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

Deliverable 1: Final Year Dissertation

22/11/2021

by

Solomon Baarda

Meng Software Engineering

Heriot-Watt University

Supervisor: Dr Benjamin Kenwright

Second Reader: Ali Muzaffar

Abstract

A fractal is a recursively created never-ending pattern that is usually self-similar. Separate from Euclidean geometry, fractal geometry describes the more non-uniform shapes found in nature, like clouds, mountains, and coastlines. Fractal patterns exist everywhere in the universe, whether we can see them or not. From DNA molecules to the structure of galaxies, and everything in between. Fractals appear everywhere in nature and many technological breakthroughs have been made through studying their patterns.

With the increasing popularity in fractal geometry, and increasing computing power, fractal rendering software has become far more common in the last decade. However, only a small number of these programs are capable of rendering 3D fractals in real time, and those that are capable, are mostly written using graphics shaders which contain lots of code duplication between scenes. This makes it hard for a beginner to get into rendering 3D fractals as they must be competent in the chosen shader language and understand must the complex theory of rendering fractals. This project aims to produce a real-time 3D fractal geometry renderer, for which it is easy for a user to create new scenes and add geometry to it.

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 1.1.i Common definitions

Word	Definition
Complex number	
Euclidian geometry	
Fractal	
Frame	
Geometry	
Method/constant overloading	
Quaternion	
Ray	
Render	

Common Abbreviations

Table 1.1.ii Common abbreviations

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

1 Introduction

All project materials can be found online in the GitHub repository
https://github.com/SolomonBaarda/fractal-geometry-renderer

1.1 Project Description

A fractal is a recursively created never-ending pattern that is usually self-similar [1]. Separate from Euclidean geometry, fractal geometry describes the more non-uniform shapes found in nature, like clouds, mountains, and coastlines. Beniot Mandelbrot, inventor of the concept of fractal geometry, famously wrote "Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line" [2]. Fractal patterns exist everywhere in our lives [3], whether we can see them or not. From DNA molecules to the structure of galaxies, and everything in between.

Fractals have many applications in the real world [1]–[4]. In medicine, fractals have been used to help distinguish between cancerous cells which grow abnormally, and healthy human blood vessels which typically grow in fractal patterns. In fluid mechanics, fractals have been used to help model both complex turbulence flows and the structure of porous materials. In computer science, fractal compression is an efficient method for compressing images and other files and uses the fractal characteristic that parts of a file will resemble other parts of the same file. Fractal patterns are also used in the design of some cell phone and Wi-Fi antennas, as the fractal design allows them to be made more powerful and compact than other designs. Even losses and gains in the stock market have been described in terms of fractal mathematics. All these applications of fractals demonstrate that this geometry is a fundamental part of our universe, both in physical materials and in theoretical concepts.

With the increasing popularity in fractal geometry, and increasing computing power, fractal rendering software has become far more common in the last decade [5]. However, only a small number of these programs are capable of rendering 3D fractals in real time, and those that are capable, are mostly written using graphics shaders which contain lots of code duplication between scenes. This makes it hard for a beginner to get into rendering 3D fractals as they must be competent in the chosen shader language and understand must the complex theory of rendering fractals. This project aims to produce a real-time 3D fractal geometry renderer, for which it is easy for a user to create new scenes and add geometry to it.

The term fractal has been used throughout this report to describe a pattern or geometry that displays the recursive self-similarity characteristic of fractals. The term fractal-like has been used to describe something that appears to display the fractal characteristics but may not truly contain infinite detail.

1.2 Aims & Objectives

There are two aims that this project hopes to achieve. These are:

1. To develop a prototype real-time rendering engine, capable of displaying 3D “fractal-like” geometry
2. When using the rendering engine, it must be easy for a user to create a new scene and to add geometry to it

To achieve these aims, some objectives have been defined which will be the main tasks completed during this project. These are:

1. Implement the basic ray marching algorithm for execution on a single CPU thread
2. Design a software structure that will minimise code duplication between scenes
3. Update the single threaded ray marching code so it can be run in parallel on the GPU
4. Update the code to use a game loop so that it can be executed in real-time
5. Add camera movement controls and a GUI
6. Add optical effects such as ambient occlusion, soft shadows, and lighting
7. Define a benchmark scene to test the performance of the renderer
8. Create documentation to assist users when creating scenes

It is necessary to complete many of the early objectives in the order they are specified, as they build upon previous objectives. From objective 5 onwards, the order of completion is less important.

These objectives have been created bearing the SMART properties [6] in mind. SMART stands for: Specific, Measurable, Achievable, Realistic and Time constrained.

Objectives 1, 3, 4, 5, 6 and 7 have been designed to help achieve aim 1, as these objectives contribute directly to the functionality that this application hopes to achieve. Objectives 2 and 8 have been created to help achieve aim 2, which is more subjective. Aim 2 is less specific and measurable than aim 1, and care must be taken that it is not overlooked. Objectives 5 and 6 have been deliberately left vague to allow them to be scaled back or extended depending on the progress made.

The objectives form the main tasks to be completed during the duration of this project and the requirements specification in section 3.2 and the Gantt chart in section 6.5 have been structured around these.

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. Objectives 5 and 6 have large amounts of flexibility and can be extended or cut back depending on time constraints. The requirements specification in section 3.2 describes the specific functionality to be implemented for each objective along with any optional stretch goals.

