Real-time Rendering of 3D Fractal Geometry

Final Year Dissertation

21/04/2022

by

Solomon Baarda

Meng Software Engineering

Heriot-Watt University

Supervisor: 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: 21/04/2022

Abstract

A *fractal* is a pattern which remains detailed at any arbitrary scale. As such, fractals appear everywhere in the natural world from the structure of clouds to the roughness of mountains to the edges of our coastlines. All objects created by nature remain detailed when viewed at our own human scale and even at an atomic scale. Fractals occur in every scale, from the structure of our own DNA molecules to the shapes created by the formation of galaxies, and everything between.

While fractal patterns look aesthetically pleasing and have featured in popular pieces of artwork, they have also led to technical breakthroughs and have range of uses in industry, from modelling the growth of cancerous cells, to the design of efficient Wi-Fi and cell phone antennas, to describing losses and gains in the stock market. Some use cases require software capable of displaying a visual depiction of the fractal being modelled, such as in the discipline of fluid mechanics, when viewing complex turbulence flows or the structure of porous materials.

Most of the currently available fractal viewing applications are designed for outputting images or videos of high visual quality, using realistic lighting and shading techniques, while there are few applications which can display changes to 3D fractals in real time. This is likely due to the performance limitations of commercially available processors and graphical processors, which for many years were not powerful enough for computing images of 3D fractals in real-time. However, over the last couple years, the parallel degree of commercially available graphical processors has increased significantly and the Nvidia's current generation may be powerful enough to allow us to utilise parallelism techniques to achieve real-time rendering of 3D fractals.

This project aimed to develop an application which can update and render 3D fractal geometry in real-time, by utilising the parallel nature of graphical processors and by implementing known algorithmic optimisations for rendering 3D fractals efficiently. The performance of the application was analysed, the work completed was discussed and evaluated in respect to the original project aim, the significance and value of the project was evaluated, and future work for the project was discussed.

Acknowledgements

I'd like to take a moment to express my gratitude to those people without whom this project would not have been successful. Firstly, I'd like to thank my supervisor *Ben Kenwright* for his continued support over the duration of this project. His advice and expertise in this field helped guide me throughout the duration of my project. I'd also like to thank *Ali Muzaffar* for his valuable feedback on my initial research report, which encouraged me to complete more thorough research in several areas. Finally, I'd like to thank *Chris Hulme*, *Callum Scott*, and *Fraser Orr*, for allowing me to use their computers to gather results for my project, without which this work would be less significant.

Tables

Table of Contents

Table of Figures

Table of Tables

Table of Equations

Table of Graphs

Definitions and Abbreviations

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 pattern which remains detailed at any arbitrary scale. As such, fractals appear everywhere in the natural world from the structure of clouds to the roughness of mountains to the edges of our coastlines. Beniot Mandelbrot, inventor of the concept of fractal geometry, famously wrote "Clouds are not spheres, mountains are not cones, coastlines are not circles, bark is not smooth, nor does lightning travel in a straight line" [1]. Fractal geometry describes the more non-uniform shapes found in nature, which remain detailed when viewed at our own human scale and even at an atomic scale. Fractals appear in all scales, from the structure of our own DNA molecules to the shapes created by the formation of galaxies, and everything between.

While fractal patterns look aesthetically pleasing and have featured in popular pieces of artwork, they have also led to technical breakthroughs and have range of uses in industry [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. Fractal patterns are a key design choice in the design of efficient Wi-Fi and cell phone antennas as some of these patterns maximise the area of antenna that can transmit and receive signals. In computer science, fractal compression is an efficient method for compressing images and other files which uses the fractal characteristic that parts of a file will resemble other parts of the same file. Fractals have been used to visualise complex data sets and to describe losses and gains in the stock market. Some use cases require software capable of displaying a visual depiction of the fractal being modelled, such as in the discipline of fluid mechanics, when viewing complex turbulence flows or the structure of porous materials.

Most of the currently available fractal viewing applications are designed for outputting images or videos of high visual quality, using realistic lighting and shading techniques, while there are few applications which can display changes to 3D fractals in real time. This is likely due to

performance limitations of commercially available Central Processing Units (CPUs) and Graphics Processing Units (GPUs), which for many years were not powerful enough for computing images of 3D fractals in real-time. However, over the last couple years, the parallel degree of commercially available GPUs has increased significantly and the Nvidia's current generation may be powerful enough to allow us to utilise parallelism techniques to achieve real-time rendering of 3D fractals.

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 used common optical effects including the Blinn-Phong lighting and surface shading method, hard and soft shadows, and geometry 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 appendix 6.9 and the Gantt chart in appendix 6.11.3 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 6.1, and all project files can be found in the project GitHub repository [5]. 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 [2]. 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 [6]. 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.

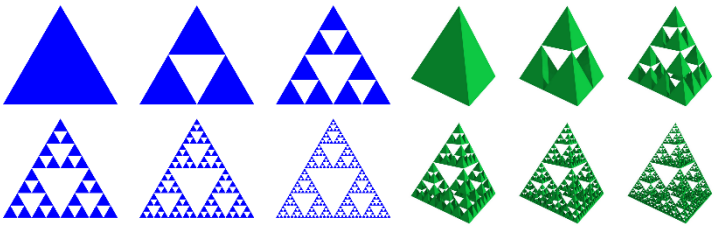


Figure 1.4.1 Sierpiński triangle (left) [7] *and tetrahedron (right)* [7] *both of recursive depth 5*

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.

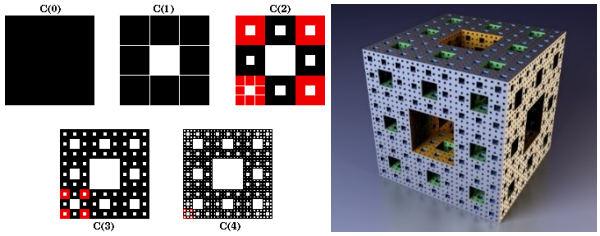


Figure 1.4.2 Sierpiński carpet (left) [8] *and Menger sponge (right)* [9] *both of recursive depth 4*

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 [10].

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) [11]. 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 [12], a few of which are shown below.

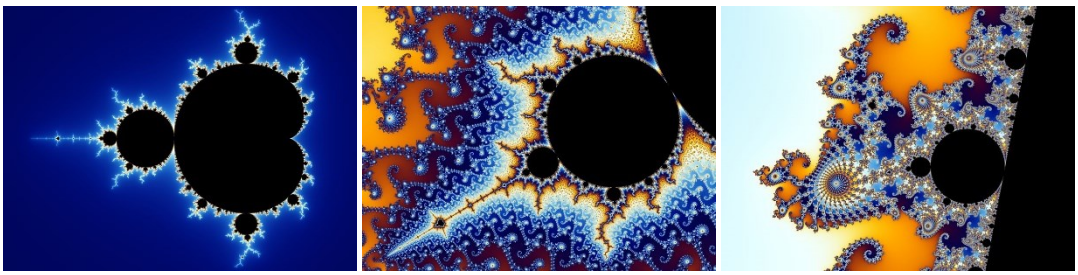


Figure 1.4.3 Mandelbrot set overview (left) [12]*, antenna (middle)* [12]*, and upside-down seahorse (right)* [12]

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 [13]. 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 [14]. 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 [14], [15] is given as:

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

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

Equation 1.4.i White and Nylander's formula for the nth power of a point in 3D space

And the addition of two points is given as:

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

Equation 1.4.ii Addition of two points in 3D space

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) [16]*, three (middle)* [16]*, and eight (right)* [16]

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 [17], 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 [18], 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 [19]. 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 [17] 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.

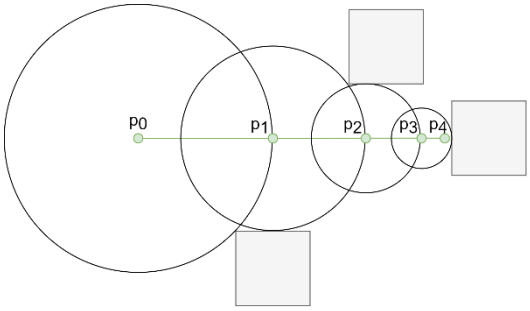


Figure 2.3.1 Ray marching diagram

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 [19]. 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 [20]. 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 [21]. 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 [20].

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 [22]. A full example of the SDF for a sphere is included in appendix 6.2. 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 [22]. 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.

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

Equation 2.3.i Equations for scaling a point down (left) and up (right)

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 [23], as given below.

$$union(a, b) = \min(a, b),\ subtraction(a, b) = max(-a, b),\ intersection(a, b) = max\ (a, b)$$
$$where\ a, b\ \in \mathbb{R}\ are\ the\ values\ returned\ from\ object\ a\ and\ b's\ SDF$$

Equation 2.3.ii Equations for union (left), subtraction (middle) and intersection (right) of two SDFs

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 6.3. The images below were rendered using an early prototype of the application.

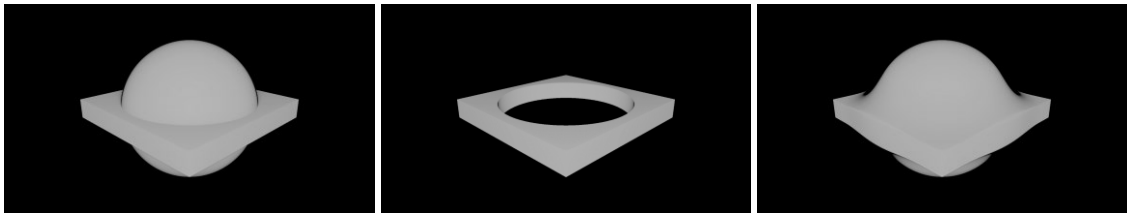


Figure 2.3.2 Ray marched sphere and box scene experiment union (left), subtraction (middle) and smooth union (right)

There are several additional alterations that can be applied to primitives once we have their SDF [22]. 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 [19] (see appendix 6.1 for an example). Lighting and surface shading functionality has been added as stretch goals in the requirements specification located in appendix 6.9. 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 6.4.

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 following 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

TODO

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 [24]. 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 [25]. 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 [26] 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 [25], [27], [28] 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

Flesh this out

Discuss evaluation techniques for these and for papers

Maybe compare my output to theirs?

Make this paragraphs and not bullet points

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.

Many fractal viewing applications already exist

There are many less 3D fractal viewers, and even less realtime fractal viewers

2.6.1 Fragmentarium

Fragmentarium [29], [30], originally released in 2011, was one of the first open-source programs capable of rendering 3D fractals. It has received many new features over the years and is still being updated to this day. The Fragmentarium interface contains three main panels: a GLSL (common shader language) IDE with syntax highlighting which defines the geometry in the scene, a main window displaying the visual output for the camera, and a panel containing a selection of sliders for controlling various parameters of the scene or visual output. This interface allows users to update parameters of the scene in real time and view the output. User can also manually edit and recompile the GLSL shader code for the scene and view its output.

Fragmentatium also contains many example scenes, and the rendering quality of the output image is good, but does vary between scenes.

