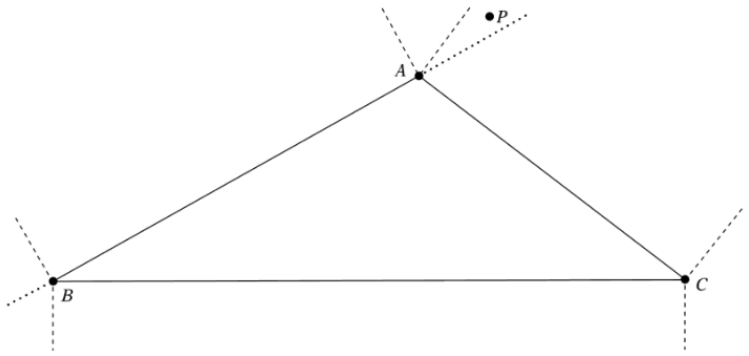


Figure 2.2: The situation of point p's projection lies on Voronoi region CA when ABC is a obtuse triangle that introduced in Ericson [24]'s paper.

Ray tracing is also a direction for solving the triangle distance problem. Tomas Akenine-Möller and Ben Trumbore [2] introduced a solution which avoids pre-compute whether the ray intersects with the triangle plane. This method can compute the distance value through matrix manipulation rather than square root computation.

2.2.2.2 Sign Function Computation

Payne and Toga [59] introduced a basic method for sign computation through scanning. The dot product of surface normal $\vec{n}$ and direction vector $\vec{d}$ can be used to test the interior or exterior problem. However, they also noticed that this method is not always working for C^1 discontinuous surfaces, which is the property of most polygonal meshes.

An alternative is to replace the surface-normal with angle weighted pseudo-normal, which was introduced by Baerentzen and Aanaes [3] in 2005. They first extend the angle weighted pseudo-normal of vertices introduced by Thürrner and Wüthrich[72] to edges and faces, then applied the upgraded method to the signed function computation algorithm. However, Xu and Barbič [76] observed that this approach did not improve the efficiency of their research.

The ray tracing method [2] mentioned in 2.2.2.1 also provide a solution for testing the direction of ray and triangle intersection. If we use this method to compute the distance, then the result of normal can be reused during the sign judgement process.

2.2.3 Algorithms

After calculating the signed distance for a single triangle, the problem becomes how to compute oriental distance values for point sets.

Payne and Toga [59] introduced the naive or brutal-force method, which simply computes distances for all points in the space. The pseudo-code is shown as Algorithm 1:

Algorithm 1 Brutal Force SDF generation

```
for each point(x,y,z) do
    dist = ∞
    for = each triangle t do
        newdist = distance_to_triangle(point, t)
        if abs(newdist)<abs(dist) then
            dist = newdist
        end if
    end for
end for
```

Inigo Quilez, the developer of ShaderToy [62] published the SDF calculation method for most of the basic geometric shapes like Boxes and Capsules [61], but decomposing the arbitrary objects into fundamental shapes is not realistic.

Both Frisken et al. [30], and [49]'s researches used the sampling method to generate the signed distance without computing every triangle. For this project, it can be a potential solution. Specifically, the testing can be used through the ray intersection test. A number of equally distributed sample rays should be generated for a single sample grid to test the intersection with models. Regarding the grid as a sphere, the sample ray can be generated by dividing the sphere equally, Ahn [1] provided a method to calculate the coordinates of arbitrary points on the sphere by splitting the sphere. This method can be used to generate the sample rays for any direction.

Building SDF for objects is time-consuming, especially when the object has a large number of triangles. Fortunately, as for a similar technique, ray tracing, the solution for the same problem has been explored for a long time and can apply to the signed distance field.

To reduce the count of the traverse, we need to reduce the range of computing. Bounding volume is a widespread solution. There are several choices for constructing a bounding volume for three-dimensional meshes. The bounding sphere is the most simple choice. However, using a sphere as a bounding volume is usually unbefitting for three-dimensional objects. The distance computing through the bounding sphere contains expensive square root computation to reduce the efficiency [67]. The Axis-aligned bounding box (AABB) has developed research, which is widely used for ray tracing[45]. This structure has a good balance of generation time and quality. Therefore, AABB is the most popular choice for acceleration [67].

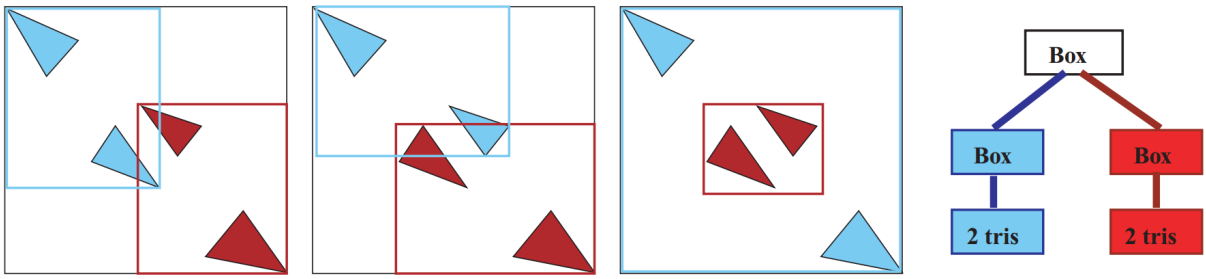


Figure 2.3: Different BVHs for 4 triangles provided by Wald [75]

Similarly, the acceleration structure for ray tracing can be extended to SDF computation. There are two primary tree-structured options: Bounding Volume Hierarchy (BVH) and K-dimensional Tree, representing two different solutions directions.

BVH split the whole object according to its primitives, for this project, triangles. Wald [75] gives a brief explanation of BVH in his research. When building a BVH, it is recursively creating newer and smaller bounding volumes according to the size of

primitives for every children node; as a result, the edge of children nodes' bounding volume can be different from its parent. Each leaf node store the bounding volume for different primitives. Ideally, different primitives will only be saved to one single leaf node and can be traversed through parent nodes. Strategies for building BVH are illustrated in figure 2.3.

The principle of the KD-tree building is similar to BVH, except the method of splitting the space is different. The KD-tree structure is mostly built using AABB bounding volume [26]. Building a KD tree for space will recursively divide the whole space with the position of primitives alongside the axis. The bounding box of the children nodes must share some edges with its parent nodes. When ray tracing through the KD-tree structure, the algorithm will test whether the ray intersects with the bounding box. If a ray only intersects one side of a box, then the ray must shoot on a primitive contained in the box node; the traverse will continue until the correct leaf node is found [26]. In the comparison of Havran[45], this structure is the best method for CPU implementation. As for SDF generation, Inigo Quilez has a WebGL implementation on ShaderToy [63], which proved that the solution fits the requirement of SDF computation. An example given by Foley [26] of KD-tree ray tracing showed in figure 2.4.

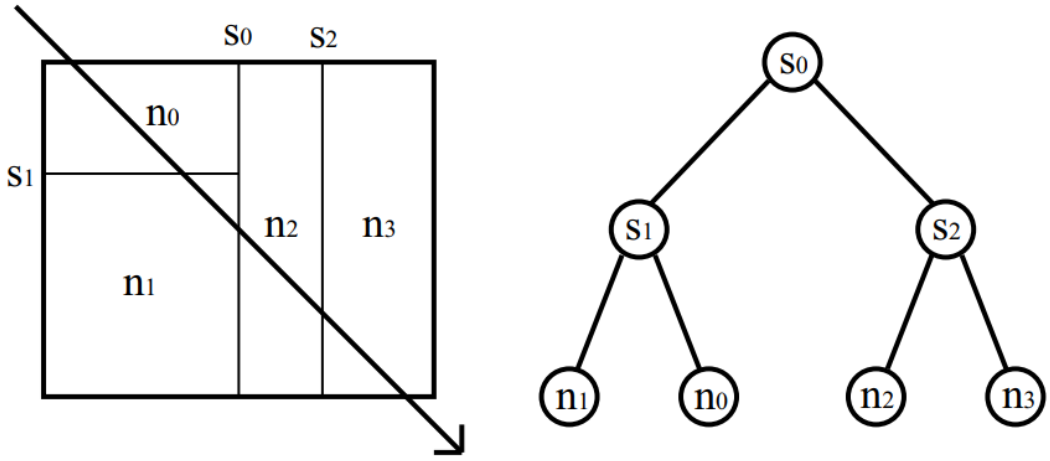


Figure 2.4: Left: The principle of kd-tree ray tracing. Right:The structure of the same tree

2.2.4 Technologies and Tools

2.2.4.1 Graphics APIs

In computer graphics, the Graphics Application Programming Interface, or Graphics API, plays a vital role in determining how the GPU manipulate the rendering tasks. For the graphics programmers, a friendly API will increase the development efficiency.

Three Graphics APIs are widely used: OpenGL, Vulkan and Direct3D.

OpenGL Open Graphic Library or OpenGL [33], is a cross-platform and language-independent Graphics API, which was initially developed by Silicon Graphics, Inc. (SGI) [51]. Khronos Group has continued to manage the development of this API since 2006. Its Version 1.0 was released in July of 1994, and the latest version was updated to 4.6. OpenGL is able to render both 2D images, and 3D graphics through a graphics processing unit (GPU) [51]. Although C++ is the most chosen programming language for developing OpenGL applications, using JavaScript, C and Java to create OpenGL applications on IOS, Android or web browsers are available.

Direc3D Direct3D [56] is the main competitor of OpenGL and Vulkan created by Microsoft. It only runs on Microsoft environments like Windows, Xbox and UWP applications. As a low-overhead Graphics API, Direct3D allows the programmer to control synchronization and multi-threading; compared with standard APIs, it has higher efficiency and saves more resources [6] [7].

Vulkan Vulkan is the newest Graphics API, which was developed by Khronos Group and first released in February of 2016. Vulkan was positioned as the successor of OpenGL, it keeps cross-platform capabilities and aims to create the graphic applications with complex, and numerous datasets [68]. Therefore, Vulkan will be widely used in industry in the future. Vulkan gives the programmer a high degree of freedom during the development process; the responsibility of synchronization, scheduling or memory management and many other features belongs to the programmer, which also means that compared with OpenGL, Vulkan gives higher requirements on the experience for application writers [68].

2.2.4.2 Languages

C++ C++ is a general-purpose programming language based on the C programming language [13], developed by Bjarne Stroustrup and released in 1985 [71]. C++ is widely used in the graphics industry. All of the Graphics APIs mentioned in 2.2.4.1 provide support for C++, and Vulkan only officially provides C++ wrapper support since the property of memory management.

Python Python [29] is a popular interpreted, general-purpose programming language developed by Guido van Rossum around 1990 [53]. With the development of deep learning, many popular and friendly python frameworks and libraries have been created. For example, TensorFlow [8] and Keras [50]. There are already successful examples applying Deep Learning technology to graphics like NVIDIA's DLSS (Deep Learning

Super Sampling) [57]. Moreover, as mentioned in 2.1.2, the research of Park [58] about signed distance field was based on deep learning technology.

2.2.4.3 Graphical User Interface libraries

GLFW GLFW [11] is a cross-platform C library for OpenGL, OpenGL ES and Vulkan. This library allows programmers to create and handle application windows, contexts and surfaces, user input and many other functions.

ImGui Dear ImGui [18] is a open-source C++ graphical user interface library developed by Omar Cornut. The core source code of Dear ImGui is only contained in several platform-independent files. Therefore, it is portable and easy to be compiled with other application codes. Most of the mainstream Graphics APIs, including DirectX, OpenGL, Vulkan, and platforms including Windows, OSX are supported. Dear ImGui also provides useful extensions, including Text Editors and Animation Editors.

Qt Qt [17] is a robust cross-platform UI framework for C++ [70], which was initially developed by Haavard Nord and Eirik Chambe-Eng in 1990, and the responsibility of development belongs to The Qt Company. Qt provides a series of tool kits for UI development, including declarative language QML [15], compilation tool qmake [14] and an integrated development environment (IDE) Qt Creator [16]. The programmer is able to create a complete application only in Qt environment.

2.2.4.4 Building Project

Premake Premake [60] is open-source utility software for source code building. Lua [46] Script is used for premake to store the information of software projects. Premake use the command line to read the project configuration for different platforms from a Script and generate project of development IDEs

CMake CMake [10] is an open-source compiler-independent for source code building, testing and packaging. It is mostly used for C and C++ languages. A C++ compiler like MinGW is necessary for project building. CMake also supports mainstream IDEs like Visual Studio and Xcode.

2.2.4.5 Model file formats

OBJ OBJ [28] file is a geometry file format developed by Wavefront Technologies for its Advanced Visualizer animation tool. This format contains the positions of vertices, UV mapping information, vertex normals and the list of each polygon face. OBJ files store the information of vertex in counter-clockwise order. Wavefront Technologies also

developed the Material Template Library (MTL) format for OBJ to store the material properties of objects.

FBX FBX (FilmBox) [27] is a file format for 3D models, which was initially developed by Kaydara for MotionBuilder and owned by Autodesk in 2006. FBX file is a tree-structured document and can be used for animation by storing the motion nodes.

2.3 Choice of Methods

2.3.1 Choice of Method and Algorithm

Considering about the difficulty of implementation and time limitation, several methods and algorithms are chosen for this project.

First, the field generation method is sampling the points in a limited bounding box range and calculating the distance values for them through a number of sample rays; when setting the appropriate parameters (e.g. resolution, number of sample rays), we can get a satisfying result. Second, the chosen method of computing signed distance values is the ray tracing solution [2] mentioned in 2.2.2.1, because this method solves the distance function computation problem through matrix rather than square root computation, which will get the exact value with relatively low consumption. Then, the acceleration strategy is applied KD-tree data structure alongside the AABB bounding volume during the process of constructing the signed distance field, which is a widely used space acceleration structure; the successful commercial game engine Unreal Engine uses this structure to generate SDF [31]. Finally, using multi-threading programming to improve efficiency. Undoubtedly, GPU implementation is a better way for acceleration for generating SDF. However, the HPG courses are not cover related content like CUDA programming; therefore, GPU implementation will be a tricky method for this project and may result in the project not being completed on time. Besides, avoiding GPU implementation brings better compatibility since there will be no requirement for specific hardware.

2.3.2 Choice of Technologies

2.3.2.1 Language and Graphics API

The libraries, frameworks and APIs mentioned in 2.2.4 provide support for C++. Besides, the compatibility and efficiency of C++ satisfied the requirement of Graphics application development. Therefore, C++ becomes the chosen language for this project.

OpenGL and Vulkan are covered by High-Performance Graphics and Games Engineering MSc modules, while Direct3D is not. Direct3D obviously is a risky option. Besides, considering the time limitation and the requirement of Vulkan API, OpenGL becomes the final chosen API for this project because of its compatibility and relatively easy to implement.

A review of the OpenGL graphics pipeline is helpful for further application design.

As shown in figure **??** [21], the pipeline is divided into several stages; each stage will collect the output data from its last stage and process the data for the next.
The first stage is named Vertex Specification. Two concepts should be cleared before continuing to introduce the pipeline. Vertex Buffer Objects (VBO) are the buffers containing a number of information (e.g. positions, normals, colours) of vertices in video memory [42]. Vertex Array Object (VAO) contains at least one VBO as well as the definition of vertex data [42]. The GPU use VAO and VBO to analyze the vertex data for rendering.

Vertex Shader Vertex Shader is a programmable stage which can be used to process the attributes of vertex [41]. The vertex shader is mostly used to make transformations of coordinates for vertex data, including local-world transformation, view transformation and clipping. Tessellation [40] and Geometry Shader [36] are two optional stages; the former can divide data into smaller primitives, while the latter can modify the input to geometric shapes like triangles.

Primitive Assembly Primitive Assembly [38] aims to modify the former primitives to a sequence of basic primitives like lines. When Tessellation or Geometry Shader is active, an early primitive assembly process will happen before both [41].

Rasterisation Then it comes to Rasterisation [39]. The output of Primitive Assembly is primitives; they will be processed to screen position by perspective and viewport transformation. The output of this stage is screen coordinates, to be specific, the position of pixels. Those pixels (fragments) will be sent to the fragment shader for colour computation in the next stage.

Fragment Process Fragment Process [39] is another programmable stage of the pipeline. Colours, depth values and possible stencil values will be computed in Fragment Shader [35] and set as the output of this shader, which will be passed to the final testing and blending stage [37] , where the scissor test, alpha test and depth test will happen. Finally, the data will be passed to the framebuffer and shown on the screen.

2.3.2.2 GUI

The user interface of the final application only needs a panel to show buttons, checkboxes and text information. Considering the complexity of setting up the development environment and writing Qt code, using Qt is too obese for this project. Dear ImGui is enough to implement the features mentioned above and can also avoid the potential problems caused by different Qt environments. Therefore, the project will be developed through Dear ImGui.

Without Qt, the only option for application window control API is GLFW. Finally, the GUI will be created through GLFW alongside Dear ImGui.

2.3.2.3 Building Project

To generate a project through CMake, the user must install and set the appropriate environment for this tool kit. However, some potential problems will happen in specific situations, like the version differences on different devices. Compared with CMake, Premake only needs to define the configuration of the project when creating Lua script, which also enhances the understanding of project structure. Therefore, this project chose Premake as the tool for building project.