Objectives 1 and 2 have already been completed. Some initial experimentation rendering still images of the Mandel bulb fractal and other geometry has been very successful. Some of these renders can be viewed in the appendix, section 9. In addition, some experimentation with OpenCL, a library that allows GPU parallel code to be written and executed, has been completed. These experiments were done to gain familiarity with this style of programming in the hope to reduce the learning curve of this new software.

2 Literature Review

This literature review contains the relevant information for understanding the complexity of this project and details of how the problem will be tackled. While this review contains explanations of all key concepts related to, basic knowledge of the following topics is assumed: recursion, vector maths, complex numbers.

This review is split into several sections, first the theory of fractals and their 3D counterparts will be discussed. Then the key concepts of ray marching, the chosen method of rendering fractals, will be discussed. And finally, a brief introduction GPU parallel programming will be given.

2.1 3D Fractals

2.1.1 Sierpiński Tetrahedron

The Sierpiński tetrahedron, also known as the Sierpiński pyramid, is a 3D representation of the famous Sierpiński tringle, named after the Polish mathematician Wacław Sierpiński [7]. The Sierpiński triangle is one of the most simple and elegant fractals and has been a popular decorative pattern for centuries. This pattern is created by recursively each splitting each solid equilateral triangle into four smaller equilateral triangles and removing the middle one. Theoretically, these steps are repeated forever, but in practice some maximum depth must be specified as computers only have finite memory.

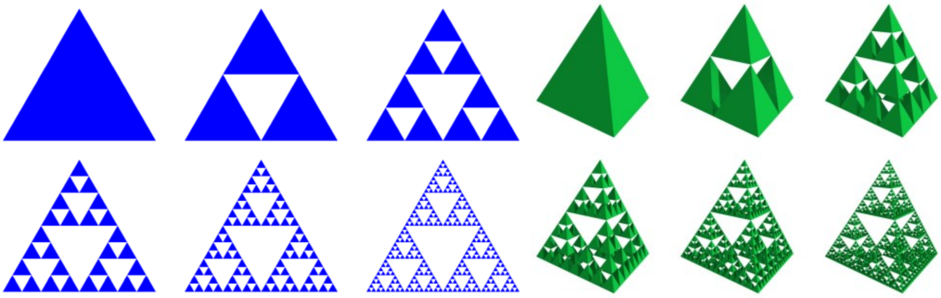

Figure 2.1.i Sierpiński triangle (left) [8] *and tetrahedron (right)* [8] *both of recursive depth 5*

As the recursive depth of the fractal increases, so does the number of objects (either triangles or tetrahedrons) in the scene. This is the limiting factor when rendering the Sierpiński tetrahedron as computer memory is finite and can only store limited number of objects. The total number of objects in the Sierpiński triangle increases by a factor of 3 each iteration and the tetrahedron by a factor of 4.

2.1.2 Menger Sponge

The Menger sponge, also known as the Menger cube or Sierpiński cube, is another 3D representation of a 2D fractal patten created by Polish mathematician Wacław Sierpiński [7]. The Menger sponge is a 3D representation of the 2D Sierpiński carpet fractal, which follows very similar recursive rules to the Sierpiński triangle but uses squares instead of triangles.

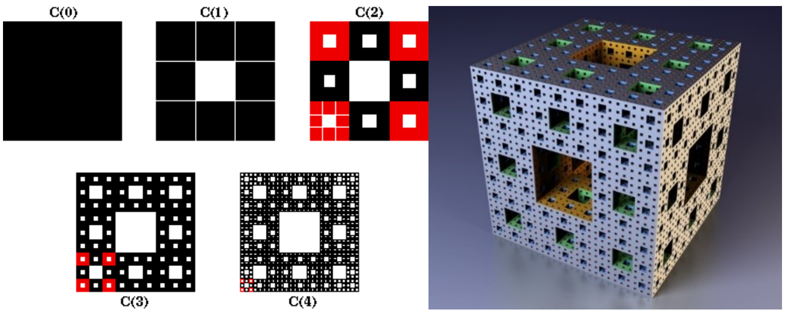

Figure 2.1.ii Sierpiński carpet (left) [9] *and Manger sponge (right)* [10] *both of recursive depth 4*

The number of objects required to create these fractals at various recursive depths increases similarly to the Sierpiński triangle and tetrahedron, but at a different rate. The Sierpiński carpet increases by a factor of 8 each iteration and the Menger sponge by a factor of 20. A Menger cube at recursive depth n is made up of 20^n smaller cubes, each with a side length of $(\frac{1}{3})^n$ [11].

2.1.3 Mandel Bulb

The Mandel bulb is a 3D fractal, created by Daniel White and Paul Nylander and is now the commonly used 3D representation of the 2D Mandelbrot fractal. The Mandelbrot fractal is defined as the set of complex numbers c for which the equation $f(z) = z^2 + c$ does not diverge to infinity, when iterated from $z = 0$ [12]. For many years, it was thought that a true 3D representation of the Mandelbrot fractal did not exist, since there is no 3D representation of the 2D space of complex numbers, on which the Mandelbrot fractal is built upon [13].