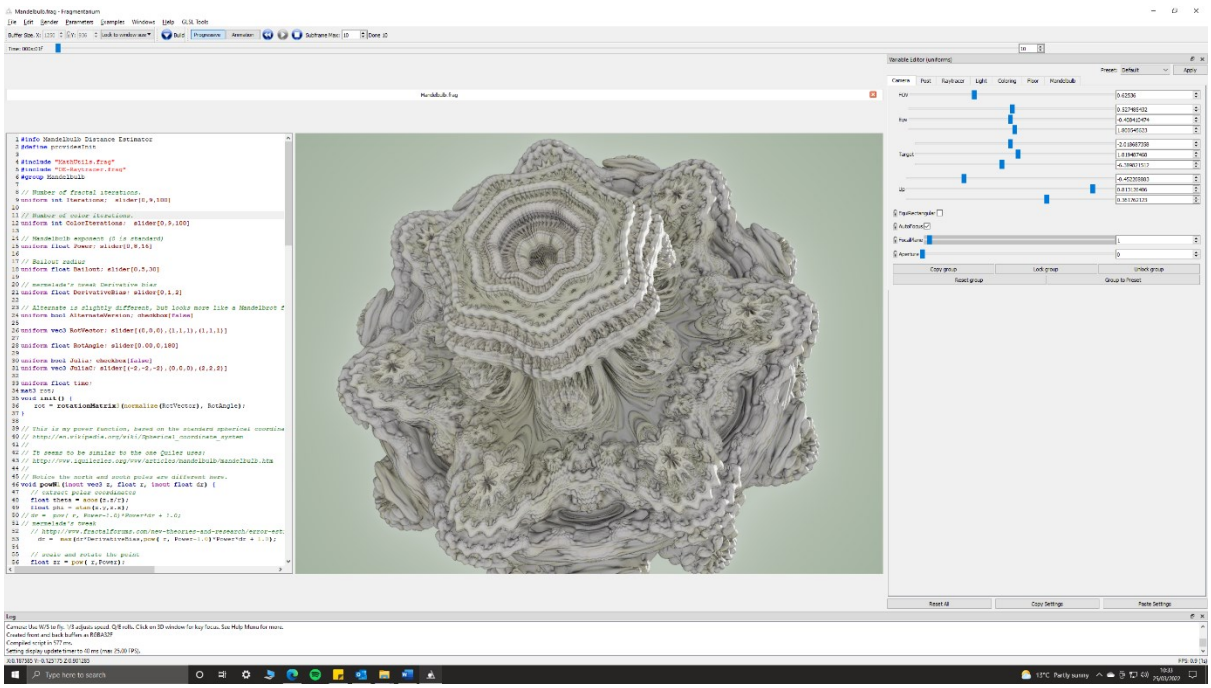
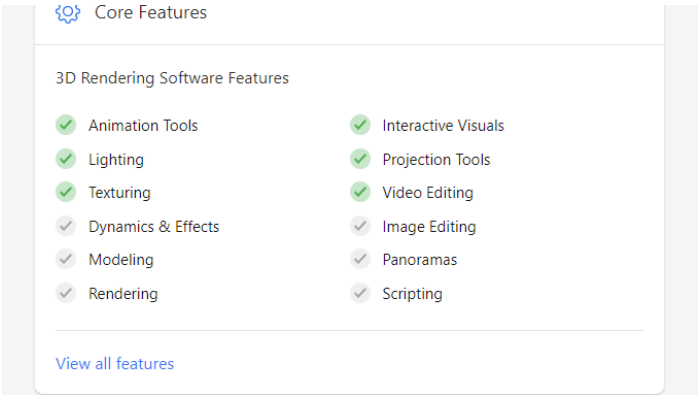


Figure 2.6.1

2.6.2 Mandelbulb3D

Mandelbulb3D [31] is a modern fractal rendering application, featuring a clean and stylish interface with.



Steep learning curve

- 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 6.6)

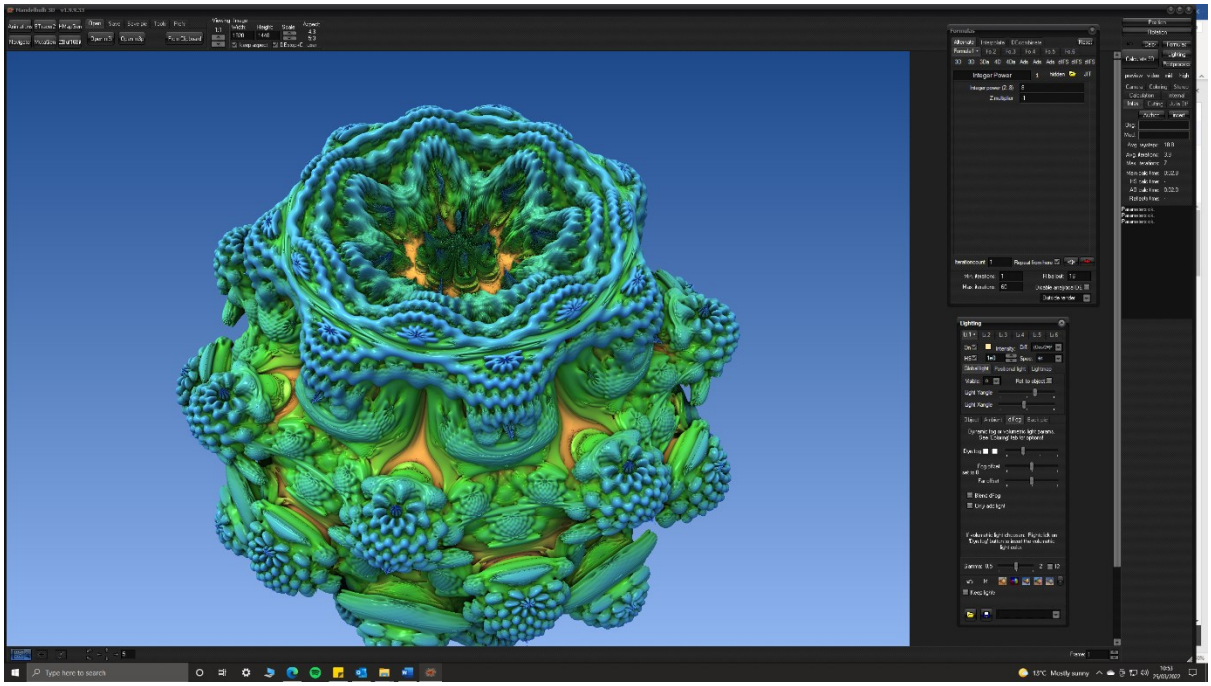


Figure 2.6.2

2.6.3 FractalLab

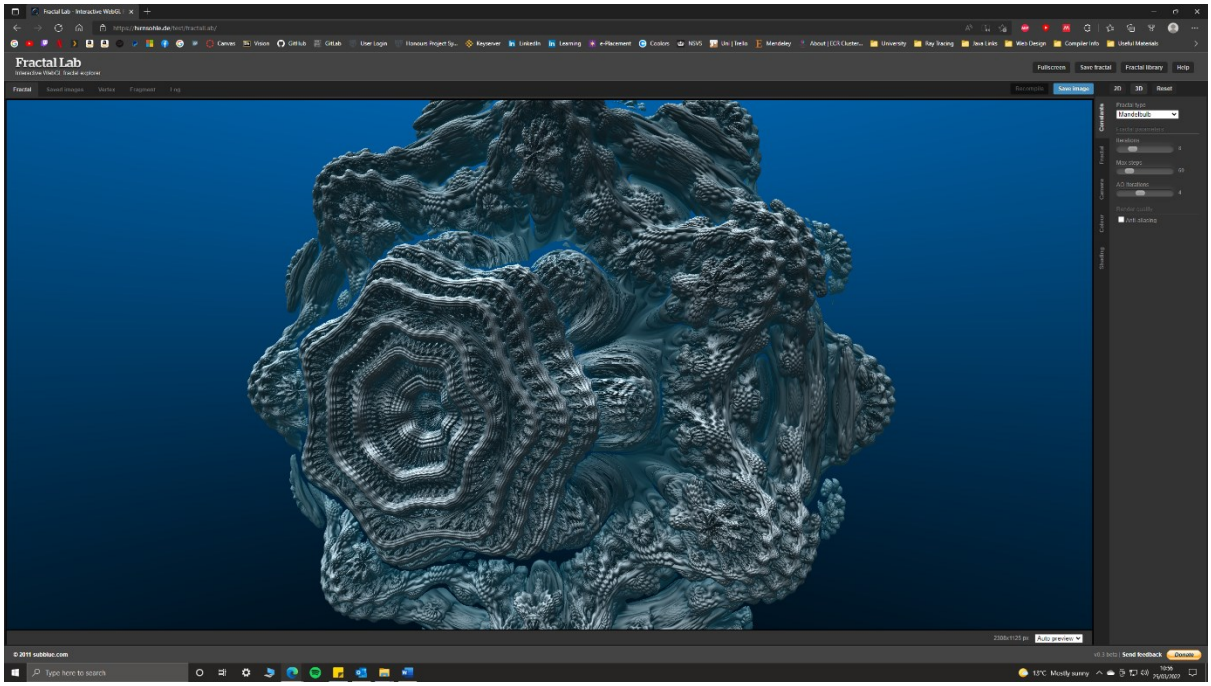


Figure 2.6.3

2.6.4 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

The following section describes the development environment and chosen design for the application, the technologies used, and development strategy employed to achieve the project aims and objectives. Additionally, the application documentation is referenced and future improvements to the application are discussed.

3.1 Development Environment

One of the most efficient ways of developing a C++ application on Windows is using Visual Studio [32], which provides powerful development tools such as debugging support, testing interface, refactoring options, code snippets, and IntelliSense, Microsoft's automatic code completion software. Visual Studio was chosen as the development IDE for these reasons, and Visual Studio 2019 was installed on a desktop Windows 10 machine to be used for development of the application. A GitHub repository was created for the project [5] early on to ensure all work carried out for the project was backed up using a version control system. GitHub Desktop was used as the version control interface as it is easy to use.

Development of the application was completed using a Windows 10 computer with a Nvidia RTX 3060 Ti graphics card, a mid-tier choice from the current new generation of Nvidia GPUs.

3.2 Application Design

3.2.1 High Level Overview

Figure 3.2.1 displays a high-level overview of how the `FractalGeometryRenderer` interacts with the CPU and GPU of the device it is currently running on. For the most part, the application is single threaded and runs on the CPU. When the application starts, the scene kernel file is loaded to the devices main memory, then it is passed to the OpenCL runtime system which compiles the scene and then loads it onto the devices GPU. Once this is done, the application enters a loop where it will instruct the kernel about *how* to render the scene, specifying information such as the resolution to output, the position of the camera, the direction that the camera is facing, and

the current time that the application has been running for. The kernel then computes the pixel data and writes it to a buffer in main memory, and the application reads that pixel data and updates the display accordingly.

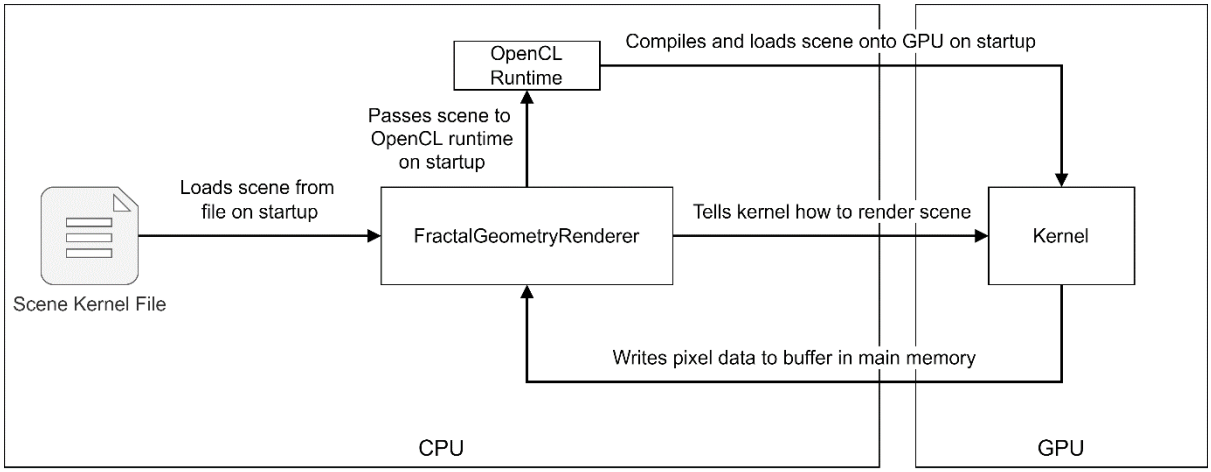


Figure 3.2.1 Application high level overview

Most of the time taken to render each frame comes from computing the RGB values for each pixel, a process which is executed on the GPU. During this time, the CPU is idle as it must wait for this process to complete before continuing. This means that the application is GPU bound, as the limiting factor for the application performance is the performance of the graphics card itself. This is important because is allows the specifications of the CPU to be ignored when benchmarking the application as they have little to no impact on performance, which allows a direct comparison of performance of performance of the application over various GPUs to be made. Additionally, as GPU performance increases, so will performance of the application.

3.2.2 Application Design Principles

The application has been developed following modern C++ practices [33]. This encourages the use of objects and emphasises the principle of *resource acquisition is initialisation* (RAII), which means that resources such as heap memory, file handles, and sockets should be *owned* by the object that creates that resource in its constructor and, therefore, is also responsible for deleting that resource in its destructor it once it is no longer used. The principle of RAII ensures that all resources are correctly returned to the operating system once an object goes out of scope. This is especially important for this application, which must allocate and deallocate memory buffers on

the GPU. There would be severe consequences for the device if GPU memory was not correctly deallocated.

The application has been structured using several core classes, which are loosely coupled and highly abstracted, which encourage the object-oriented principle of encapsulation. An object-oriented programming style was chosen to ensure the RAII principle was followed. Table 3.2.1 lists the core classes of the application and their responsibilities.

Table 3.2.1 Class responsibilities

Class name	Responsibilities
`FractalGeometryRenderer`	• Drives the application using a game loop • Contains instances of `Renderer` and `Window`
`Renderer`	• Calculating the pixel data for the current frame, and writing that data to main memory
`Window`	• Reading pixel data from main memory and updating the window accordingly • Reading window events and user input events

The colour format chosen by the application is RGBA8888, which means that each red, green, blue, and alpha channel of a colour uses eight bits, representing a value from 0 to 255. Theoretically, this allows for 4,294,967,296 different colours, but in practice only 16,777,216 of these colours will be used by the application as the alpha channel is always set to 255 to make the colour opaque. Leaving this channel unused is not ideal, but it is the simplest solution and many GPUs are optimised for rendering textures using this colour format.

