Real-time Rendering of 3D "Fractal-like" Geometry

Deliverable 1: Final Year Dissertation

03/11/2021

by

Solomon Baarda

Meng Software Engineering

Heriot-Watt University

Supervisor: Dr Benjamin Kenwright

Second Reader: Ali Muzaffar

Abstract

TODO

a short description of the project and the main work to be carried out – probably between one and two hundred words.

I, Solomon Baarda confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

Date:

Table of Contents

Table of Figures

Table of Tables

Common Definitions

Table 1: Common Definitions

Word	Definition
Frame	
Geometry	
Ray	
Render	
Method/constant overloading	

Common Abbreviations

Table 2: Common Abbreviations

Word	Abbreviation
CPU	Central Processing Unit
DF	Distance Function
FPS	Frames per second
GPU	Graphics Processing Unit
PC	Personal computer
SDF	Signed distance function

1 Introduction

TODO

https://github.com/SolomonBaarda/dissertation

1.1 Aims & Objectives

The aim of this project is to develop a prototype real-time rendering engine, capable of viewing complex 3D “fractal-like” geometry. The performance of the engine will be benchmarked across various systems to determine whether the “real-time” requirement of the project has been achieved.

TODO finish

1.2 Project Description

The application will be benchmarked across several computers of varying spec to determine if the real-time requirement of the application has been achieved. For the scope of this project, real-time has been defined as running at 1920x1080 with a minimum of 60 frames per second (fps), as this is the current industry standard for PC applications.

The benchmark scene has yet to be fully defined, but it must be non-trivial to render. This means it should contain multiple geometries (both fractal and primitive) and multiple lights while also making use of advanced rendering features like ambient occlusion, soft shadows, and reflections. It is important that the scene is consistent as possible between separate runs, therefore, the camera should be either stationary or move through the scene on a fixed path to view the geometries.

The benchmark scene should run for a fixed duration (so it takes the same amount of time on all machines), and the total frame count can be recorded and compared between systems. In addition, the minimum fps and maximum fps achieved should also be recorded and compared.

1.3 Scope

The scope of the project has been carefully considered, and several stretch goals have been included in the requirements specification if good progress is made. Some initial experimentation has proved promising as the Mandel bulb fractal was able to be rendered using a modified ray marcher.

TODO finish

2 LITERATURE REVIEW

2.1 FRACTALS

In mathematics, a fractal is a complicated pattern built from simple repeated shapes, which are reduced in size every time they are repeated [1]. The term fractal was proposed by Benua Mandelbrot in 1975 to describe this new geometry and comes from the Latin word frāctus, meaning fractured [2]. Fractal geometry bears a striking resemblance to geometry created by nature, which often has complex repeated patterns while also being irregular and distorted.

Figure 1: Fractal in nature (https://www.flickr.com/photos/genista/2447322/in/photolist-dxvd)

In mathematics, fractals are created using equations TODO more theory here

Some recent work, such as Quilez [3], uses pixel shaders to render a 3D Julia set.

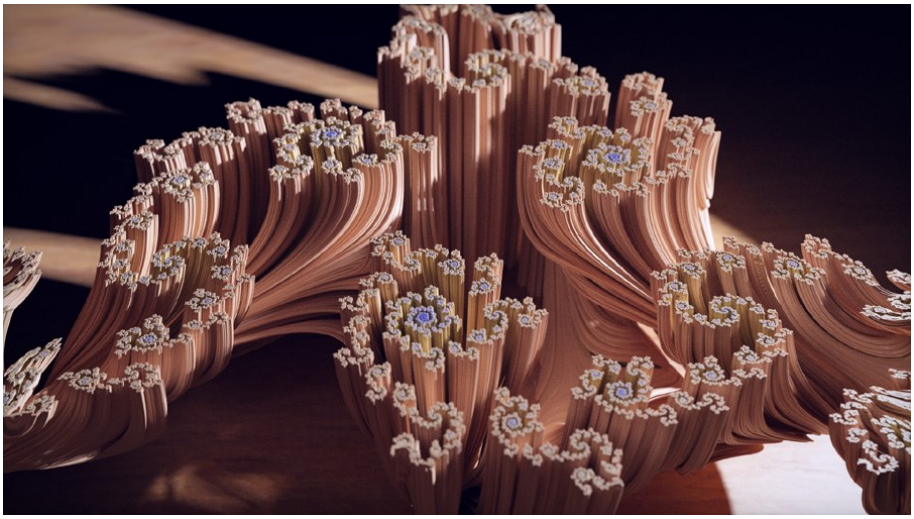

Figure 2: Ray marched Julia set, cut in half to expose the fractal interior (https://www.iquilezles.org/www/articles/juliasets3d/juliasets3d.htm)

The recent work da Silva et al in 2021 [4] took this a step further, using Nvidia DirectX Raytracing (DXR) shaders to render the visualisations of the 3D Julia set and Mandel bulb fractal.