2.3.2.4 Model file formats

There is no requirement to make animation for the project; the OBJ file satisfies the load test models' demand and becomes the chosen format.

2.3.2.5 Third Party Libraries

Moreover, some third-party libraries help implement the application. An OpenGL Loading Library is used to load the pointers to OpenGL functions. The project chose glad [20] as the final option of this library because of a similar reason to choosing Dear ImGui. For mathematics and model load libraries, the situations are analogical; there is one widely used library for each field with no obvious alternative solutions, GLM [19] (for mathematics), and tinyobjloader [73] (for loading obj file). Therefore, they become the chosen third-party libraries.

Chapter 3

Software Requirements and System Design

3.1 Software Requirements

The requirements should be defined correctly and briefly to design and implement a software product. In order to achieve this object, a detailed analysis is necessary. The following section will discuss the potential requirements from several different aspects.

3.1.1 Main Function

Initially, to figure out the requirements of a software product, the most vital work is ensuring this project's main function, or the primary target, and giving a brief description. Fortunately, the main function of this project is simple to summarise from the name, generating the signed distance field for arbitrary three-dimensional objects.

Since the Wavefront OBJ has been chosen as the input file format in 2.3.2.4, the explanation can become more specific, generating SDF for triangular meshes. Therefore, the software is supposed to read in the data from .obj files and store the process of the information. Once the data has been stored in the memory, the software needs to analyse the data and compute signed distance functions, then store the result in a new file, and finally visualise the SDF on the screen.

After defining the main requirement of this project, more specific demands need to be discussed. The following sections focus on more detailed aspects: Visualisation in 3.1.2, GUI in 3.1.3, Generation Algorithm in 3.1.4 and Evaluation in 3.1.5.

3.1.2 Visualisation

Before discussing the generation method requirement, as a computer graphic application, the contents of the graphic should be determined at first. The product should read in and visualise a triangle mesh as analysed in 3.1.1. Therefore, a camera and a simple scene are required to implement for rendering, which can be reused when rendering the SDF result. To visualise the SDF result, an intuitive solution is using different colours to represent the distance values; we can also use ray tracing or ray marching to test the practicality of SDF. The camera is supposed to observe the mesh from any angle, which can be implemented through a sphere orbit camera.

3.1.3 Graphical User Interface

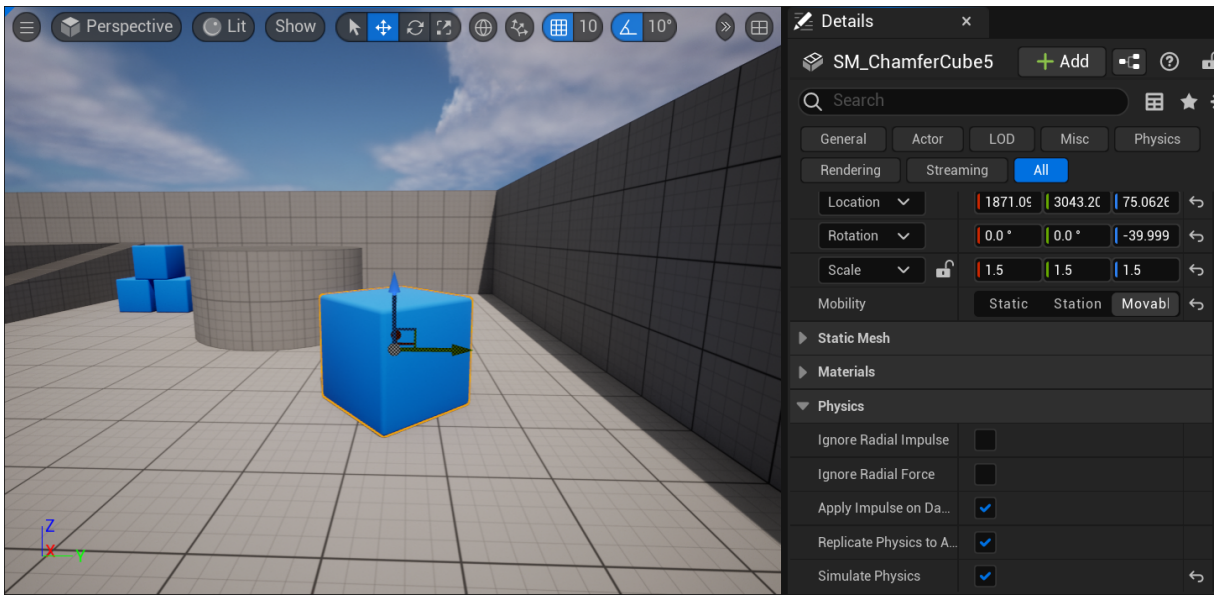


Figure 3.1: GUI screenshot of Unreal

An appropriate graphical user interface is helpful for users to understand the functions the software product provides to them. Besides, the programmer can also complete the testing and debugging job through a well-designed GUI. Then we have to figure out what should a good interface looks like. The design of a successful commercial game engine inspires this question. As is shown in the screenshot of Unreal Engine (figure 3.1), the attributes of an object are illustrated on the GUI. In addition, the content of the visualisation can be changed through buttons and checkboxes. For this project, as discussed in 3.1.2, the GUI is supposed to render a single model and its SDF in the same scene while the camera can move. Therefore, the requirement of an SDF generator GUI should contain the attributes info of the camera and several controls like checkboxes or buttons to change the visual contents of the object.

3.1.4 Algorithm

The generation algorithm itself has been discussed in detail in the section 2.2.2 and 2.2.3 of Chapter 2. The algorithm requirement should meet the choice mentioned in 2.3.1. Hence, two algorithms should be constructed for signed distance field computation, ray-tracing [2] and KD-tree.

3.1.5 Evaluation

As for evaluation, accuracy and efficiency should be considered. The result of the calculation should be saved as a text file to validate simply, while the efficiency can be evaluated through the time cost.

3.1.6 Others

Other requirements like the multi-threading optimization should be applied to the computation process, and the project source code should be able to be compiled and running, as discussed in Chapter 2. Those requirements will be considered in the system design.

3.2 System Design

3.2.1 Overview

According to the system requirement, this project will be divided into three main parts: GUI, algorithm and visualisation. In the GUI part, the Dear ImGui library will be used to implement a GUI class. In the algorithm part, three data structures of KD-Tree will be implemented as a class and used by the SDF calculation class. In the visualisation part, shaders, models, camera, and OpenGL render application is necessary for this project. Therefore, four classes will be implemented for each one. The software architecture of this project is shown in figure 3.2.

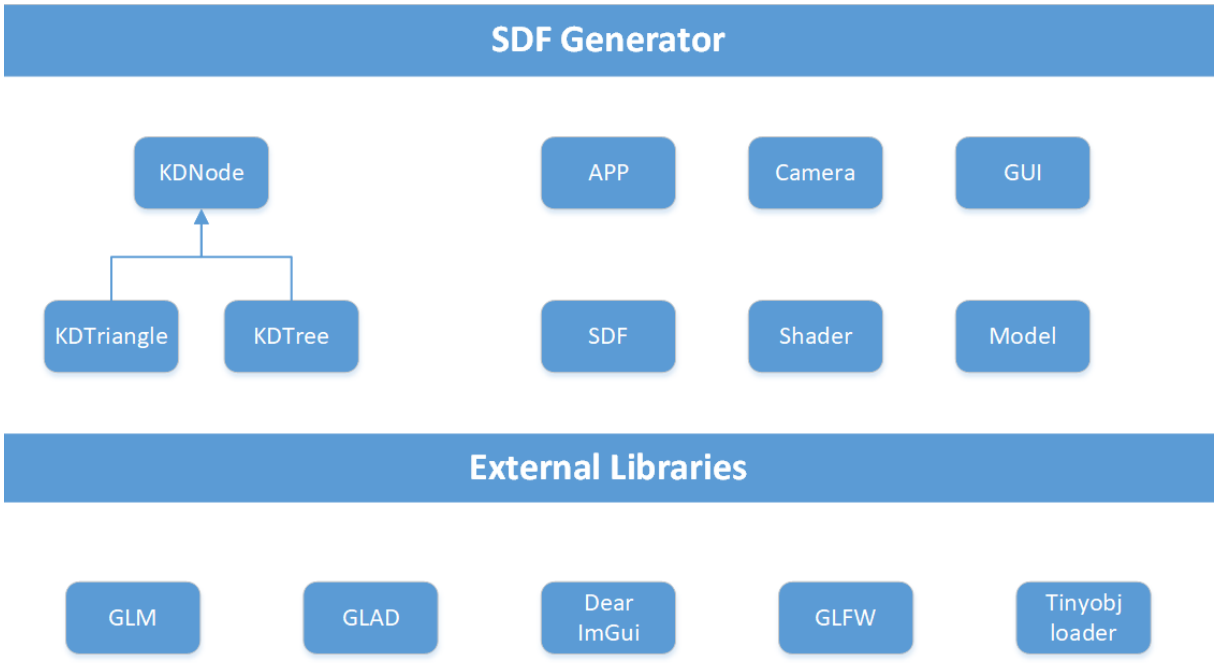


Figure 3.2: The software architecture

The detailed design of GUI, SDF algorithm and visualisation will be discussed in the following part of this chapter.

3.2.2 Graphical User Interface

As mentioned in 2.3.2, the user interface will be implemented through Dear ImGui. ImGui is easily used to create a moveable control panel in front of the rendering widget. Therefore, the control options for rendering, the attributes of the camera and the debug info like generation time will be set on a moveable panel. The status of rendering control options can be changed through checkboxes. The GUI design is shown below in figure 3.2.2.

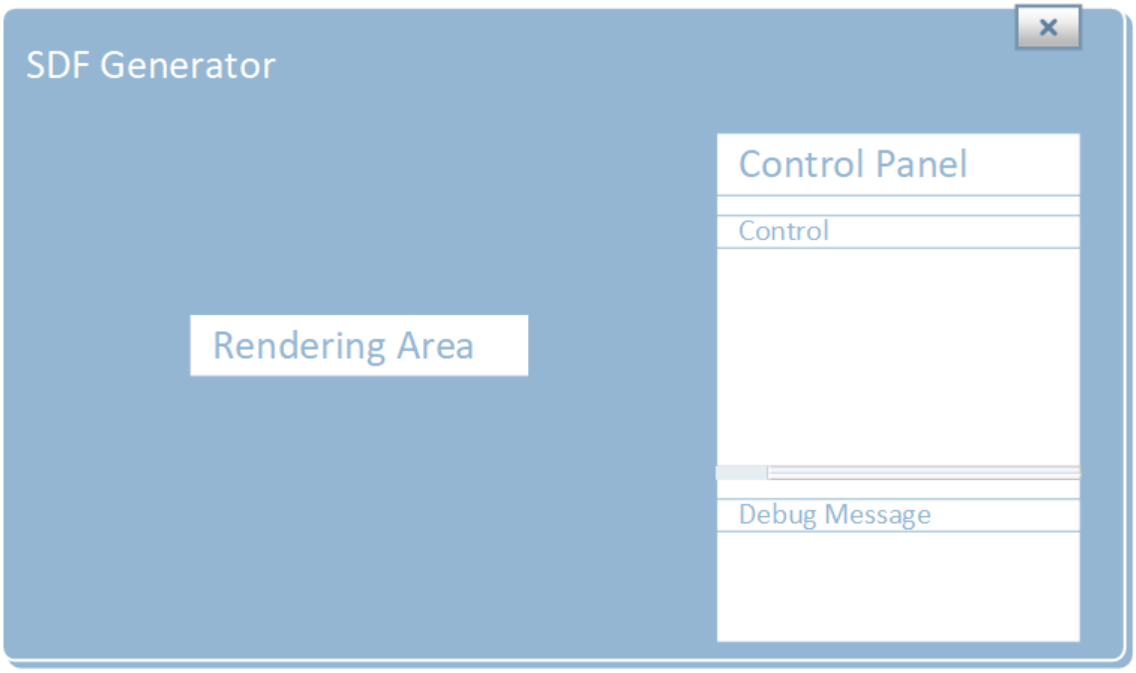


Figure 3.3: The application GUI design

3.2.3 Algorithm

There are three main algorithms to complete the signed distance field computation: the KD-tree structure, ray intersection and field generation.

3.2.3.1 KD-Tree

The tree should be built initially to use the KD-tree as the acceleration structure. As the project aims to generate a signed distance field for triangular meshes, the primitives of the tree should be triangles, which will be set as the tree node's data structure. Therefore, the nodes are constituted by the essential elements of triangles, i.e. vertices and edges. Besides, the bounding boxes and normal information will be stored in the nodes. To accelerate the process of ray intersection judgement, the function will be contained in the node structure.

The structure of the KD-tree is shown in figure 3.4. The pseudocode of building the KD-tree algorithm is shown as Algorithm 2.

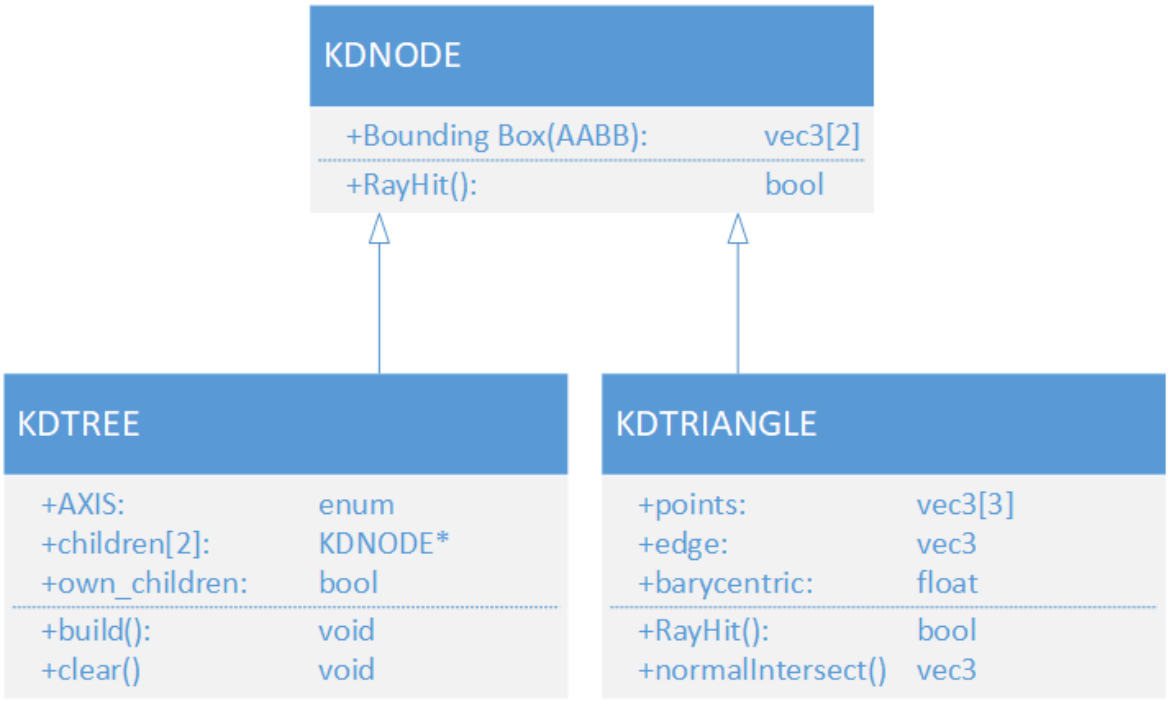


Figure 3.4: The structure of KD-tree

3.2.3.2 Ray Instersection

To apply the ray intersection algorithm, a number of sample rays must be generated equally. A sample voxel can be recognised as a sphere and split according to a fixed step. In Ahn's solution [1], the sphere is divided into a number of stacks and sectors, shown as figure 3.5.