White and Nylander considered some of the geometrical properties of the complex numbers. The multiplication of two complex numbers is a kind of rotation, and the addition is a kind of transformation. White and Nylander experimented with ways of preserving these characteristics when converting from 2D to 3D, and their solution was to change the squaring part of the formula to instead use a higher power, a practice sometimes used with the 2D Mandelbrot fractal to produce snowflake type results [14]. This change gives the equation $f(z) = z^n + c$.

White and Nylander's formula for the nth power of a point [15] is given as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}^n = r^n \begin{bmatrix} \sin(n\theta)\cos(n\varphi) \\ y\sin(n\theta)\sin(n\varphi) \\ \mathrm{z}\cos(n\theta) \end{bmatrix}$$

$$where\ r = \sqrt{x^2 + y^2 + z^2}, \theta = \ atan2(\sqrt{x^2 + y^2}, z), \varphi = \ atan2(y, x)$$

2.1.4 Julia Set

TODO

The Julia set is a set of fractals named after the French mathematician Gaston Julia [16]. Fractals in the Julia set come from the convergence of the system given by the quadratic function $f(z) = z^2 + c$, where different values of c produce different fractals.

Similar to the Mandelbrot fractal, 2D Julia sets use the complex number plane as the domain of the quadratic function f. Unlike the Mandel bulb, the Julia set can be extended into 4D using quaternions as the domain of f [17], and then 3D slices can be taken of the quaternions to view the fractal in 3D space.

Some recent work, such as Quilez [18], uses pixel shaders to render a 3D Julia set.

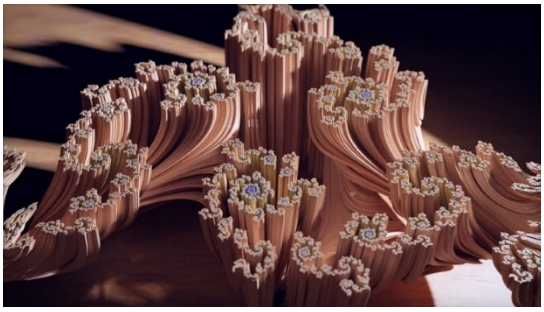

Figure 2.1.iii Ray marched Julia set, cut in half to expose the fractal interior [18]

The recent work da Silva et al in 2021 [13] took this a step further, using Nvidia DirectX Raytracing (DXR) shaders to render the visualisations of the 3D Julia set and Mandel bulb fractal.

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

2.1.5 3D Fractals Summary

TODO

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 [19].

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 Euclidian geometry, such as primitive shapes or meshes, as points of intersection can be calculated relatively easily on these shapes. However, these functions do not exist for fractal geometry [20]. Instead, a slightly different approach must be used.

2.3 Ray Marching

Ray marching is a variation of ray tracing, which only differs in the method of detecting intersections between the ray and objects. Instead of a using a ray-surface intersection function which returns the position of intersection, ray marching uses a distance estimation (DE) function, which simply returns the distance from any given position in the scene, to the closest object. Instead of shooting the ray in one go, ray marching uses an iterative approach, where the current position 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, the DE function is called and marched forward by that distance, and the process is repeated until the ray lands on the surface of an object. 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.

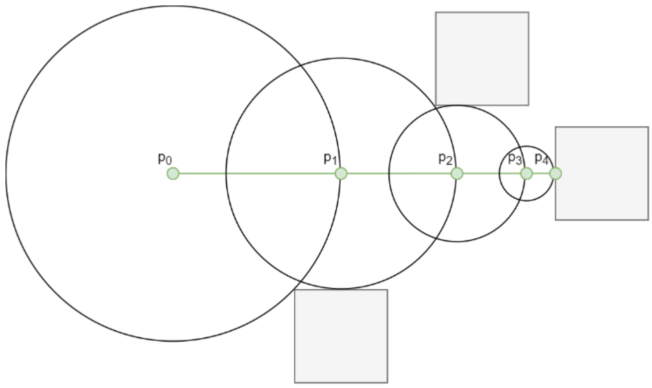

Figure 2.3.i Ray marching diagram

Technically, 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. However, if the value is too small, then the ray marching algorithm becomes 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. As mentioned before, 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 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 [21]. 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 [22]. This property can be used to implement ambient occlusion 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 [23]. 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 [22].

2.3.2 Signed Distance Functions

A signed distance function (SDF) for a geometry, is a function which given any position in 3D space, will return the distance to the surface of that geometry. If the distance contains a positive sign if the position is outside of the object, and a negative sign if the position is inside of the object. If a distance function returns 0 for any position, then the position must be exactly on the surface of an object. Every single geometry in a scene must have its own SDF. The scenes distance estimation (DE) function will loop through all of the SDF values for the geometry in the scene and will return the minimum.

The sign returned by the SDF is useful as it allows the ray marcher to determine if a camera ray is inside of a geometry 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.

Signed distance functions are already known for most primitive 3D shapes [24], 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.

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

$$where\ p = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, |p|\ is\ the\ magnitude\ of\ the\ vector\ p,\ \ R\ is\ the\ circle\ radius\ in\ world\ units$$

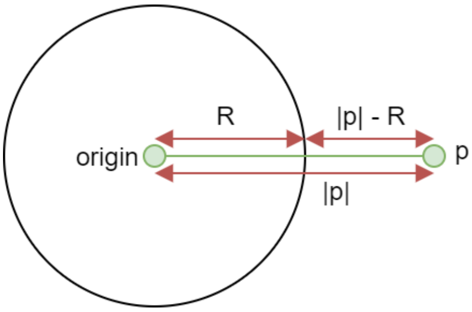

Figure 2.3.ii Sphere SDF diagram

2.3.3 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. The images below were rendered using an early version of the application.