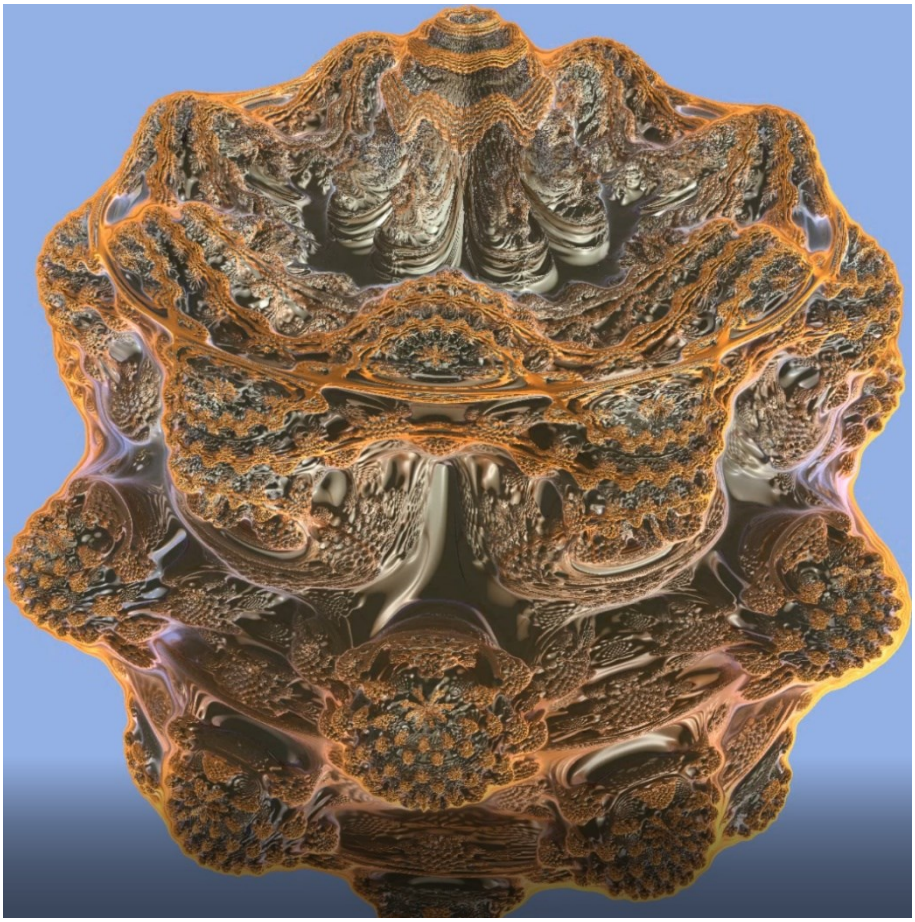

Figure 3: Render of Mandel bulb fractal created using DXR shaders (https://github.com/dsilvavinicius/realtime_rendering_of_complex_fractals)

TODO talk about pros/cons of existing work

Colour – orbit trap, as surface point transforms, look at how far away it gets from origin as it iterates through the transformation, min, max, sum, x,y,z etc

TODO talk about my experiments?

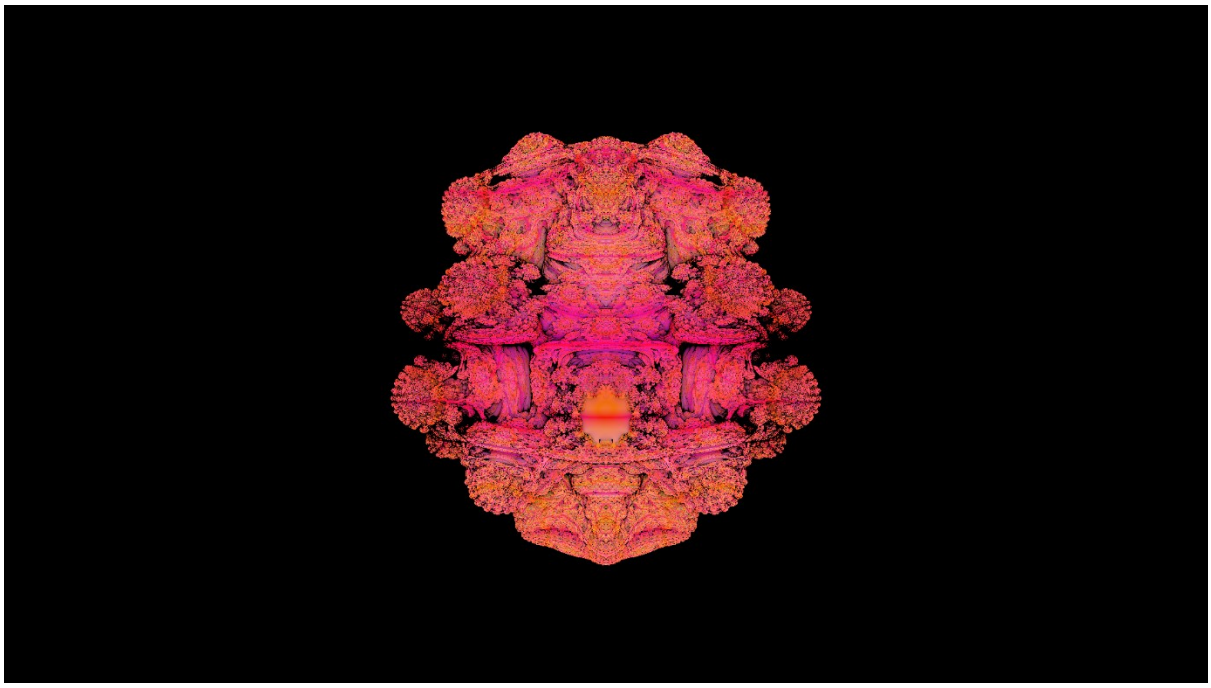

Figure 4: Mandel bulb experiment

Emphasise all existing work done in shaders, each scene in its own file – massive code duplication, no code reusability. Can we do better? Code inheritance + performance reasons for choosing C++ application.