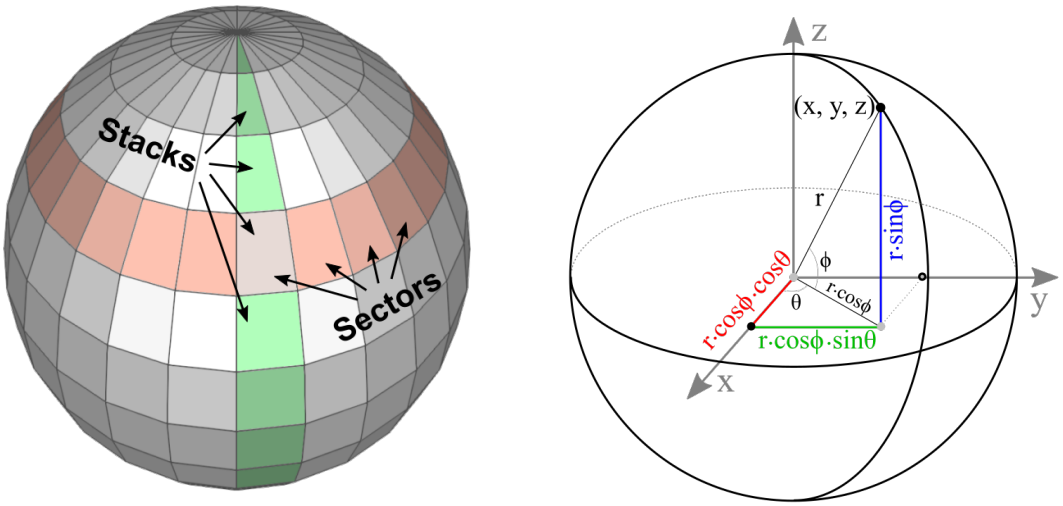


Figure 3.5: Calculate the coordinate of arbitrary point on sphere

Then the coordinate of an arbitrary point p(x,y,z) can be computed as equation 3.1:

$$\begin{aligned} x &= (r \cdot \cos\phi) \cdot \cos\theta \\ y &= (r \cdot \cos\phi) \cdot \sin\theta \\ z &= r \cdot \sin\phi \end{aligned} \tag{3.1}$$

Where the sector angle θ and stack angle ϕ can be calculated by equation 3.2:

$$\begin{aligned} \theta &= 2\pi \cdot \frac{sectorStep}{sectorCount} \\ \phi &= \frac{\pi}{2} - \pi \cdot \frac{stackStep}{stackCount} \end{aligned} \tag{3.2}$$

After generating the rays, the intersection algorithm needs to be used for distance computation. The algorithm of Tomas Akenine-Möller and Ben Trumbore [2] represent a point on triangle as $T(u, v)$, while (u, v) are the barycentric coordinates and satisfy $u \geq 0, v \geq 0$. Then T can be represent as 3.3:

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2 \tag{3.3}$$

V_0, V_1 and V_2 are the vertices of triangle.

We use $R(t)$ to denote a ray of length t, then the formula for $R(t)$ is as 3.4:

$$R(t) = O + tD \tag{3.4}$$

O is the origin while D is the unit vector of direction. Therefore, we can calculate the intersection problem by simply solving the equation 3.5:

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2 \tag{3.5}$$

Which can be reordered as the linear system of equation3.6:

$$\begin{bmatrix} -D, & V_1 - V_0, & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0 \tag{3.6}$$

let $M = [-D, V_1 - V_0, V_2 - V_0]$, the ray and triangle can be transformed as 3.6:
let $E_1 = V_1 - V_0, E_2 = V_2 - V_0$ and $T = O - V_0$, use the Cramer's rule, the equation 3.6 is able to be rearranged as 3.7:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D, E_1, E_2|} \begin{bmatrix} |T, E_1, E_2| \\ |-D, T, E_2| \\ |-D, E_1, T| \end{bmatrix} \tag{3.7}$$

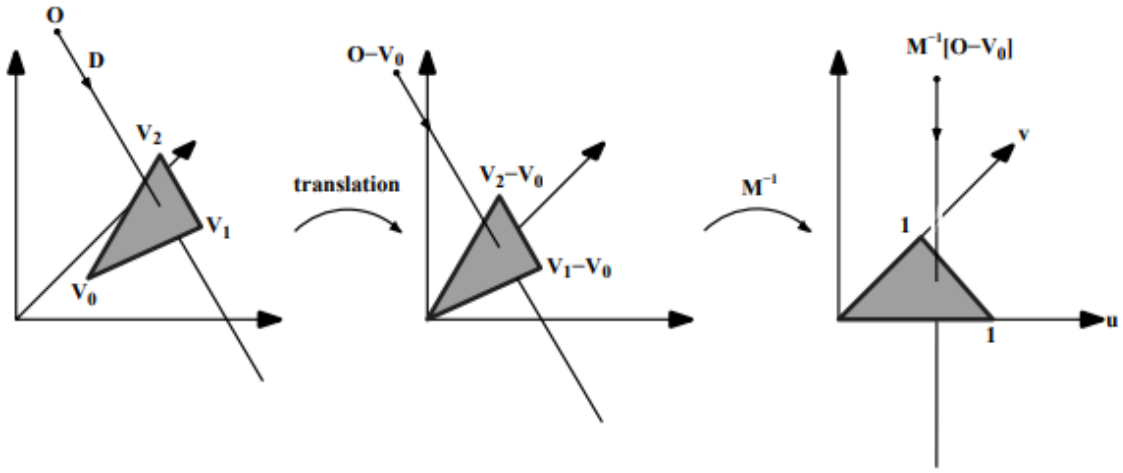


Figure 3.6: Translation and change of base of the ray origin

Applying $|A, B, C| = -(A \times C) \cdot B = -(C \times B) \cdot A$ to 3.7, we finally get equation 3.8:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix} \tag{3.8}$$

where $P = (D \times E_2)$ and $Q = (T \times E_1)$.
Finally, the distance t can be calculated as 3.9:

$$t = \frac{Q \cdot E_2}{P \cdot E_1} \tag{3.9}$$

There are two branches of the ray-intersection problem. The first one is to determine whether a ray intersects with the bounding box; id Software provides a solution in DOOM3 [74]. If the ray intersects with the bounding box, then the application will make a further determination, which is the second branch: intersect with the triangle. This algorithm will be used in the second branch to compute the distance function.

3.2.3.3 Signed Distance Field Generation

Finally, the whole distance field can be generated with the values of signed distance functions. The method of ray intersection judgement should be implemented through KD-tree acceleration and invoked by the generation algorithm.

The data members of the SDF class should contain the file path, bounding box, resolutions, distance values, and sample voxels that should be stored into vectors. Besides, a generating function is necessary, and the computed values should be able to output and read in again.

Figure 3.7 provide the structure of SDF class.

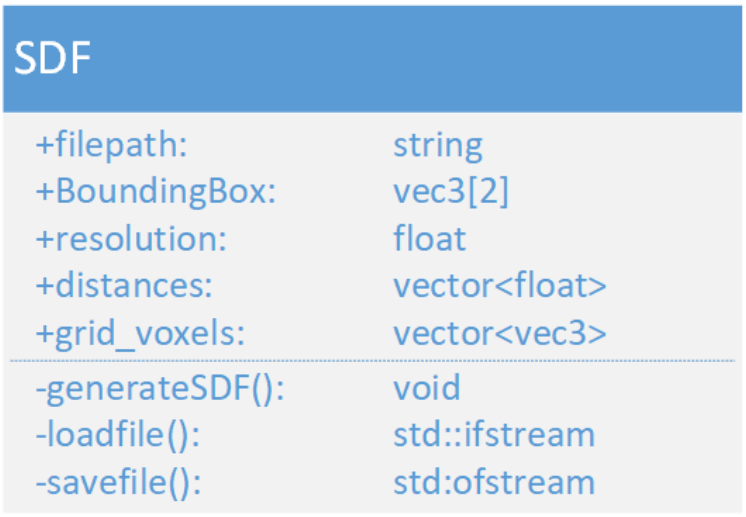


Figure 3.7: The structure of SDF class

3.2.4 Visualisation

The design of the visualisation module should contain resource loading (including shaders and models), a render scene, a scene camera and a render window.

3.2.4.1 Resources Loading

Shader The shader class should be about to load different types of shader files. Several enumerations can distinguish the shader type. After setting the type of shader, we can invoke the gl functions like glCompileShader() to compile shaders. Finally, the shader will be created successfully.

The structure of the shader class and its members are shown in figure 3.8.

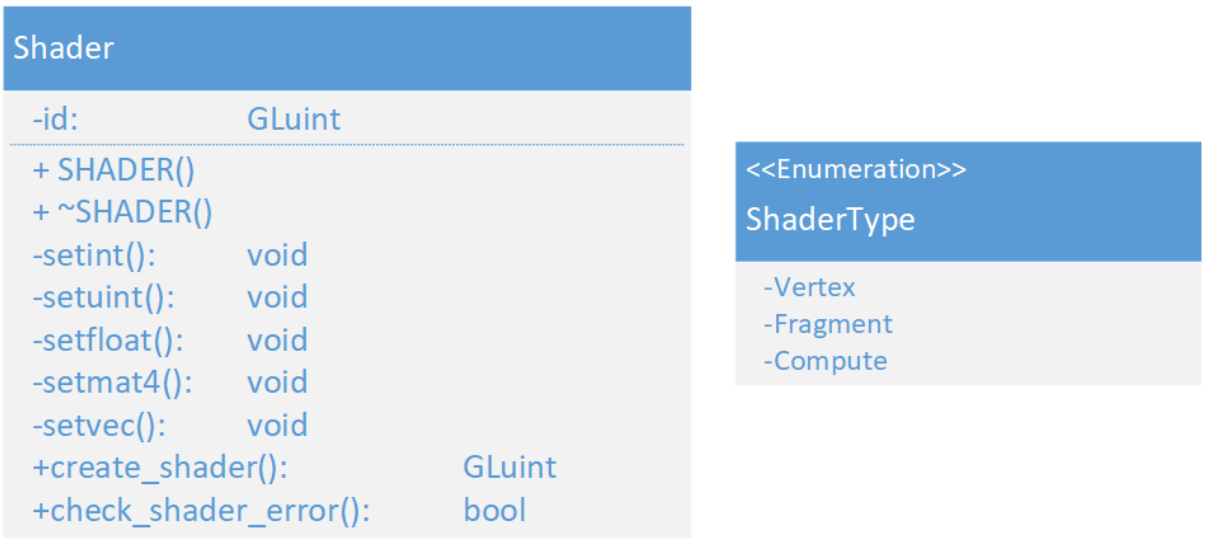


Figure 3.8: The structure of Shader class

Model The model load class used the tiniobjloader library. Considering about the requirements for computation, the information of vertices and normals will be read from the file. Then the vertices need to be reordered to the primitives, in this project, triangles. Then VAO and VBO should be created and allocated. Finally, the KD-tree should be built for the model, and the sample ray should test the intersection with the model. Therefore, both functions need to be implemented in the class.

The structure of model class is shown in figure 3.9.

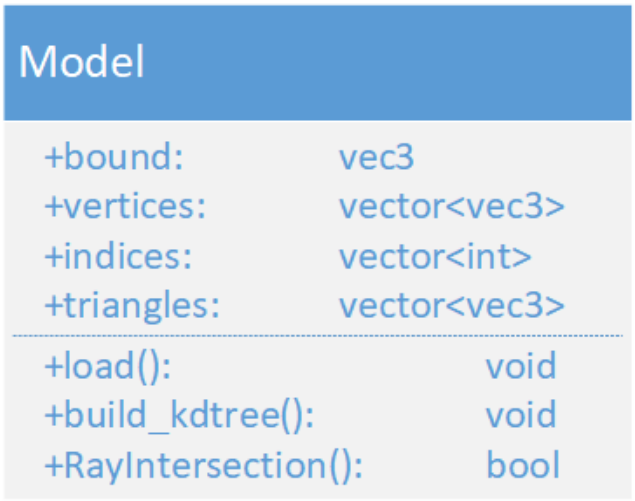


Figure 3.9: The structure of Model class

3.2.4.2 Scene

As for the scene, a grid-structured plane which can be extended infinitely is enough; the main coordinate axis will be highlighted. This plane can be implemented through a shader and can be directly reused by other applications or graphic APIs like Vulkan. Marie [54] provide the methods to create such a scene in the following order. First, create a basic grid plane vertex and fragment shader. Second, set the plane coordinates as infinity. Then draw the floor plane(i.e. y=0) and make the axis and grids on the floor. Finally, use the fragment shader to add a fade-out effect when the grid is far away.

3.2.4.3 Camera

The camera should be able to control by the mouse. The GLFW window will collect the user input; therefore, the camera class need to provide transformation functions like translation, rotation and zoom. Besides, the camera status should be updated by time step, and independent of the frame rate, the timer can be used for implementing the update method.

Figure 3.10 provides the structure of Camera class.

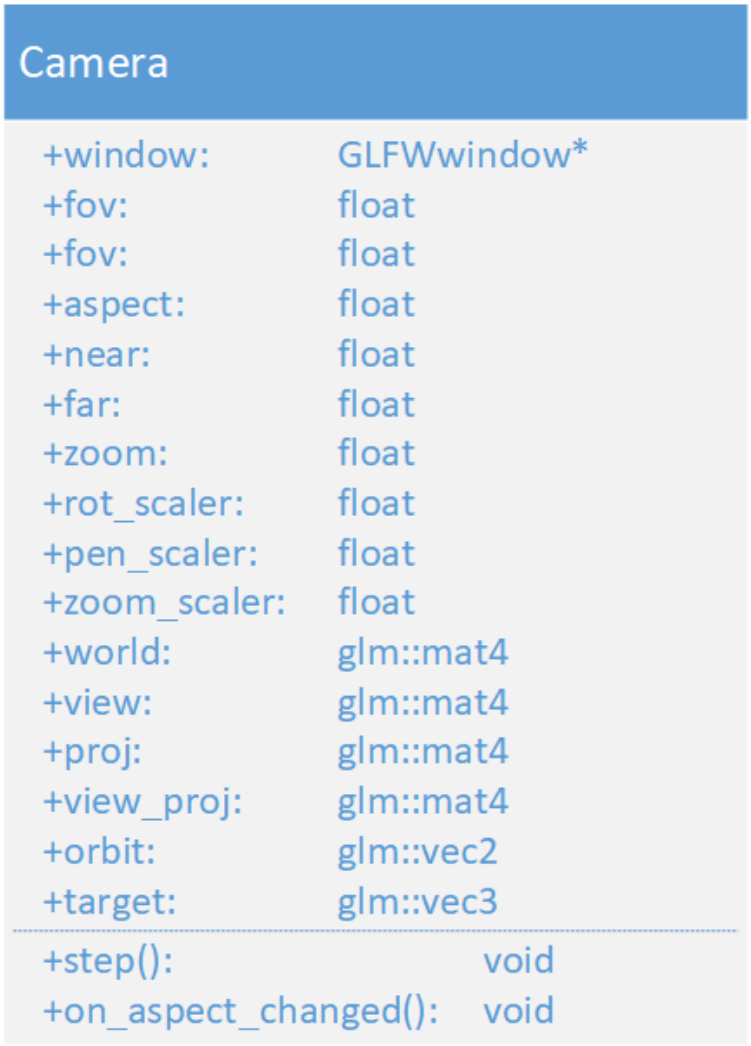


Figure 3.10: The structure of Camera class

3.2.4.4 Rendering Process

The rendering process is divided into three main stages: initialize, terminate and rendering loop.

Initialize This stage contains the primary setting of an OpenGL application. A function for creating a GLFW window will be defined; other operations are contained in this function, like setting up a version and receiving the user input, which will be used to manipulate the camera. Besides, this function is also necessary for testing if the development environment has been set correctly. Then the rendering resources need to be read in, including models and shaders. If the last compute result of the signed distance field exists, it will also be loaded. If not, then the computation function will be running. Before entering the rendering loop, the parameters of the GUI will be set, and the GUI components will be bound to the same GLFW window.

Terminate The functions that tell the libraries to terminate and release the resources like glfwTerminate() have been contained in their codes, which only needs to be invoked by the terminate function.

Rendering In the rendering loop, three tasks need to be completed. First, the status of the timer needs to be stored and updated to ensure the camera transformation's stability. Then, the application will get framebuffer status and update continuously. For example, if the window has been resized, the viewport should be updated so that the pixels will be rendered properly; the swapbuffer technique [34] is applied in this process to prevent tearing or flickering issues. In addition, the GUI panel will be configured, and the contents like checkboxes and textboxes will be created and updated. Finally, the rendering function will bind different shaders when changing the modes of rendering.

Chapter 4

Software Implementation

The details of implementation will be described in this chapter. The modules will be explained in the chronological order in which they were implemented.

4.1 Project Building

For arbitrary Graphics API programming, setting up the development environment is complex and time-consuming and error-prone. As discussed in 2.2.4.4 and 2.3.2.3 the Visual Studio solution will be generated through Premake5. Premake5 only contains an execute program; the configuration needs to be written to Lua scripts. The syntax of premake scripts is fixed; it should contain the basic configurations of the project, including languages, names, platforms, source code formats and others. Premake has many valuable tokens for setting the relative directories for input and output, the different settings for platforms are distinguished through the filters. Besides, in this project, many external libraries (e.g. GLFW, GLM, ImGui) need to be contained and compiled with the source file, premake can be used to set the libraries as StaticLib projects and linked to the main project, which will keep the source code of main project independent with the libraries.

After editing the scripts, the solution can be generated by Premake correctly as shown in figure 4.1 and 4.2.