While this format contains more than enough colours, the main benefits of this format are memory usage and performance. Each pixel requires one byte of memory for storage, meaning a resolution of 1920 by 1080 takes up only 2.1 MB of memory. In this application, the contents of the pixel data buffer is copied to another buffer so that the graphics library SDL2 can efficiently stream the data to a texture and display it on the window. If the device supports it, SDL2 will stream this pixel data very efficiently using the GPU. The pixel data must be duplicated in memory as OpenCL and SDL2 should not have access to the same memory location as this can cause

conflicts and unexpected behaviour as SDL2 may edit the pixel data when streaming it to the window. The duplication of pixel data is not ideal though it is the simplest solution.

3.2.3 Kernel Design Principles

The kernel code is structured into several key OpenCL C files. The file `main.cl` contains all the algorithmic ray marching logic and contains the main kernel method which the C++ side of the application interacts with. The file `types.cl` contains `struct` definitions for `Material`, `Light` and `Ray`. The file `defines.cl` contains all pre-processor definitions used by `main.cl` and is used for assigning default values if a definition is not specified in the scene.

Compiler directives are used throughout the file `main.cl` to enable and disable specific branches of code, depending on the settings defined in the scene. This means that features that are not wanted in the current scene aren't even compiled, which gives much better performance than making this check at runtime as these is no redundant code.

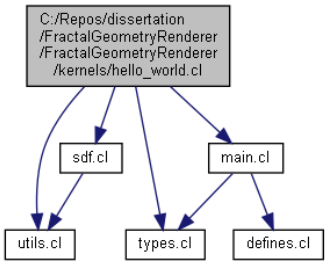


Figure 3.2.2 Dependency graph for example scene hello_world.cl

Each scene kernel file must implement several key methods which will be called by `main.cl`. These methods include:

Table 3.2.2 Key scene methods to implement

Method signature	Description
`float DE(float3 p, float t)`	The signed distance estimation between point p and geometry in the scene, sampled at time t
`Material getMaterial(float3 p, float t)`	The material used to render the geometry at point p and time t
`Light getLight(float t)`	Properties of the scene light at time t

`float boundingVolumeDE(float3 p, float t)`	The signed distance estimation between point p and bounding volume used by all geometry, sampled at time t Only used if `USE_BOUNDING_VOLUME` is defined as `true`

The design choice to separate the scene signed distance estimation function and the scene material sampling function has significant implications. By separating the two functions, it gives significantly better performance for scenes for which the material is not calculated as part of the distance estimation function. For example, the scene `mandelbrot.cl` which maps the 2D Mandelbrot set onto a 3D plane, benefits significantly from this design choice as the graphics rendered onto the plain can be calculated independently to the distance estimation to the plane. This means that the graphics are only calculated one time, when the ray has collided with the geometry. While this design choice gives significantly better performance for some scenes, it does provide slightly worse performance for others. For example, the scene `mandelbulb.cl` will perform slightly worse as one extra call will be made to the scene geometry function, since the material of this geometry is calculated as a by-product of calculating the distance to the geometry. In most cases, any negative performance is negligible as the computation cost of calling these functions is low, especially considering these functions are called hundreds of times for each pixel on the screen.

Another important design choice was the use of the OpenCL type `float3`, which contains three float values and is used to represent a vectors and colours. Behind the scenes, OpenCL uses a `float4` to store the `x`, `y`, and `z` values of the `float3`, and it initialises the fourth component `w` to zero. This means that for every `float3` type that is used in the kernel code, four floats are allocated, and one is not used. This is quite wasteful of memory, though the benefits of using this type are significant. Most GPU architectures use SIMD operations (Single Instruction Multiple Data) which makes use of the parallel nature of the GPU to combine multiplication or addition operations on float types into a single instruction. This means that the addition or multiplication

of two `float4` types is completed in one operation, instead of taking four operations. The performance benefits gained through using this type are significant, in our case giving around a 3x speedup assuming the GPU architecture supports this technique, which in most cases is worth wasting the memory of a 32-bit float type. If a GPU did not support this technique, then it would fallback to SISD operations (Single Instruction Single Data) meaning that addition or multiplication would be completed in four operations. This is very unlikely as most GPUs have supported the basic addition and multiplication SIMD operations for many years now, though it is worth noting as this could contribute to poor performance on older devices.

It was initially planned to use the new OpenCL C++ kernel language [34], which allows kernels to be written using many C++17 features, most notably allowing classes (instead of C structs) and method/operator overloading, as these features would have allowed the kernel code to follow similar principles to those used by the main application. However, support for this new language was made optional in OpenCL 3.0 and was not supported on the device used for development, so unfortunately it could not be used. Also, if this new language had been used, scenes written in this language might not have been backwards compatible with devices using older OpenCL runtimes.

3.3 Technologies

The application uses OpenCL, a parallel programming framework that allows code to be executed on the GPU. OpenCL has bindings in several languages, most notably C++ which is a high performant system language. CMake is a cross platform language designed for automating the build process for large scale applications. CMake provides a simple API for including files and linking libraries and works well with package managers on most platforms. In the Windows environment, the vcpkg package manager was used for installing development packages. Table 3.3.1 lists the packages used by the application. All packages chosen are cross platform, as this provides an abstraction layer over platform specific libraries, which allows the program implementation to remain decoupled from the deployment platform. The application has only

been built and tested on Windows x64, however, the use of CMake and cross platform libraries means that building the application on other architectures should be relatively simple.

SDL2 was chosen as the graphics library as it is low level, relatively lightweight, and gives good performance for uploading 8-bit colour values directly to the window. A more complex graphics library was not required as no GUI is needed by the application.

Table 3.3.1 Packages used by the application

Technology	Description	Justification
OpenCL	GPU parallel programming framework	• Standardised parallel programming framework for GPUs • Version 1.2 is the mandatory minimum for all graphics cards • Allows code to be deployed to any brand of GPU
SDL2	Graphics library	• Provides window manipulation, user input event polling, and high-performance timers • Has extensive documentation and examples
Eigen3	Maths library	• Provides useful vector and quaternion maths functions, used when calculating the camera view direction
GoogleTest	Unit testing framework	• Powerful unit testing framework • Comes bundled with the Visual Studio 2019 C++ development package

3.4 Development Strategy

An Agile [35] development strategy has been used throughout the duration of the project to ensure that work was completed on time and that the project aims and objectives set out were fully achieved. The Agile approach uses small sprints of work to complete specific and defined tasks, which allows teams to respond to change quickly as requirements and plans are updated. While this an individual project and the Agile approach is normally used in teams, it was beneficial in this project as it encouraged continuous reflection and improvement of the application features and design. To ensure that requirements were implemented in time, a Gantt chart was created which broke down development into eight key sprints, each two weeks in length. The highest priority requirements were implemented in the earlier sprints which left the less important functionality and stretch goals to be implemented later in development.

Development of the application began in sprint zero, which aimed to set up the project environment and refactor existing code that was created when experimenting with a simple ray

tracing program. This ray tracing code was refactored and translated to the OpenCL C kernel language so that it could be executed on the GPU. Additionally, SDL2 was set up which allows manipulation of windows so that the output of the program can be viewed by the user.

The first functionality of the application was implemented in sprint one, which aimed to add a game loop, basic benchmarking tools, and user input controls to move the camera around the scene at runtime. This ended up being very simple to implement as the progress made in sprint zero allowed these tasks to be completed easily. The dynamic scenes task from the next sprint was brought forward and completed in this sprint instead.

Sprint two aimed to add the first visual features of the application, including ambient occlusion, lighting, and shadows. Lighting and shadows were implemented relatively easily, as these features don't differ much from traditional ray tracing, however, there were problems finding a suitable ambient occlusion algorithm to implement the one chosen in the initial research report suffered from a significant flaw, which makes it unusable in some situations. This task was put on the back burner and materials were implemented instead using the Blinn-Phong surface shading method. Additionally, the benchmark scene was not defined during this sprint as was originally planned due to lack of time, so this task was pushed back to the next sprint.

Sprint three aimed to complete the final release of the application and to create a full user guide explaining how users should use the application and how developers could develop the application. It was during this sprint that progress began to significantly fall behind the project plan, as there were performance optimisations that had to be made before the application could be released, and there were tasks from the previous sprint still to implement. These optimisations were implemented along with the first draft of the user guide and several key fractals were also added as scenes. More research into ambient occlusion algorithms was completed, and it was decided to move this task to the stretch goals and to instead focus on defining the benchmark scenes and producing a stable version of the application.

Sprint four aimed to complete the first draft of the final report and record all results for the project. Since progress had fallen behind the original estimation these tasks were not completed and instead the benchmark scene was defined, and final build of the application created. This required lots of fine tuning of parameters in example scenes, ensuring that all features were accessible from the command line interface, and that there were no bugs in the final build.

In sprint five progress was made towards the first draft of the report and the application benchmarks were prepared and run on several different computers.

In sprint six the first draft of the report was completed, and in sprint seven changes were made to the draft based on feedback from readers and the final draft was completed.

Overall, an Agile development strategy worked well for this project as the continuous reflection on the project ensured that tasks were reprioritised when required and the project scope was adjusted if needed. The free time allocated in the original project plan was required as tasks were delayed and had to be pushed back. Several risks identified in the original risk analysis did occur, specifically *R-2 Change in Requirements* and *R-6 Delays due to Bugs*, but the negative impact of these risks was minimal thanks to effective mitigation plans. No unidentified risks occurred during the duration of the project.

3.5 Documentation

Extensive documentation for the application was created using Doxygen and is hosted using GitHub pages [36]. The *Home* page of the documentation contains system requirements, an installation guide and a basic user guide explaining how to use the application.

The *Manual Builds* page contains information explaining how to build the application manually on your own device should you wish to modify the application or build it on your own platform. Documentation of all classes used by the application is also included.

The *Scene Development Guide* page contains information explaining how one would create their own scenes for the application. This page contains links to the pages for some core kernel files,

which list all methods and compiler directives that can be used to modify the contents of the scene and quality of the render.

3.6 Interface Design

The application features a simple command line user interface. A full list of commands can be found in the *Home* documentation page.

A graphical based user interface was not chosen as it simply was not necessary. All the scene configuration information is located within the scene kernel file itself, the only parameters that need to be passed into the application at runtime describe the scene to be rendered, what resolution to render the scene at, and the mouse sensitivity for controlling the camera.

4 Evaluation

To accurately evaluate the successfulness of the project in respect to the original aims and objectives, several different evaluation strategies must be employed. First, the unit testing strategy will be discussed as this form of testing was vital during development in ensuring that the application behaviour was consistent with the expected behaviour. 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 in terms of performance, functionality, and scalability. The overall success of the project in terms of requirements implemented and aims and objectives achieved will then be evaluated using a goal-based evaluation strategy. Finally, possible future developments will be discussed.

4.1 Unit Testing

Unit testing was a small but vital part of ensuring that 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 ensured 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 whether 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 detected using automated tools. Instead, a goal-based evaluation strategy will employed to keep track of which requirements have been implemented and the corresponding objectives that each requirement falls under.

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 responsible for calculating the time taken to execute key pieces of code. The performance benchmarking tools were added to the application to enable comparisons to be made between the performance of each new feature and the value brought by each new feature, in terms of visual quality or functionality.

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 which 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 time by at most the duration of one frame, which varies between systems as this is dependent on system performance.

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	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	$\frac{benchmark\ duration\ (seconds)}{total\ number\ of\ frames\ rendered}$
Mean frames per second	Frames per second	$\frac{1}{mean\ frame\ time\ (seconds)}$
Maximum frames per second	Frames per second	$\frac{1}{minimum\ frame\ time\ (seconds)}$
Minimum frames per second	Frames per second	$\frac{1}{maximum\ frame\ time\ (seconds)}$
Upper difference	Percent	$\frac{maximum\ FPS - mean\ FPS}{\frac{maximum\ FPS + mean\ FPS}{2}}$
Lower difference	Percent	$\frac{mean\ FPS - min\ FPS}{\frac{min\ FPS + mean\ FPS}{2}}$

4.3 Fractals Implemented

The application features several scenes containing 3D fractal geometry, including the Sierpiński cube and tetrahedron fractals, and the Mandelbulb fractal. Two core 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 which are calculated by distorting the space at which the ray position is sampled from. The application also contains several trivial scenes such as `hello_world.cl`, `trivial.cl` and `sphere_box.cl`, which are all useful as examples for helping new users. The scenes `planet.cl` and `trivial.cl` have also been benchmarked.

4.4 Evaluation of Application

4.4.1 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.

