Real-time Rendering of 3D Fractal Geometry

Final Year Dissertation

21/04/2022

by

Solomon Baarda

Meng Software Engineering

Heriot-Watt University

Supervisor: Dr Benjamin Kenwright

Second Reader: Ali Muzaffar

Declaration

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: 09/03/2022

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 patterns appear everywhere in nature and many technological breakthroughs have been made through studying them.

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 must understand 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.

Table of Contents

Table of Figures

Table of Tables

Common Definitions

Table 1.1.1 Common definitions

Word	Definition
Complex number	Number system containing an imaginary unit, defined as $i = \sqrt{-1}$
Convex polyhedron	3D equivalent of a regular 2D polygon
Euclidian geometry	A geometry containing basic objects like points and lines
Fractal	Recursively created never-ending pattern that is usually self-similar
Frame	One of many still images that make up a moving image
Geometry	Branch of mathematics concerned with the properties of space, distance, shape, size, and positions
Method overriding	Object-oriented programming technique which allows a subclass to change the implementation of a method that a parent class is providing
Object-oriented programming	Programming style that organises its software design around reusable objects
Polygon	2D shape with three or more sides
Polyhedrons	3D shape with six or more faces
Quaternion	Extension of the complex numbers, often used for storing position, translation, and scale values in 3D space
Ray	Line in 3D space with a beginning and an end
Regular polygon	Shape where all edges are the same length, and all angles between vertices are also equal
Render	Process of creating a still image using a computer
Vector	Mathematical object representing a position in space relative to another

Common Abbreviations

Table 1.1.2 Common abbreviations

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

1 Introduction

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 objects 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 the complex theory of rendering

fractals. This purpose of this project, therefore, is to design and implement a real-time 3D fractal geometry renderer, for which it is easy for a user to create and add geometry to scenes, in the hopes that it will fill the gap in the market for entry-level 3D fractal viewing software.

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

The aim of this project was to develop an application which can update and render 3D fractal geometry in real-time. The render uses common optical effects including the Blinn-Phong surface shading method, lighting, soft and hard shadows, and glow. In addition, it should be possible to create scenes and view them using the application, and this process should be as straight forward as possible.

Listed below are the key objectives that this project set out to achieve. These objectives helped guide the project in the correct direction as to achieve its aim.

Objective 1: Research topic

Background research of the project area to gain a better understanding of the chosen topic and the scope of the project. Once this first stage was completed, the research must be kept up to date and any significant developments in the project research area should be explored.

Objective 2: Investigate existing solutions

Several relevant existing solutions exist, and an analysis of their strengths and flaws helped guide the project in the correct direction. This information was kept up to date and any relevant newly released applications were added and reviewed.

Objective 3: Core functionality

Implement the core functionality of a non-real-time 3D fractal renderer. This objective formed the safe core of the project, and the following objectives built upon this.

Objective 4: Additional functionality

This involves adding additional functionality to the renderer, such as making the application capable of real-time rendering, adding a game-loop, adding a controllable camera, making scenes dynamic, and adding optical effects and lighting. The scope of this objective can be increased or decreased as necessary, and several additional stretch goals have been included in the requirements specification.

Objective 5: Evaluation

Benchmark the performance of the application across various systems and evaluate how successfully the project aim was achieved.

Objective 6: Create user documentation

Create documentation to assist users or future developers of the application.

These objectives form the main tasks to be completed during the duration of this project, and the requirements specification in section 7.9 and the Gantt chart in section 5.3 have been structured around these. It is necessary to complete some of the objectives in the order they are specified, as they build upon previous objectives. Objectives one and two will run for the entire duration of the project, to ensure that the project stays up to date with current developments. Objectives three to six, however, relate to specific functionality to be implemented, and must be completed in order.

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

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. Objective 4 leaves large amounts of flexibility and can be extended or cut back depending on time constraints.

At the time of writing this report, the first stage of objectives one and two have already been completed. Additionally, progress has been made towards objective three and initial experimentation rendering still images of the Mandel bulb fractal and other geometry has been successful. Some of these renders can be viewed in the appendix, section 7.1, and all project files can be found in the project GitHub repository [6]. In addition, some experimentation with OpenCL, a library that allows code to be executed in parallel on the graphics card 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.

1.4 Document Structure

Continuing from section 1, section 2 discusses relevant background literature for the project. Section 3 contains the requirements specification for the application and Section 4 discusses the technical design of the application. Section 5 discusses the strategy for testing and evaluating the application. Section 6 contains a plan for the project and Section 7 concludes the document.

2 Literature Review

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

This review is split into several sections, first the theory of 2D fractals and their 3D counterparts will be discussed. Then the key concepts of ray marching, the chosen method of rendering fractals, will be outlined. This will be followed by a brief introduction to the core concepts of GPU parallel programming, which is followed by a short analysis of several relevant existing solutions.

As discussed in the introduction, a fractal is a recursively created never-ending pattern that is usually self-similar [1]. This concept defines fractal geometry, which describes up the more non-uniform shapes found in nature, like clouds, mountains, and coastlines. There exist several ways of artificially creating a fractal, from manually defining a simple repeating pattern to studying the convergence of equations. This section will discuss several common 3D fractals (and their 2D counterparts) and the methods used for creating them.

2.1.1 Sierpiński Tetrahedron

The Sierpiński tetrahedron, also known as the Sierpiński pyramid, is a 3D representation of the famous 2D Sierpiński tringle fractal, 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 when creating this fractal using a computer, some maximum depth must be specified as computers only have finite memory.

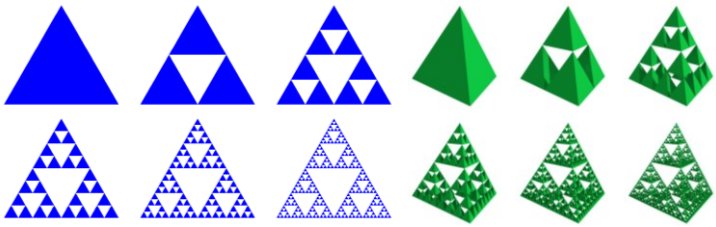


Figure 1.4.1 Sierpiński triangle (left) [8] *and tetrahedron (right)* [8] *both of recursive depth 5*

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

2.1.2 Menger Sponge

The Menger sponge, also known as the Menger cube or Sierpiński cube, is another 3D representation of one of Sierpiński's 2D fractals. This 2D fractal is known as the Sierpiński carpet, which follows very similar recursive rules to the Sierpiński triangle but uses squares instead of triangles.

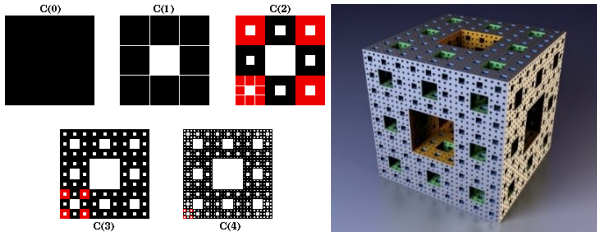


Figure 1.4.2 Sierpiński carpet (left) [9] *and Menger sponge (right)* [10] *both of recursive depth 4*

The number of objects required to create these fractals at various recursive depths increases similarly to the Sierpiński triangle and tetrahedron, but at a larger rate. The Sierpiński carpet increases by a factor of 8 each iteration while the Menger sponge by a factor of 20. A Menger sponge at recursive depth n is made up of 20^n smaller cubes. As with the Sierpiński tetrahedron, the limiting factor when trying to create this fractal is the exponential growth of the number of objects in the scene as the recursive depth increases.

In addition to the Sierpiński tetrahedron and Menger sponge, Sierpiński variations of other convex polyhedrons exist. A convex polyhedron is the 3D equivalent of a 2D regular polygon. While there are an infinite number of regular polygons, there are only fixe possible convex polyhedrons. These are the tetrahedron, cube, octahedron, dodecahedron, and icosahedron. These polyhedrons are known as the platonic solids, and their fractal counterparts are called the platonic solid fractals. This Wikipedia page contains a concise and informative list of all these shapes [11].

2.1.3 MandelBulb

While the platonic solid fractals are created by making many copies of a primitive shape, another method for creating fractals is to plot the convergence of values for certain mathematical equations. This can instead generate much more "natural" looking fractal patterns. One of the first fractals of this type to be discovered was the Mandelbrot set, discovered by Adrien Douady and named after Beniot Mandelbrot, the inventor of the concept of fractal geometry. The Mandelbrot fractal is defined as the set of complex numbers c for which the iteration from $z = 0$ in the equation $z_{n+1} = z_n^2 + c$ remains bounded (does not diverge to infinity) [12]. This definition is commonly written in the form $f_c(z) = z^2 + c$, where the value of c is varied. While this equation is staggeringly simple, when plotted on a graph using the value of c (a complex number in the form $x + iy$) as the position on the axis, with x and y corresponding to the position on the x and y axis, a colour can be assigned based on how quickly that value of c tends towards infinity. This results in a beautiful shape that when zoomed in on, shows more fractal shapes within. This Wikipedia page contains a fantastic image gallery showing common shapes found within the Mandelbrot set [13], a few of which are shown below.

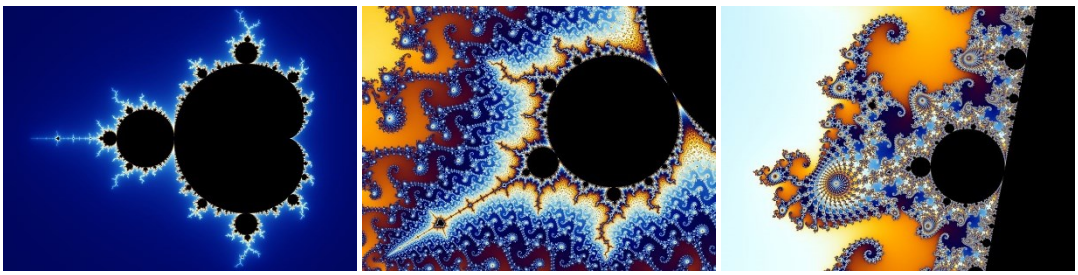


Figure 1.4.3 Mandelbrot set overview (left) [13]*, antenna (middle)* [13]*, and upside-down seahorse (right)* [13]

The Mandel bulb is a commonly used 3D representation of the 2D Mandelbrot fractal, created by Daniel White and Paul Nylander. 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 [14]. While this is still the case, White and Nylander made a significant breakthrough which resulted in the creation of a 3D fractal bearing similar characteristics to the 2D Mandelbrot set.