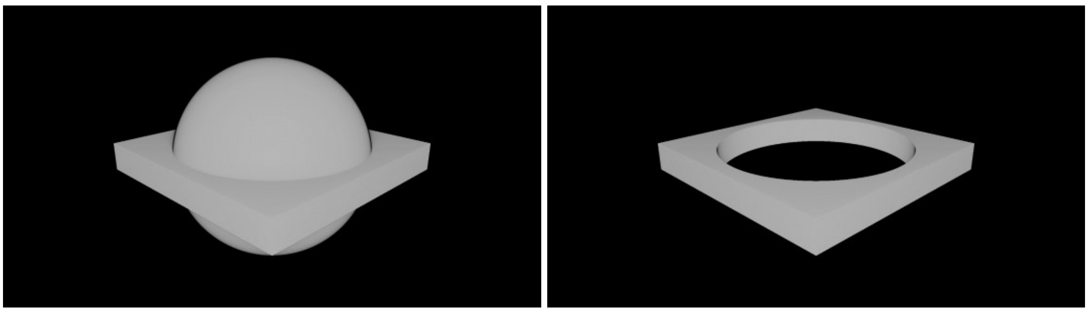

Figure 2.3.iii Ray marched sphere and box scene experiment union (left) and subtraction (right)

The union, intersection and subtraction of two SDFs can be taken as

$$union(a, b) = \min(a, b),$$

$$subtraction(a, b) = \max(-a, b),$$

$$intersection(a, b) = \max(a, b),$$

$where\ a, b\ \in \mathbb{R}\ are\ the\ values\ returned\ from\ object\ a\ and\ b's\ SDF$

There also exist variations of formulas above which can be used to apply a smoothing value to the operation. Formulas have been included in the appendix section 9.1.

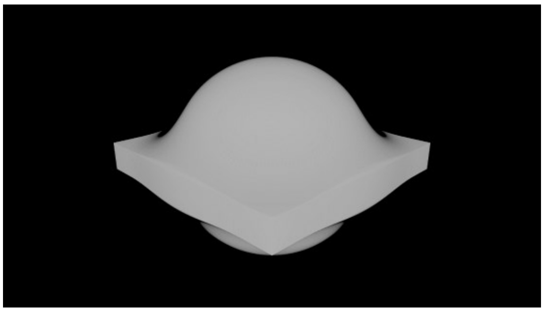

Figure 2.3.iv Ray marched sphere and box scene experiment smooth union

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. All of these techniques mentioned will be essential when creating more complex geometry.

2.3.4 Surface Normal

The surface normal of a position on the surface of a geometry is a normalised vector that is perpendicular to the that surface [25]. This information is is essential for most lighting calculations.

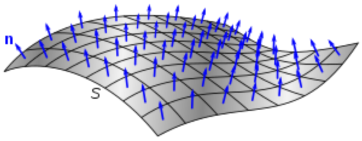

Figure 2.3.v Surface normal vectors for a curved surface [25]

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

$$normal = normalise(\begin{bmatrix} DE(p + \begin{bmatrix} e \\ 0 \\ 0 \end{bmatrix}) - DE(p - \begin{bmatrix} e \\ 0 \\ 0 \end{bmatrix}) \\ DE(p + \begin{bmatrix} 0 \\ e \\ 0 \end{bmatrix}) - DE(p - \begin{bmatrix} 0 \\ e \\ 0 \end{bmatrix}) \\ DE(p + \begin{bmatrix} 0 \\ 0 \\ e \end{bmatrix}) - DE(p - \begin{bmatrix} 0 \\ 0 \\ e \end{bmatrix}) \} \end{bmatrix})$$

$$where\ p\ is\ a\ vector\ in\ the\ form\ \begin{bmatrix} x \\ y \\ z \end{bmatrix}, e\ is\ an\ arbitrary\ epsilon\ value,$$

$$DE\ is\ the\ scene\ distance\ estimation\ function$$

The value of a normal can be converted to a colour by mapping the x, y, and z values to r, g, and b.

This is useful

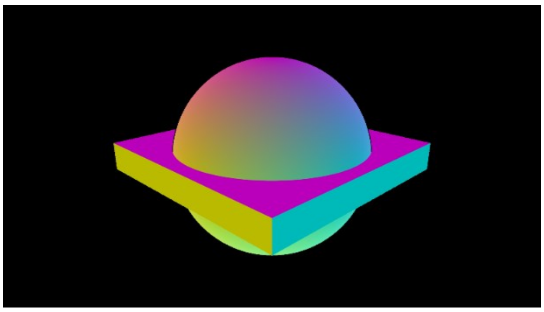

Figure 2.3.vi Surface normal of ray marched sphere and box scene experiment

2.3.5 Ray Marching Summary

TODO

2.4 GPU Parallel Programming

TODO

Explain background, how stuff is executed, all at the same time

2.4.1 OpenCL Kernel Language

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 3.2.i Functional requirement specification

ID	Name	Description	Objective	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	7	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	6	MUST	Unit test
F-5	Optional optical effects	The application could support the following optical effects: • Reflections • Depth of field	6	COULD	Unit test

		• Transparency			
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	5	MUST	
F-7	Fixed camera paths	The application camera could support fixed camera paths	5	COULD	

Table 3.2.ii Non-functional requirement specification

ID	Name	Description	Objective	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

There are several types of testing that will be used to ensure the correctness of the application. Unit tests will be used throughout the development of the application, to ensure

TODO

Benchmark – to test for real-time project requirement – see evaluation strategy section for more

Unit tests – several requirements + code correctness

Other type of test for other requirements? User test?