4.4.1.1 Linear Epsilon

The first optimisation, called the linear epsilon optimisation, involves linearly increasing the epsilon value used for ray intersections as the ray gets further away from the camera. This means that the further a ray gets from the camera, the less precise the collision between the ray and geometry needs to be. This optimisation has a nice side effect of reducing the amount of noise generated on geometry far away, as distant geometry becomes less detailed. This makes scenes feel more natural and realistic, while also increasing performance significantly. The rate at which the epsilon value linearly increases can be controlled using the `LINEAR_INTERSECTION_EPSILON_MULTIPLIER` compiler directive which is useful for fine tuning the performance boost with respect to visual quality. This optimisation performs best when the camera is far away from geometry and has little effect on anything close.

4.4.1.2 Bounding Volumes

The second key optimisation involved adding bounding volumes around geometry in the scene. A bounding volume is any shape which contains the main geometry inside of its volume. This can be used when ray tracing or ray marching as a collision with geometry only needs to be calculated if the ray collides with the bounding volume first. Since the collision function for a primitive sphere or box is considerably cheaper to compute than that of the Mandelbulb fractal for example, this can result in a significant performance boost as it saves unnecessary computation when the camera is far away from the geometry or facing in a different direction.

Bounding volumes only work well when the bounding volume function is simple and cheap to compute, while also being as close to the original geometry shape as possible. This means that

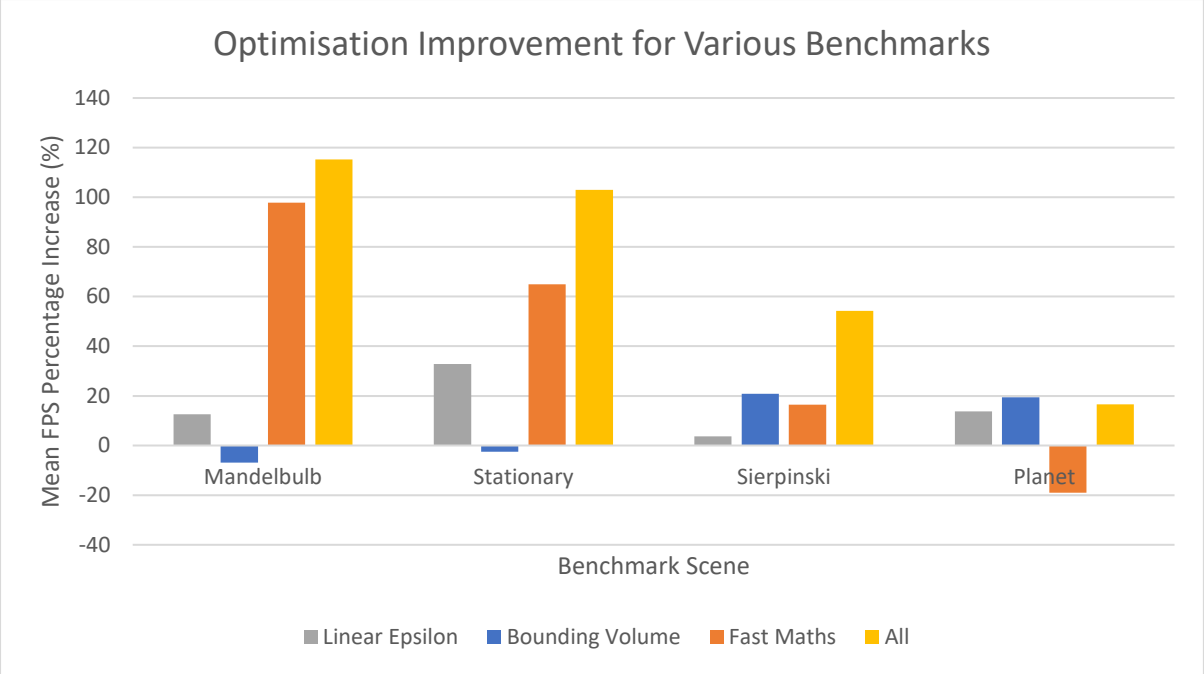
bounding volumes are less suitable for geometry which contains lots of empty space or does not fit easily within a primitive shape. A single geometry could be subdivided into multiple bounding volumes, but this would add more complexity and computation cost and may not be worth it depending on the scene. Each scene must implement bounding volumes into a separate distance estimation function which the application uses when it doesn't require a precise distance estimation. Since bounding volumes may not be required for all scenes, this optimisation is disabled by default but can be enabled using the directive `USE_BOUNDING_VOLUME`. Some scenes may not see any performance boost when using bounding volumes or may even see worse performance if the bounding volume is too complex.

4.4.1.3 Fast Maths Build Flag

The final key optimisation results from using the `-cl-fast-relaxed-math` OpenCL build flag when compiling 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 $a \times b + c$) [37]. These optimisations provide a trade-off between performance and result correctness and subsequently violate several IEEE 754 conventions, which define floating point arithmetic. This optimisation only effects the compiled OpenCL kernel code, not the C++ side, meaning that only the visual output might be 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 6.8. No noticeable visual difference can be seen when comparing the two images so this optimisation is enabled by default but can be disabled using the application command line argument `--force-high-precision`.

4.4.1.4 Evaluation

The performance of the three optimisations discussed above were benchmarked using the scenes defined in section 4.2 and the results are displayed in Graph 4.4.1. Each benchmark was run three times and an average of the raw data was taken.



Graph 4.4.1 Performance of optimisations in the Mandelbulb scene

The linear epsilon optimisation resulted in between a 3% and 32% increase in mean frames per second when compared to the base version. This optimisation behaves similarly to the bounding volume optimisation in the sense that a significant performance boost is only seen when the camera is further away from the geometry, though in this case it performed significantly better than the bounding volume optimisation as the computational cost of using this optimisation is trivial, and so it has no negative performance effects when in suboptimal conditions.

Interestingly, the Mandelbulb scene performed worse when using the bounding volume optimisation. This is likely due to a combination of the camera being close to the geometry and not getting the benefit of using a bounding volume, and because the bounding volume for the Mandelbulb is substantially larger than the geometry itself, to ensure that all the fractal is within the bounding volume since it changes shape over time. The efficiency of a bounding volume optimisation is highly dependent on the scene and camera position, and in this case, it didn't provide a performance increase. However, if the benchmark had spent more time further away from the geometry, then the performance could be significantly better. The bounding volume optimisation performed better when viewing the Mandelbulb from a distance in the Stationary

scene, though performance of this optimisation was still worse than not using it at all. For the Sierpinski and planet scenes, the bounding volume optimisation performed significantly better.

The fast maths optimisation performed significantly better for the Mandelbulb and stationary Mandelbulb scenes and performed acceptably for the Sierpinski scene. However, this optimisation performed significantly worse for the Planet benchmark scene. This is due to this scene including a C header file?

In all cases, the use of all optimisations at the same time resulted in a greater FPS increase than the sum of their component parts. This is due to the synergistic nature of the linear epsilon optimisation and bounding volume optimisation, as when combined this reduces the accuracy of rays that are far away from the camera but interacting with a bounding volume. Additionally, the use of the fast maths optimisation results in better performance in most cases and when combined with the other two optimisations, this results in even better performance as the fast maths can optimise the computations performed by the other optimisations.

4.4.2 Unmeasurable Features

There are several features of note provided by the application which aren't measurable but are still significant reasons to use the application. 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 rendered on the screen can be rendered using a different material, allowing geometries to blend between colours seamlessly. This gives a massive amount of flexibility and is incredibly powerful as it allows geometry to behave more like real world objects, which contain impurities and aren't made from one solid material. This feature is not common among rasterization rendering, which requires that geometry be grouped using the material by which it should be rendered.

The second feature of note is that there is no performance cost for moving geometries or lights around the scene. All lighting and pixel data calculated by the application is computed in real time and is computed every frame, even if the camera, light, or geometry hasn't moved. This does leave

room for future improvements, but real time lighting is a benefit of using ray tracing over rasterization rendering, which commonly "bakes" lighting data into a texture for use later. This makes lighting calculations very cheap for rasterization rendering as all data is precomputed, but also means that it is difficult for lighting to be updated in real time. Ray tracing completely negates this problem by making all lighting update in real time, which requires more computation but gives more realistic results.

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

4.4.3 Measurable Features

This section evaluates the features of the application that can measured, by calculating their performance cost and comparing this against the visual benefits of that feature.

4.4.3.1 Surface Shading

The Blinn-Phong surface shading model is a physically based rendering method which combines ambient colour, diffuse colour and specular highlights to efficiency approximate how real-world materials behave under light. This is a common surface shading model that is computationally cheap while giving a good visual result.

4.4.3.2 Glow

A glow can be applied to the space outside of an object. This is calculated for each pixel by keeping track of the minimum distance to that this ray was from the scene geometry, and if this ray didn't collide with geometry, then the glow value should be calculated using $\frac{minimum\ distance\ to\ geometry}{maximum\ glow\ distance}$.

4.4.3.3 Shadows

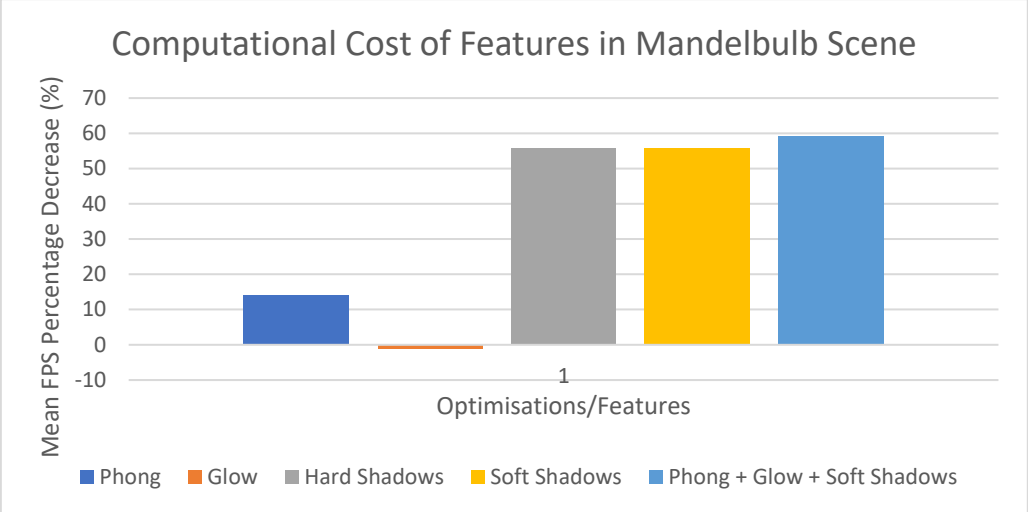
Both hard shadows and soft shadows have been implemented and either can be added to a scene. To calculate if a point on a geometry is in shadow, a ray must be cast from the point towards the main light in the scene, and if the ray collides with any other geometry, then the original point should be in shadow. Hard shadows mean that a point is either in shadow or not, which makes the shadows have very harsh edges and gives a stylised, cartoon look. Soft shadows use a falloff

value to determine how much distance should be used to blend between shadow and no shadow, which gives much more realistic shadows at the cost of more computation.

To generate accurate shadows, the main scene distance estimation function is called for each position on the ray, however, this could be optimised to use the bounding volume distance estimation. This was not implemented as incorrect shadows could be introduced accidently by allowing the bounding volume to cast shadows on geometry.

4.4.3.4 Evaluation

The performance of the four features discussed above were benchmarked using the Mandelbulb benchmark scene, where scene was run three times and an average of the raw data was taken. The results are displayed in Graph 4.4.2.



Graph 4.4.2 Computational Cost of Features in Mandelbulb Scene

The Blinn-Phong feature saw a performance decrease of only 14.1%, which in most cases is well worth it considering how substantial the visual payoff is. Figure 4.4.1 compares basic ambient shading to the Blinn-Phong surface shading model, note the darker areas where no diffuse light reaches and the specular highlights created by the light source.

Figure 4.4.1 Basic shading (left) and Blinn-Phong shading (right)

The glow feature resulted in a 1.1% performance increase, which results from inconsistencies when running the benchmark such as background tasks running on the machine. The computation cost of this feature is negligible and is essentially free for the ray marching algorithm and can produce a nice in some scenes. While this feature is very cheap, it doesn't look good in every scene which is why it is disabled by default. Figure 4.4.2 displays the geometry glow effect on the Mandelbulb scene.

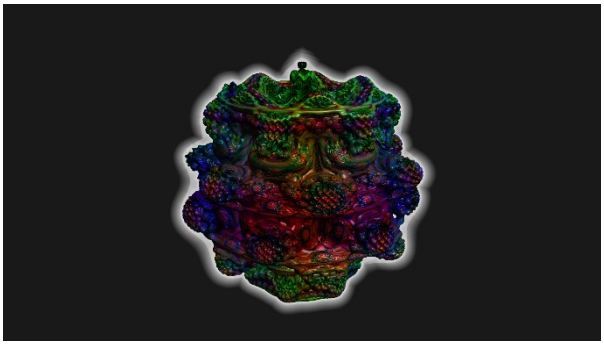


Figure 4.4.2 Geometry glow applied to Mandelbulb

The use of hard shadows decreased the average frames per second by 55.6%, which is a substantial performance loss. This performance decrease comes from the additional ray that must be cast for each pixel of a geometry. The performance of this feature is affected by both the computational cost of the scene distance estimation function, and by the distance between the geometry surface and the scene light. If the light is far away from the geometry, then it will take

more iterations of calling the distance estimation function for the ray to reach the light, and therefore will take more computation time.