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 [15]. This change leads us to the equation $f(z) = z^n + c$, where z and c are triplex numbers, representing a point with an x, y, and z coordinate. The value of n is varied to give different results, where $n = 8$ is commonly used as this results in a good amount of fractal detail when zooming in.

White and Nylander's formula for the nth power of a point in 3D space [15], [16] is given as:

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

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

And the addition of two points is given as:

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \end{bmatrix}$$

Using both formulas, the equation $f(z) = z^n + c$ can now easily be solved and when rendered in 3D results in some beautiful images. Some renders from Daniel White's website are shown below.

Figure 1.4.4 Mandel bulb power of two (left) [17]*, three (middle)* [17]*, and eight (right)* [17]

The power of two version of the function (left) results in a filled 3D Mandelbrot set, with little fractal detail. Power of three (middle) contains more detail, and power of eight (right) is the sweet spot for this formula in the trade-off between fractal detail and performance. There exist several variations of Mandel bulb formula, each balancing performance with fractal detail.

The most notable performance improvement for this algorithm was discovered by David Mankin [18], and uses a distance estimation function, which for any point in 3D space, returns an estimation of the distance to the surface of the geometry. Distance estimation functions exist for many other 3D fractals as well, meaning that this is a valid generalised method for rendering 3D fractals.

In addition to the Mandel bulb, there exist many other 3D fractals. One interesting example are the Julia sets, which come from the same $f(z) = z^2 + c$ equation as the Mandelbrot set, but the value of z is varied instead of the value of c. An analysis of this fractal and will be completed in the future.

2.1.4 3D Fractals Summary

In summary, there exist two main ways of creating 3D fractals. The first approach is used for creating the platonic solid fractals and relies on taking primitive shapes and applying transformations, rotations, and scaling operations to them. The main bottleneck of rendering this type of fractal is the number of objects in the scene that must be rendered. The second approach is used for rendering more natural looking fractals and relies on plotting the convergence of equations and arbitrarily colouring them depending on how quickly the values converge. The main bottleneck of this approach is the number of iterations required for the values to converge,

which can be improved using a distance estimation function which for any point in 3D space, returns an estimation of the distance to the surface of the geometry.

Both approaches discussed create completely different looking fractals, and the application will contain examples of both. However, to view these fractals, a suitable rendering approach which supports both primitive object transformation, rotation, and scaling operations, and supports the rendering of a surface defined by a series of points in 3D space. This will allow both fractal rendering approaches to be supported.

2.2 Fractal Rendering Methods

2.2.1 Rasterization

In computer graphics, there are two commonly used methods of rendering an image of a 3D scene. The first is called rasterization, which renders an image of a scene by looping through all objects in that scene, determining which pixels on the screen are affected by that object, and modifying them accordingly. This approach has many benefits [19], most notably its speed when rendering an image. Additionally, when doubling the number of pixels in an image, the time taken to rasterize the image increases by less than double. Because of these benefits, rasterization is the most common rendering method, and all graphics cards (GPUs) are designed to efficiently render images using this approach.

Rasterization is very well suited for rendering 3D objects that are stored as meshes, which contain vertices, edges and faces. Unfortunately, a 3D fractal could not be converted into a mesh without losing detail, as a mesh only contains a finite number of points, and a 3D fractal must contain infinite detail.

2.2.2 Ray Tracing

Ray tracing is another method of rendering an image of a 3D scene. 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. Ray tracing is ideal for photorealistic rendering as it takes into consideration many of the properties of light, through simulating reflections, light refraction, and reflections of reflections [20]. Often, ray tracers do not render images in real-time as the process is computationally expensive. To make a ray tracer capable of rendering in real-time, many approximations must be made, or hybrid approaches used.

With a little modification, the ray tracing algorithm can be modified to render an image represented by a distance estimation function instead. This approach is called ray marching, and it can be used to render 3D fractals since fractal geometry can be represented by a distance estimation function. The optimisation for the Mandel bulb fractal which uses distance estimation [18] gives such a massive performance increase when compared to ray tracing, that this approach can be used to render fractals in real time.

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 geometry. 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 surface of the closest geometry. 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 zero 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.

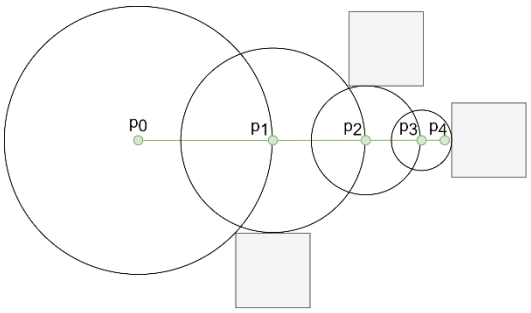


Figure 2.3.1 Ray marching diagram

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 above, ray marching does not require a surface intersection function like ray tracing does, which means it can be used to render geometry for which these functions do not exist.

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 approximate how exposed each point in a scene is to ambient lighting [20]. 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 [21]. 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 [22]. 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 collided with the geometry, a glow can be applied using the minimum distance the ray was from the object, a strength value, and colour specified [21].

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 zero 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 the SDF values for the geometry in the scene and will return the closest to the point specified.

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, such as spheres, boxes, and planes. A full list of these functions can be found on Inigo Quilez's web page [23]. A full example of the SDF for a sphere is included in appendix section 7.2. Since distance estimation can be used to represent primitive shapes, the only thing still required to render the platonic solid fractals is the ability to transform, rotate and scale primitives.

2.3.3 Transforming SDFs

Translating and rotating objects is trivial when using distance estimation. All that is required, is to transform the point being sampled with the with the inverse of the position and rotation used to place an object in the scene [23]. A function to transform a point by a translation and rotation is given below.

$$transform(p, t) = invert(t) * p$$

$$where\ p = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, t\ is\ a\ 3 \times 4\ transformation\ matrix\ storing\ only\ translation\ and\ rotation$$

In addition, uniform scaling of an SDF can be completed by first decreasing the scale back to one, sampling the point, and then increasing the scale back up again. Functions for decreasing and increasing the scale of a point in 3D space are given below.

$$scaleDown(p, s) = \frac{p}{s}, \qquad scaleUp(p, s) = p \times s, \qquad where\ p = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, s\ \in \mathbb{R}\ is\ the\ scale$$

2.3.4 Combining SDFs

Another feature that would be nice to include in the renderer is the ability to combine SDFs, and to have multiple objects in the same scene. SDFs can be combined using the union, subtraction, and intersection operations [24], as given below.

$$union(a, b) = \min(a, b),\ subtraction(a, b) = max(-a, b),\ intersection(a, b) = max\ (a, b)$$

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

There also exist variations of formulas above which can be used to apply a smoothing value to the operation. These formulas have been included in the appendix section 7.3. The images below were rendered using an early prototype of the application.

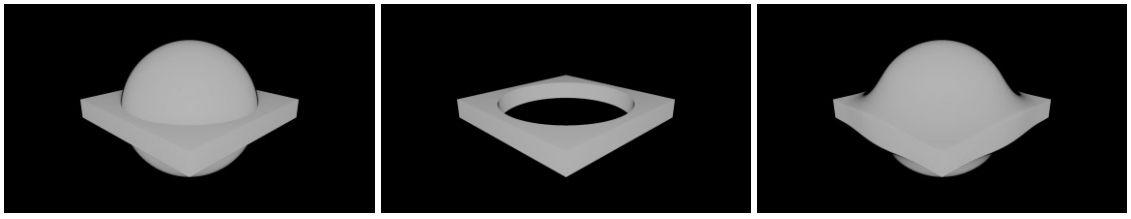


Figure 2.3.2 Ray marched sphere and box scene experiment union (left), subtraction (middle) and smooth union (right)

There are several additional alterations that can be applied to primitives once we have their SDF [23]. 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 such as a noise function or sin wave, though these alterations are more expensive. All these techniques mentioned will be essential when creating more complex geometry.

2.3.5 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. This information is is essential for most lighting calculations and surface shading techniques, such as phong shading [20] (see appendix 7.1 for an example). Lighting and surface shading functionality has been added as stretch goals in the requirements specification section 7.9. When using distance estimation, the surface normal of any point on the geometry in a scene can be determined by probing the SDF function on the x, y and z axis, using an epsilon value of arbitrary value. The formula used to calculate the surface normal of any point using distance estimation can be found in appendix 7.4.

2.3.6 Ray Marching Summary

In summary, ray marching is a variation of ray tracing which contains all the building blocks required for rendering both types of 3D fractals. It uses distance estimation and signed distance functions for calculating intersections with geometry, primitives can be modelled using SDFs, and SDFs can be translated, rotated, and scaled. In addition, there are many mathematical operations that can be used for combining SDFs to create more complex scenes and the surface normal of geometry can be calculated which means that it is possible to create advanced rendering features like surface shaders. The next section will give a brief introduction to GPU computing and how it will be used in the project to execute the rendering code in parallel on the GPU.

2.4 Surface Shading

2.4.1 Phong Shading

2.5 GPU Computing

GPU computing is when a GPU (Graphics Processing Unit) is used in combination with a CPU (Central Processing Unit) to execute some code [25]. The correct use of GPU computing improves the overall performance of the program by offloading some computation from the CPU to GPU. A CPU is designed for executing a sequence of operations, called a thread. A CPU can execute a few tens of threads in parallel and is designed to execute them as fast as possible [26]. On the other hand, a GPU is designed to execute a few thousand of these threads in parallel but does it significantly slower than the CPU would, achieving a higher overall throughput. This difference in architecture between CPUs and GPUs means that CPUs are better suited for computations requiring data caching and flow control, while GPUs are better suited for highly parallel computations.

Implementing a graphics renderer using GPU computing is the perfect choice, as this is a massively parallel task requiring a computation to be executed for every single pixel on the screen. Making use of this architecture is the only feasible way to implement a real-time renderer. When implementing a real-time fractal renderer, the extra features that libraries like OpenGL [27] provide, such as compute shaders, aren't needed. Instead, it makes more sense to make use of general-purpose GPU computing library, as the overhead of this software will be significantly less, giving us better performance.

2.5.1 CUDA vs OpenCL

Currently, there are two suitable general-purpose GPU computing interfaces, CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language). Both interfaces make use

of a kernel language, a specialised programming language in which the code to be executed in parallel is written. The table below breaks down the main advantages and disadvantages [26], [28], [29] of CUDA and OpenCL.