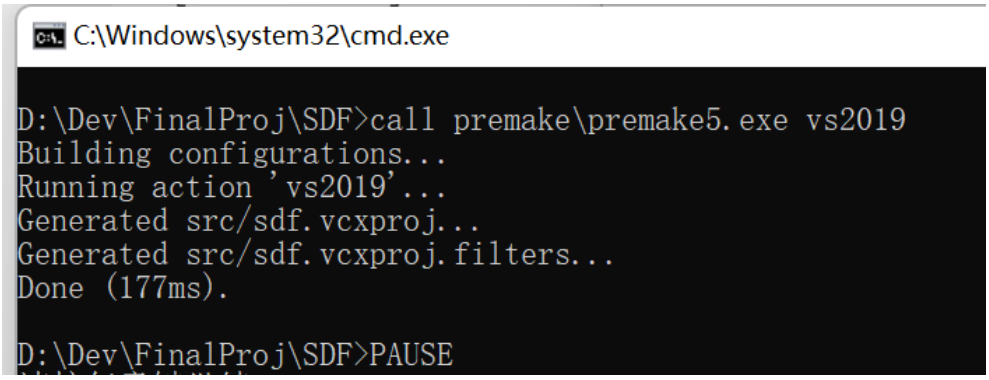


Figure 4.1: Generate Project Solution by Premake

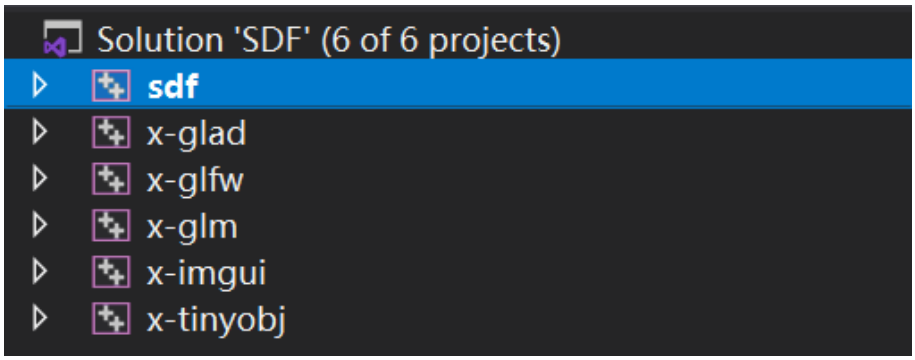


Figure 4.2: The VS2019 Solution of this project

The script code for whole solution and external libraries can be seen in Appendix A.

4.2 Basic Visualization Application

In the first stage, a basic visualization application must be created to test whether the development environment is running properly. The modules will be implemented according to the design of the rendering process, which has been discussed in 3.2.4.

Shader Class The Shader class must load and compile shader files according to their type, and several overloaded constructors are implemented for the vertex shader, fragment shader and other options. The class invoke several gl functions for binding and compiling.
The code of constructor and creating function are shown in Listings 4.1 and 4.2.

```
1     SHADER::SHADER(const char* vertex_path, const char* fragment_path)
2 {
3     GLuint vertex_shader = create_shader(vertex_path, ShaderType::Vertex
      );
4     GLuint fragment_shader = create_shader(fragment_path, ShaderType::
      Fragment);
5
6     id = GL( glCreateProgram());
7     GL( glAttachShader(id, vertex_shader) );
8     GL( glAttachShader(id, fragment_shader) );
9     GL( glLinkProgram(id) );
10
11    if (check_shader_errors(id, ShaderType::Program)) exit(1);
12
13    GL( glDeleteShader(vertex_shader) );
14    GL( glDeleteShader(fragment_shader) );
15 }
```

Listing 4.1: Constructor of vertex and fragment shader

```
GLuint SHADER::create_shader(const char* path, ShaderType type) const
{
    std::stringstream shader_stream;
    shader_stream << file.rdbuf();
    file.close();
    const std::string& shader_source = shader_stream.str();
    const char* shader_source_c_str = shader_source.c_str();
    GLenum shader_type = GL_VERTEX_SHADER;
    switch (type)
    {
        case ShaderType::Fragment:
            shader_type = GL_FRAGMENT_SHADER;
            break;
        case ShaderType::Compute:
            shader_type = GL_COMPUTE_SHADER;
            break;
    }
    GLuint SHADER = GL( glCreateShader(shader_type) );
    GL( glShaderSource(SHADER, 1, &shader_source_c_str, 0) );
    GL( glCompileShader(SHADER) );
    if (check_shader_errors(SHADER, type))
        return GLuint(-1);
    return SHADER;
}
```

Listing 4.2: Function of creating shader

Camera The camera class is implemented as a sphere orbit camera; the camera's position will rotate on a sphere orbit outside the object, whose centre is the centre of the object. There are several parameters needed to store the transformation matrices and scales. The project directly used the data structure provided by the GLM library for storing and computing. As for the operation mode, hold the right or middle mouse button, drag to move the camera, and use the mouse wheel to zoom.

The camera matrix transformation is implemented in the step() function and updated per frame, whose implementation can be seen in Listing 4.3.

```
    void step( float elapse, const mouse_t &mouse ) {
    const glm::vec2& mouseD = mouse.mouseD;
    const glm::vec2& wheelD = mouse.wheelD;
    glm::vec3 delta( mouseD.x , -mouseD.y, 0 );
    // right button + mouse :rotation
    if ( mouse.button_stats[2] ) {
        orbit.x = constrain360( orbit.x + delta.y * rot_scaler );
        orbit.y = constrain360( orbit.y + delta.x * rot_scaler );
    }
    zoom = glm::clamp( -wheelD.y * zoom_scaler + _zoom, 10.f, 1000.f  );
    glm::mat4  rotation = glm::mat4( glm::rotate( glm::radians( _orbit.y
    ), Y ) ) * glm::mat4( glm::rotate( glm::radians( _orbit.x ), X ) )
    ;
    if ( mouse.button_stats[1] ) {
        target += glm::mat3(rotation) * delta * pen_scaler ;
    }
    world = glm::translate( _target ) *  rotation * glm::translate( glm
    ::vec3(0.f,0.f, _zoom) ) ;
    view = glm::inverse( _world);
    proj = glm::perspective( glm::radians(_fov), _aspect, _near, _far );
    view_proj = _proj * _view;
}
```

Listing 4.3: Function of matrix transformation

GLFW Window An application window is necessary for rendering the GUI and models. As discussed in 3.2.4.4, this project uses the GLFW library to create a rendering application. In this stage, a simple shader is used to test if the window is running correctly. A GLFW window should be instantiated to achieve the target and set the compulsory attributes, then get the user input data to control the camera. The step() function is invoked in the rendering loop function; frame time and the status of the mouse are stored in an object and passed to the Camera class for matrix computation. Listing 4.4 and 4.5 shows the code of GLFW window instantiation and rendering.

```
    GLFWwindow* window = nullptr ;
    glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GL_TRUE);
    // GL 4.5
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 5);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    // Create window with graphics context
    window = glfwCreateWindow(1280, 720, "SDF", NULL, NULL);
    if (!window)    exit(1);
```

Listing 4.4: GLFW window instantiation

```
    while (!glfwWindowShouldClose(window)) {
        // Camera update
        this->mouse.wheelD = {0,0};
        this->mouse.mouseD = {0,0};
        glfwPollEvents();
        float current = glfwGetTime();
        delta = current - last;
        accum += delta;
        last = current;
        camera.step( delta, mouse );
        // Scene render
        glfwGetFramebufferSize(window, &screen_size.x, &screen_size.y);
        if (screen_size.x!=0 && screen_size.y!=0 ) {
            glViewport(0, 0, screen_size.x, screen_size.y);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

            app_draw_scene( delta );
            gui_draw(clear_color);
            glfwSwapBuffers(window);
        }
    }
```

Listing 4.5: Rendering loop implementation

Scene The infinite grid scene will be implemented through the vertex shader and fragment shader as the method introduced by Marie [54]. The GLSL source code can be seen in the Appendix B. The shaders are loaded to the application and compiled by the shader class to create a shader program object, then bind the object as a part of the rendering state.

Graphical User Interface The GUI is implemented through the Dear ImGui library, and the GUI panel is created by invoking the functions of the library. First, it needs to be initialized to set the window pointer for the GUI panel. Then members of the GUI will be created. For example, ImGui::CollapsingHeader() function can be used to create an independent area on the panel, and ImGui::Checkbox() can be used to create a checkbox object on the panel.

Listing 4.6 shows the GUI implementation code.

```
    // Setup Platform/Renderer backends
    ImGui_ImplGlfw_InitForOpenGL(window, true);
    ImGui::Begin( "Panel", 0 );
    // Visibility of model
    if (ImGui::CollapsingHeader("Visibility",
   ImGuiTreeNodeFlags_DefaultOpen ))
```

```
6     {
7         static bool shows[5];
8         ImGui::Checkbox("Grid", &show_grids);
9         ImGui::Checkbox("Polygon", &show_polygon );
10        ImGui::Checkbox("Coloured Model", &show_color );
11
12        ImGui::Checkbox("SDF", &show_iso );
13        ImGui::Checkbox("RT Approximation", &show_sdf);
14    }
15    // Camera Status
16    if (ImGui::CollapsingHeader("Cemera", ImGuiTreeNodeFlags_DefaultOpen
    )) {
17        ImGui::LabelText( "Orbit", "%2.0f %2.0f" , camera._orbit.x,
    camera._orbit.y );
18        ImGui::LabelText( "Target", "%2.0f %2.0f %2.0f", camera._target.
    x, camera._target.y ,camera._target.z );
19        ImGui::LabelText( "Zoom", "%2.0f", camera._zoom  );
20    }
21    ImGui::End();
22    // Rendering
23    ImGui::Render();
24    ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
```

Listing 4.6: ImGui panel implementation

Once all of the processes have been completed, a window that renders an infinity grid scene and GUI panel can be seen on the screen as figure 4.3.

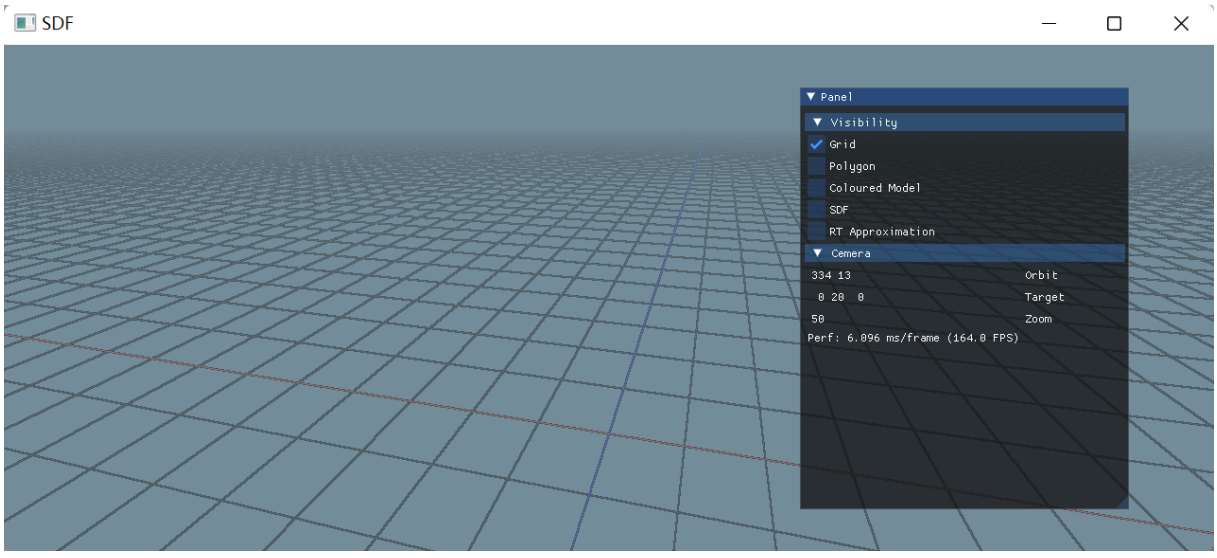


Figure 4.3: The basic infinity grid scene and GUI implementation

4.3 Model Rendering

Model Class To load a triangular mesh, the tiniobjloader is used to get the vertices and face indices information. The vertices and the face indices are used to create triangle primitives for rendering and the signed distance function calculation. Several gl functions are used in the primitive creating stage.
The pseudocode of the load model is shown in Listing 4.7.

```
    load( obj_file ) {
    //Create empty containers
    tinyobj::attrib_t attrib;
    std::vector<tinyobj::shape_t> shapes;
    tinyobj::LoadObj(&attrib, &shapes, obj_file) // Load obj file
    // Clear the vectors
    vertices.clear();
    indices.clear();
    for (loop shape : shapes) { // Loop meshes to get vertices
        for (loop index : shape.mesh.indices) {
        vertex.pos = {
            attrib.vertices[3 * index.vertex_index + 0],
            attrib.vertices[3 * index.vertex_index + 1],
            attrib.vertices[3 * index.vertex_index + 2]    };
        vertices.push_back(vertex);     }   }   }
    triangles.resize( _indices.size()/3 );
    for ( size_t i=0;i<_triangles.size(); ++i) {
        auto i0 = _indices[i*3+0];
        auto i1 = _indices[i*3+1];
        auto i2 = _indices[i*3+2];
        triangles[i]._init( vertices[i0],vertices[i1],vertices[i2]); }
    make_primitive( vertices, indices );    }
```

Listing 4.7: Pseudocode of model loading

The code for making primitives are illustrated through Listing 4.8.

```
void MODEL::make_primitive( const  vertex_t *Vdata, GLuint Vcount,
        const uint32_t *Idata, GLuint Icount ) {
    _primitive.ibo_count = Icount;
    // Create VAO
    glGenVertexArrays(1, &_primitive.vao);
    glBindVertexArray( _primitive.vao);
    // create VBO
    glGenBuffers(1, &_primitive.vbo );
    glBindBuffer(GL_ARRAY_BUFFER, _primitive.vbo );
    // copy vertex attribs data to VBO
    GLuint vsize = sizeof(Vdata[0]) * Vcount;
    glBufferData(GL_ARRAY_BUFFER,  vsize, Vdata, GL_STATIC_DRAW);
    // Setup vertex attributes
    glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE, sizeof(Vdata[0]), 0);
```

```
    glVertexAttribPointer(1,3, GL_FLOAT, GL_FALSE, sizeof(Vdata[0]),
        (GLvoid*) long( sizeof( Vdata[0].pos ) ));
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    // Create index buffer
    glGenBuffers(1, & _primitive.ibo );
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _primitive.ibo );
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, Icount * sizeof(Idata[0]) ,
        Idata , GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER,0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,0);
    glBindVertexArray(0);   }
```

Listing 4.8: Function of creating triangular primitives

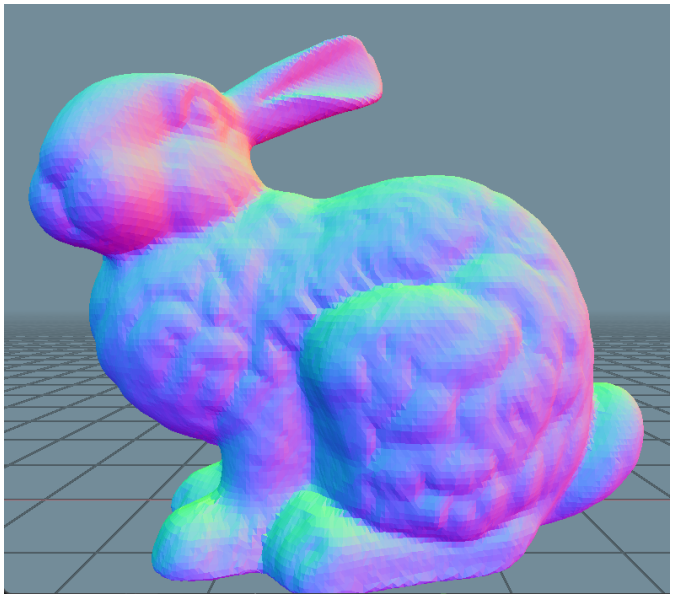
Rendering In the GUI implementation, two modes are set for model rendering, polygonal mode and coloured mode. Both can bind the same shaders, while the polygonal mode can be implemented by invoking the glPolygonMode(). The primitives of the model are set as triangles. Therefore, the draw mode should be set as GL_TRIANGLES.

Listing 4.9 shows the code of both modes.

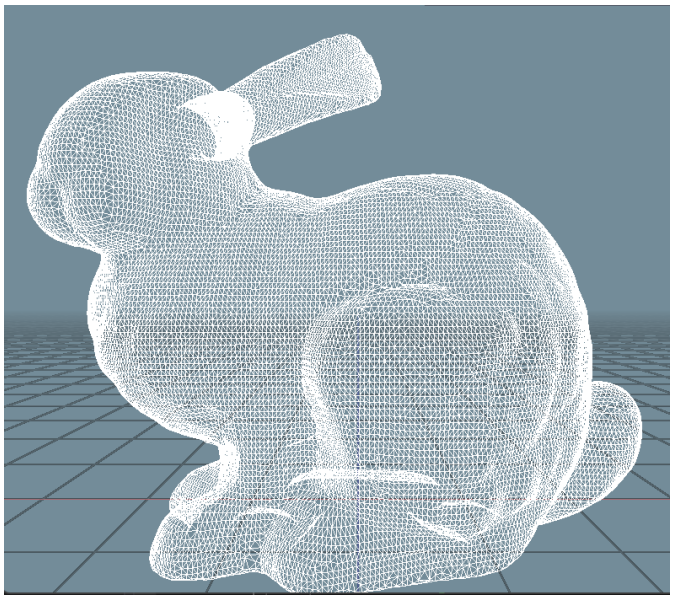
```
    auto prim = &model._primitive; // Draw model primitives
    model_prog->bind(); // Bind shader
    // Setup transfrom
    model_prog->set_mat4( "MVP", glm::value_ptr( view_proj ) );
    prim->bind();
    if ( show_color ) { // Coloured draw mode
        model_prog->set_vec4( "COLOR", 0,0,0,0 );
        glDrawElements( GL_TRIANGLES , prim->ibo_count,
            GL_UNSIGNED_INT, (0) );
    }
    if ( show_polygon ) { // Polygonal draw mode
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        model_prog->set_vec4( "COLOR", 1,1,1,1 );
        glDrawElements( GL_TRIANGLES , prim->ibo_count,
            GL_UNSIGNED_INT, (0) );
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    }
    prim->unbind();
    model_prog->unbind();
```

Listing 4.9: Function of creating triangular primitives

The result of both rendering modes can be seen in figure 4.4.