Soft shadows performed similarly to hard shadows, resulting in a 55.6% mean FPS decrease, the same as hard shadows. The computation required to calculate soft shadows is very similar to hard shadows but does require several more operations and temporary variables. It would be expected for soft shadows to perform slightly worse than hard shadows, but in this case inconsistencies between runs makes up this difference. Figure 4.4.3 compares hard shadows and soft shadows, note the smoother blending between shadow and light for soft shadows.

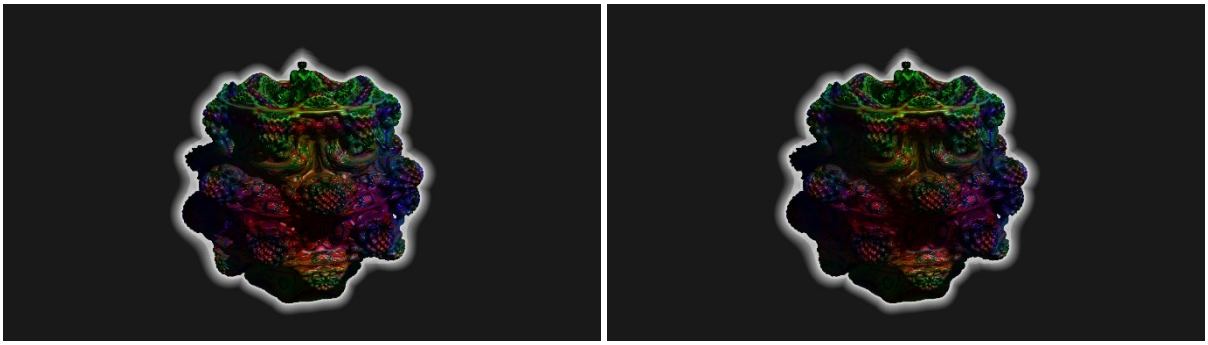


Figure 4.4.3 Hard shadows (left) and soft shadows (right)

Interestingly, when combining Blinn-Phong, soft shadows, and glow, the performance decrease is significantly less than the sum of their component parts, which is an unexpected result. It is expected for optimisations to perform better when combined due to their synergistic nature but combining features should not have the same effect. Well, WHY DOES IT HAPPEN THEN?

While the visual output of the application is aesthetically pleasing, it doesn't look anywhere near as good as an offline render of a fractal as many compromises have been made to ensure that the application can render the image in real time. However, in the future, this may not always be the case as Moore's law observes that the number of transistors in a dense integrated circuit (such as a CPU or GPU) doubles approximately every two years, and therefore performance also approximately doubles. While the upper limit for the number of transistors in CPUs and GPUs has likely been reached, the relationship of doubling performance roughly every two years is almost

still being achieved, through increasing the parallel degree of chips allowing more operations to be completed per second [38]. If this trend keeps up, higher visual quality real time rendering will be achievable in the next couple years, though offline renders will always create a superior output.

4.4.4 Application Scalability

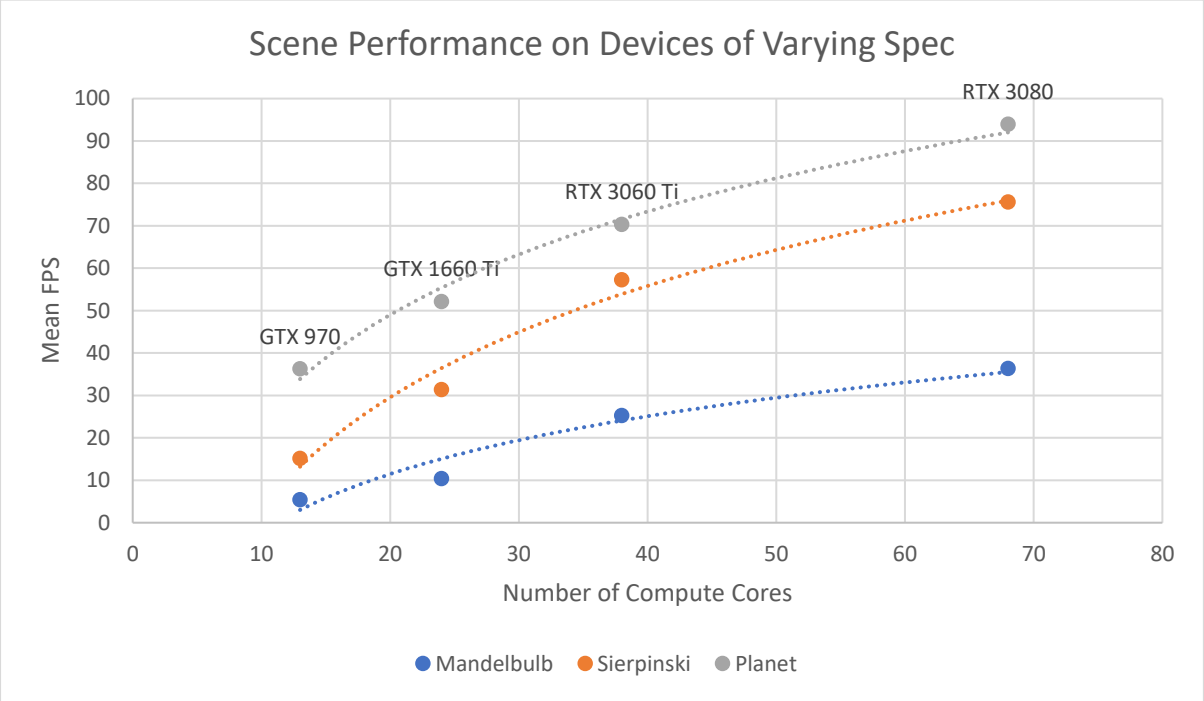
Relate these results to existing results from literature

The benchmark scenes were profiled across several different systems to determine how well the application scaled over systems of varying performance and age. Table 4.4.1 lists the graphics cards used by the benchmarking system and the age of their respective architectures. The devices tested against contains variety of current generation and older generation Nvidia GPUs. Unfortunately, no devices containing AMD or Intel graphics were available to benchmark the application on. Windows was the only operating system that the application was tested on. Windows 11 is still in early access and is not yet fully supported, so performance on this platform may not be fully accurate.

Table 4.4.1 Graphics cards used by the benchmarking systems

Graphics Card	Date Released	Price Range	Operating System
NVIDIA GeForce GTX 970	2013/2014	Mid-tier	Windows 10
NVIDIA GeForce GTX 1660 Ti	2018/2019	Mid-tier	Windows 11
NVIDIA GeForce RTX 3060 Ti	2020 (current generation)	Mid-tier	Windows 10
NVIDIA GeForce RTX 3080	2020 (current generation)	High-end	Windows 11

The core scenes were benchmarked on the systems listed in Table 4.4.1 and results plotted in Graph 4.4.3. The axis “Number of Compute Cores” refers to the total number of work groups that each graphics card can execute simultaneously. This is the parallel degree of the graphics card, which has increased substantially over the last decade.

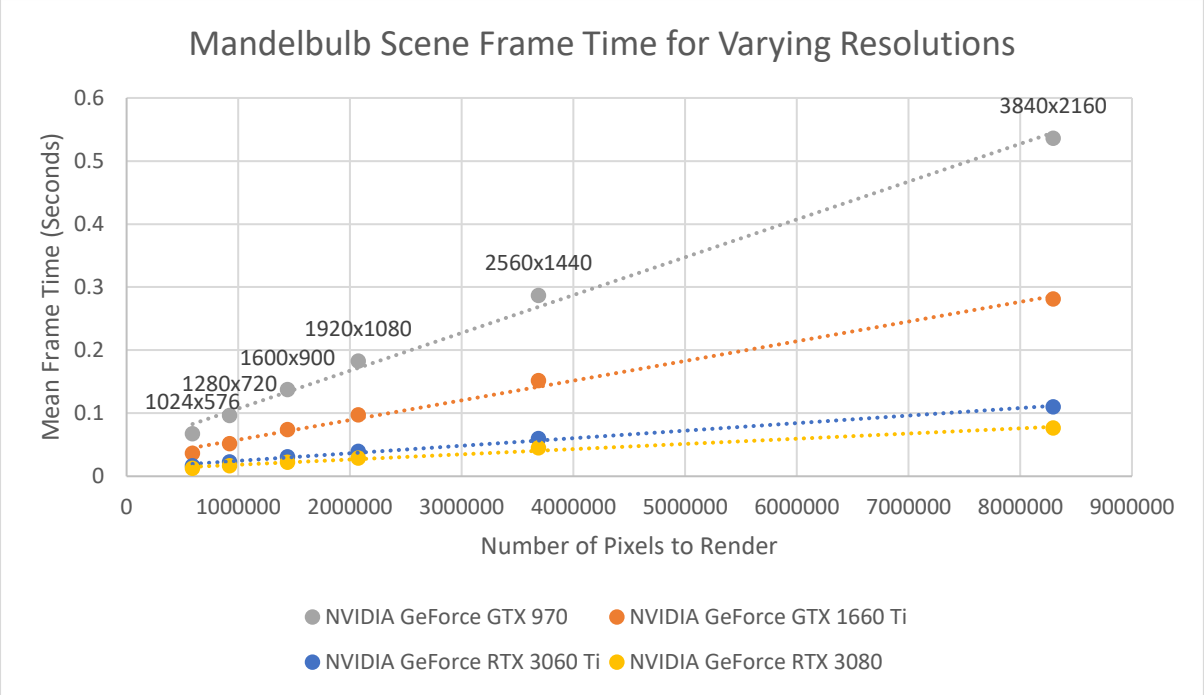


Graph 4.4.3 Scene Performance on Devices of Varying Spec

The mean frames per second increases logarithmically as the number of compute cores increases linearly. This result is expected as per Amdahl’s law, which states that all performance improvements are limited by the amount of time that the improved section is spent in execution. This means that because it will always take at least 3ms per frame to draw pixels onto a window for a resolution of 1920x1080, the total computation time for one frame can never fall below 3ms. This is a hard limit for performance and is the theoretical best performance achievable.

To determine the point at which this limit is reached, further benchmarking would be required on much higher power systems. Benchmarking of the application using a high performance system such as the Heriot-Watt Robotarium cluster [39] or Google Colaboratory [40] was considered, but was decided to be out of scope for this project and instead is discussed in section 5.3 Future Work.

The performance of the application over various resolutions was also benchmarked, and results displayed in Graph 4.4.4.



Graph 4.4.4 Mandelbulb Scene Frame Time for Varying Resolutions

As expected, the average time taken to render a frame increased linearly as the number of pixels in the frame also increases linearly. WHY DOES IT NOT INCREASE 2x pixels = 2x slower? This result is important as it means upscaling is a valid optimisation improvement that can be made. The resolution can easily be halved and image upscaled back to the original resolution resulting in significantly better performance, at the cost of reduced detail. In most cases this would work very well as ray tracing creates very detailed images which would upscale well.

To summarise, application scales pretty well

4.5 Evaluation of Requirements Specification

The full project requirement specification can be found in appendix 6.9. This specification only received small changes since the initial research report as thorough project planning ensured that all requirements were realistically achievable and within a reasonable scope. 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

All thirteen of the MUST priority requirements have been fully implemented. In addition, the one SHOULD priority requirement was also implemented. Of the five COULD priority requirements, which were included as stretch goals to be implemented if very good progress was made with the project, two were fully implemented, one was partially implemented and two were not implemented.

In terms of requirements implemented, this project was definitely a success as 100% of the MUST and SHOULD requirements were implemented, and 40% of the optional COULD requirements were implemented.

4.6 Evaluation of Aims and Objectives

A goal-based evaluation strategy has been used to determine if the projects aim and objectives have been achieved. 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.

The purpose of objective one was to complete background research into the project area to gain a better understanding of the chosen topic and the scope of the project. This objective was mostly completed when creating the initial research report, though it did run for the entire duration of the project to ensure that the project stayed up to date with current developments.

The purpose of objective two was to investigate existing solutions and to analyse their strengths and flaws. This objective was also mostly completed when creating the initial research report, though popular fractal communities were monitored so that any new fractal rendering software

could be investigated. No new fractal rendering software of note was released during the duration of this project.

The purpose of objective three was to implement the core functionality of a non-real-time 3D fractal renderer, which formed the safe core of the project. This objective was linked directly to the MUST priority requirements FR-1.1, FR-1.2, FR-1.3, FR-1.4 and FR-1.5, all of which were fully implemented. Therefore, objective three has been achieved as all requirements associated with the objective have been implemented.