2.2 Ray Tracing

In computer graphics, ray tracing is a method of rendering an image of a 3D scene, often with photorealistic detail. This is done by tracing the paths of light and simulating their effects on geometry by taking into consideration reflections, light refraction, and reflections of reflections [1].

When rendering an image using ray tracing, for each pixel in the camera, a ray (simply a line in 3D space) is extended or traced forwards from the camera position until it intersects with the surface of an object. From there, the ray can be absorbed or reflected by the surface and more rays can be sent

out recursively, which can be used to take into consideration light absorption, reflection, refraction, and fluorescence.

Ray tracing is ideal for photorealistic graphics, as it takes into consideration many of the properties of light, but because of this, it is computationally expensive. Often, ray tracers do not render images in real-time, and they can take hours to render a couple seconds of video. To make a ray tracer capable of rendering in real-time, many approximations must be made, or hybrid approaches used.

One of the limitations of ray tracing is that an accurate ray-surface intersection function must exist for every object in the scene. This is well suited for any Euclidian surfaces, such as primitives and meshes, which are made up of vertices, faces and edges, as points of intersection can be calculated relatively easily on these shapes. While in general, computing intersections is not cheap, there is a bigger problem. What do we do for any geometries for which a ray-surface intersection function does not exist [2], such as fractal geometries? The solution is ray marching.

2.3 Ray Marching

Ray marching is a variation of ray tracing, which differs in the method of detecting intersections between the ray and objects. Instead of a using a ray-surface intersection function, ray marching uses an iterative approach, where the current point is moved/marched along the ray in small increments until it lands on the surface of an object. For each point on the ray that is sampled, a distance estimator (DE) function is called, which returns the distance to the closest object in the scene. The ray is then marched forward by that distance, and the process repeated. If the distance function returns 0 at any point (or is close enough to an arbitrary epsilon value), then then the ray has collided with the surface of the geometry.

The diagram below shows a ray being marched from position p0 in the direction to the right. The distance estimation for each point is marked using the circle centred on that point.

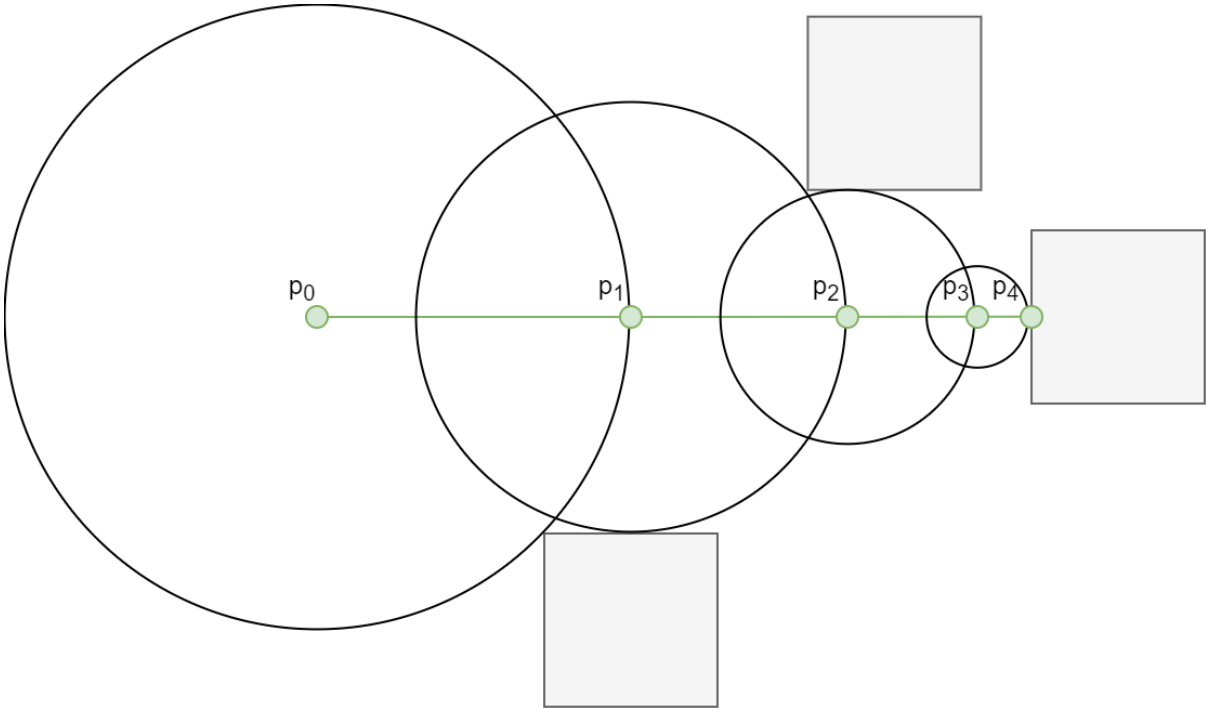

Figure 5: Ray marching diagram

The DE does not have to return the exact distance to an object, as for some objects this may not be computable, but it must never be larger than the actual value. If the value is too small though, then the ray marching algorithm becomes very inefficient so a fine balance must be found between accuracy and efficiency.

2.3.1 Benefits of Ray Marching

Ray marching may sound more computationally complex than ray tracing since it must complete multiple iterations of an algorithm do what ray tracing does in a single ray-surface intersection function, however, it does provide several benefits. Most notably, ray marching does not require a surface intersection function like ray tracing does, so it can be used to render geometry for which these functions do not exist. This property will be used to render 3D fractal-like geometry in this project. While many effects such as reflections, hard shadows and depth of field can be implemented almost identically to how they are in ray tracing, there are several optical effects that the ray marching algorithm can compute very cheaply.