(a) Coloured Mode



(b) Polygonal Mode

Figure 4.4: The result of model rendering

4.4 Algorithm

4.4.1 KD-Tree

To implement the KD-tree structure, the tree node needs to define a tree node class and then define the tree class as the derived class of the node. Since the primitives of a model are triangles, the triangle tree nodes need more data members to store their attributes like vertices and edges, while the tree class need children pointers and the

split axis. Therefore, the node class is defined as an abstract class inherited by the triangle and tree class to add more variables.

Algorithm 2 KD-Tree Building

```
Require: objects
  Building bounding box
  Set split Axis

  split bounding box
  for each objects i do
    if object[i].Bbox.center < objects.Bbox.middle then
                                        ▷ Bbox is the bounding box of current object
      Insert to left tree
    else
      Insert to right tree
    end if
  end for
  newBbox = left.Bbox + right.Bbox
```

4.4.2 Ray Intersection

There are three situations of ray-model intersection problem. If the ray start point is inside the bounding box, then the intersection status is set as true. If the ray intersects with any axis, the return value is also set as true. Otherwise, there is no intersection.

The pseudocode is provided in Algorithm 3 below.

Algorithm 3 Bounding Box Intersection

```
  if ray origin is inside of Bbox then
    return true
  else if ray intersect with bounding box then
    return true
  else
    return false
  end if
```

After completing the judgement of the intersection between ray and bounding box, the application uses the algorithm of Tomas Akenine-Möller and Ben Trumbore [2] to calculate the distance function and sign function. Then the signed distance value will be stored in the intersection object for distance value comparison.

The pseudocode is provided in Algorithm 4. The definition of variables in Algorithm 4 will follow the definitions of [2], which can be double reviewed in Section 2.2.2.1.

Algorithm 4 Triangle Intersection

Require: EPSILON ▷ minimal interval from [2]
ray r ▷ denote origin as O, direction as D
triangle vertices V_0, V_1 and V_2
triangle edges E_1 and E_2
previous intersection inter ▷ default as infinity

$P = D \times E_2$
$determinant = E_1 \cdot P$ ▷ wiil be denoted as det for short
$T = O - V_0$

if det $<$ EPSILON **then**
    return false
end if

$u = T \cdot P$
if u $<$ 0 or u $>$ det **then**
    return false
end if

$Q = T \times E_1$
$v = D \cdot Q$

if $v < 0$ or $u + v > det$ **then**
    return false
end if

$t = Q \cdot E_2$
$InvDet = 1 \div det$
$t = t * InvDet$
if $t < inter.t$ and $t > EPSILON$ **then**
    update inter variable
    return true
else return false
end if

4.4.3 Signed Distance Field Generation

Finally, the ray intersection algorithm is applied to the field generation function. As discussed in the 2.2.3, each sampling voxels generate a number of rays from its centre to test and compute the distance. The first step is to loop a single voxel's ray group and get its minimal distance; the Algorithm 4 is used for single ray intersection judgement. Then test if the ray start point is inside the model or if the model has a hole to determine the sign of this voxel. Finally, record the signed distance value and repeat the process to each voxel. The ray intersection computation algorithm is implemented in the KD-tree structure for pruning.

Listing 4.10 provides the code of sample rays generation.

```
vector<vec3> sampleRays;
sampleRays.clear();
vec3 sum(0);
float phiOffset = PI / phiSteps * 0.5f;
for (int phiStep = 0; phiStep < phiSteps; phiStep++)
{
    float phi = -PI / 2.f + (PI * phiStep) / phiSteps + phiOffset;
    for (int thetaStep = 0; thetaStep < thetaSteps; thetaStep++)
    {
        float theta = (PI * 2.f * thetaStep) / thetaSteps;
        vec3 direction(cosf(theta), sinf(theta), sinf(phi));
        sum += normalize(direction);
        sampleRays.push_back(normalize(direction));
    }
}
```

Listing 4.10: Function of generating sample rays for voxels

The pseudocode of the field generation algorithm is shown in Algorithm 5:

Algorithm 5 SDF Generation

Require: mesh, dimensions, sampleRays

Bbox = mesh.bbox ▷ The bounding box of the mesh
d = dimensions
hit = 0
hitBack = 0

for each point p(x,y,z) in Bbox **do**
    minDistance = ∞
    for each ray in sampleRays **do**
        if ray intersect with model **then**

```
            hit++
            if ray intersect with backside then
                hitBack++
            end if
            if currentDistance < minDistance then
                minDistance = currentDistance
            end if
        end if
    end for
    if Over half of rays hit backside then
        The ray origin inside mesh
        sign is negative
    end if
    if The meshes has hole then
        The ray origin inside mesh
        sign is negative
    end if
    Distance at p(x,y,z) = minDistance
end for
```

The asynchronous multi-threading strategy is also applied to reduce the generation time. The voxels are divided into many groups, and each group will run the generation function in parallel. The method is implemented by using the std::async() function.

Listing 4.11 gives the code of multi-threading acceleration. The Algorithm 5 is implemented in function generate_layer().

```
auto future = std::async(std::launch::async, [&] {
    std::vector<int> zid (dfDimensions.z);
    std::iota (std::begin(zid), std::end(zid), 0);

    std::for_each(
    std::execution::par,
    zid.begin(),
    zid.end(),
    [&](auto&& zIndex)
    {
        cout << "\tProcessing layer: " << zIndex << endl;
        generate_layer(model,sampleRays,
            zIndex, _layer_progress[zIndex] );
    });
    return 0;
});
```

Listing 4.11: Function of generating sample rays for voxels

4.5 SDF output and debug

The result of the signed distance field is stored in a text file since it is easy to be generated and validate. Operator « is overloaded to output the values. The implementation can be seen in Listing 4.12

```
1    ofstream &operator<<(ofstream &o, const SDF &data) {
2        o << data._source << endl
3        << data._grids.x << " " << data._grids.y << " " << data._grids.z
     << endl
4        << data._resolution << endl;
5        for (float f : data._voxels) //distance values
6        {   o << f << endl;    }
7        for (auto v : data._vertices) // sample voxels
8        {   o << v.x << " " << v.y << " " << v.z << endl;   }
9        return o;
10   }
```

Listing 4.12: SDF output function

Some debug information needs to be shown on the interface, including generation time, triangle number and sampling resolutions. Therefore, the GUI panel implementation should add a new capsule area.

The final GUI panel implementation is shown in figure 4.5.

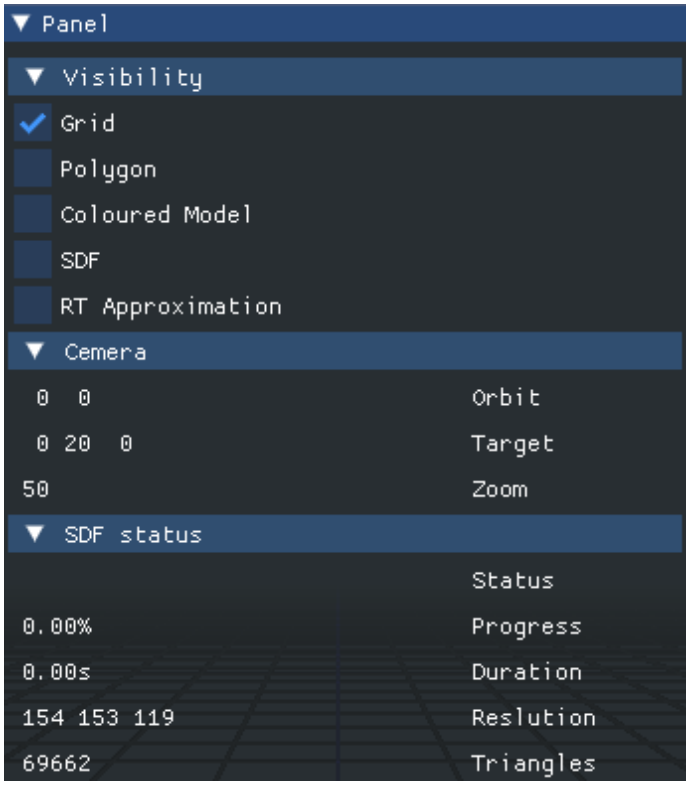


Figure 4.5: GUI panel with debug info

4.6 SDF Visualization

Finally, the result of the SDF calculation should be visualized. Two methods are implemented in this project, and the first one is using distance values to adjust the colour value; the volume of colour is changed according to the distance. The result is shown in figure 4.6, the volume of green is set as the highest on the boundary of the model while the colour of inside voxels is visualized as red. The other one used ray tracing to approximate the model shape, which is shown in figure 4.7.

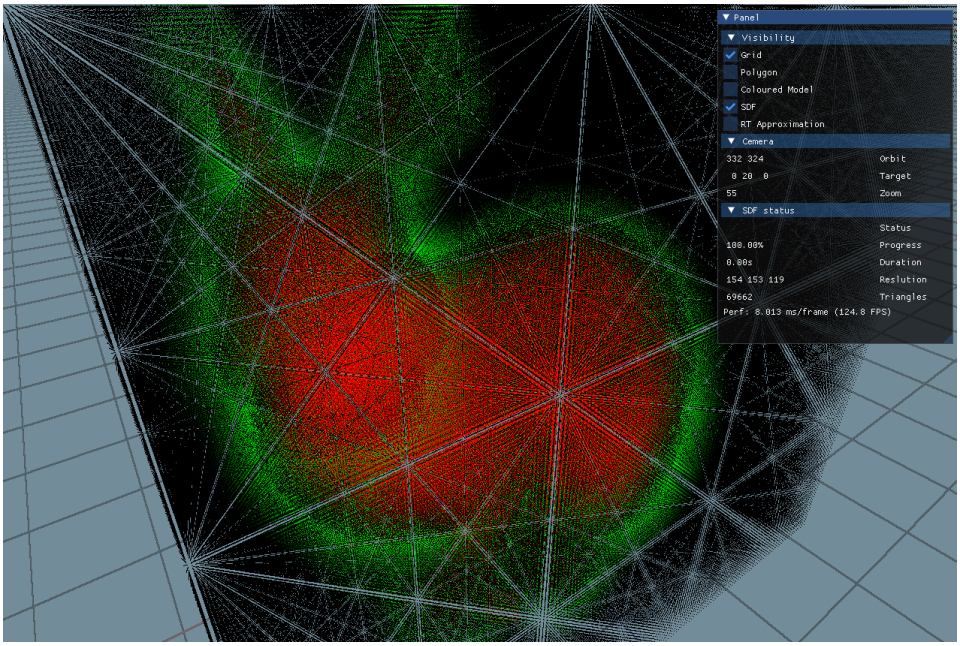


Figure 4.6: The colour volume mode of SDF visualization

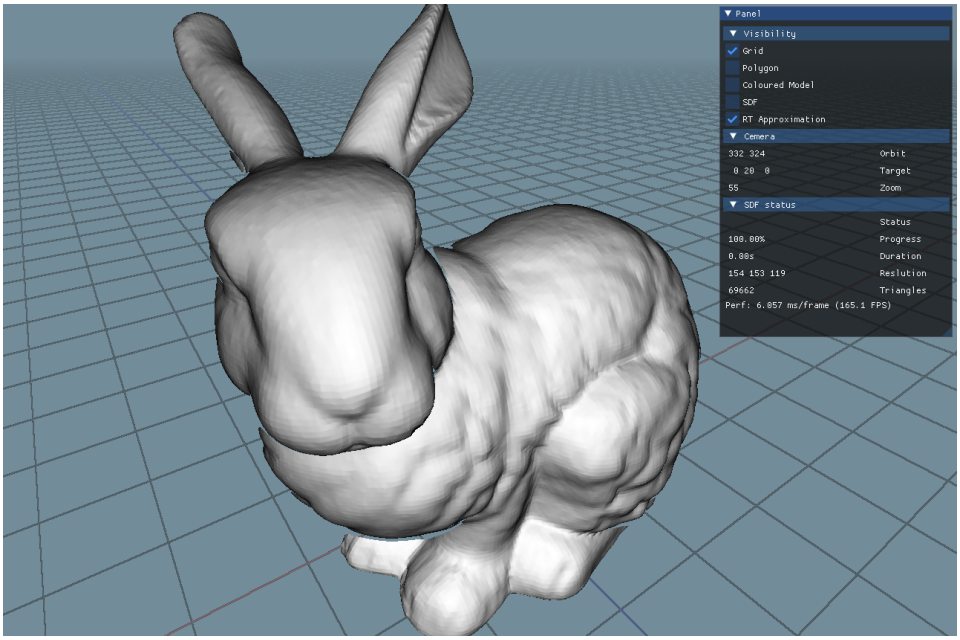


Figure 4.7: The ray tracing mode of SDF visualization

Chapter 5

Software Testing and Evaluation

To assess the quality of implementation, ensuring each component of this project works properly has top priority, and then the different input settings should be explored. Therefore, this chapter covers the unit test and performance evaluation to prove that the project satisfied the requirements of the initial objectives.

5.1 Unit Test

The unit test needs to verify several components: basic model rendering, GUI update and SDF.

5.1.1 External Libraries

Most of the components are implemented based on several third-party libraries. For example, the model class use the vector definition from the GLM library to store the vertices. Therefore, all libraries should ensure that they are imported correctly so that further programming and testing tasks.

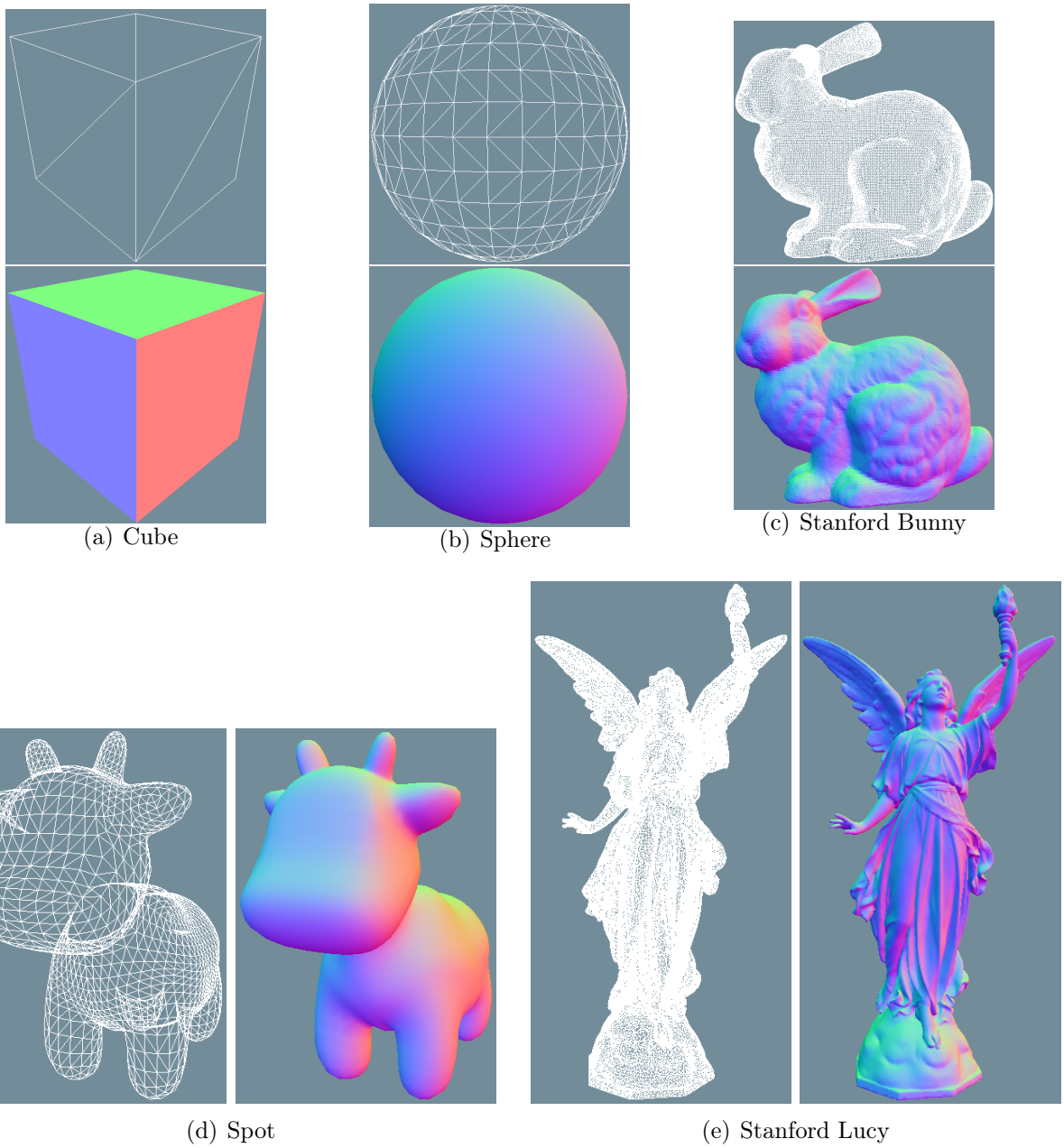
GLFW and GLAD testing can be verified through an executable rendering window. Once the libraries are appropriately set, the rendering pipeline is built automatically so that the GPU will render the correct image. GLM and tinyobjloader are required to be adequately included in the source code, and both can be verified through read-in a simple mesh like a cube and converting the object data to the format of GLM; if the code can be compiled and running correctly, both libraries are successfully imported. Similarly, the Dear ImGui library's import status can be verified by rendering a simple GUI component to the application window.