The purpose of objective four was to add additional functionality to the renderer making it capable of real-time rendering by adding a game-loop, adding a controllable camera, making scenes dynamic, and adding optical effects and lighting. The scope of this objective was left flexible so that it could be increased or decreased as necessary, and several stretch goals were included in the requirements specification 6.9. The requirements associated with this objective were FR-2.1, FR-2.2, FR-2.3 and FR-2.4, all of which were fully MUST priority and fully implemented. Additionally, two of the COULD priority requirements representing the project stretch goals were fully implemented, FR-2.7 and FR-2.8, though the remaining three stretch goals FR-2.5, FR-2.6, and FR-2.9 were not fully implemented, though some progress had been made towards one of them. Since all MUST priority requirements associated with this objective were fully implemented, and good progress was made implementing additional stretch goals, this objective been achieved to a large degree.

The purpose of objective five was to benchmark the application performance across various systems. This objective was tied to the MUST priority requirement FR-3.1 which was achieved and is evaluated in section 4.4.

The purpose of objective five was to create documentation to aid users and developers of the application. This objective was not tied to any requirements but was still achieved and is located on the online documentation page [36].

The project objectives were created to help guide the project in the correct direction by spitting up the work to be completed into several key chunks. These objectives were definitely achieved.

The project aim was to develop an application which can update and render 3D fractal geometry in real-time. This was, without a doubt, achieved. Additionally, the aim specified that the render should use common optical effects including the Blinn-Phong surface shading method, lighting, hard and soft shadows, and geometry glow, which as discussed in section 4.4.3 have been fully implemented and evaluated. Finally, the aim also specified that it should be possible to create scenes and view them using the application, and this process should be as straight forward as possible. This part of the aim is more subjective than the previously discussed parts, as some users will find it more straight forward to create and view a scene than others, though for the most part, this process is as simple as possible for the current setup. In the future it could be worth completing a usability study to determine how simple most users thought this process was.

In summary, all six of the project objectives have been completed and several stretch goals were also completed. The project aim has been achieved and additional constraints mostly met, though in the future a usability study of the user interface would be valuable. Overall, the project has certainly achieved the aims and objectives that it set out to achieve.

4.7 Future Work

4.7.1 Features

There are many visual features that could be added to the application. Additional surface shading methods like Lambert, Oren-Nayar, or a subsurface scattering algorithm could be implemented. Additionally, ambient occlusion would be a valuable addition. The application would benefit visually from the addition of transparency and reflections, however, these features are costly as they both require additional rays to be cast. It is likely both features would have a performance cost similar to shadows, which resulted in approximately a 50% drop in mean frames per second. For any of these visual features to be viable, the application performance would first have to be significantly improved.

Gradient background

A graphical user interface could also be valuable for the application. Currently, a GUI is not needed as there are not any features that would make use of it, however, if more features are added in the future then a GUI may be considered.

4.7.2 Algorithmic Optimisations

There are several small optimisations that would result in a moderate performance boost, such as optimising the shadow ray cast to use the geometry bounding volume, and to use an alternate method for calculating the surface normal of a geometry, which uses four calls to the scene distance estimation function instead of six [41]. These optimisations would be easy to implement.

A more substantial optimisation would be to make use of an acceleration structure for storing the geometry in the scene. An acceleration structure is a data structure (usually some kind of tree) which stores all objects in a scene. This means that when calculating ray intersections, the tree can be iterated over to find the collision, instead of all objects in the scene. In the best case, this decreases the complexity of searching for the object to $O(\log N)$, down from $O(N)$. However, in practice it is hard to actually reach this theoretical performance as the scene must be partitioned efficiently. Additionally, a performance gain is only seen when the scene contains many objects, which our application does not. Finally, the OpenCL C kernel language would be unpractical for implementing this optimisation as it does not feature classes and work items cannot access a shared state. This optimisation may be beneficial in some cases but would require rewriting most of the application code.

4.7.3 Design Optimisations

An important optimisation to make would be to reduce the duplication of pixel data as this is an unnecessary waste use of memory. Additionally, it could be worth changing colour format as the alpha channel is completely unused by the application.

It may also be worth investigating other graphics libraries and finding one that would allow the pixels to be displayed asynchronously using the CPU, which could be completed while the

application is using the GPU to compute the next frame. This would result in quite a significant performance gain as it takes approximately 3 ms to render a 1920×1080 frame to the window using the development device.

Additionally, some of the sequential code in the application could be replaced with concurrency, such as reading the user input in another thread. This would require a significant rework as threads would now be required to synchronise before computing a frame, and the performance gain would be minimal, but possibly worth it as it would reduce the amount of sequential code in the application, therefore increasing the theoretical maximum FPS as per Amdahl's law.

Finally, it could be worth implementing upscaling in the application which would give a significant performance gain as the application could render a lower quality image, and then upscale it to the desired resolution.

4.7.4 Fractals

There are many more fractals that could be implemented as scenes. Including fractals within the Julia bulb family, kaleidoscopic fractal family, and IFS (Iterated Function System) fractals.

4.7.5 Evaluation

More sophisticated evaluation of the application could be performed. This could include performance benchmarks on more commercial devices, or even on high-performance systems such as the Heriot-Watt Robotarium cluster [39] or Google Colaboratory [40]. Additionally, it could be worth benchmarking the application using some ray tracing hardware, though the application renderer might have to be rewritten using a different language to make use of platform specific API features, such as the CUDA ray tracing API. This would likely result in significantly better performance but would require thorough research and planning.

5 Conclusion

5.1 Achievements

The aim of this project was to develop an application which can update and render 3D fractal geometry in real-time. This has been achieved without a doubt, as the application created can render the Sierpiński cube, tetrahedron, or the Mandelbulb fractal in real time, running at a resolution of 1920×1080. The render uses the Blinn-Phong lighting and surface shading technique, hard or soft shadows, and geometry glow. Additionally, users can create their own scenes using the library functions provided by the application, and scenes can be viewed in the application by passing the scene path as a command line argument.

All six of the project objectives have been achieved. Objectives one and two were met in section 2, which evaluated relevant background literature surrounding the project area, accompanied by a critical analysis of existing solutions. Objectives three and four specify the functionality that the application should provide in terms of software requirements, which are evaluated in section 4.5. Objective five specifies the evaluation to be performed on the application and project, which is completed in section 4. Finally, objective six specifies the documentation to be created to aid users and developers, which is located on the online documentation page [36].

All the MUST and SHOULD priority requirements specified have been met, and approximately half of the COULD priority stretch goals have also been implemented. In addition, all the non-functional constraints have been met and validated using unit tests where applicable.

5.2 Limitations

At its core, the application is only a basic implementation, providing limited functionality. Most currently available fractal viewing applications provide more visual features than just surface shading, shadows, and glow, and contain many more example scenes. Additionally, the application was only tested using several fractals - the Mandelbulb and Sierpiński cube and tetrahedron, and it was only tested on Windows 10 and 11 using Nvidia GPUs.

5.3 Future Work

A thorough discussion of possible future improvements and features for the application is included in section 4.7, which covers five main categories, features, algorithmic optimisations, design optimisations, fractals, and evaluation.

There are many visual features that could benefit the application, such as additional surface shading techniques, ambient occlusion, transparency, and reflections. However, these features are costly, and the application would need to be optimised further before these could be implemented. Algorithmically, there are several simple optimisations that could be made, such as modifying the shadow calculation to use bounding volumes and to use a cheaper alternative method for calculating the geometry surface normal. The application design could also be improved by removing the duplication of pixel data in main memory and by using an alternative library for updating the display window asynchronously. Additionally, some sequential code could be made concurrent and image upscaling could be considered. There are many more fractals that could also be implemented, and more sophisticated evaluation of the application could be completed using different hardware and operating environments.

6 Appendices

6.1 Experimentation Renders

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

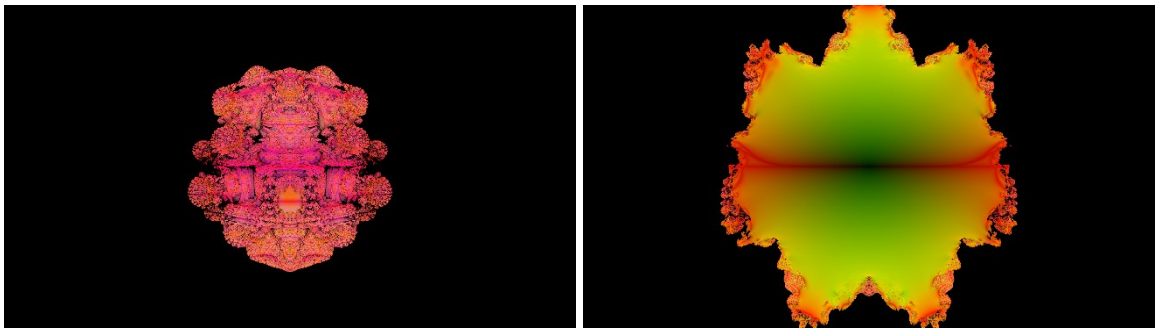


Figure 6.1.1 Render of the Mandel bulb fractal (left) and cross section (right) using equation from [42]

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.

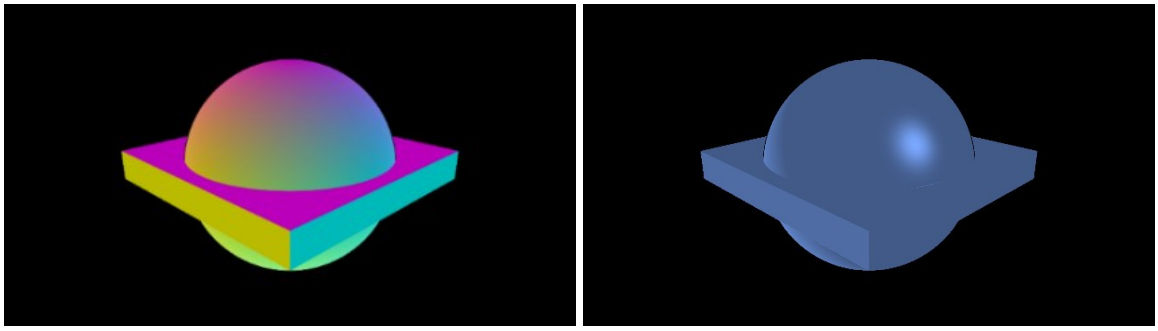


Figure 6.1.2 Surface normal visualised (left) and phong shading experiment (right)

6.2 Sphere SDF Example

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

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

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

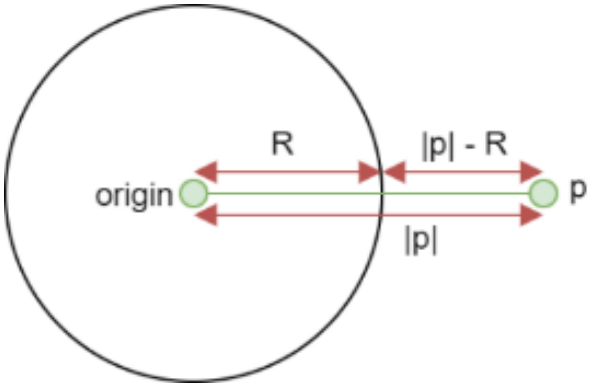


Figure 6.2.1 Sphere SDF diagram

6.3 Smooth SDF Combinations

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

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

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

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

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

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

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

6.4 SDF Surface Normal Calculations

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

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

6.5 Fragmentarium Renders

The following screenshots were taken when experimenting with Fragmentarium [29], [30].

Figure 6.5.1 A recursive scene (left) and Mandel bulb (right)

Figure 6.5.2 Tree fractal

6.6 Mandelbulb3D Renders

The screenshot below was rendered using MandelBulb3D [31].