Ambient occlusion is a technique used to determine how exposed each point in a scene is to ambient lighting [3]. This means that the more complex the surface of the geometry is (with creases, holes etc), the less ways ambient light can get into it those places and so the darker they should be. With ray marching, the surface complexity of geometry is usually proportional to the number of

steps taken by the algorithm [4]. This approximation works well in practice and comes with no extra computational cost at all.

Soft shadows can also be implemented very cheaply, by keeping track of the minimum angle from the distance estimator to the point of intersection, when marching from the point of intersection towards the light source [5]. This second round of marching must be done anyway if any type of lighting is to be taken into consideration, so minimum check required for soft shadows is practically free.

A glow can also be applied to geometry very cheaply, by keeping track of the minimum distance to the geometry for each ray. Then, if the ray never actually intersected the geometry, a glow can be applied using the minimum distance the ray was from the object, a strength value and colour specified [4].

2.3.2 Signed Distance Functions

A signed distance function (SDF) for a geometry, is a function which given any point in 3D space, will return the distance to the surface of that geometry. If the distance contains a positive sign if the point is outside of the object, and a negative sign if the point is inside of the object. If a distance function returns 0 for any point, then the point must be exactly on the surface of an object.

The scenes distance estimation (DE) function will make the required calls to SDFs for every geometry in the scene.

The sign returned by the SDF is useful as it allows the ray marcher to determine if a camera ray is inside of an object or not, and from there it can use that information to render the objects differently. We may want to render geometry either solid or hollow, or potentially add transparency.

The image below shows renders when the camera is located inside of a hollow sphere on the left, and a solid sphere on the right.

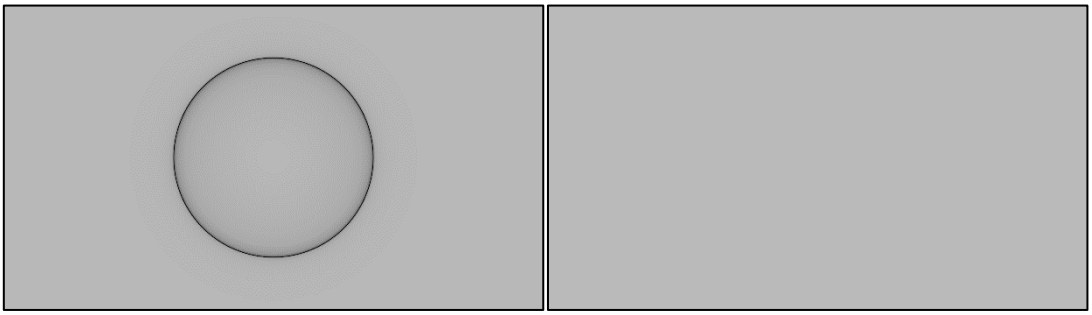

Figure 6: Hollow vs solid interior geometry experiment

2.3.3 Primitives

Signed distance functions are already known for most primitive 3D shapes [6], such as spheres, boxes and planes. Some of these functions are trivial, such as the SDF for a sphere with radius R, positioned on the origin.

$$sphereSDF(p) = |p| - R$$

Where:

$p\ is\ a\ vector\ in\ the\ form\ \{x, y, z\}$

$|p|\ is\ the\ magnitude\ of\ the\ vector\ p$

$R\ is\ the\ circle\ radius\ in\ world\ units$

2.3.4 Alterations & Combinations

Signed distance functions can be translated, rotated, and scaled. In addition, they can also be combined using the union, subtraction, and intersection operations.