Figures 4.3 and 4.4 have proven that the rendering pipeline, GUI library, and obj loader library have been imported properly.

5.1.2 Model Rendering

The application is supposed to process arbitrary obj files. Therefore, both simple and complex meshes are tested for this component, including the simple cube, sphere meshes and common 3D models like Spot, Stanford Bunny and Stanford Lucy. All test models can be read-in and rendered correctly through polygonal and coloured mode.

The result of model rendering testing for different meshes is shown in figure 5.1.



(a) Cube (b) Sphere (c) Stanford Bunny

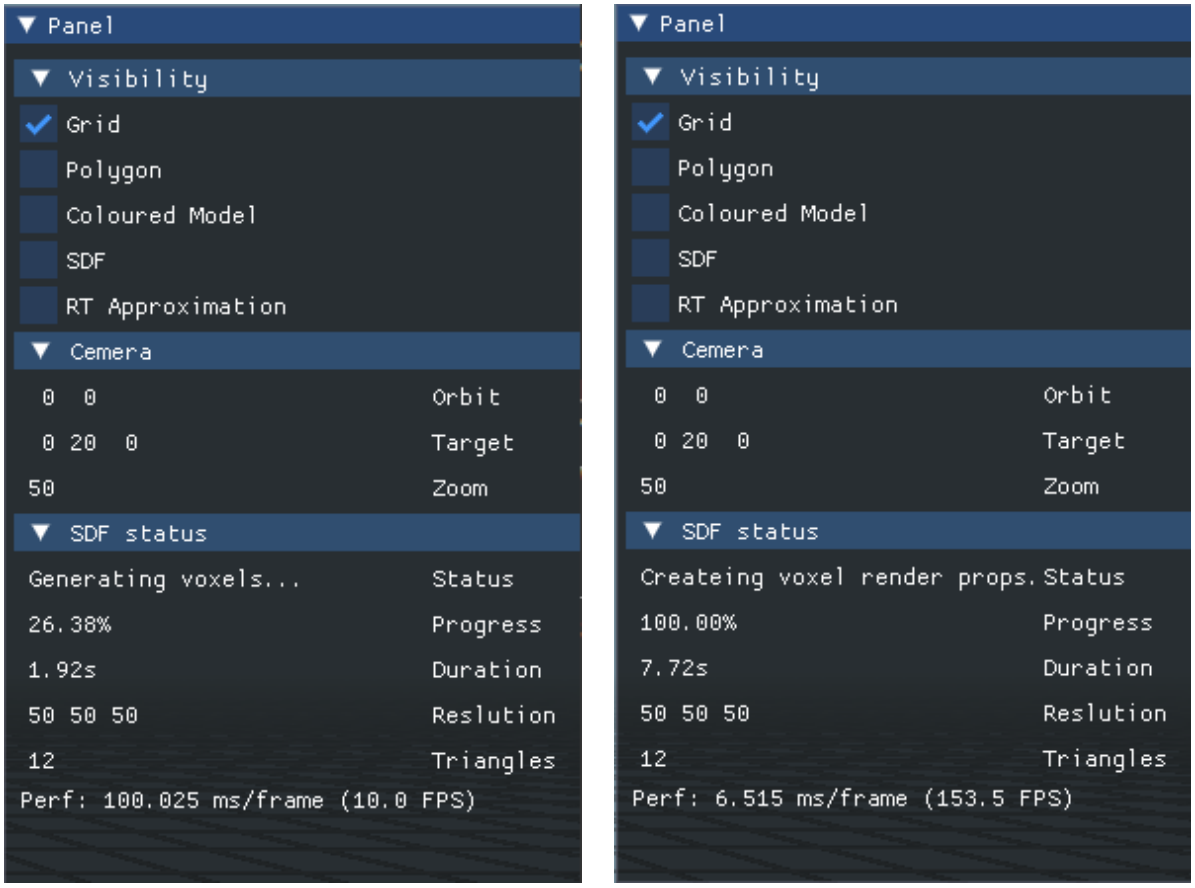
(d) Spot (e) Stanford Lucy

Figure 5.1: Rendering test for several models

5.1.3 GUI update

Since the computation is time-consuming, the application needs to monitor the progress of computation; the percentage of the whole model computation is shown in the debug area of the GUI panel. The GUI debug status should keep updating during the process of computing the signed distance field and provide a hint when it is finished.

The GUI panel updates the computation status properly and provides the rendering hint when the computation is finished, as shown in figure 5.2.

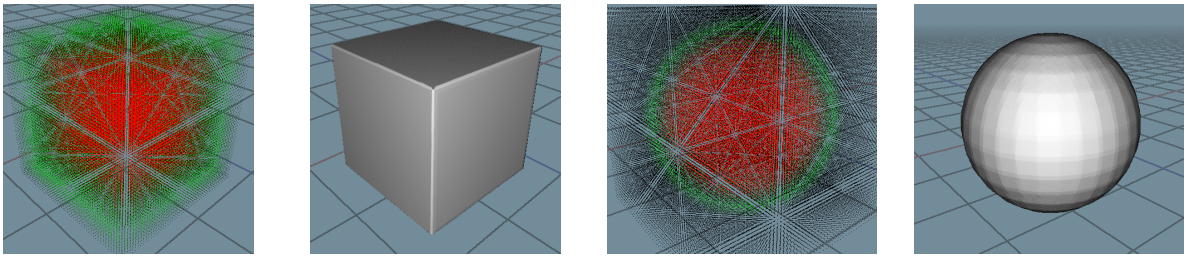


(a) Update (b) Sphere

Figure 5.2: The Debug status update properly

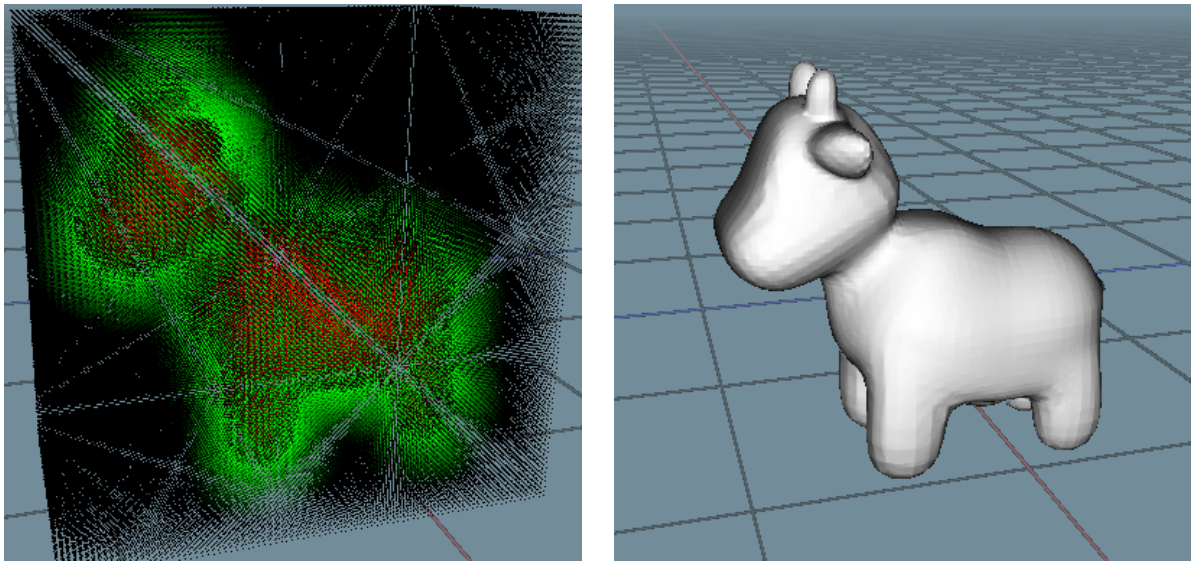
5.1.4 SDF

If the SDF has been calculated, the application reads in the generated SDF file to visualize the signed distance field. As shown in 4.6, the field of Stanford Bunny can be visualized properly. The result of other models can be seen in figure 5.3, 5.4 and 5.5. The effect of different parameter settings on the quality of the SDF is discussed in 5.2.1.



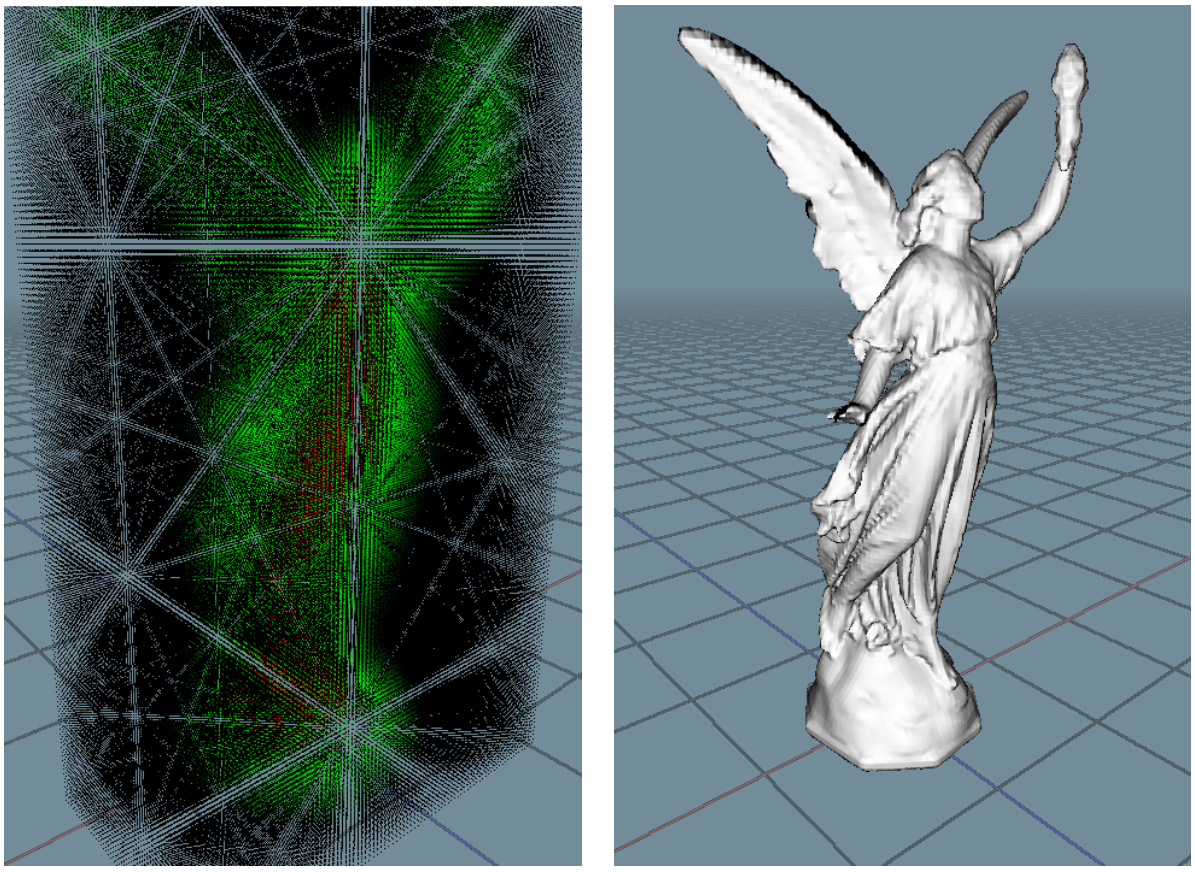
(a) Colour Volume (b) Ray Tracing (c) Colour Volume (d) Ray Tracing

Figure 5.3: The SDF visualization of Cube(a,b) and Sphere(c,d)



(a) Colour Volume (b) Ray Tracing Approximation

Figure 5.4: The SDF visualization of Spot



(a) Colour Volume (b) Ray Tracing Approximation

Figure 5.5: The SDF visualization of Stanford Lucy

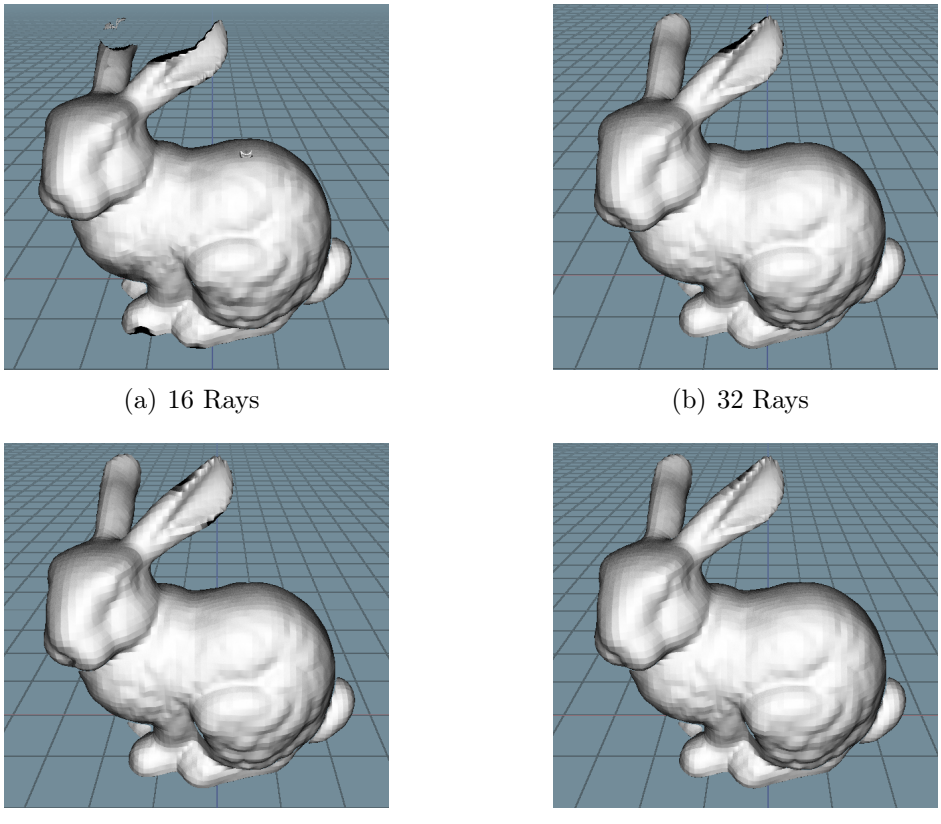
5.2 Evaluation

Three parameters influence the quality and generation performance of the SDF. The first one is the number of sample rays in each voxels. Another one is the resolution, which influence the number of sample voxels. The last one is triangle number of the mesh. The effect of these three parameters will be explored.

5.2.1 SDF quality

5.2.1.1 Sample Ray Number

The number of sample rays for each voxel affect the accuracy of distance function computation, if the ray is sparse, then the error of distance value for each sample voxel can be serious. Figure 5.6 provides the SDF approximation results in different ray number settings, the resolution is set as 50*63*64.



(a) 16 Rays

(b) 32 Rays

(c) 64 Rays

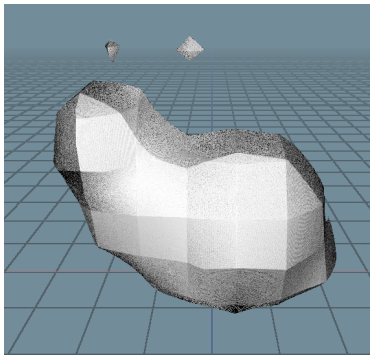
(d) 128 Rays

Figure 5.6: The SDF approximation results for different ray number

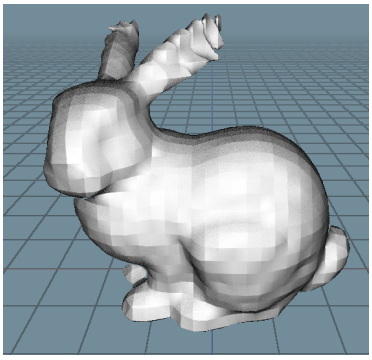
As can be seen in figure 5.6, when the ray number is low (a,b), there are obvious gaps in the parts of bunny'ears. As the number of rays increases, the error gradually disappears and the shape of Stanford Bunny has been nicely reproduced. Therefore, with a rational setting of ray number, the qualify of SDF generation meets the requirement.