Table 2.5.1 CUDA and OpenCL comparison

Comparison	CUDA	OpenCL
Portability	Only runs on NVIDIA hardware	Runs on pretty much all hardware - NVIDIA, AMD, Intel, Apple, Radeon, etc
Open Source	Propriatery framework of NVIDIA	Open source standard
Technicalities		Compiles kernel code at runtime, which could take a lot of time, though this also means that the compiled code is more optimised for the device currently running it
Interface Languages	C, C++, Fortran, Java Python Wrappers, DirectCompute, Directives (e.g. OpenACC)	C, C++, Python wrapper
Kernel Language	KUDA C++	OpenCL C (C99) and OpenCL C++ (C++17)
Libraries	Extensive and powerful libraries	Good libraries, but not as extensive as CUDA
Performance	No clear advantage	No clear advantage
OS Support	Runs on Windows, Linux and MacOS	Runs on Windows, Linux and MacOS

The main difference between CUDA and OpenCL, is CUDA can only run on NVIDIA branded GPUs. While this technically does allow CUDA to make full use of the hardware, it makes the application deployment far less portable as the code will not run on other types of GPUs. OpenCL is, therefore, the superior choice when aiming to write portable GPU code.

2.6 Review of Existing Applications

A review of relevant and popular existing 3D fractal renderers was completed, and the key features of each application was recorded. A summary containing points to take away from the review is included at the end.

2.6.1 Fragmentarium [30], [31]

- Features an editor within the application, which allows scenes to be created and previewed in real time
- Contains many example scenes
- Provides very detailed parameters to edit the scene
- Rendering quality of the output image is acceptable (see appendix 7.5)

2.6.2 Mandelbulb3D [32]

- User interface is clunky, and features are hard to find
- Only contains a couple example scenes
- The performance of the real-time preview feature is sub optimal, making it hard to navigate the scene
- The output image is very detailed, is coloured beautifully and contains many advanced optical effects (see appendix 7.6)

2.6.3 Existing Applications Summary

After reviewing existing 3D fractal rendering applications, there are several key points that should be considered when designing the application.

- The application must contain lots of example scenes, and they must be easy to find
- The real-time preview of the scene must have good performance and be easy to control
- It would be nice to be able to output an image of the scene, with an easy-to-use interface
- It must be relatively easy to create a new scene and preview it

Additionally, Fragmentarium and Mandelbulb3D are both built using OpenGL with GLSL shaders, as this makes creating scenes simple as it uses an already existing language and format. However, using OpenGL is unnecessary as not many of the features it provides are being used.

2.7 Literature Review Summary

In summary, there are two main methods of creating fractal patterns - recursively applying transformations to primitive shapes and plotting the convergence of equations. To be able to render both types of fractals, a variation of ray tracing called ray marching must be used which uses a distance estimation function to calculate the distance to the closest piece of geometry in the scene. In addition, to be able to render 3D fractals in real time, GPU computing techniques must be used to execute code in parallel. Several existing applications have already used similar approaches to great success, however, these applications do contain some flaws and a note of these have been made so that this project will not make the same mistakes.

3 Development

3.1 Development Environment

Windows 10

Visual studio 2019

Github desktop

Cmake which is cross platform

Links to packages from package managers

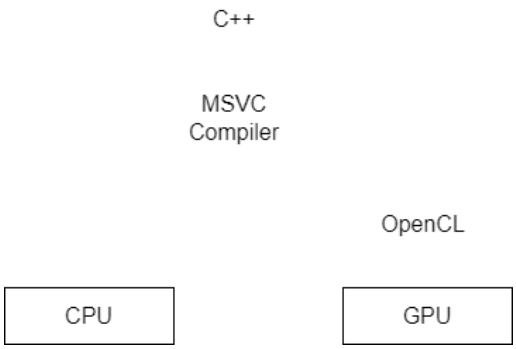
Development of the application and documentation will be assisted the following technologies:

Table 3.1.1 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

3.2 Application Structure

3.2.1 High Level Overview



3.2.2 Main Application

The application will be developed following the main object-oriented principles [33]. 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.

The application will be structured using an object-oriented programming style to promote code reusability and data encapsulation. The application will consist of several key classes which are listed in the table below. A provisional class diagram has been included in appendix 7.7.

Table 3.2.1 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
Input	Reading keyboard and mouse input from the user

The Display and Input classes are basic and only provide a wrapper around corresponding SDL2 window methods for setting pixels in the display, and for reading user input. The Renderer class,

however, is much more complex and requires discussion. The Renderer class contains the current pixels to be displayed on the screen this frame, which is 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 in appendix **Error! Reference source not found.** below show which methods and constants in the kernel are reusable between scenes and which are not.

A solution to reducing code duplication between the kernel files is to use the new OpenCL C++ kernel language, released in March 2021, which supports most C++17 features. The OpenCL C++ kernel language documentation can be found here [34]. The Renderer class will contain a main kernel file which contains the implementation of all reusable methods, along with an empty distance estimation method which the other kernel scene files must override, using method overriding. Method overriding is an object-oriented programming technique which allows a subclass to change the implementation of a method that a parent class is providing. This will drastically reduce the amount of code duplication between scenes as it allows the scenes to share common code. This is important for making scene creation accessible for newer users, who would be overwhelmed by a scene file containing hundreds of lines of code.

3.2.3 Kernel

3.3 Technologies

The application will be developed using the following technologies:

Table 3.3.1 Application technologies

Technology	Description	Justification
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
CMake		•
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
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
OpenMP		•
Eigen		•
GoogleTest	Unit testing framework	• Unit testing is essential for ensuring code is correct and functionality has been implemented • This framework comes with the Visual Studio 2019 C++ development package which is also being used

3.4 Development Strategy

The project will be developed using an Agile [35] 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. While this an individual project and the Agile approach is normally used in teams, the continuous improvement gained through this approach will be incredibly valuable for the project.

3.5 Documentation

3.5.1 User Guides

3.6 Interface Design

The application will feature a simple and intuitive user interface which contains a main display window, with a ribbon toolbar at the top. The toolbar will contain several tabs, including Settings, Load Scene, Create Scene, and Help. A mock user interface has been included in appendix 7.7.

4 Evaluation

To accurately evaluate the successfulness of the project in respect to the original aims and objectives, several different evaluation strategies must be consulted. First, the unit testing strategy will be discussed as this form of testing was vital in ensuring that the application behaved as expected during development. Then, the application benchmarking framework must be discussed, and results gathered from this framework will be analysed and used to assess the value of the application. Finally, the overall success of project will be evaluated using a goal-based evaluation strategy to determine to what extent the project met the original aims and objectives.

4.1 Unit Testing

Unit testing was a small but vital part of ensuring the application meets the requirements specified and behaves as expected. Unit tests were created for key classes used by the application, including profiling tools such as the high-performance timer and benchmark data classes, and classes containing complex behaviour such as the scene class which is responsible for moving the camera along a camera path during benchmarks. In addition, unit tests were created for all the key scenes in the application, and these tests ensure that each scene can be loaded by the application without crashing.

Due to the nature of the application, automated tests cannot be used to determine if functionality has been implemented or if the functionality behaves as expected, as most of the features of the application have a visual output which cannot be automatically checked.

4.2 Benchmarking Framework

The benchmarking framework in the application is responsible for storing performance data of the application and outputting that data when the application closes. The benchmark provides two main pieces of functionality, firstly it compares the duration taken to render each frame to the current minimum and maximum and updates their values respectively. And secondly it is also responsible for calculating the time taken to execute key pieces of code. The performance

benchmarking tools were added to the application to evaluate the value in achieving real time brought by each optimisation, and the performance hit of each graphical improvement. The performance benchmark always runs.

The performance benchmark is interfaced solely through the scene kernel file, as this is more flexible than using a command line argument, and it keeps all scene parameters within one location. For the benchmark scenes, either a stationary camera or camera path was used as this ensures that the geometry viewed by the camera is consistent across runs. It would not be appropriate to allow the user to control the camera as this could significantly increase the variance of results. The camera path is defined using the `CAMERA_POSITIONS_ARRAY` and `CAMERA_FACING_DIRECTIONS_ARRAY` compiler directives, which both contain an array of float4 values, for a position/facing direction and a time at which this value should occur. Additionally, `BENCHMARK_START_STOP_TIME` is used to specify the duration of time that the benchmark should be active and `CAMERA_DO_LOOP` is used to move the camera back to the starting position when it reaches the end. This information is loaded from the scene file on start-up and data is passed to the Scene class, which is responsible for linearly interpolating between camera position/facing values, and for closing the application once the duration is exceeded.

While the benchmark is running, the time taken to render each frame is compared to the current minimum and maximum that has occurred and is updated accordingly. Additionally, this frame time is also added to the total benchmark duration and the number of frames that have occurred is incremented. This total duration value is important as the benchmark will not run for the exact amount of time specified by the `BENCHMARK_START_STOP_TIME` directive, as it will always have to finish rendering the current frame before exiting. This means that the total duration value is likely to exceed the benchmark duration by the duration of one frame at most. However, since different systems take different amounts of time to compute each frame, this value can vary a bit between systems.

Once the benchmark has concluded, the results are appended to the file "results.txt" which can then be easily copy and pasted into a spreadsheet. Additionally, the results are also displayed on the terminal window so that the user can read them. The table below describes all the benchmark results that are recorded by the application.

Table 4.2.1 Benchmark framework results

Description	Units	Origin
Scene name		Runtime Parameter
OpenCL build options		Runtime Parameter
Resolution width	Pixels	Runtime Parameter
Resolution height	Pixels	Runtime Parameter
Device name		OpenCL Query
Device version		OpenCL Query
Work group size		OpenCL Query
Clock frequency	MHz	OpenCL Query
Number of parallel compute units		OpenCL Query
Global GPU memory	Bytes	OpenCL Query
Local GPU memory	Bytes	OpenCL Query
Constant GPU memory	Bytes	OpenCL Query
Total benchmark duration	Seconds	Benchmark
Total number of frames rendered		Benchmark
Maximum frame time	Seconds	Benchmark
Minimum frame time	Seconds	Benchmark

From the raw data specified above, the following results were calculated for each benchmark run.