4 Software Design

4.1 Technologies

The application will be developed using the following technologies:

Table 4.1.i 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 piece of code simultaneously for many different input values • GPU parallelism is far better suited for this task than CPU parallelism as the same piece of code must be executed for every pixel on the screen • 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 decoupled 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 4.1.ii Development technologies

Technology	Description	Justification
Visual Studio 2019	Code editor	• Contains powerful development tools such as refactoring options, code snippets, documentation preview etc
GitHub	Version control software	• Version control is essential for any large coding project • GitHub is being used to store all project materials, including documents, papers, and the coding project
Microsoft Word	Word processing software	• Powerful and easy to use word processing software • Documents will be edited locally and backed up to GitHub
Mendeley	Reference manager	• This reference manager has a web interface which is very convenient when researching • It also has a Microsoft Word add in which manages all referenced sources automatically
Microsoft Teams	Video communication software	• For communication with the project supervisor and second readers

4.2 Class Structure

The application will be structured using several key classes:

Table 4.2.i 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

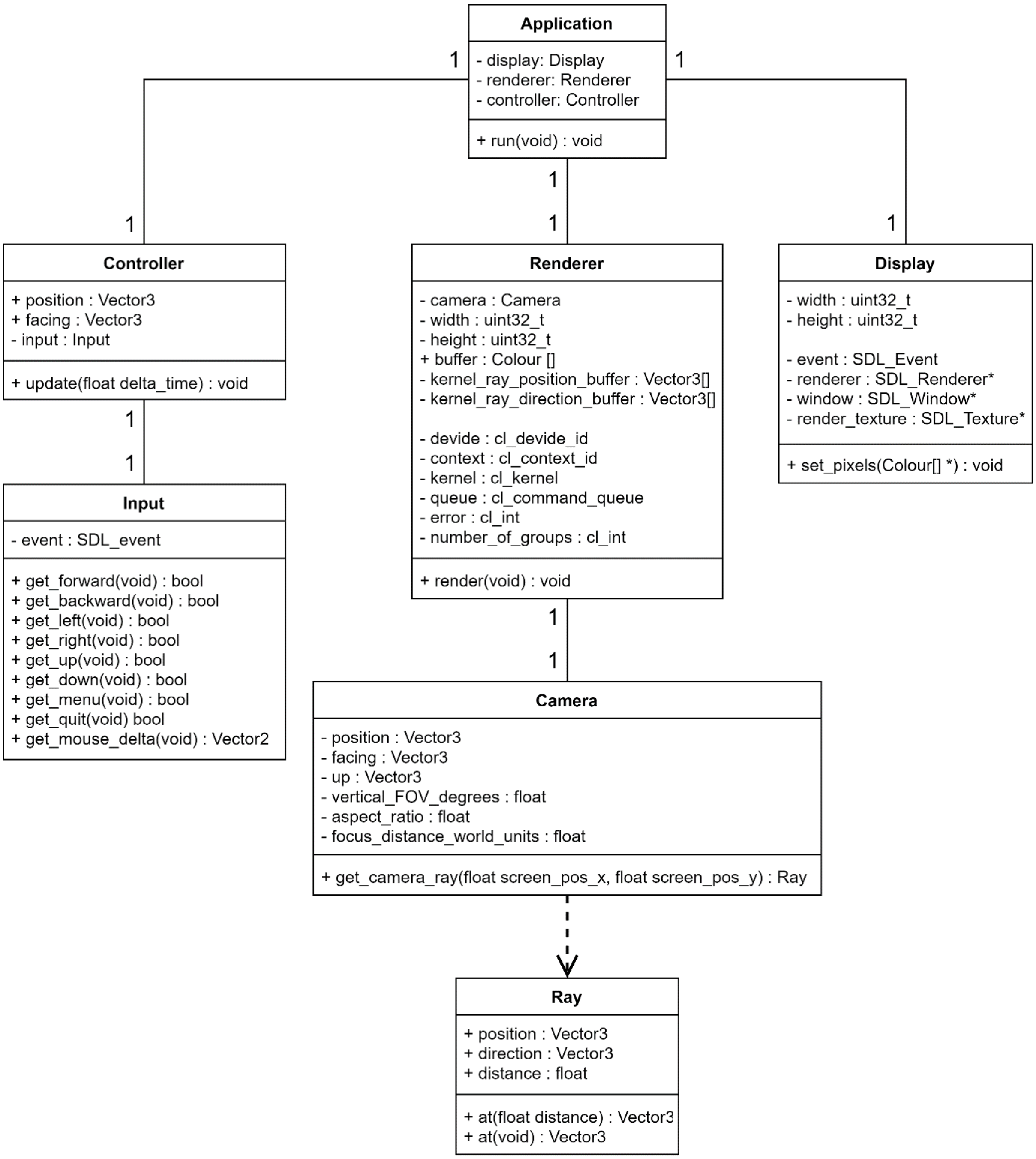

Figure 4.2.i Application class diagram

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 an OpenCL kernel. The OpenCL kernel is a piece of code written in one of the OpenCL kernel languages, and is the code that is executed in parallel on the GPU. 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 4.2.ii 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 4.2.iii 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 [26] is good.