5.2.1.2 Resolution

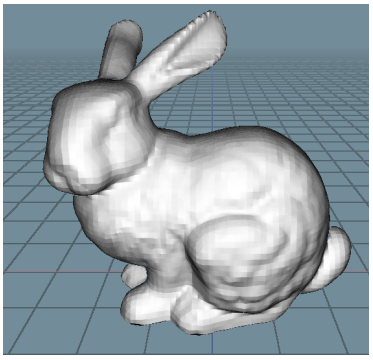
A high quality signed distance field can be generated with more sample voxels. However, using more sampled voxels can significantly increase the amount of computation and increase the generation time, especially for CPU implemented projects. In this section, the Stanford Bunny is used to explore the influence the effect of resolution on the quality of the generated SDFs since it is one of the most common 3D test model and can easily compare with other's work. The ray tracing visualisation mode is used to show the result.



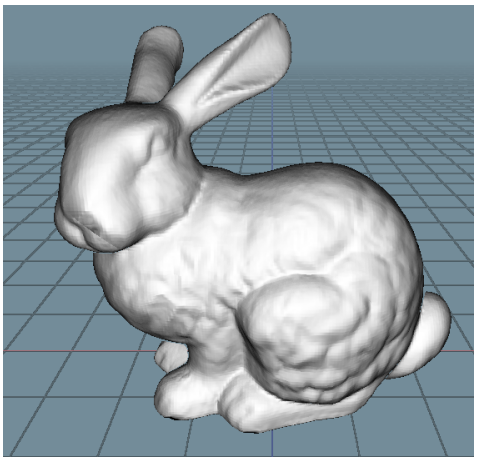
(a) 10*12*12



(b) 25*31*32



(c) 50*63*64



(d) 120*153*154

(e) DeepSDF [58]

Figure 5.7: The SDF approximation results for different resolutions and comparison with DeepSDF [58]

As shown in figure 5.7, the shape can be roughly reproduced with a resolution of 25*31*32. The recover result of DeepSDF [58] is more smooth but lose a lot of details like the fur balls. Conversely, when setting the voxel number on the shortest axis as 120, the most details of the Stanford Bunny mesh can be recovered. Therefore, the quality of SDF is satisfying in this resolution.

5.2.2 Performance

This section will explore the influence of resolution, sample ray number and mesh triangle number on the generation performance. The evaluation has experimented on the computer of the School of Computing.

The configuration of the testing computer is as follows:

CPU	Intel 11th Gen Core i7-11700 2.5GHz
RAM	32GB DDR4 3200MHz
GPU	NVIDIA GeForce RTX 3080
OS	Windows 10 Enterprise x64 20H2

5.2.2.1 Sample Ray Number

The cube with 12 triangles is used to test the influence of the sample ray number since the influence of the triangle number can be avoided in this setting. The number of sample voxels is set as 6.4k to show the change in generation time. The results can be seen in figure 5.8.

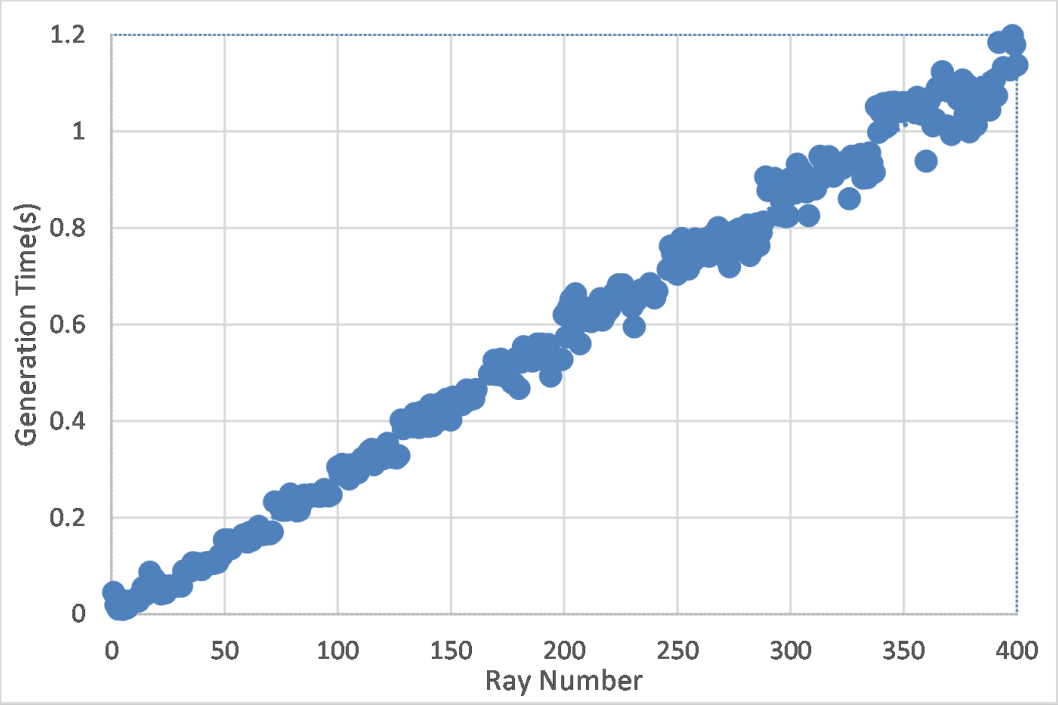
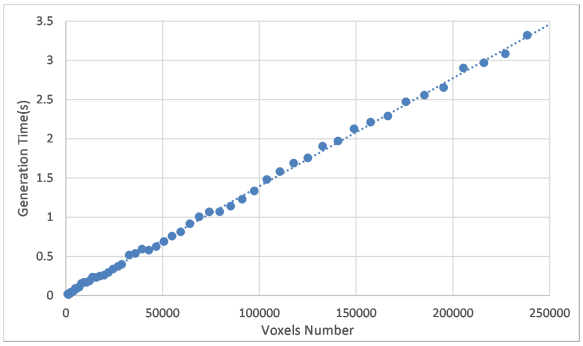


Figure 5.8: Generation time results with various ray number

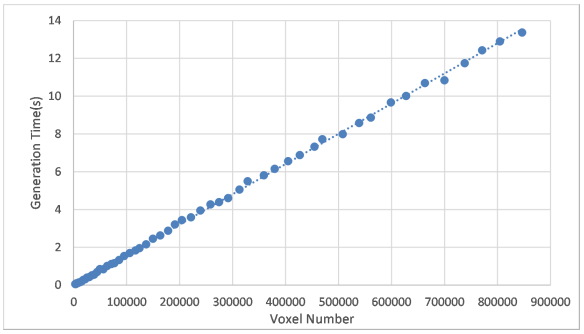
As shown in figure 5.8, the generation time rises linearly as the ray number increases. Therefore, the generation time is only related to the number of rays for each sample voxel when other parameters are unchanged.

5.2.2.2 Resolution

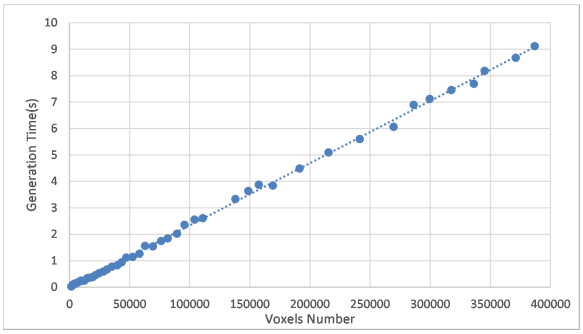
The resolution influence is tested through common 3D models: SPhere, Spot, Stanford Bunny and Stanford Lucy, while the ray number is set as 64 per voxel. The results can be seen in figure 5.2.2.2.



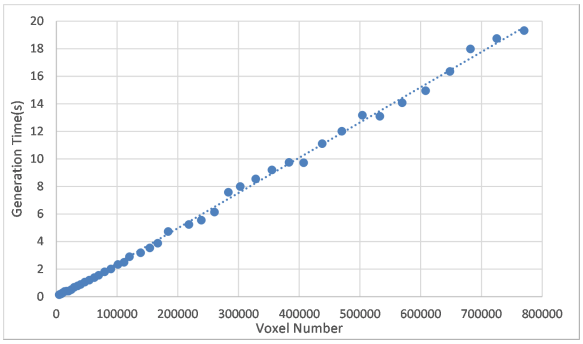
(a) Sphere



(b) Spot



(c) Stanford Bunny



(d) Stanford Lucy

As shown in figure 5.2.2.2, as the voxel number increase, the SDF generation time experience a linear growth. At this point, the generation time is only positively correlated with the number of voxels. Besides, as seen from the figure, generating the SDF with around 800 thousand sample voxels can be finished within 20 seconds.

5.2.2.3 Mesh Triangle Number

The sphere meshes with different numbers of triangles are used to test the influence of mesh triangle numbers, which can avoid the influence of model shape. The sphere with 1k, 5.5k, 7.5k, 9k and 13k triangles are used, while the ray number per voxel is still set as 64. The results can be seen in figure 5.9.

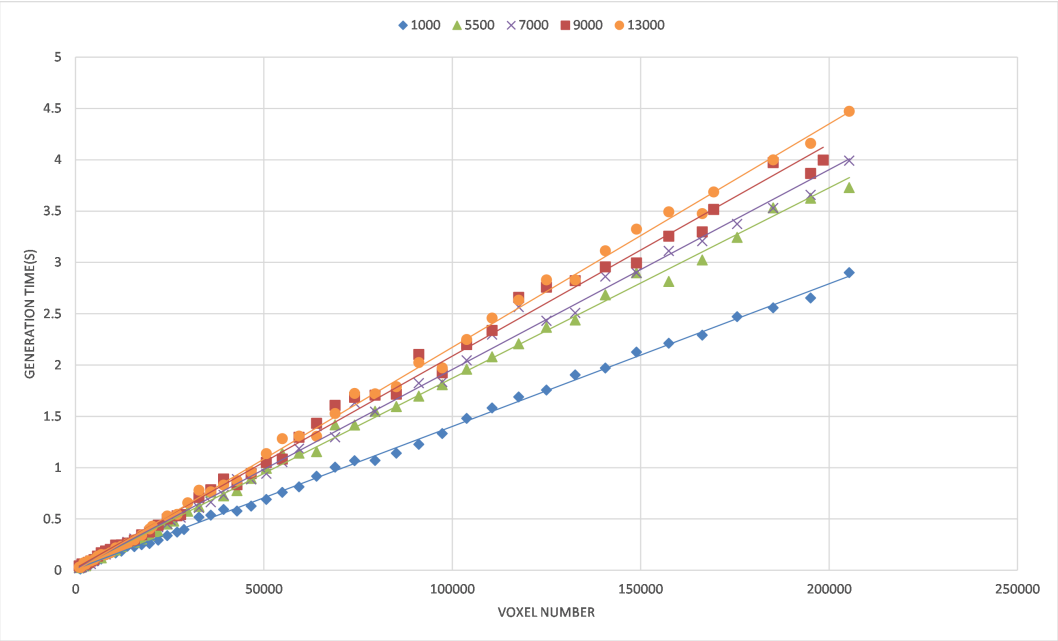


Figure 5.9: Generation time results with various triangle number

As shown in figure 5.9, the different triangle number affects the slope of the generation time with respect to the number of voxels. For the models with shape, generating an SDF with the same sample voxels cost more time if the mesh triangle number is larger.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this section, the objectives will be reviewed and evaluated based on the implementation, and then the existing problems will be summarised and finally, a conclusion.

There are five objectives set in the planning stage. The first one is an OpenGL model visualisation application completed on June 27th, three days earlier than initially planned. The implementation of the SDF generation algorithm consumes the most time. The generation function initially used the brutal force method to ensure the data structure was built correctly; this process cost one week. Then the ray-intersection algorithm was implemented in four days and used for two days to adapt to the existing code. After that, the task became the KD-tree structure implementation, which takes the longest time to complete and test. This implementation was finished on July 20th. Then the visualisation shader of SDF was implemented four days later. Finally, the application was tested and evaluated on a personal laptop and the computer of the School of Computing and was compiled and run correctly on both. The implementation and evaluation results have been shown in Chapter 4 and 5. Therefore, all of the objectives have been achieved.

The main problem of this project, for now, is the performance. Although the performance is efficient for CPU implementation, using the current implementation to generate the SDF for a single complex model with satisfying quality costs of at least dozens of seconds is unacceptable for real-time rendering. In addition, the visualisation can be improved, as shown in 5.1.4, but the result of colour volume mode is still not continuous. Besides, more visualisation modes like Ray Marching can be added. Finally, the application only supports a single model for now, which cannot meet the requirement of the modern graphic industry.

In conclusion, the targets of the initial plan have been completed, and it is helpful to enhance the understanding of SDF, ray tracing and KD-tree structure. More optimisation strategies and algorithms can be added to the implementation to satisfy the requirement of modern industry and make the comparison between different solutions. Since my previous experience in computer graphics is zero, implementing this project improved my knowledge of Graphics APIs, SDF, system design, programming,

evaluation and academic writing. I hope these skills will be applied to my future career.

6.2 Future Work

The main problem for the current implementation is the generation time, which can be solved by switching the implementation to GPU. The compute shader is an ideal choice. The first work for the future is implementing a GPU compute version to accelerate the generation speed. Besides, the BVH structure mentioned in 2.2.3 is widely used in the industry. Adding BVH to the application will help compare the pros and cons of different acceleration structures. The next plan is to provide support for the complex scene. The model class should be upgraded to process the multiple meshes, which meet the realistic situation of the industrial applications. Finally, the application can be applied to more problems like sphere-tracing, soft shadow, and ray marching, and these scenarios will be gradually added to the implementation in the future.

Bibliography

[1] S. H. Ahn. OpenGL Sphere, 2018. Accessed: 17/05/2022.

[2] T. Akenine-Möller and B. Trumbore. Fast, minimum storage ray/triangle intersection. *ACM SIGGRAPH 2005 Courses*, page 7–es, 2005. Accessed: 09/06/2022.

[3] J. Baerentzen and H. Aanaes. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):243–253, 2005. Accessed: 10/06/2022.

[4] S. Balaji and M. S. Murugaiyan. Waterfall vs. v-model vs. agile: A comparative study on sdlc. *International Journal of Information Technology and Business Management*, 2(1):26–30, 2012. Accessed: 22/03/2022.

[5] G. Barill, N. G. Dickson, R. Schmidt, D. I. W. Levin, and A. Jacobson. Fast Winding Numbers for Soups and Clouds. *ACM Trans. Graph.*, 37(4), 2018. Accessed: 18/03/2022.

[6] D. Blythe. The Direct3D 10 System. In *ACM SIGGRAPH 2006 Papers*, page 724–734, 2006. Accessed: 20/06/2022.

[7] D. Blythe. Direct3D 10. *GPU Shading and Rendering, SIGGRAPH*, 6, 2006. Accessed: 18/06/2022.

[8] G. Brain. TensorFlow, 2022. Accessed: 13/06/2022.

[9] R. Bán and G. Valasek. First Order Signed Distance Fields. In A. Wilkie and F. Banterle, editors, *Eurographics 2020 - Short Papers*, pages 33–36, 2020. Accessed: 07/04/2022.

[10] A. Cedilnik, B. Hoffman, B. K. adn Ken Martin, and A. Neundorf. CMake Documentation, 2022. Accessed: 25/04/2022.

[11] D. S. Champion Pierre, Coşku Baş. GLFW Documentation, 2022. Accessed: 25/04/2022.

[12] D. Cohen-Or, A. Solomovic, and D. Levin. Three-Dimensional Distance Field Metamorphosis. *ACM Trans. Graph.*, 17(2):116–141, 1998. Accessed: 17/05/2022.

[13] C. S. Committee. ISO International Standard ISO/IEC 14882: 2020, Programming Language C++. Technical report, Tech. Rep., 2020, http://www.openstd.org/jtc1/sc22/wg21, 2020. Accessed: 13/05/2022.

[14] T. Q. Company. qmake Manual, 2022. Accessed: 25/04/2022.

[15] T. Q. Company. QML Applications, 2022. Accessed: 25/04/2022.

[16] T. Q. Company. Qt Creator Manual, 2022. Accessed: 25/04/2022.

[17] T. Q. Company. Qt, 2022. Accessed: 25/04/2022.

[18] O. Cornut. DearImGui, 2022. Accessed: 25/04/2022.

[19] G.-T. Creation. GLM Documentation, 2022. Accessed: 25/04/2022.

[20] Dav1dde. GLAD, 2022. Accessed: 25/04/2022.

[21] J. de Vries. Hello Triangle, 2022. Accessed: 15/03/2022.

[22] D. Eberly. Distance between point and triangle in 3D. *Magic Software*, 2008. Accessed: 14/05/2022.

[23] D. Enright, S. Marschner, and R. Fedkiw. Animation and Rendering of Complex Water Surfaces. *ACM Trans. Graph.*, 21(3):736–744, 2002. Accessed: 24/04/2022.

[24] C. Ericson. *Real-time collision detection*. Crc Press, 2004. Accessed: 15/05/2022.

[25] S. Fisher and M. C. Lin. Deformed Distance Fields for Simulation of Non-Penetrating Flexible Bodies. In *Computer Animation and Simulation 2001*, pages 99–111. Springer Vienna, 2001. Accessed: 21/03/2022.

[26] T. Foley and J. Sugerman. KD-Tree Acceleration Structures for a GPU Raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, page 15–22, 2005. Accessed: 28/05/2022.

[27] F. Format. FBX File Format, 2022. Accessed: 25/04/2022.