Table 4.2.2 Results calculated from benchmark

Description	Units	Calculation
Total number of pixels		$resolution\ width \times resolution\ height$
Global GPU memory	Gigabytes	$\frac{global\ GPU\ memory\ (bytes)}{1000000000}$
Local GPU memory	Kilobytes	$\frac{local\ GPU\ memory\ (bytes)}{1000}$
Constant GPU memory	Kilobytes	$\frac{constant\ GPU\ memory\ (bytes)}{1000}$
Mean frame time	Seconds	$\frac{benchmark\ duration}{total\ number\ of\ frames\ rendered}$
Mean frames per second	Frames per second	$\frac{1}{average\ frame\ time}$
Maximum frames per second	Frames per second	$\frac{1}{minimum\ frame\ time}$

Minimum frames per second	Frames per second	$\frac{1}{maximum\ frame\ time}$
Upper difference	Frames per second	$maximum\ FPS - mean\ FPS$
Lower difference	Frames per second	$mean\ FPS - \min FPS$

4.3 Fractals Implemented

The application features several scenes containing 3D fractal geometry, which include the Sierpiński cube and tetrahedron fractals, and the Mandelbulb fractal. Two benchmark scenes were created to analyse the performance of the application when rendering these fractals. The scene sierpinski_collection.cl contains both the Sierpiński cube and tetrahedron fractals, while the scene mandelbulb.cl contains the mandelbulb fractal.

Several other non-trivial scenes have been implemented not featuring 3D fractal geometry. The scene mandelbrot.cl contains the 2D Mandelbrot fractal mapped to a 3D plane, which the user can explore. The scene mandelbrot_zoom.cl contains a camera path zooming into the Mandelbrot fractal. The scene terrain.cl contains mountainous terrain generated using 2D Perlin noise and the scene planet.cl contains an Earth like planet generated using 3D Perlin noise. Additionally, the scene infinite_spheres.cl contains an infinite number of spheres.

4.4 Evaluation of Features

4.4.1 Key Optimisations

Ensuring that the performance of the application was good enough to render 3D fractals in real time was a challenge, and there are several key optimisations that were implemented to achieve this. The first optimisation, called the linear epsilon optimisation below, involves linearly increasing the epsilon value used for determining ray intersections as the ray gets further away from the camera. This optimisation has a nice side effect as it reduces the amount of noise generated on geometry far away while also increasing performance significantly.

The second key optimisation involved adding bounding volumes around geometry in the scene. BOUNDING VOLUME DESC? One draw back of this optimisation is that it breaks the geometry glow feature, which relies on keeping track of the closest distance that each ray got to the geometry. When geometry glow is used with a bounding volume, this makes the bounding volume glow as well as the geometry, which in most cases completely ruins the effect, as can be seen below. This problem would be quite hard to fix without negating the performance gain from using bounding volumes. One way would be to have two distance estimation functions in the scene, one containing the bounding volume and the other containing the geometry, instead of combining the two estimations into one function. Then, whenever the distance estimation to the bounding volume falls below a threshold, the program could call the main distance estimation function. When calculating the glow value all values returned from the bounding volume distance estimation function could be ignored, and distance to the actual geometry could be sampled instead. This would result in a more accurate glow shape, though there would be visual artifacts. This separation of the two distance functions would require quite a significant rework of the application, though it would have some benefits.

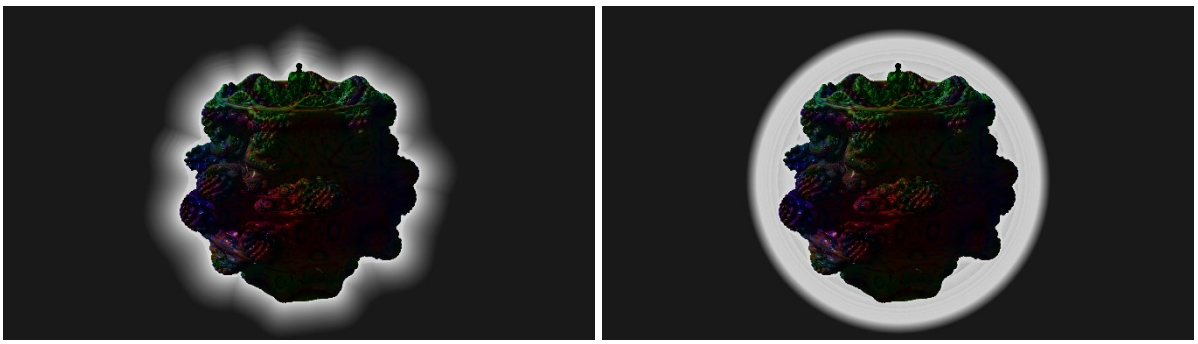
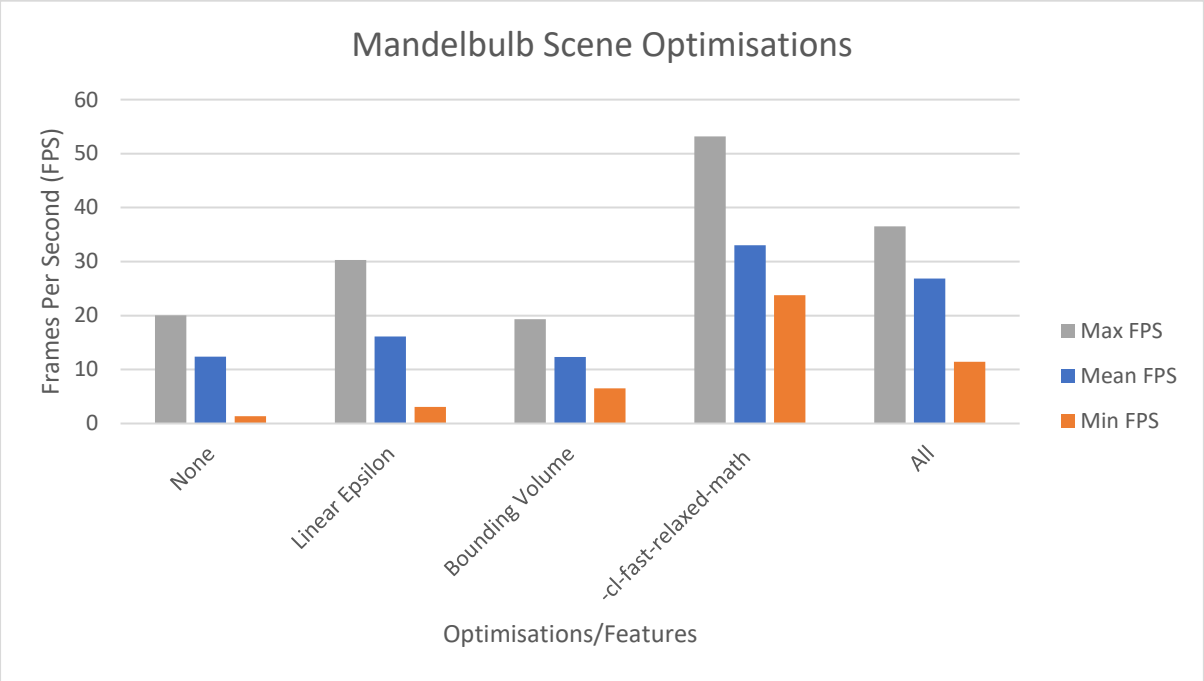


Figure 4.4.1 Mandelbulb scene with glow without bounding volume (left) and with bounding volume (right)

The final key optimisation results from using the `-cl-fast-relaxed-math` OpenCL build flag when loading the scene at runtime. This build flag instructs the compiler to assume that the input will never be NaN or infinity, to ignore the signedness of zero, and to reduce the precision of certain arithmetic operations (such as $a \times b + c$). These optimisations provide a trade-off between performance and result correctness and violate several IEEE 754 conventions. This optimisation only effects the compiled OpenCL kernel code, not the C++ side, meaning that only

the visual output is affected and not the accuracy of the profiling timers. Several screenshots showing the output of the application when this optimisation is enabled and disabled are included in appendix section 7.8. No noticeable visual artifacts can be seen when comparing the two images so this optimisation is enabled by default but can be disabled using the command line argument `--force-high-precision`.

The performance of the three optimisations discussed above were benchmarked using the Mandelbulb benchmark scene, and the graph is shown below.



The bounding volume optimisation for this scene resulted in an average of less frames per second than without any optimisations. This is because the benefit gained by using bounding volumes only occurs when the camera is far away from the object, outside of the bounding volume. When the camera is near or inside of the bounding volume, then the distance estimation to the bounding volume and geometry must both be calculated resulting in overall worse performance as more computations than normal are performed. The efficiency of a bounding volume optimisation is highly dependant on the scene, and in this case it wasn't efficient. However, if the benchmark had spent more time further away from the geometry, then the performance would be significantly better.

On average, the linear epsilon optimisation resulted in a 30% increase in mean frames per second when compared to the base version. This optimisation behaves similarly to the bounding volume optimisation, as a significant performance boost is only seen when the camera is further away from the geometry, as in this case it reduces the surface complexity of the geometry to render.

The most significant optimisation was the `-cl-fast-relaxed-math` compiler flag which resulted in a 166% increase in average frames per second. TALK ABOUT IT MORE

When all three optimisations were used in combination the performance increase was only 116%, which is surprising as it would be expected for the performance of the optimisations to be combined.

4.4.2 Key Unmeasurable Features

There are also several other features of note provided by the application, some of which aren't conventionally available when using rasterization rendering. The first, and most significant is that every point that is sampled using the distance estimation function can have its own material. This means that every pixel on the screen that can see geometry will be using a different material to render it. This give a massive amount of flexibility and is incredibly powerful. This is not common among rasterization rendering, which requires that geometry be should grouped using the material that by which it should be rendered.

Additionally, the application is capable of rendering updateable geometry in real time. Since geometry is modelled using a distance estimation function this gives us far more flexibility than conventional means.

While these two features are not measurable, they are significant reasons to use this application.