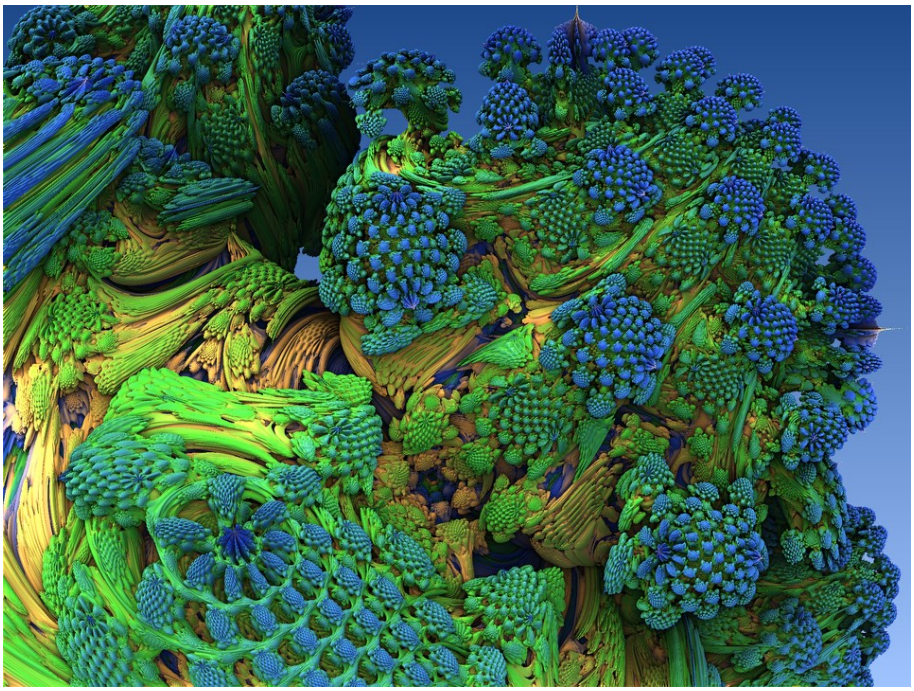


Figure 6.6.1 Mandel bulb fractal with natural looking colouring

6.7 Application Class Diagram

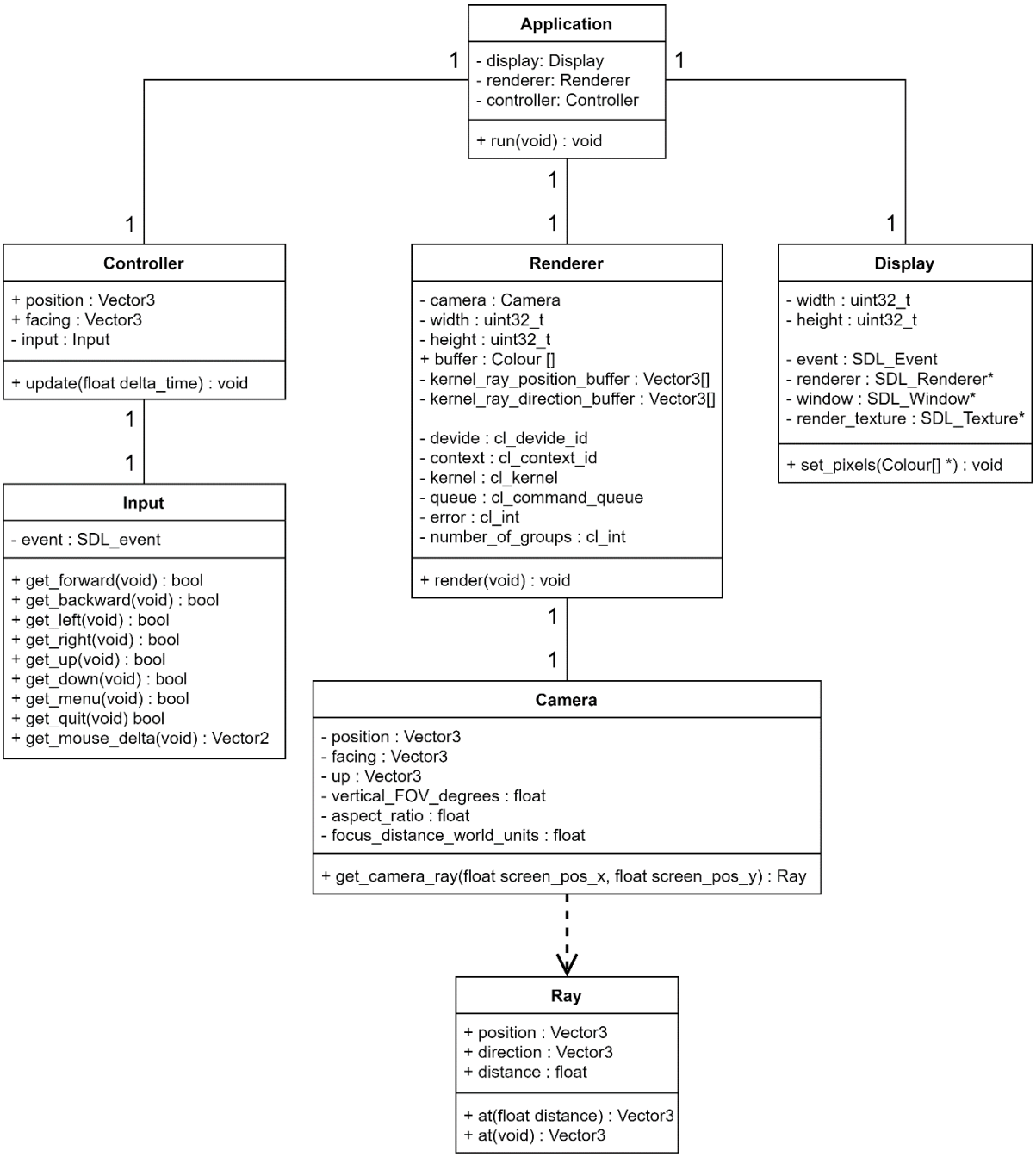
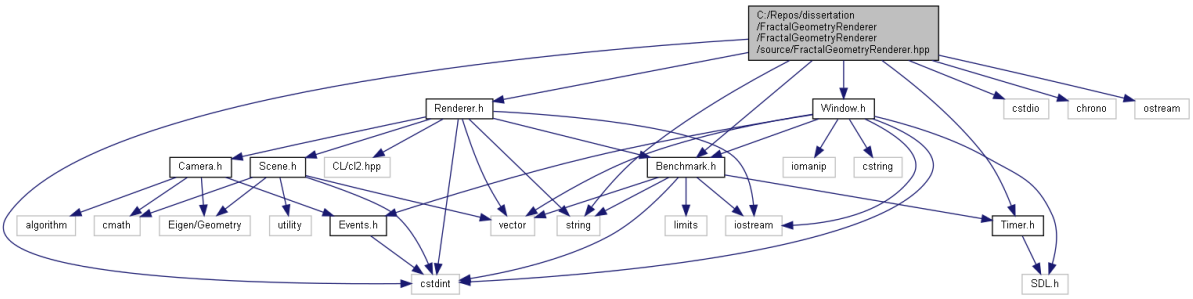


Figure 6.7.1 Application class diagram


C:/Repos/dissertation
/FractalGeometryRenderer
/FractalGeometryRenderer
/source/FractalGeometryRenderer.hpp
Renderer.h
Window.h
cstdio
chrono
ostream
Camera.h
Scene.h
CL/cl2.hpp
Benchmark.h
iomanip
cstring
algorithm
cmath
Eigen/Geometry
utility
Events.h
vector
string
limits
iostream
Timer.h
cstdint
SDL.h

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

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

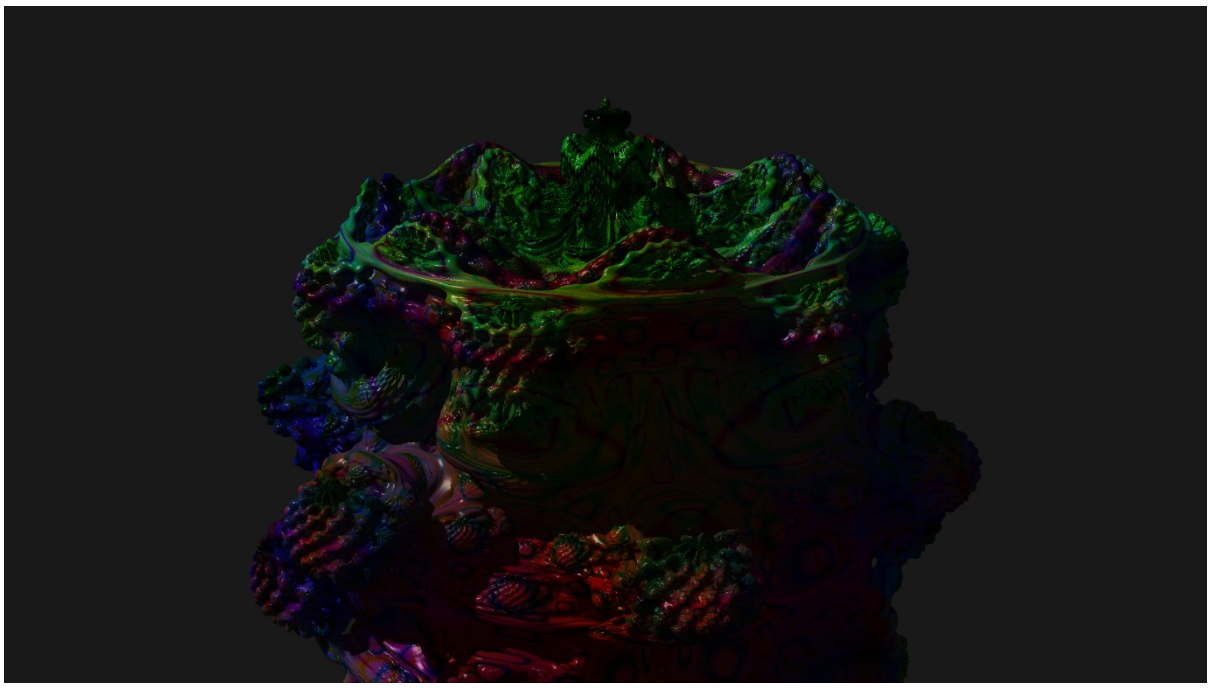


*Figure 6.8.1 Mandelbulb.cl scene with -cl-fast-relaxed-math **enabled***

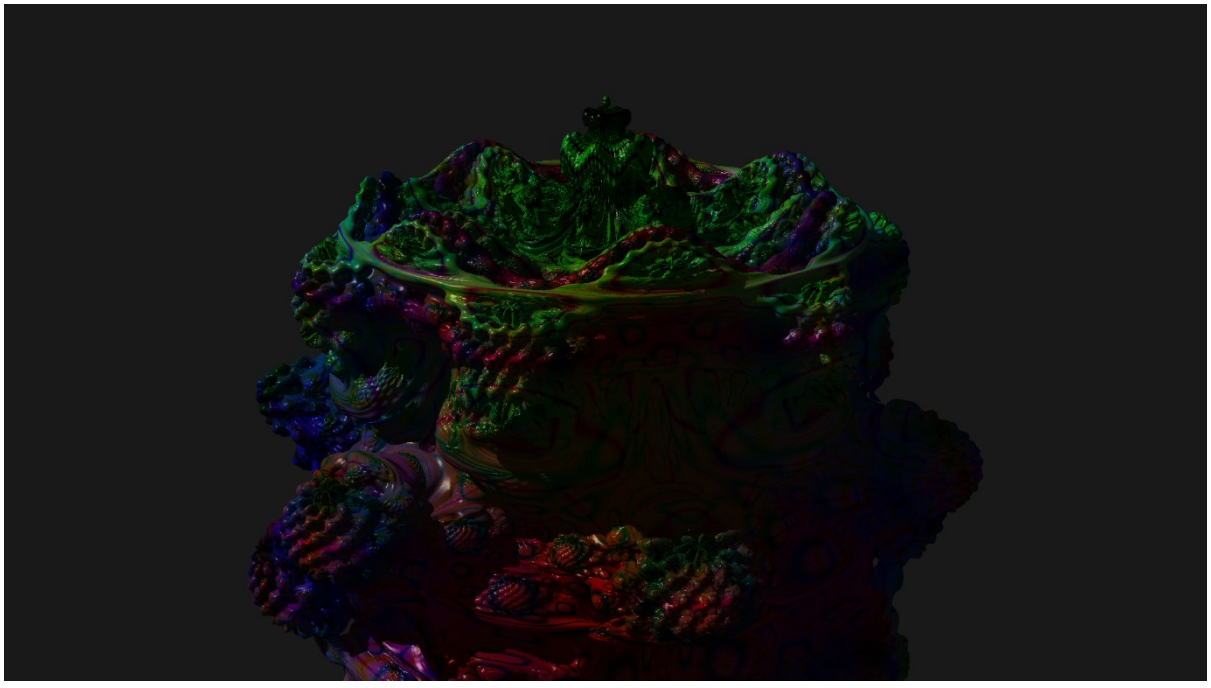


*Figure 6.8.2 Mandelbulb.cl with -cl-fast-relaxed-math **disabled***

6.9 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 6.9.1 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 • 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 6.9.2 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: 1366×768, 1920×1080, 2560×1440 and 3840×2160	MUST	Achieved
NF-4	GPU Parallel Computing	The application must run in parallel on the GPU	MUST	Achieved

6.10 Use Cases

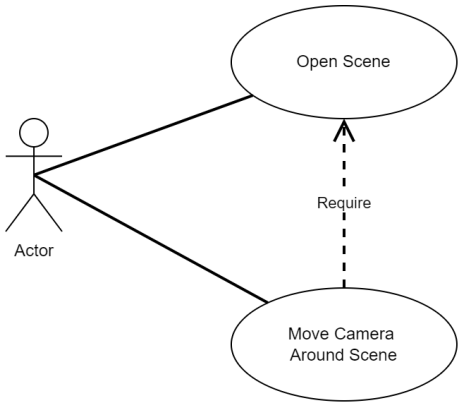


Figure 6.10.1 Use case diagram for the application

6.11 Project Plan

REMOVE THIS SECTION?