5 EVALUATION STRATEGY

A goal-based evaluation strategy will be used. Both unit tests (specified in section 3.3) and a benchmark scene will be used to determine how many of the requirements specified in section 3.2 have been fully implemented.

For an objective to be considered achieved, all requirements related to that objective must have been implemented. For an aim to be considered achieved, all objectives related to that aim must have been achieved.

5.1 AIM 1

Aim 1, "To develop a prototype real-time rendering engine, capable of displaying 3D "fractal-like" geometry" will be evaluated in two parts, the real-time aspect through a benchmark scene, and displaying fractals through unit tests. The tables below list the results that must be returned from the benchmark scene, and the values that they are used to calculate.

Table 5.1.i Results recorded from benchmark

Description	Units
Window display resolution	-
Duration of the benchmark scene	Seconds
Total number of frames rendered	-
Minimum frame time	Milliseconds
Maximum frame time	Milliseconds

Table 5.1.ii Results calculated from benchmark

Description	Units	Calculation
Average frame time	Milliseconds	$\frac{duration\ of\ the\ benchmark\ scene}{total\ number\ of\ frames\ rendered} \times 1000$
Average frames per second	Frames per second	$\frac{1000}{average\ frame\ time}$
Minimum frames per second	Frames per second	$\frac{1000}{minimum\ frame\ time}$
Maximum frames per second	Frames per second	$\frac{1000}{maximum\ frame\ time}$

In addition, the PC specs of the computer running the benchmark must be recorded. This information can be exported to a file by running the "System information" program on windows and selecting the File>Export option. The exported file contains a list of the computer specifications and relevant driver information. The important information that will be extracted from the file is included in the table below.

Table 5.1.iii Relevant PC specification values

Description	Units	Line number in exported file
Operating system name	-	6
Processor name	-	15
Processor base clock speed	Megahertz	15
Processor number of cores	-	15
Processor number of logical cores	-	15
Total physical memory	Gigabytes	34
Total virtual memory	Gigabytes	36
Graphics card name	-	Varies, located under [Display] heading
Graphics card dedicated memory	Gigabytes	N/A
Graphics card shared memory	Gigabytes	N/A

Unfortunately, the dedicated and shared memory of the graphics card is not included in the file and must be recorded manually from viewing the task manager.

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 that it takes the same amount of time to run on all machines.

The benchmark should be run multiple times and averages taken of the results. This is to reduce impact of any background tasks that may be running on the machine.

5.2 Aim 2

Aim 2, "When using the rendering engine, it must be easy for a user to create a new scene and to add geometry to it" is a much more subjective aim and will be harder to evaluate.

TODO

5.3 Unique Characteristics

- Users will have different experiences

Details of the evaluation and analysis to be conducted.

stats, analysis, what and I going to do, how to compare results

emphasise visualising

pros and cons

6 PROJECT PLAN

6.1 PROJECT MANAGEMENT

The project will be developed using an Agile [27] approach, which uses small sprints of work to complete specific and defined tasks. This approach allows teams to respond to change quickly as requirements and plans are updated regularly.

6.2 DESIGN METHODOLOGY

The application will be developed following the main object-oriented principles [28]. These include encapsulation, abstraction, inheritance, and polymorphism. An object-oriented style of programming has been chosen to promote code reusability within the application in the hopes that this will allow the kernel code to be as simple and easy to use as possible.

6.3 LEGAL, ETHICAL & SOCIAL ISSUES

TODO

open source standards

BCS

license

6.4 RISK ANALYSIS

Table 6.4.i Risk analysis matrix

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	Time was assigned during the planning stage to experiment with new software	**MEDIUM**

				Free time has been allocated at the end of the timetable to allow for delays	
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**