4.4.3 Key Measurable Features

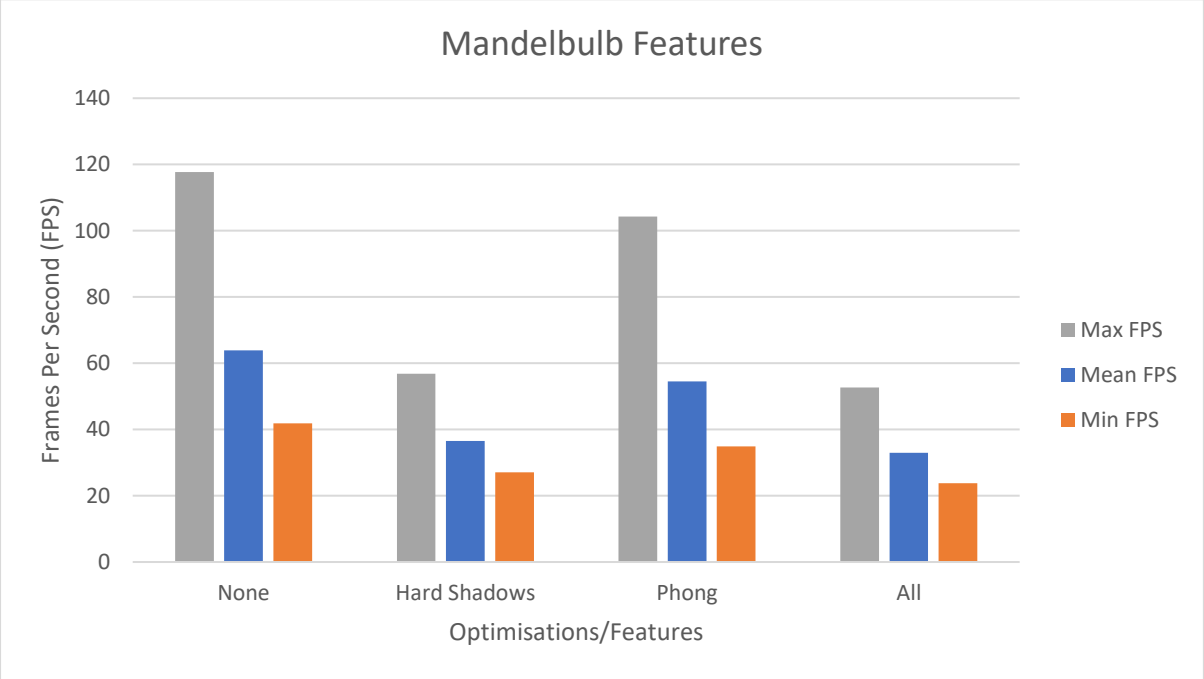
In addition to key performance optimisations, several key measurable visual features have also been implemented. These include geometry materials using the Blinn-Phong shading surface shading model, a physically based rendering method which combines ambient colour, diffuse

colour and specular highlights to efficiency approximate how real-world materials behave in light.

In addition, both hard shadows and soft shadows have been implemented and either can be added to a scene. Soft shadows doesn't always work due to weird sdf

Additionally, a geometry glow can be added.

The measurable features

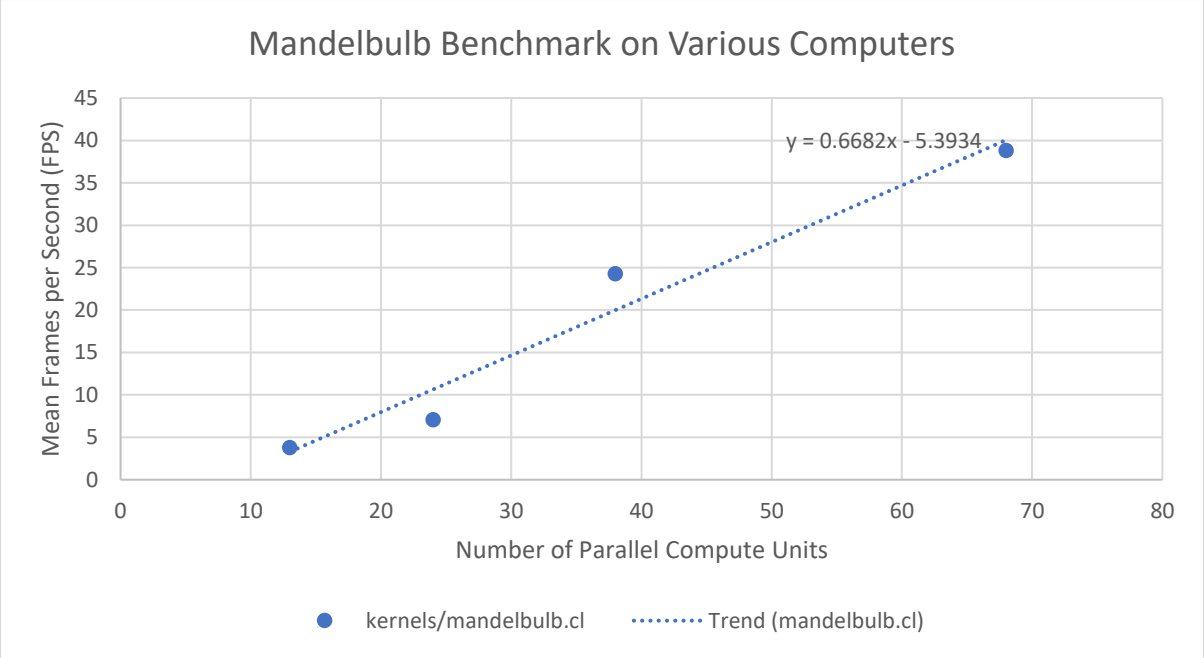


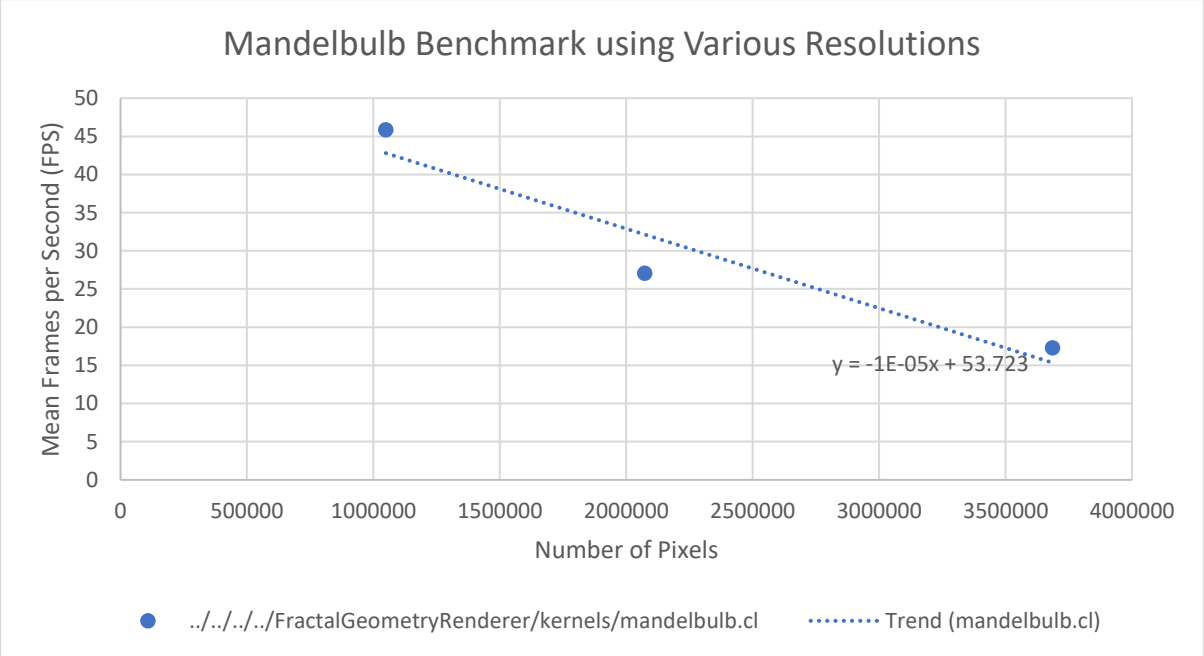
The Blinn-Phong feature saw a performance decrease of 15%, which is well worth it considering how substantial the visual payoff is. Image comparing both here

The use of hard shadows decreased the average frames per second by 43%, which is very substantial. The performance decrease is this substantial as once the ray intersects with geometry, another ray must be cast from the point of intersection in the direction of the light source. This increases the number of says cast per pixel from one to two.

Soft shadows could not be tested on the Mandelbulb scene as they don't work correctly.

4.4.4 Application Scalability


Mandelbulb Benchmark on Various Computers
Mean Frames per Second (FPS)
45
40
35
30
25
20
15
10
5
0
y = 0.6682x - 5.3934
0
10
20
30
40
50
60
70
80
Number of Parallel Compute Units
kernels/mandelbulb.cl
Trend (mandelbulb.cl)


Mandelbulb Benchmark using Various Resolutions
Mean Frames per Second (FPS)
50
45
40
35
30
25
20
15
10
5
0
y = -1E-05x + 53.723
0
500000
1000000
1500000
2000000
2500000
3000000
3500000
4000000
Number of Pixels
../../../../FractalGeometryRenderer/kernels/mandelbulb.cl
Trend (mandelbulb.cl)

4.5 Evaluation of Requirements Specification

The full project requirement specification can be found in appendix 7.9. This specification only received small changes since the initial research report as thorough project planning ensured that all.

13 must

1 should implemented

2 fully, 1 partly /5 could

4.6 Evaluation of Aims and Objectives

A goal-based evaluation strategy will be used to determine if the projects aim and objectives have been achieved. Unit tests and a benchmark scene will be used to determine how many of the requirements specified in section 7.9 have been fully implemented. For an objective to be considered achieved, all requirements prioritised MUST and SHOULD related to that objective must have been implemented. For the project aim to be considered achieved, all objectives must be achieved.

4.7 Future Developments

5 Project Plan

5.1 Professional, Legal, Ethical & Social Issues

The main principles of the open-source definition [36] state that a product must have publicly available source code, must allow modifications and derived works of the product, and allow free redistribution of the product. This project is licensed under the GNU General Public License v3.0 [37] which complies with the open-source definition and grants permission for modification, distribution, and commercial use of this product. However, all changes made to the licensed material must be documented, the modified source code must be made public, and the modified work must be distributed under the same license as this product. The license can be viewed on the project GitHub repository [6], within the LICENSE file.

The British Computer Society (BCS) codes of conduct [38] specifies the professional standards expected to be displayed by a member of the BCS, which includes working for the interest of the public, displaying professional integrity, accepting relevant authority, and showing a commitment to the profession. These standards will be respected and complied with for the duration of the project.

This project does not require any ethical approval as no studies requiring participants will be completed. Instead, the application's performance will be benchmarked over several available PCs, and results will be calculated to determine the how successful the project was.

5.2 Risk Analysis

A risk analysis for the project has been completed to identify any potential risks that may affect the successfulness of the project, and a mitigation plan has been drawn up for each risk. The mitigation plan has been designed to reduce the chances of each risk happening and to reduce the negative impact of the risk if it were to happen. Each risk was then assigned a rating using the table below.