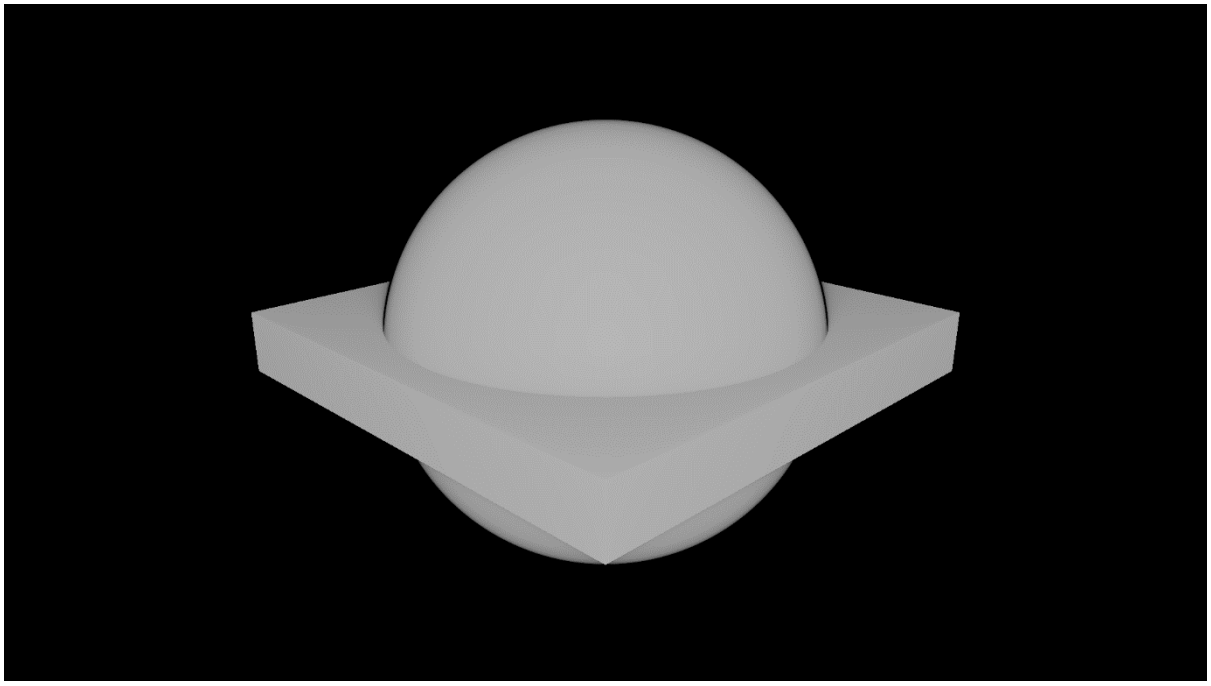

Figure 7: Ray marched union of sphere and box experiment

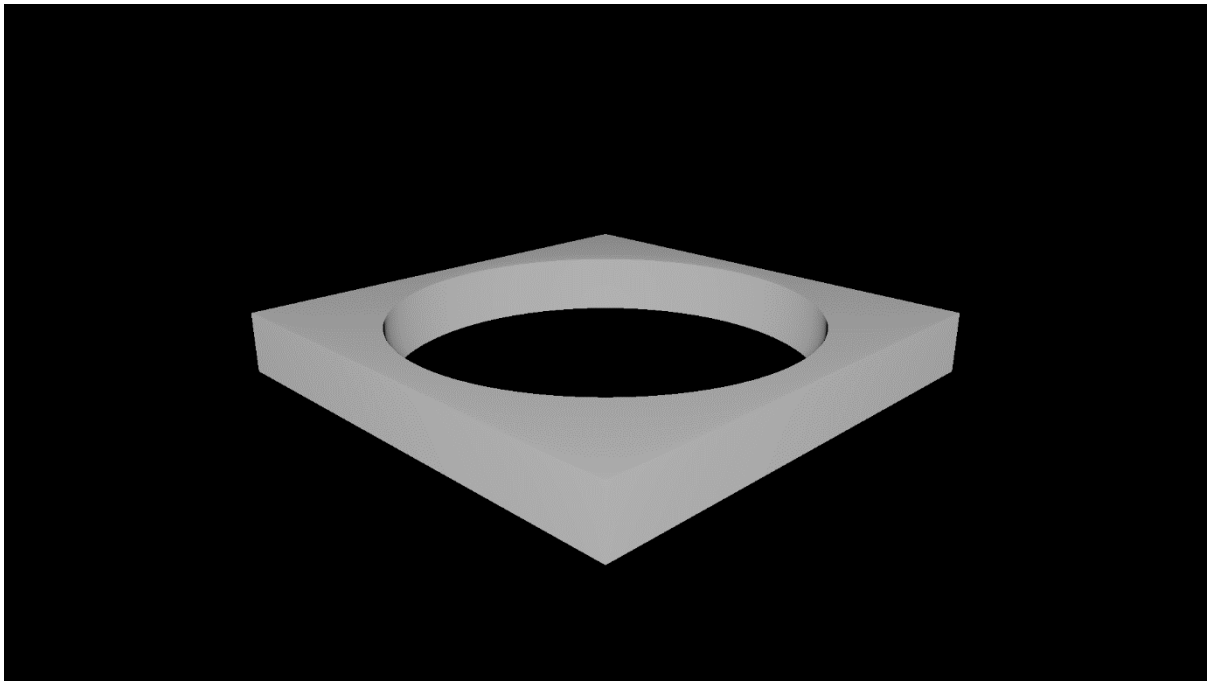

Figure 8: Ray marched intersection of sphere and box experiment

SDFs can also be combined using smooth union, subtraction, or intersection operations.