6.11.1 Professional, Legal, Ethical & Social Issues

The main principles of the open-source definition [43] 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 [44] 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 [5], within the LICENSE file.

The British Computer Society (BCS) codes of conduct [45] 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.

6.11.2 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 6.11.1 Risk rating matrix

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

Table 6.11.2 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 In addition, performance analysis of a non-real-time renderer could be performed instead	**LOW**

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.

6.11.3 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 [46] and has been included below. Tasks have been coloured using the following strategy: documentation (blue), implementation (red), and unallocated (green).

6.11.3.1 Initial Gantt Chart

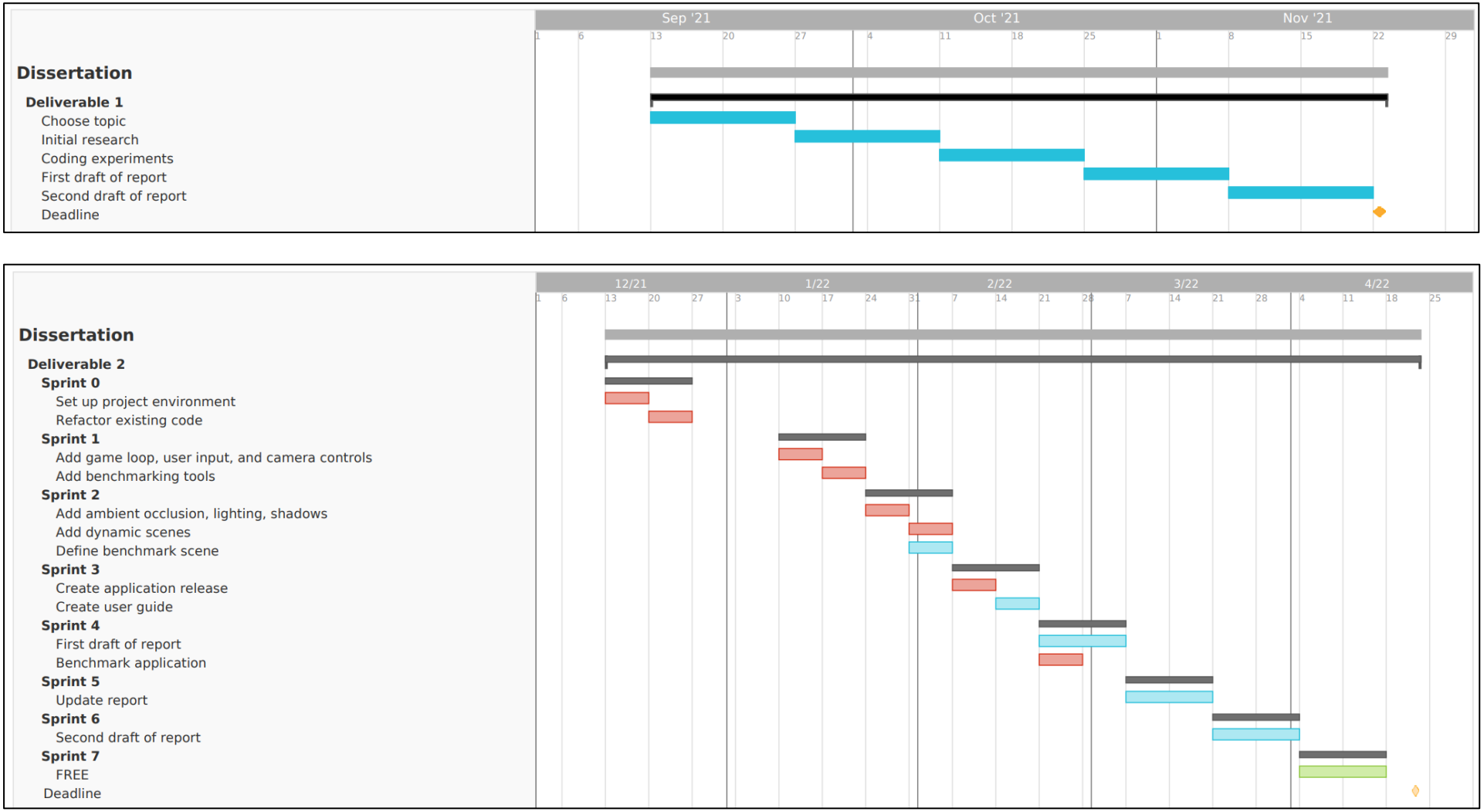


Figure 6.11.1 Initial Gantt chart for deliverable one (top) and deliverable two (bottom)

6.11.3.2 Revised Gantt Chart

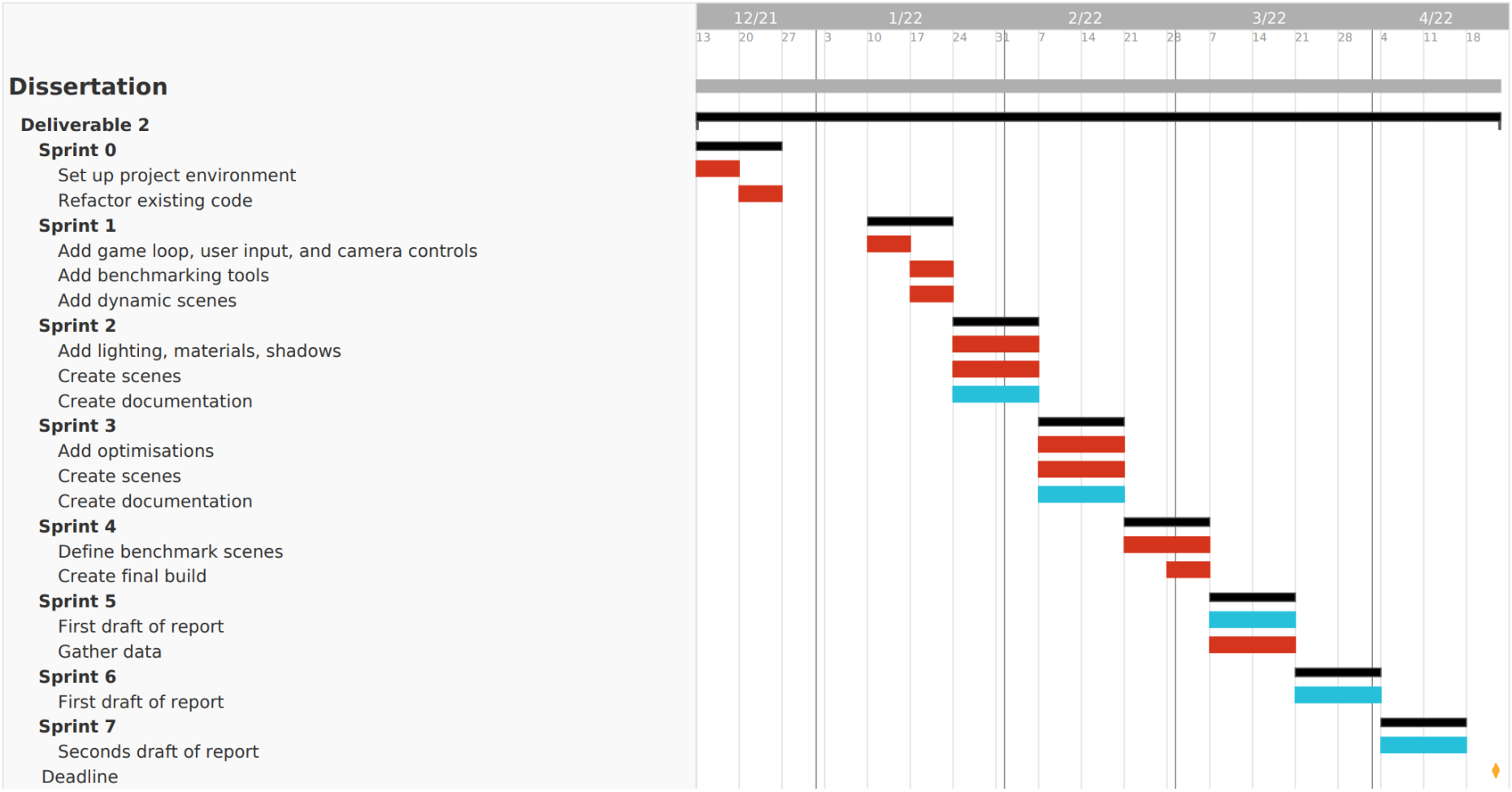


Figure 6.11.2 Revised Gantt chart for deliverable two

DUMPING GROUND

One drawback 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 boost gained from using bounding volumes.

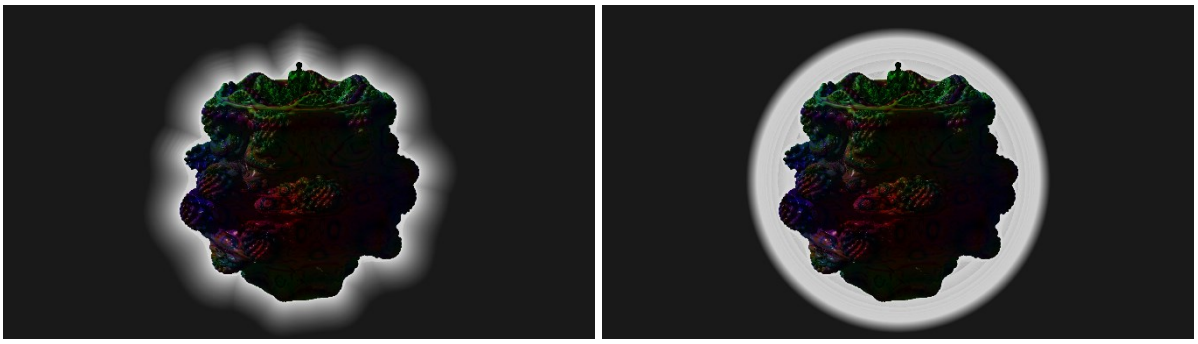


Figure 6.11.3 Mandelbulb scene with glow without bounding volume (left) and with bounding volume (right)

One way to fix this problem 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 its likely there would be visual artifacts. This separation of the two distance functions would require quite a significant rework of the application, which is likely not worth it just to fix the glow feature.

7 References

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

[2] University of Waterloo, "Top 5 applications of fractals." https://uwaterloo.ca/math/news/top-5-applications-fractals (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] S. Baarda, "3D Fractal Geometry Renderer," 2021. https://github.com/SolomonBaarda/fractal-geometry-renderer (accessed Sep. 24, 2021).

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

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

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

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

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

[11] 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.

[12] Wikipedia, "Mandelbrot set." 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," in *Ray Tracing Gems II*, 2021, pp. 529–544. doi: https://doi.org/10.1007/978-1-4842-7185-8_33.

[14] D. White, "The Unravelling of the Real 3D Mandelbrot Fractal," 2009. 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, Bachelor Thesis," 2010.

[16] 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).

[17] 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).

[18] 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

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

[20] 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).

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

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

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

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

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

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

[27] 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).

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

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

[30] "Fragmentarium." https://github.com/3Dickulus/FragM (accessed Nov. 17, 2021).

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

[32] "Visual Studio Development Features | Visual Studio." https://visualstudio.microsoft.com/vs/features/ (accessed Mar. 16, 2022).

[33] "Welcome back to C++ - Modern C++ | Microsoft Docs." https://docs.microsoft.com/en-us/cpp/cpp/welcome-back-to-cpp-modern-cpp?view=msvc-170 (accessed Mar. 16, 2022).

[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] "Realtime Fractal Renderer Documentation: Home." https://solomonbaarda.github.io/fractal-geometry-renderer/FractalGeometryRenderer/documentation/html/index.html (accessed Mar. 16, 2022).

[37] "clBuildProgram." https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clBuildProgram.html (accessed Mar. 09, 2022).

[38] "A new era of innovation: Moore's Law is not dead and AI is ready to explode - SiliconANGLE." https://siliconangle.com/2021/04/10/new-era-innovation-moores-law-not-dead-ai-ready-explode/ (accessed Mar. 30, 2022).

[39] "About | ECR Cluster Documentation Site." https://ecr-cluster.github.io/ (accessed Mar. 30, 2022).

[40] "Google Colab." https://research.google.com/colaboratory/faq.html (accessed Mar. 30, 2022).

[41] "Inigo Quilez :: fractals, computer graphics, mathematics, shaders, demoscene and more." https://www.iquilezles.org/www/articles/normalsSDF/normalsSDF.htm (accessed Mar. 31, 2022).

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

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

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

[45] 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

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