Table 5.2.1 Risk rating matrix

		Severity		
		LOW	**MEDIUM**	**HIGH**
Probability	**LOW**	LOW	LOW	MEDIUM
	MEDIUM	LOW	MEDIUM	HIGH
	HIGH	MEDIUM	HIGH	HIGH

Table 5.2.2 Risk analysis matrix

ID	Description	Probability	Severity	Mitigation plan	Rating
R-1	Loss of work	LOW	HIGH	All work will be backed up regularly using online version control	**MEDIUM**
R-2	Change in requirements	LOW	MEDIUM	A thorough requirements specification has been prepared to reduce the probability of this happening In addition, an Agile development approach will be used, which by design minimises the negative effects of changes in requirements	**LOW**
R-3	Change of deadlines	LOW	HIGH	Communication from the course leaders will be regularly monitored This risk is very unlikely to happen as assurances have been made that the course deadlines will not change	**MEDIUM**
R-4	Delays due to learning new software	HIGH	LOW	Time was assigned during the planning stage to experiment with new software like OpenCL and GLSL shaders Additionally, some free time has been allocated at the end of the project 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**
R-7	Renderer can't be made real-time	LOW	MEDIUM	Thorough research has been completed with the conclusion that this project is feasible There is a safe core to the project and room for scaling back or adding extra functionality to the application if this were to occur	**LOW**

				In addition, performance analysis of a non-real-time renderer could be performed instead	

The project risks will be regularly monitored, and mitigation of higher priority risks will be prioritised. Each sprint of work will conclude with an analysis of the current state of the project, and from there any risks likely to occur will be identified and corresponding mitigation plans will be consulted.

5.3 Project Timeline

The project timeline for the second deliverable has been split into a total of eight sprints of work, each sprint being two weeks in duration. Sprint zero will occur in December during the Christmas holidays, and its purpose is to set up the project working environment and to refactor the existing experimentation code, created when researching and investigating existing solutions. Once this has been completed, Objective 3, the safe core of the project, will almost have been achieved. From there, sprints one and two will be used to achieve Objective 3, to add additional functionality to the application. Then, during sprint three and four, the project evaluation will be completed, and any remaining documentation will be created. Sprints five and six will be used to write up the project evaluation and complete the report, and sprint seven has been allocated as free time, in full expectation that there will be delays when completing tasks and the timeline will have to be pushed back.

A Gantt chart for the project was created using an online tool [39] and has been included below. Tasks have been coloured using the following strategy: documentation (blue), implementation (red), and unallocated (green).

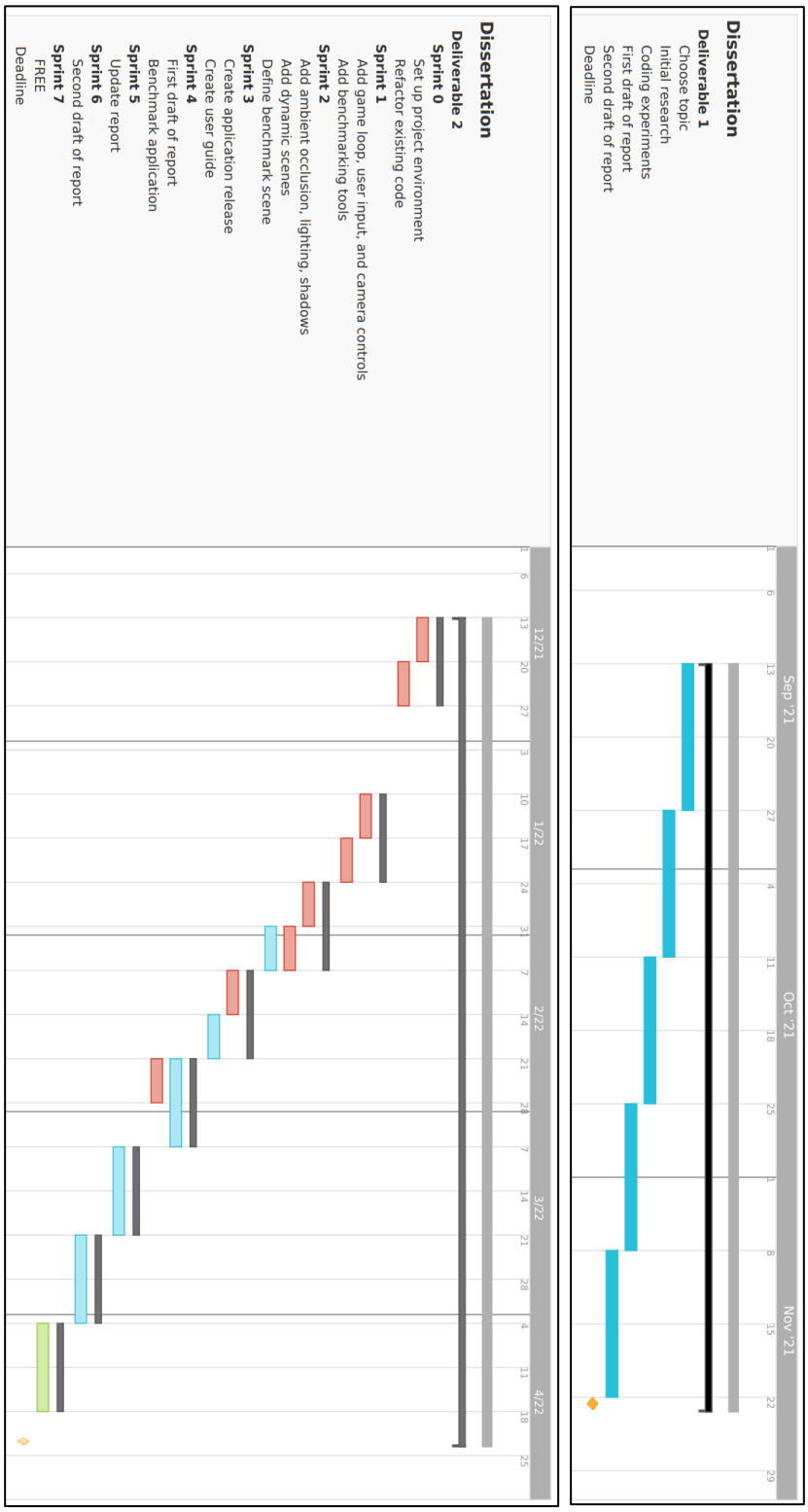


Figure 5.3.1 Project timeline Gantt chart deliverable two (left) and deliverable one (right)

6 Conclusion

The aim of this project is to develop an application which can render 3D fractal geometry in real-time. Several objectives have been defined to help guide this process. A thorough review of

literature relevant to the project has been completed, which identified the two main methods of creating 3D fractals - recursively applying transformations to primitive shapes and plotting the convergence of equations. To be able to render both types of fractals, a variation of ray tracing called ray marching must be used which uses a distance estimation function to calculate the distance to the closest piece of geometry in the scene. In addition, to be able to render 3D fractals in real time, GPU computing techniques must be used to execute code in parallel on the graphics card.

Relevant existing solutions have also been analysed and their key advantages and disadvantages have been considered when creating the application design. The project has specific and measurable requirements that will be implemented over the course of many two-week sprints, as specified in the project timeline. A thorough risk analysis and investigation into professional, legal, ethical, and social issues has been completed and a strategy has been defined for evaluating how successfully the project aim will be achieved.

6.1 Achievements

6.2 Limitations

6.3 Future Work

7 Appendices

7.1 Experimentation Renders

The images below were rendered using a very early version of the application, during the research and experimentation stage.

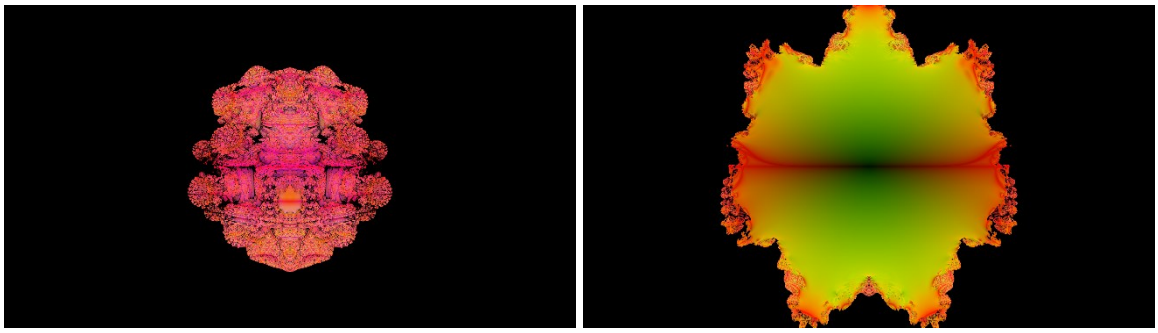


Figure 7.1.1 Render of the Mandel bulb fractal (left) and cross section (right) using equation from [40]

The value of a surface normal can be converted to a colour by mapping the x, y, and z values to r, g, and b, which can be useful when debugging. The images below show the surface normals of the sphere and box experiment scene, and an example phong shading, a surface shading technique.

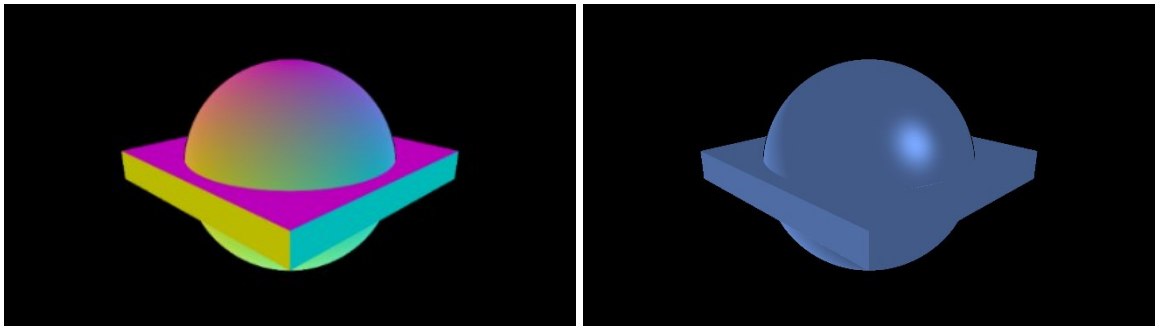


Figure 7.1.2 Surface normal visualised (left) and phong shading experiment (right)

7.2 Sphere SDF Example

Included is the signed distance function (SDF) for a sphere with radius R, positioned on the origin.