6.5 Project Timeline

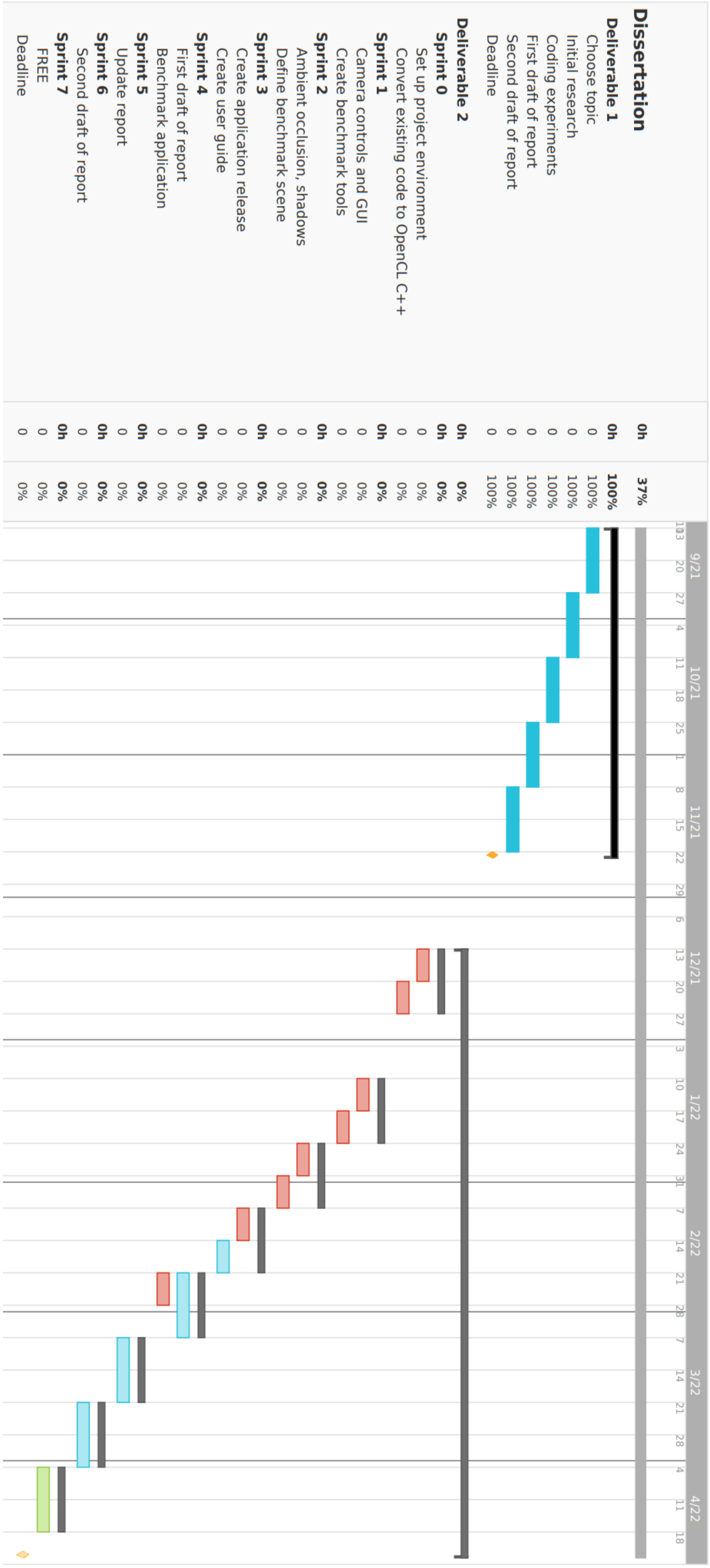

Figure 6.5.i Project timeline Gantt chart

The project timeline for the second deliverable has been split into 8 sprints, each 2 weeks long. Project objectives one and two have already been completed, leaving objectives three to eight.

Sprint 0 will be completed during the Christmas holiday, to set up the project environment and refactor the existing experimentation code into something usable. Once the semester starts, sprint

7 Conclusion

TODO

8 REFERENCES

[1] "Top 5 applications of fractals | Mathematics | University of Waterloo." https://uwaterloo.ca/math/news/top-5-applications-fractals (accessed Nov. 10, 2021).

[2] "How Mandelbrot's fractals changed the world - BBC News." https://www.bbc.co.uk/news/magazine-11564766 (accessed Nov. 10, 2021).

[3] "Fractals in nature and applications." https://kluge.in-chemnitz.de/documents/fractal/node2.html (accessed Nov. 10, 2021).

[4] "Fractal Foundation Online Course - Chapter 12 - FRACTAL APPLICATION." http://fractalfoundation.org/OFC/OFC-12-2.html (accessed Nov. 10, 2021).

[5] "Fractal-generating software - Wikipedia." https://en.wikipedia.org/wiki/Fractal-generating_software (accessed Nov. 14, 2021).

[6] "Writing Dissertations: Aims and objectives." https://learn.solent.ac.uk/mod/book/view.php?id=116233&chapterid=15294 (accessed Nov. 11, 2021).

[7] "Wacław Sierpiński - Wikipedia." https://en.wikipedia.org/wiki/Wac%C5%82aw_Sierpi%C5%84ski (accessed Nov. 11, 2021).

[8] H. Segerman, "Fractals and how to make a Sierpinski Tetrahedron", Accessed: Nov. 11, 2021. [Online]. Available: http://www.segerman.org

[9] "Sierpinski Carpet." https://larryriddle.agnesscott.org/ifs/carpet/carpet.htm (accessed Nov. 11, 2021).

[10] "Menger Sponge | Visual Insight." https://blogs.ams.org/visualinsight/2014/03/01/menger-sponge/ (accessed Nov. 11, 2021).

[11] "Menger sponge - Wikipedia." https://en.wikipedia.org/wiki/Menger_sponge (accessed Nov. 11, 2021).

[12] "Mandelbrot set - Wikipedia." https://en.wikipedia.org/wiki/Mandelbrot_set (accessed Nov. 13, 2021).

[13] 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

[14] "Mandelbulb: The Unravelling of the Real 3D Mandelbrot Fractal." https://www.skytopia.com/project/fractal/mandelbulb.html (accessed Nov. 12, 2021).