[28] F. Format. OBJ File Format, 2022. Accessed: 25/04/2022.

[29] P. S. Foundation. Python, 2022. Accessed: 13/06/2022.

[30] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, page 249–254, 2000. Accessed: 14/05/2022.

[31] E. Games. Mesh Distance Fields, 2022. Accessed: 03/04/2022.

[32] C. Green. Improved Alpha-Tested Magnification for Vector Textures and Special Effects. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, page 9–18, 2007. Accessed: 22/04/2022.

[33] K. Group. OpenGL Wiki. Accessed: 27/05/2022.

[34] K. Group. Default Framebuffer, 2022. Accessed: 23/04/2022.

[35] K. Group. Fragment Shader, 2022. Accessed: 23/04/2022.

[36] K. Group. Geometry Shader, 2022. Accessed: 23/04/2022.

[37] K. Group. Per-Sample Processing, 2022. Accessed: 23/04/2022.

[38] K. Group. Primitive Assembly, 2022. Accessed: 23/04/2022.

[39] K. Group. Rendering Pipeline Overview, 2022. Accessed: 23/04/2022.

[40] K. Group. Tessellation, 2022. Accessed: 23/04/2022.

[41] K. Group. Vertex Shader, 2022. Accessed: 23/04/2022.

[42] K. Group. Vertex Specification, 2022. Accessed: 23/04/2022.

[43] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex Rigid Bodies with Stacking. *ACM Trans. Graph.*, 22(3):871–878, 2003. Accessed: 21/04/2022.

[44] J. C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996. Accessed: 24/04/2022.

[45] V. Havran. Heuristic ray shooting algorithms. 2000. Accessed: 19/04/2022.

[46] R. Ierusalimschy, W. Celes, and L. H. de Figueiredo. Lua Documentation, 2022. Accessed: 25/04/2022.

[47] M. Jones, J. Baerentzen, and M. Sramek. 3D distance fields: a survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, 2006. Accessed: 24/04/2022.

[48] M. W. Jones. 3D distance from a point to a triangle. *Department of Computer Science, University of Wales Swansea Technical Report CSR-5*, 1995. Accessed: 15/05/2022.

[49] T. Ju. Robust Repair of Polygonal Models. In *ACM SIGGRAPH 2004 Papers*, page 888–895, 2004. Accessed: 16/05/2022.

[50] Keras. Keras, 2022. Accessed: 13/06/2022.

[51] J. Kessenich, G. Sellers, and D. Shreiner. *OpenGL Programming Guide: The official guide to learning OpenGL, version 4.5 with SPIR-V*. Addison-Wesley Professional, 2016. Accessed: 29/05/2022.

[52] D. Koschier, C. Deul, M. Brand, and J. Bender. An hp-adaptive discretization algorithm for signed distance field generation. *IEEE Transactions on Visualization and Computer Graphics*, 23(10):2208–2221, 2017. Accessed: 27/03/2022.

[53] M. Lutz. *Programming python*. " O'Reilly Media, Inc.", 2001. Accessed: 13/06/2022.

[54] Marie. How to make an infinite grid., 2020. Accessed: 26/04/2022.

[55] C. Maurer, R. Qi, and V. Raghavan. A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(2):265–270, 2003. Accessed: 05/05/2022.

[56] Microsoft. Direct3D, 2022. Accessed: 17/06/2022.

[57] NVIDIA. DLSS, 2022. Accessed: 13/06/2022.

[58] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove. DeepSDF: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 165–174, 2019. Accessed: 15/04/2022.

[59] B. Payne and A. Toga. Distance field manipulation of surface models. *IEEE Computer Graphics and Applications*, 12:65–71, 1992. Accessed: 10/04/2022.

[60] J. Perkins. Premake5 Documentation, 2022. Accessed: 25/04/2022.

[61] I. Quilez. distance functions, 2018. Accessed: 17/04/2022.

[62] I. Quilez. ShaderToy, 2022. Accessed: 17/04/2022.

[63] I. Quilez. kd-tree sdf, 2022. Accessed: 17/04/2022.

[64] A. Rosenfeld and J. Pfaltz. Distance functions on digital pictures. *Pattern Recognition*, 1(1):33–61, 1968. Accessed: 25/04/2022.

[65] A. Rosenfeld and J. L. Pfaltz. Sequential Operations in Digital Picture Processing. *J. ACM*, 13(4):471–494, 1966. Accessed: 24/04/2022.

[66] M. Sanchez, O. Fryazinov, P.-A. Fayolle, and A. Pasko. Convolution Filtering of Continuous Signed Distance Fields for Polygonal Meshes. *Computer Graphics Forum*, 34(6):277–288, 2015. Accessed: 18/04/2022.

[67] M. Sanchez, O. Fryazinov, and A. A. Pasko. Efficient evaluation of continuous signed distance to a polygonal mesh. In *SCCG*, 2012. Accessed: 15/06/2022.

[68] G. Sellers and J. Kessenich. *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016. Accessed: 29/05/2022.

[69] D. Seyb, A. Jacobson, D. Nowrouzezahrai, and W. Jarosz. Non-Linear Sphere Tracing for Rendering Deformed Signed Distance Fields. *ACM Trans. Graph.*, 38(6), 2019. Accessed: 18/05/2022.

[70] N. Sherriff. *Learn Qt 5: Build modern, responsive cross-platform desktop applications with Qt, C++, and QML*. Packt Publishing Ltd, 2018. Accessed: 24/04/2022.

[71] B. Stroustrup. *The C++ programming language*. Addison-Wesley, 1985. Accessed: 13/04/2022.

[72] G. Thürrner and C. A. Wüthrich. Computing Vertex Normals from Polygonal Facets. *Journal of Graphics Tools*, 3(1):43–46, 1998. Accessed: 20/05/2022.

[73] tinyobjloader. tinyobjloader, 2022. Accessed: 25/04/2022.

[74] W. van der Laan. Doom 3 GPL source release Bounds.cpp, 2014. Accessed: 23/04/2022.

[75] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. 26(1):6–es, 2007. Accessed: 25/05/2022.

[76] H. Xu and J. Barbič. Signed Distance Fields for Polygon Soup Meshes. In *Proceedings of Graphics Interface 2014*, page 35–41, 2014. Accessed: 17/04/2022.

Appendices

Appendix A

Premake Script

```
workspace "SDF"
  language "C++"
  cppdialect "C++17"

  platforms { "x64" }
  configurations { "release", "debug" }

  flags "NoPCH"
  flags "MultiProcessorCompile"

  startproject "sdf"

  debugdir "%{wks.location}"
  objdir "prebuild/%{cfg.buildcfg}-%{cfg.platform}-%{cfg.toolset}"
  targetsuffix "-%{cfg.buildcfg}-%{cfg.platform}-%{cfg.toolset}"

  filter "toolset:gcc or toolset:clang"
    linkoptions { "-pthread" }
    buildoptions { "-march=native", "-Wall", "-pthread" }
  filter "toolset:msc-*"
    defines { "_CRT_SECURE_NO_WARNINGS=1" }
    defines { "_SCL_SECURE_NO_WARNINGS=1" }
    buildoptions { "/utf-8" }

  filter "*"

  -- default libraries
  filter "system:linux"
    links "dl"
  filter "system:windows"

  filter "*"

  -- default outputs
  filter "kind:StaticLib"
    targetdir "prebuild/lib/"
  filter "kind:ConsoleApp"
    targetdir "prebuild/bin/"
    targetextension ".exe"

  filter "*"

```

```
43      --configurations
44      filter "debug"
45        symbols "On"
46        defines { "_DEBUG=1" }
47      filter "release"
48        optimize "On"
49        defines { "NDEBUG=1" }
50      filter "*"
51
52    include "extern"
53
54    project "sdf"
55      includedirs( "src" );
56      local sources = {
57        "src/**.cpp",
58        "src/**.hpp",
59        "src/**.hxx",
60        "src/**.h"
61      }
62
63      kind "ConsoleApp"
64      location "src"
65
66      files( sources )
67
68      links "x-glad"
69      links "x-glad"
70      links "x-glfw"
71      links "x-tinyobj"
72      links "x-imgui"
73      dependson "x-glm"
74
75      filter "system:linux"
76        links "tbb"
```

Listing A.1: Premake Script for solution generation

```
    -- Third party projects
    includedirs( "glad/include" );
    includedirs( "glfw/include" );
    includedirs( "glm" );
    includedirs( "tinyobjloader" );
    includedirs( "imgui" );

    project( "x-glad" )
      kind "StaticLib"
      location "."
      files( "glad/src/glad.c" )


    project( "x-glfw" )
      kind "StaticLib"
      location "."
      filter "system:linux"
        defines { "_GLFW_X11=1" }
      filter "system:windows"
        defines { "_GLFW_WIN32=1" }
      filter "*"

      files {
        "glfw/src/context.c",
        "glfw/src/egl_context.c",
        "glfw/src/init.c",
        "glfw/src/input.c",
        "glfw/src/internal.h",
        "glfw/src/mappings.h",
        "glfw/src/monitor.c",
        "glfw/src/null_init.c",
        "glfw/src/null_joystick.c",
        "glfw/src/null_joystick.h",
        "glfw/src/null_monitor.c",
        "glfw/src/null_platform.h",
        "glfw/src/null_window.c",
        "glfw/src/platform.c",
        "glfw/src/platform.h",
        "glfw/src/vulkan.c",
        "glfw/src/window.c",
        "glfw/src/osmesa_context.c"
      };
      filter "system:linux"
        files {
          "glfw/src/posix_*",
          "glfw/src/x11_*",
          "glfw/src/xkb_*",
```

```
48            "glfw/src/glx_*",
49            "glfw/src/linux_*",
50          };
51        filter "system:windows"
52          files {
53            "glfw/src/win32_*",
54            "glfw/src/wgl_*",
55          };
56        filter "*"
57
58      project( "x-glm" )
59        kind "Utility"
60        location "."
61        files( "glm/glm/**.h" )
62        files( "glm/glm/**.hpp" )
63        files( "glm/glm/**.inl" )
64
65      project( "x-tinyobj" )
66        kind "StaticLib"
67        location "."
68        files( "tinyobjloader/tiny_obj_loader.cc" )
69
70      project( "x-imgui" )
71        kind "StaticLib"
72        location "."
73        files( "imgui/imgui.cpp" )
74        files( "imgui/imgui_draw.cpp" )
75        files( "imgui/imgui_widgets.cpp" )
76        files( "imgui/imgui_tables.cpp" )
77        files( "imgui/imgui_demo.cpp" )
78        files( "imgui/backends/imgui_impl_glfw.cpp" )
79        files( "imgui/backends/imgui_impl_opengl3.cpp" )
```

Listing A.2: Premake Script for external library

Appendix B

Infinity Grids Scene Shaders

```
#version  460 core

uniform mat4 MVP_INV;

layout(location = 1) out vec3 nearPoint;
layout(location = 2) out vec3 farPoint;

const vec2 positions[4] = vec2[](
  vec2(-1, -1),
  vec2(+1, -1),
  vec2(-1, +1),
  vec2(+1, +1)
);
vec3 unproj(float x, float y, float z ) {
  vec4 hp = MVP_INV *vec4(x, y, z, 1.0);
  // hp.y = -hp.y;
  // hp.z = -hp.z;
  return hp.xyz / hp.w;
}
void main() {
  vec3 p = vec3( positions[gl_VertexID], 1 );
  nearPoint = unproj(p.x, p.y, 0.0 ).xyz; // unprojecting on the near
 plane
  farPoint = unproj(p.x, p.y, 1.0 ).xyz; // unprojecting on the far
 plane

  gl_Position = vec4( positions[gl_VertexID], 1, 1.0); // using
 directly the clipped coordinates
}
```

Listing B.1: Vertex shader of Infinity Grids Scene

```
#version  460 core

uniform mat4 MVP;
uniform vec2 NF;
uniform vec4 BCOLOR;

layout(location = 1) in vec3 nearPoint;
layout(location = 2) in vec3 farPoint;
layout(location = 0) out vec4 outColor;

vec4 grid(vec3 fragPos3D, float scale) {
  vec2 coord = fragPos3D.xz * scale; // use the scale variable to set the distance between the lines
  vec2 derivative = fwidth(coord);
  vec2 grid = abs(fract(coord - 0.5) - 0.5) / derivative;
  float line = min(grid.x, grid.y);
  float minimumz = min(derivative.y, 1);
  float minimumx = min(derivative.x, 1);
  vec4 color = vec4( 0.25,0.25,0.25, 1.0 - min(line, 1.0));

  float bs = 5;

  // z axis
  if(fragPos3D.x > -bs * minimumx && fragPos3D.x < bs * minimumx)
    color.z = 1.0;
  // x axis
  if(fragPos3D.z > -bs * minimumz && fragPos3D.z < bs * minimumz)
    color.x = 1.0;
  if (color.w<=0 ) {
    discard;
  }
  return color * color.w;
}


float compute_linear_depth(vec3 pos) {
  vec4 clip_space_pos =  MVP * vec4(pos.xyz, 1.0);
  float clip_space_depth = (clip_space_pos.z / clip_space_pos.w) * 2.0 - 1.0; // put back between -1 and 1
  float near = NF[0];
  float far = NF[1];

  float linear_depth = (2.0 * near * far) / (far + near - clip_space_depth * (far - near)); // get linear value between 0.01 and 100
  return linear_depth / far; // normalize
}
```

```
44   void main() {
45     float t = -nearPoint.y / (farPoint.y - nearPoint.y);
46     if ( t<=0)
47       discard;
48
49     vec3 frag_pos = nearPoint + t * (farPoint - nearPoint);
50
51     vec4 clip_space_pos = MVP * vec4(frag_pos.xyz, 1.0);
52     clip_space_pos =  (clip_space_pos / clip_space_pos.w);
53
54     //gl_FragDepth = clip_space_pos.z;
55
56     float linear_depth = compute_linear_depth(frag_pos);
57     float fading = max(0, (0.5 - linear_depth));
58
59
60     outColor = mix ( BCOLOR, grid( frag_pos , 0.1 ), fading) ;
61 }
```

Listing B.2: Fragment shader of Infinity Grids Scene

Appendix C

Shader of ray-tracing SDF visualisation

```
#version  460 core

uniform mat4 MVP_INV;


layout(location = 1) out vec3 far_point;



const vec2 positions[4] = vec2[](
  vec2(-1, -1),
  vec2(+1, -1),
  vec2(-1, +1),
  vec2(+1, +1)
);

vec3 unproj(float x, float y, float z ) {
  vec4 hp = MVP_INV *vec4(x, y, z, 1.0);
  return hp.xyz / hp.w;
}

void main() {
  vec3 p = vec3( positions[gl_VertexID], 1 );
  far_point = unproj(p.x, p.y, 1.0 ).xyz; // unprojecting on the far
   plane

  gl_Position = vec4( positions[gl_VertexID], 1, 1.0); // using
   directly the clipped coordinates
}
```

Listing C.1: Vertex shader of ray-tracing SDF visualisation

```
#version 460

uniform mat4  MVP;
uniform vec3  CAMERA_POS;
uniform vec3  BOX0;
uniform vec3  BOX1;

uniform sampler3D VOXELS;

layout(location = 1) in vec3 far_point;
layout(location = 0) out vec4 out_color;

const float ESP = 0.1;


float sdf_box(vec3 pos, vec3 hsize) {
vec3 q = abs(pos) - hsize;
return length(max(q, 0.0)) + min(max(q.x, max(q.y, q.z)), 0.0);
}

// Definition of signed distance funtion called from
float SDF(vec3 pos) {
vec3 center = (BOX1 + BOX0) * 0.5;
vec3 size = (BOX1 - BOX0);
vec3 hsize = size * 0.5;

float s =  sdf_box(pos - center, hsize);

if ( s < ESP ) {


  vec3 uvw = (pos - BOX0) /size ;

  return texture( VOXELS,uvw ).x;

}
return s ;

}

void main() {


vec3 D = normalize( far_point - CAMERA_POS );
vec3 P = CAMERA_POS; // the current ray position
for(int i = 0; i < 30; ++i) {
  float s0 = SDF(P);
```

```
48      // if(s0 < 0 ) { // ray starting from inside the object
49      //   out_color = vec4(1, 0, 0, 1); // paint red
50      //   return;
51      // }
52      if( s0 < ESP ) { // the ray hit the implicit surfacee
53      float eps = ESP;
54      float sx = SDF(P + vec3(eps, 0, 0)) - s0;
55      float sy = SDF(P + vec3(0, eps, 0)) - s0;
56      float sz = SDF(P + vec3(0, 0, eps)) - s0;
57      vec3 nrm = normalize(vec3(sx, sy, sz)); // normal Dection
58      float c = - dot(nrm, D); // Lambersian reflection. The light is at
        the camera position.
59      out_color = vec4(c, c, c, 1);
60
61      vec4 clip_space_pos = MVP * vec4( P.xyz, 1.0);
62      clip_space_pos =  (clip_space_pos / clip_space_pos.w);
63      gl_FragDepth = clip_space_pos.z;
64
65      return;
66      }
67      P += s0 * D; // advance ray
68    }
69    discard;
70  }
```

Listing C.2: Fragment shader of ray-tracing SDF visualisation