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

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

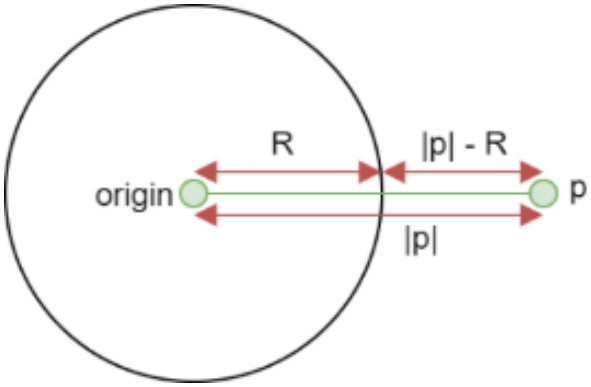


Figure 7.2.1 Sphere SDF diagram

7.3 Smooth SDF Combinations

The following equations can be used to combine two SDF values using a smoothing value.

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

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

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

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

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

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

7.4 SDF Surface Normal Calculations

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

$$where\ p = \begin{bmatrix}x\\y\\z\end{bmatrix}, e\ is\ the\ epsilon\ value, DE\ is\ the\ scene\ distance\ estimation\ function$$

7.5 Fragmentarium Renders

The following screenshots were taken when experimenting with Fragmentarium [30], [31].

Figure 7.5.1 A recursive scene (left) and Mandel bulb (right)

Figure 7.5.2 Tree fractal

7.6 Mandelbulb3D Renders

The screenshot below was rendered using MandelBulb3D [32].

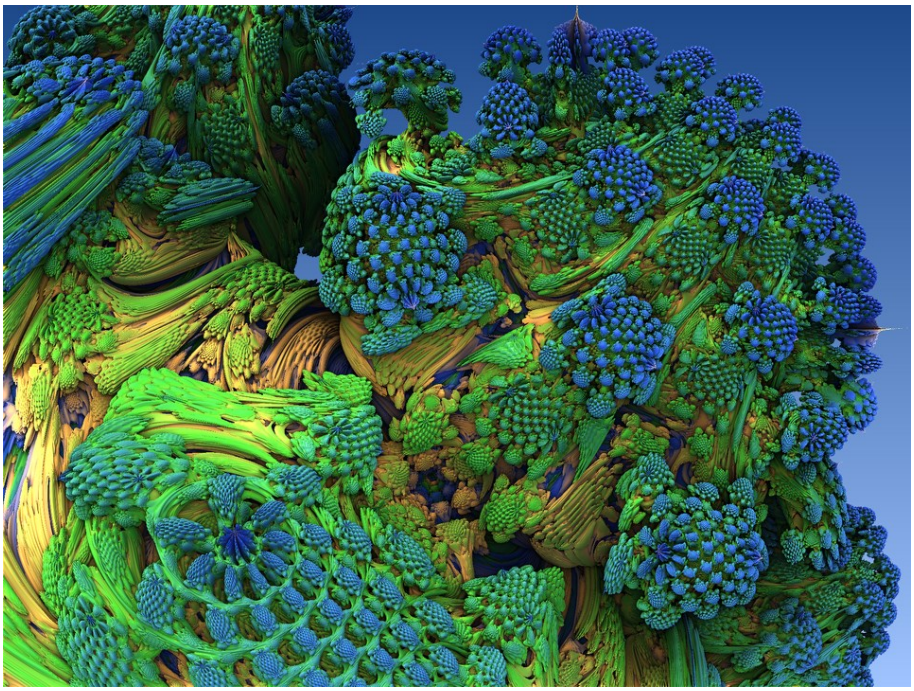


Figure 7.6.1 Mandel bulb fractal with natural looking colouring

7.7 Application Class Diagram

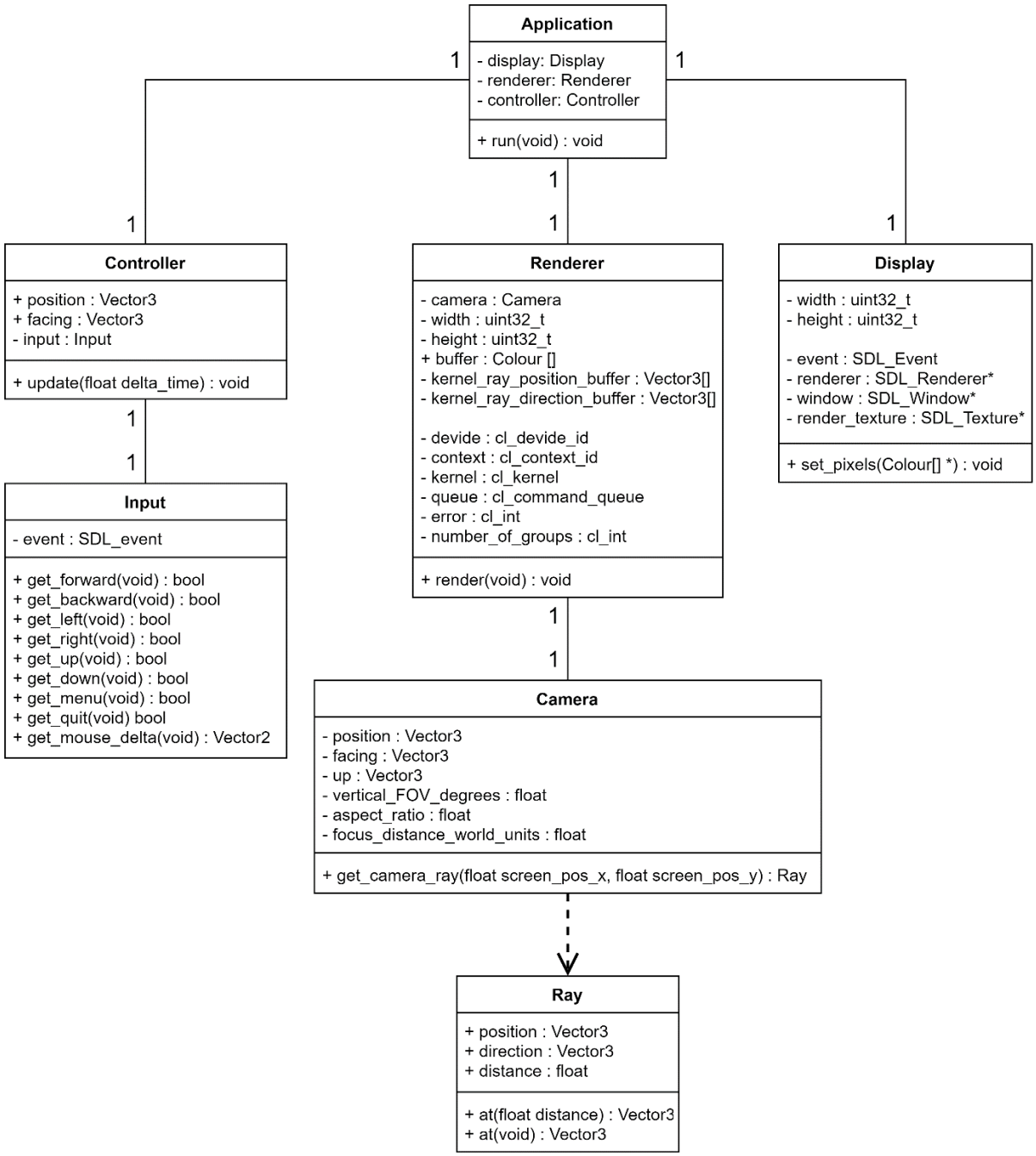


Figure 7.7.1 Application class diagram

7.8 Comparison of -cl-fast-relaxed-math Optimisation

No visual differences between -cl-fast-relaxed-math enabled or disabled.