[15] R. Englund, S. Seipel, and A. Hast, "Rendering Methods for 3D Fractals," 2010.

[16] "Gaston Julia - Wikipedia." https://en.wikipedia.org/wiki/Gaston_Julia (accessed Nov. 12, 2021).

[17] A. Norton, "Julia sets in the quaternions," *Computers & Graphics*, vol. 13, no. 2, pp. 267–278, Jan. 1989, doi: 10.1016/0097-8493(89)90071-X.

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

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

[20] J. C. Hart, D. J. Sandin, and L. H. Kauffman, “Ray Tracing Deterministic 3-D Fractals,” 1989.

[21] “Ambient occlusion - Wikipedia.” https://en.wikipedia.org/wiki/Ambient_occlusion (accessed Nov. 03, 2021).

[22] 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).

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

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

[25] “Normal (geometry) - Wikipedia.” https://en.wikipedia.org/wiki/Normal_(geometry) (accessed Nov. 13, 2021).

[26] 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).

[27] “What is Agile? | Atlassian.” https://www.atlassian.com/agile (accessed Nov. 13, 2021).

[28] “What are four basic principles of Object Oriented Programming? | by Munish Chandel | Medium.” https://medium.com/@cancerian0684/what-are-four-basic-principles-of-object-oriented-programming-645af8b43727 (accessed Nov. 13, 2021).

[29] Inigo Quilez, “Mandelbulb,” 2009. https://www.iquilezles.org/www/articles/mandelbulb/mandelbulb.htm (accessed Nov. 04, 2021).

9 APPENDICES

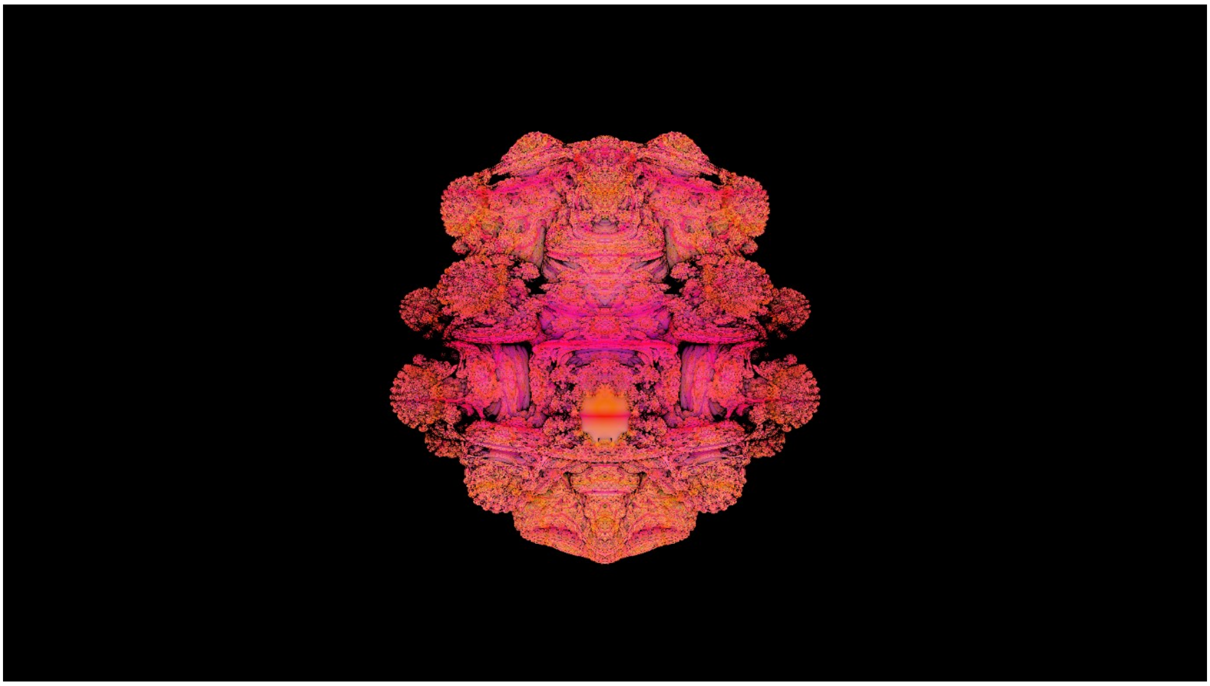

Figure 6.5.i Render of the Mandel bulb fractal, created using fractal equation from [29]

9.1 SMOOTH SDF COMBINATIONS

$$smoothUnion(a,b,s) = \min(a,b) - h^2 \times \frac{0.25}{k}$$

$where\ a,b\ \in \mathbb{R}, s\ \in \mathbb{R}\ is\ the\ smoothing\ value, h = \max(s - abs(a-b), 0)$

$$smoothSubtraction(a,b,s) = \max(-a,b) + h^2 \times \frac{0.25}{k}$$

$where\ a,b\ \in \mathbb{R}, s\ \in \mathbb{R}\ is\ the\ smoothing\ value, h = \max(s - abs(-a-b), 0)$

$$smoothIntersection(a,b,s) = \max(a,b) + h^2 \times \frac{0.25}{k}$$

$where\ a,b\ \in \mathbb{R}, s\ \in \mathbb{R}\ is\ the\ smoothing\ value, h = \max(s - abs(a-b), 0)$