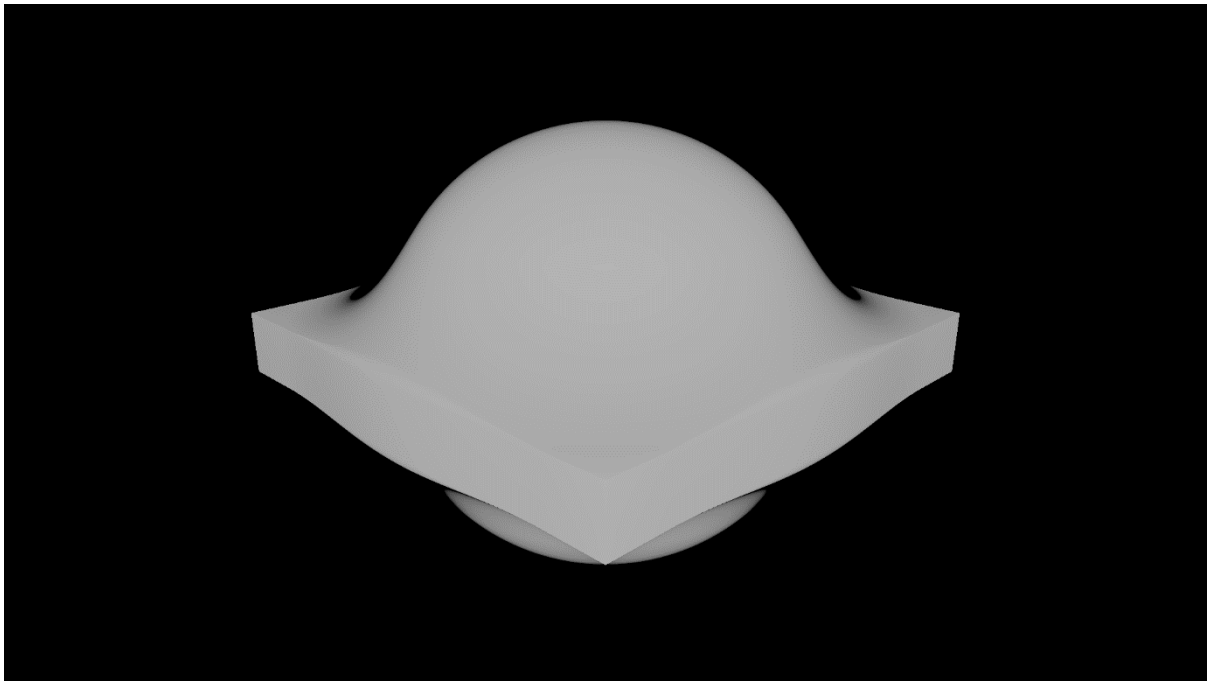

Figure 9: Ray marched smooth union of sphere and box experiment

There are several additional alterations that can be applied to primitives once we have their signed distance function. A primitive can be elongated along any axis, its edges can be rounded, it can be

extruded, and it can be “onioned” – a process of adding concentric layers to a shape. All these operations are relatively cheap. Signed distance functions can also be repeated, twisted, bent, and surfaces displaced using an equation e.g., a noise function or sin wave, though these alterations are more expensive.

2.3.5 Surface Normal

The surface normal of any point on the surface of an SDF can be determined by probing the SDF function on each axis, using an arbitrary epsilon value.

$$\begin{aligned} normal = normalise(\{\, & SDF(p + \{e, 0,0\}) - SDF(p - \{e, 0,0\}), \\ & SDF(p + \{0, e, 0\}) - SDF(p - \{0, e, 0\}), \\ & SDF(p + \{0,0, e\}) - SDF(p - \{0,0, e\}) \,\}) \end{aligned}$$

where

$p\ is\ a\ vector\ in\ the\ form\ \{x, y, z\}$

$normalise: \{x, y, z\} \rightarrow \{x, y, z\}$

$SDF: \{x, y, z\} \rightarrow \mathbb{R}$

$e\ is\ an\ arbitrary\ epsilon\ value$

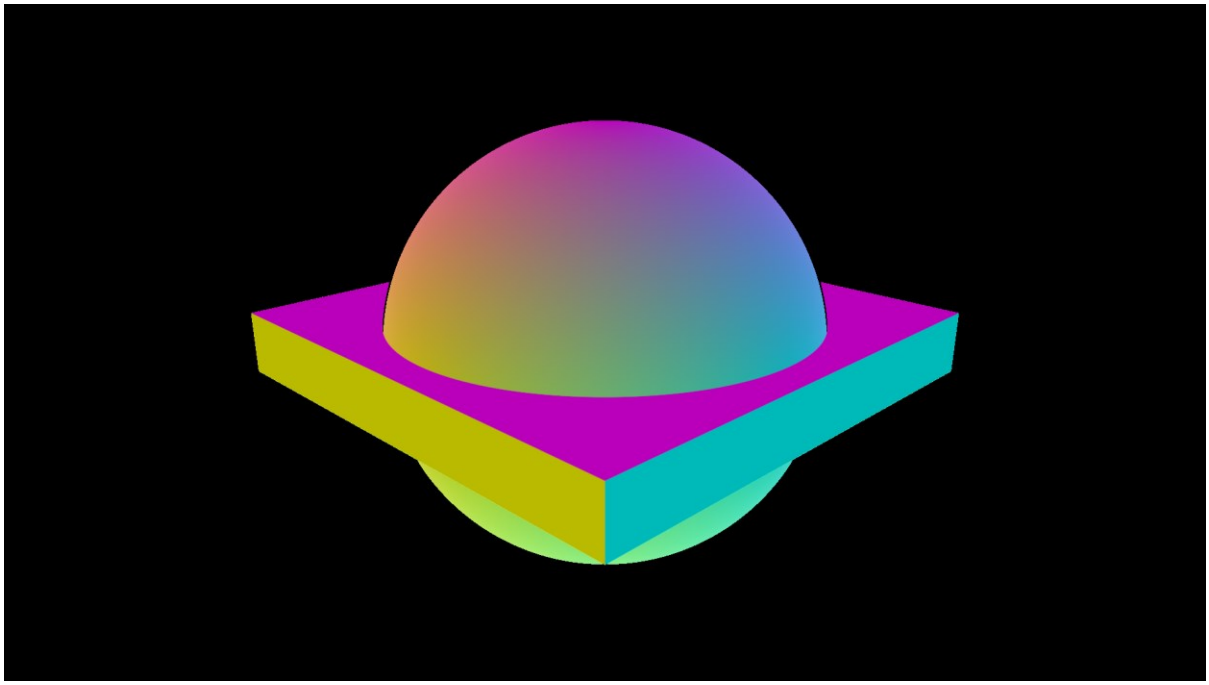

Figure 10: Surface normal of ray marched sphere and box scene experiment

The surface normal of a geometry is essential for most optical effects, such as lighting calculations and shadows.

2.4 Existing Projects

Do I need this section? Or maybe integrate it into the existing lit review content?