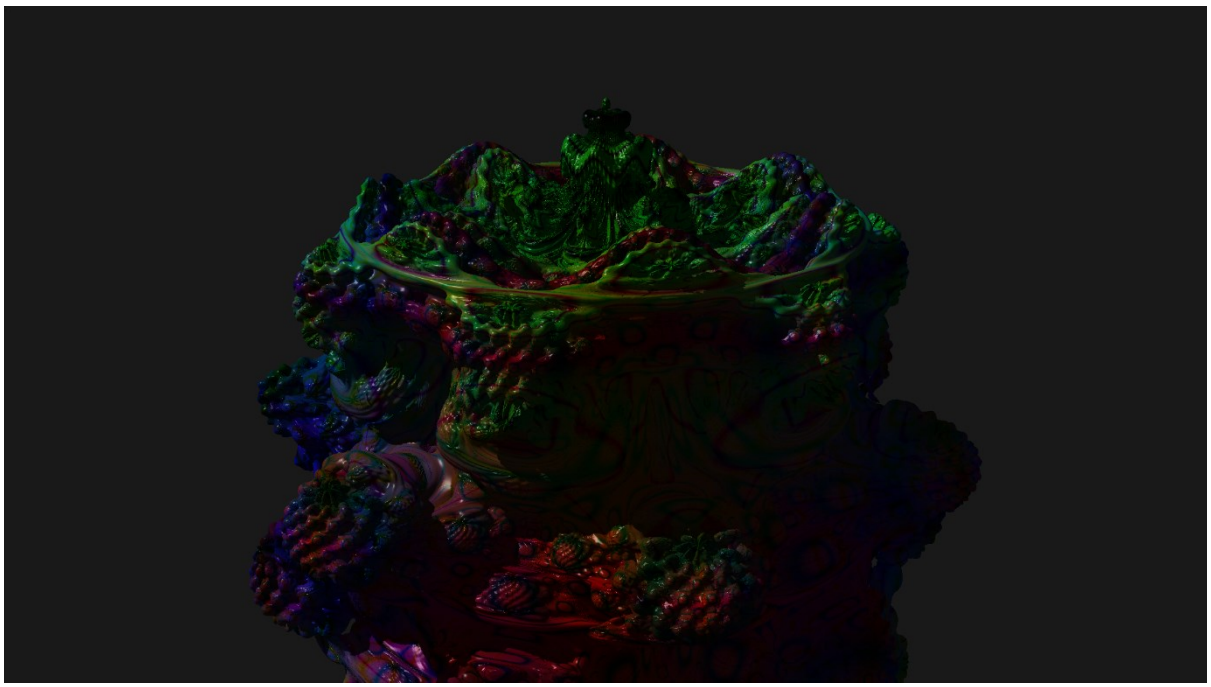


*Figure 7.8.1 Mandelbulb.cl scene with -cl-fast-relaxed-math **enabled***

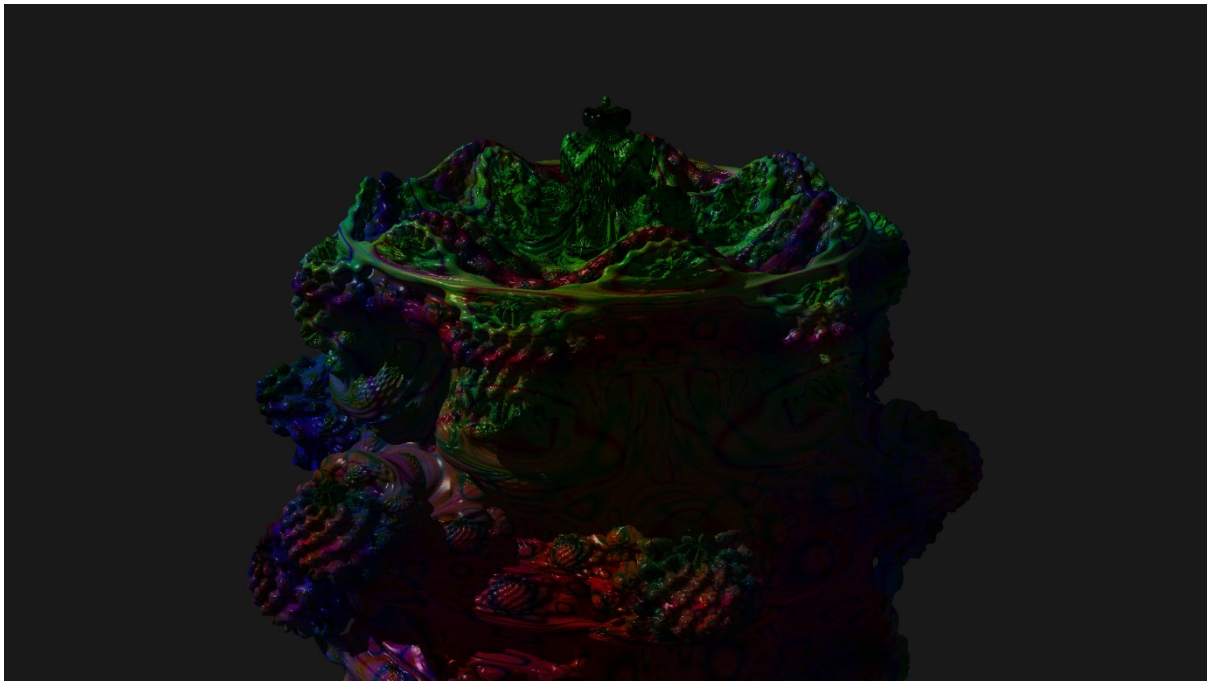


*Figure 7.8.2 Mandelbulb.cl with -cl-fast-relaxed-math **disabled***

7.9 Requirements Specification

The tables below display the key functionality to be implemented to achieve the projects aim and objectives, specified in section 1.2. Requirements have been grouped by the project objective that they fall under. Requirements were prioritised using the following strategy:

- MUST – a requirement that is of the highest priority to the project
- SHOULD – a requirement that is not essential, but it would be good if the project had it
- COULD – a requirement that is optional
- WON'T – a requirement that would be implemented if the project could run for more time

Table 7.9.1 Functional requirement specification

ID	Name	Description	Priority	Status
FR-1	**Objective 3**	**Core functionality**		
FR-1.1	Scene requirements	A scene must contain: • Geometry • A light • Camera	MUST	Achieved
FR-1.2	Example scenes	The application must contain multiple example scenes, some of which should include: • Sierpinski tetrahedron fractal • Menger sponge fractal • Julia set fractal • Mandel bulb fractal	SHOULD	Achieved
FR-1.3	Mandatory optical effects	The application must support the following optical effects: • Ambient occlusion • Hard and soft shadows • Glow	MUST	Achieved
FR-1.4	Load scene	Scenes must be defined each in their own kernel file and the user must be able to load them into the application at runtime	MUST	Achieved
FR-1.5	Settings	Basic application settings must be editable. This includes: • Output resolution • Optical effects enable/disable • Controls • Mouse sensitivity	MUST	Achieved
FR-2	**Objective 4**	**Additional functionality**		
FR-2.1	Real-time	The application must be capable of rendering the view of the scene in real time	MUST	Achieved

FR-2.2	Game Loop	The application must make use of a game loop to update the scene. Methods should be called in the following order: 1. Poll for user input 2. Update scene 3. Render scene	MUST	Achieved
FR-2.3	Controllable camera	The user must be able to control a camera, using a keyboard and mouse to move it around the scene	MUST	Achieved
FR-2.4	Dynamic scenes	The contents of a scene (requirement FR-1.1) must be able to move around at runtime	MUST	Achieved
FR-2.5	Optional optical effects	The application could additionally support the following optical effects: • Reflections • Depth of field • Transparency	COULD	Not Achieved
FR-2.6	Optional Surface Shading	The application could support the following surface shading algorithms: • Lambert • Oren-Nayar • Phong or Blinn-Phong • Subsurface scattering	COULD	Partly Achieved
FR-2.7	Fixed camera paths	The application camera could additionally move using a specified camera path	COULD	Achieved
FR-2.8	Image output	It could be possible for the application to output an image of the current camera view	COULD	Achieved
FR-2.9	Video output	It could be possible for the application to output a video sequence for the current scene	COULD	Not Achieved
FR-3	**Objective 5**	**Evaluation**		
FR-3.1	Performance benchmark	The application must contain a performance benchmark scene	MUST	Achieved

Table 7.9.2 Non-functional requirement specification

ID	Name	Description	Priority	Testing strategy
NF-1	Executable	The application must run from a compiled executable	MUST	Achieved
NF-2	Operating system	The application must run on Windows 10. Where possible, cross platform libraries should be used, though other operating systems will not be officially supported	MUST	Achieved
NF-3	Display resolutions	The application must support the following common display resolutions: 1366x768, 1920x1080, 2560x1440 and 3840x2160	MUST	Achieved
NF-4	GPU Parallel Computing	The application must run in parallel on the GPU	MUST	Achieved

8 References

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

[2] Jack Challoner, “How Mandelbrot’s fractals changed the world - BBC News,” 2010. https://www.bbc.co.uk/news/magazine-11564766 (accessed Nov. 10, 2021).

[3] T. Kluge, “Fractals in nature and applications,” 2000. https://kluge.in-chemnitz.de/documents/fractal/node2.html (accessed Nov. 10, 2021).

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

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

[6] S. Baarda, “3D Fractal Geometry Renderer,” 2021. https://github.com/SolomonBaarda/fractal-geometry-renderer (accessed Sep. 24, 2021).

[7] Wikipedia, “Wacław Sierpiński.” 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] J. Baez, “Menger Sponge | Visual Insight,” 2014. https://blogs.ams.org/visualinsight/2014/03/01/menger-sponge/ (accessed Nov. 11, 2021).

[11] Wikipedia, “n-flake.” https://en.wikipedia.org/wiki/N-flake (accessed Nov. 16, 2021).

[12] A. Douady, “Julia Sets and the Mandelbrot Set,” *The Beauty of Fractals*, pp. 161–174, 1986, doi: 10.1007/978-3-642-61717-1_13.

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

[14] V. da Silva, T. Novello, H. Lopes, and L. Velho, “Real-time Rendering of Complex Fractals,” in *Ray Tracing Gems II*, 2021, pp. 529–544. doi: https://doi.org/10.1007/978-1-4842-7185-8_33.

[15] D. White, “The Unravelling of the Real 3D Mandelbrot Fractal,” 2009. https://www.skytopia.com/project/fractal/mandelbulb.html (accessed Nov. 12, 2021).

[16] R. Englund, S. Seipel, and A. Hast, “Rendering Methods for 3D Fractals, Bachelor Thesis,” 2010.

[17] D. White, “The Unravelling of the Real 3D Mandelbrot Fractal page 2,” 2009. https://www.skytopia.com/project/fractal/2mandelbulb.html (accessed Nov. 20, 2021).

[18] D. Mankin, “True 3D mandelbrot type fractal, distance estimation optimisation,” 2009. http://www.fractalforums.com/3d-fractal-generation/true-3d-mandlebrot-type-fractal/msg8073/#msg8073 (accessed Nov. 20, 2021).

[19] M. Mcguire, E. Enderton, P. Shirley, and D. Luebke, “Real-time Stochastic Rasterization on Conventional GPU Architectures,” *Proceedings of the conference on high performance graphics*, pp. 173–182, 2010, Accessed: Nov. 20, 2021. [Online]. Available: http://research.nvidia.com

[20] J. Peddie, *Ray Tracing: A Tool for All*. 2019. doi: 10.1007/978-3-030-17490-3.

[21] 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).

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

[23] I. Quilez, “Distance Functions,” 2013. https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm (accessed Oct. 28, 2021).

[24] I. Quilez, “Distance to Fractals,” 2004. https://www.iquilezles.org/www/articles/distancefractals/distancefractals.htm (accessed Nov. 04, 2021).

[25] Boston, “What Is GPU Computing?” https://www.boston.co.uk/info/nvidia-kepler/what-is-gpu-computing.aspx (accessed Nov. 16, 2021).

[26] NVIDIA, “Programming Guide :: CUDA Toolkit Documentation.” https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed Nov. 17, 2021).

[27] The Khronos Group Inc, “OpenGL Overview.” https://www.khronos.org/opengl/ (accessed Nov. 17, 2021).

[28] D. Exterman, “CUDA vs OpenCL: Which to Use for GPU Programming,” 2021. https://www.incredibuild.com/blog/cuda-vs-opencl-which-to-use-for-gpu-programming (accessed Nov. 16, 2021).

[29] The Khronos Group Inc, “OpenCL Overview.” https://www.khronos.org/opencl/ (accessed Nov. 17, 2021).

[30] “Fragmentarium (original version, now unmaintained).” https://github.com/Syntopia/Fragmentarium (accessed Nov. 17, 2021).

[31] “Fragmentarium.” https://github.com/3Dickulus/FragM (accessed Nov. 17, 2021).

[32] “Mandelbulb3D.” https://github.com/thargor6/mb3d#Coloring (accessed Nov. 17, 2021).

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

[34] 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).

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

[36] Open Source Initiative, “The Open Source Definition,” 2007. https://opensource.org/osd (accessed Nov. 18, 2021).

[37] Open Source Initiative, “GNU General Public License version 3,” 2007. https://opensource.org/licenses/GPL-3.0 (accessed Nov. 18, 2021).

[38] British Computer Society, “Code of Conduct for BCS Members,” 2021. Accessed: Nov. 18, 2021. [Online]. Available: https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf

[39] TeamGantt, “Online Gantt Chart Maker.” https://www.teamgantt.com/ (accessed Nov. 18, 2021).

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