2.4.1 Fragmentarium

https://github.com/Syntopia/Fragmentarium

https://github.com/3Dickulus/FragM

2.4.2 Smallpt

https://www.kevinbeason.com/smallpt/

2.4.3 Ray Tracing in One Weekend

https://github.com/RayTracing/raytracing.github.io

3 Requirements Analysis

3.1 Use Cases

TODO

Relate to real world - model objects in nature? Cell structure, coastlines, etc

3.2 Requirements Specification

Table 3: Functional Requirement Specification

ID		Name	Description	Priority	Testing strategy
F-1		Real-time	The application must be capable of rendering scenes in real-time, running at 1920x1080 with a minimum fps of 60	MUST	Benchmark
F-2		Scene requirements	A scene must contain: • Geometry • Lights • Camera	MUST	Unit test
F-3		Example scenes	The application must contain multiple example scenes, some of which could include: • Julia set fractal • Mandel bulb fractal • Sierpinski tetrahedron fractal • Menger sponge fractal	MUST	Unit test

F-4		Mandatory optical effects	The application must support the following optical effects: • Ambient occlusion • Hard and soft shadows • Glow	MUST	
F-5		Optional optical effects	The application could support the following optical effects: • Reflections • Depth of field • Transparency	COULD	
F-6		Controllable camera	The user must be able to control the scene camera using a keyboard and mouse to move it around the scene	MUST	
F-7		Fixed camera paths	The application camera could support fixed camera paths	COULD	

Table 4: Non-functional Requirement Specification

ID	Name	Description	Priority	Testing strategy
NF-1	Executable	The application must run from a compiled executable	MUST	
NF-2	Display resolutions	The application must support the following common display resolutions: 1366x768, 1920x1080, 2560x1440 and 3840x2160	MUST	Unit test

3.3 Testing Strategy

TODO

Benchmark – to test for real-time project requirement

Unit tests – several requirements + code correctness

Other type of test for other requirements? User test?

3.4 Evaluation Strategy

TODO

Same as testing strategy? Do I need this section?

4 SOFTWARE DESIGN

4.1 TECHNOLOGIES

The application will be developed using the following technologies:

Table 5: Application Technologies

Technology	Description	Justification
OpenCL	Programming language which allows code to be run in parallel on the GPU	• GPU parallel computing gives a massive performance boost when executing the same code simultaneously many different values • GPU parallelism is far better suited for this task than CPU parallelism • OpenCL was chosen as it has good documentation and examples, contains C and C++ programming interfaces, and allows deployments to different platforms
C++	Common system programming language	• C++ is a low-level language with good performance • C++ was chosen over C to allow an object-oriented style of programming
SDL2	Cross platform C++ library for manipulating windows and reading user input	• Cross platform libraries provide an abstraction layer over platform specific libraries, which allows the program implementation to remain separate from the deployment platform • SDL2 was chosen as it provides both window display interaction and user input event polling, and has good documentation and examples

Development of the application and documentation will be assisted the following technologies:

Table 6: Development Technologies

Technology	Description	Justification
GitHub	Version control software	• TODO
Microsoft Word	Word processing software	•
Mendeley	Reference manager	•
Microsoft Teams	Video communication software	•

4.2 Class Structure

The application will be structured using several key classes:

Table 7: Class Responsibilities

Class name	Responsibilities
Application	Contains the run method, the main application loop which drives the application This class contains instances of Display, Renderer and Controller
Display	Setting pixels in the display window and controlling any GUI elements
Renderer	Calculating the colour for each pixel of the display window
Controller	Reading keyboard and mouse input from the user

CLASS DIAGRAM TODO

The Display and Controller classes are basic and only provide an interface to some SDL2_Event, SDL2_Renderer, SDL2_Window and SDL2_Texture instances. The Renderer class, however, is much more complex and requires discussion.

The Renderer class provides an interface to get the current pixels to be displayed on the screen, which are calculated using OpenCL kernels. Most of the ray marching code should be written in this kernel language to give the best performance to the application. Each scene will be defined within its own kernel file and will be loaded into the application at runtime. However, this makes it hard to reuse code between kernel files as the implementations of several methods, and the values of several constants will differ between scenes. The tables below show the main methods and constants used in the kernel, and their reusability status between scenes.

Table 8: Kernel Method Reusability Matrix

Method name	Purpose	Reusable across scenes?
render()	Calculates the colour for all pixels in the display and puts the values into a buffer	YES
calculatePixelColour(Ray)	Calculates the colour for the camera pixel with the specified ray direction	YES
DE(Vector3)	Calculates the distance to the nearest geometry surface in the current scene	NO
calculateNormal(Vector3)	Calculates the surface normal vector of the geometry for the position specified	YES

Table 9: Kernel Constant Reusability Matrix

Constant name	Purpose	Reusable across scenes?
MAXIMUM_MARCH_STEPS	Maximum number of iterations the ray marching algorithm can make	NO
MAXIMUM_MARCH_DISTANCE	Maximum distance the ray can be marched in the scene	NO
SURFACE_INTERSECTION_EPSILON	A very small value used to determine when the DE has converged to zero	YES
SURFACE_NORMAL_EPSILON	Arbitrary distance to probe the DE function when calculating the surface normal	YES

A solution to reducing code duplication between the kernel files is to use the new OpenCL C++ kernel language, which supports most C++17 features. Method and constant overloading will be used within the kernel file for each scene to override the implementation of the distance estimation (DE) function and MAXIMUM_MARCH_STEPS and MAXIMUM_MARCH_DISTANCE constants defined in a main kernel file. This main kernel file will contain the implementation of all other methods, such as the render and calculatePixelColour methods, and will contain an empty DE method for the other kernels to overload.

The OpenCL C++ kernel language is a new addition to OpenCL, released in March 2021, and as such there are few examples of C++ kernels, though the official documentation [7] is good.

5 Project Plan

5.1 Design Methodology

TODO

Do I need this section? Is this like sprints, agile etc?

5.2 Legal, Ethical & Social Issues

TODO

5.3 Risk Analysis

Table 10: Risk Analysis

ID	Description	Probability	Severity	Strategy	Rating
R-1	Loss of work	LOW	HIGH	All work will be backed up regularly using version control	**MEDIUM**
R-2	Change in requirements	LOW	MEDIUM	A thorough requirements specification has been prepared to reduce the probability of this happening	**MEDIUM**
R-3	Change of deadlines	LOW	HIGH		**LOW**
R-4	Delays due to learning new software	HIGH	MEDIUM	Experiments with the new software to familiarise have Free time has been allocated at the end of the timetable to allow for delays	**MEDIUM**
R-5	Delays due to illness	LOW	MEDIUM	Free time has been allocated at the end of the timetable to allow for delays	**LOW**
R-6	Delays due to bugs	MEDIUM	MEDIUM	Free time has been allocated at the end of the timetable to allow for delays	**MEDIUM**

5.4 TIMETABLE

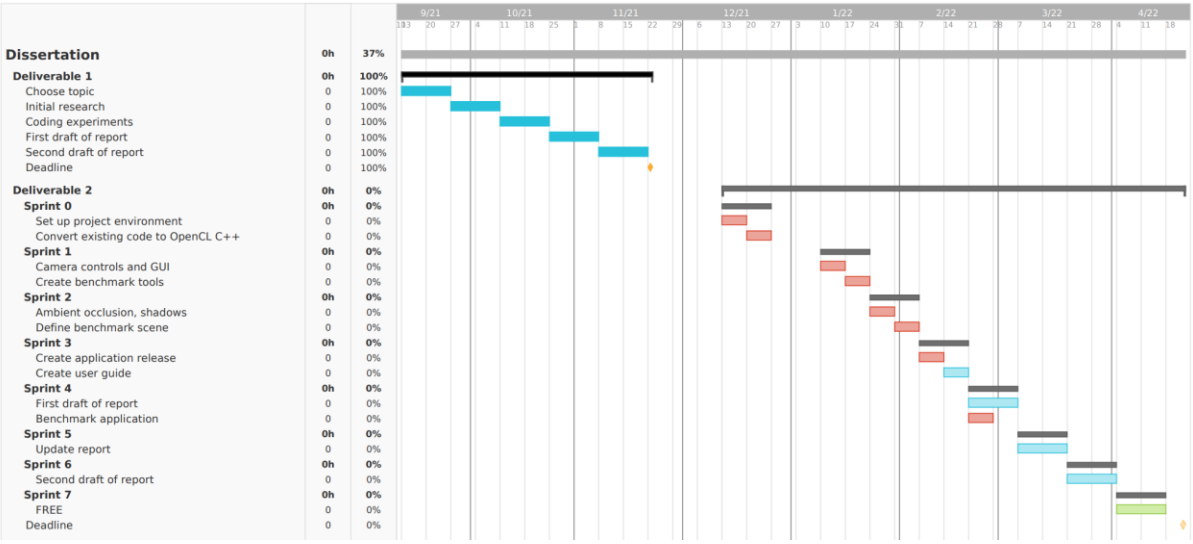

Figure 11: Project timeline Gantt chart

6 REFERENCES

[1] Cambridge English Dictionary, “FRACTAL | meaning in the Cambridge English Dictionary.” https://dictionary.cambridge.org/dictionary/english/fractal (accessed Oct. 18, 2021).

[2] B. B. Mandelbrot, “Fractals: the geometry of nature,” *CME*, vol. 12, pp. 1059–1064, 1977.

[3] Inigo Quilez, “3D Julia sets.” https://www.iquilezles.org/www/articles/juliasets3d/juliasets3d.htm (accessed Nov. 04, 2021).

[4] V. da Silva, T. Novello, H. Lopes, and L. Velho, “Real-time rendering of complex fractals,” Feb. 2021, [Online]. Available: http://arxiv.org/abs/2102.01747

[5] J. Peddie, “Ray Tracing: A Tool for All,” 2019.

[6] *What is Ambient Occlusion? Does it Matter in Games?* Accessed: Nov. 03, 2021. [Online]. Available: https://thewiredshopper.com/ambient-occlusion/?nonitro=1

[7] Mikael Hvidtfeldt Christensen, “Distance Estimated 3D Fractals,” 2011. http://blog.hvidtfeldts.net/index.php/2011/08/distance-estimated-3d-fractals-ii-lighting-and-coloring/ (accessed Nov. 04, 2021).

[8] Inigo Quilez, “Soft Shadows in Raymarched SDFs,” 2010. https://iquilezles.org/www/articles/rmshadows/rmshadows.htm (accessed Nov. 04, 2021).

[9] Inigo Quilez, “distance functions,” 2013. https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm (accessed Oct. 28, 2021).

[10] Khronos®, “The C++ for OpenCL 1.0 Programming Language Documentation,” 2021. https://www.khronos.org/opencl/assets/CXX_for_OpenCL.html#_the_c_for_opencl_programming_language (accessed Nov. 04, 2021).

7 APPENDICES