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

Deliverable 1: Final Year Dissertation

03/11/2021

by

Solomon Baarda

Meng Software Engineering

Heriot-Watt University

Supervisor: Dr Benjamin Kenwright

Second Reader: Ali Muzaffar

Abstract

TODO

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

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

Signed:

Date:

Table of Contents

Table of Figures

Table of Tables

Common Definitions

Table 1.1.i Common Definitions

Word	Definition
Frame	
Geometry	
Ray	
Render	
Method/constant overloading	

Common Abbreviations

Table 1.1.ii Common Abbreviations

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

1 INTRODUCTION

1.1 PROJECT DESCRIPTION

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

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

We ned some way to visualise fractal patterns. 3D fractals. These exists many programs designed to render 3D fractals, however, the majority are created using some from of shader language code, and contain massive amounts of code duplication between scenes. These programs require a solid grasp of the shader language and proper understanding of the ray tracing algorithm. There is a severe lack of fractal rendering applications for which a user can pick up and start experimenting straight away. This project aims to change that.

The term fractal has been used throughout this report to describe a pattern or geometry that displays the recursive self-similarity characteristic of fractals. The term fractal-like has been used to describe a pattern or geometry that appears to look like a fractal but may show some differences. For example, many fractals that this engine will render will not truly be infinite, and so are not true fractals.

Why is this project useful?

Visualise fractals – art, help discover new equations for modelling nature

Fractals define a new geometry – one that can potentially be used to define the universe we live in.

Snow flakes

Art

Model cities

Why fractals are important

brainwaves, bacteria, visualising

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

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

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

The performance of the engine will be benchmarked across various systems to determine whether the “real-time” requirement of the project has been achieved.

1.2 Aims & Objectives

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

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

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

1. Implement the basic ray marching algorithm for execution on a single CPU thread
2. Design a software structure that will minimise code duplication between scenes

3. Update the single threaded ray marching code so it can be run in parallel on the GPU
4. Update the code to use a game loop so that it can be executed in real-time
5. Add movement controls and a GUI
6. Add optical effects such as ambient occlusion, soft shadows and lighting
7. Define a benchmark scene to test the performance of the renderer
8. Create documentation to assist users when creating scenes

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

These objectives have been created bearing the SMART properties [5] in mind. SMART stands for: Specific, Measurable, Achievable, Realistic and Time constrained.

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

The objectives form the main tasks to be completed during the duration of this project and the requirements specification in section 3.2 and the Gantt chart in section 5.4 have been structured around these.

1.3 SCOPE

The scope of the project has been carefully considered, and several stretch goals have been included in the requirements specification if good progress is made. Objectives 5 and 6 have large amounts of flexibility and can be extended or cut back depending on time constraints. The requirements

specification in section 3.2 describes the specific functionality to be implemented for each objective along with any optional stretch goals.

Objectives 1 and 2 have already been completed. Some initial experimentation rendering still images of the Mandel bulb fractal and other geometry has been very successful. Some of these renders can be viewed in the appendix, section 8. In addition, some experimentation with OpenCL, a library that allows GPU parallel code to be written and executed, has been completed. These experiments were done to gain familiarity with this style of programming in the hope to reduce the learning curve of this new software.

1.4 A

https://github.com/SolomonBaarda/fractal-geometry-renderer

All renders from experiments can be found in the Experimentation folder. The main application is located in the Project folder.

2 Literature Review

2.1 Fractals

In mathematics, a fractal is a complicated pattern built from simple repeated shapes, which are reduced in size every time they are repeated [6]. The term fractal was proposed by Benua Mandelbrot in 1975 to describe this new geometry and comes from the Latin word frāctus, meaning fractured [7]. Fractal geometry bears a striking resemblance to geometry created by nature, which often has complex repeated patterns while also being irregular and distorted.

Figure 2.1.i Fractal in nature [8]

This project takes a fractal to mean this…

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

Typical 2d fractals

2.2 3D FRACTALS

How we create 3d fractals

2.2.1 Sierpiński Tetrahedron

The Sierpiński tetrahedron, also known as the Sierpiński pyramid, is a 3D representation of the famous Sierpiński tringle, named after the Polish mathematician Wacław Sierpiński [9]. The Sierpiński triangle is one of the most simple and elegant fractals and has been a popular decorative pattern for centuries. This pattern is created by recursively each splitting each solid equilateral triangle into four smaller equilateral triangles and removing the middle one. Theoretically, these

steps are repeated forever, but in practice some maximum depth must be specified as computers only have finite memory.

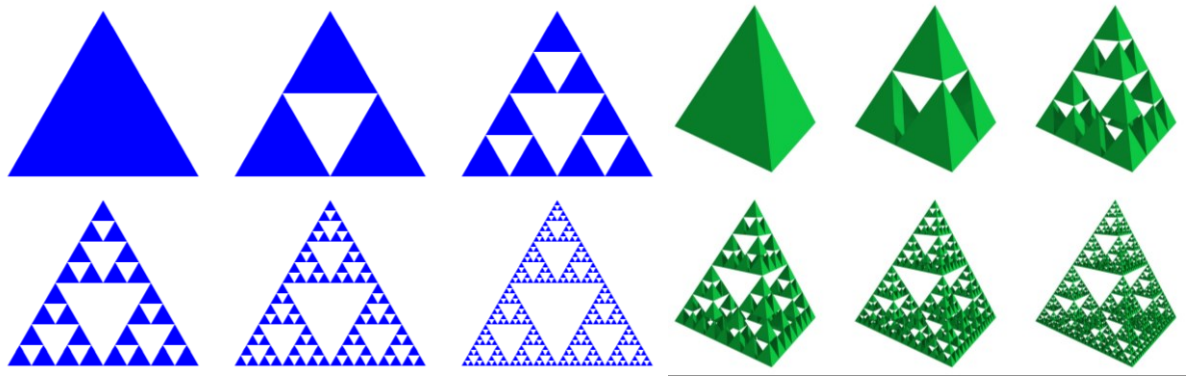

Figure 2.2.i Sierpiński triangle (left) [10] *and tetrahedron (right)* [10] *both of recursive depth 5*

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

2.2.2 Menger Sponge

The Menger sponge, also known as the Menger cube or Sierpiński cube, is another 3D representation of a 2D fractal patten created by Polish mathematician Wacław Sierpiński [9]. The Menger sponge is a 3D representation of the 2D Sierpiński carpet fractal, which follows very similar recursive rules to the Sierpiński triangle but uses squares instead of triangles.

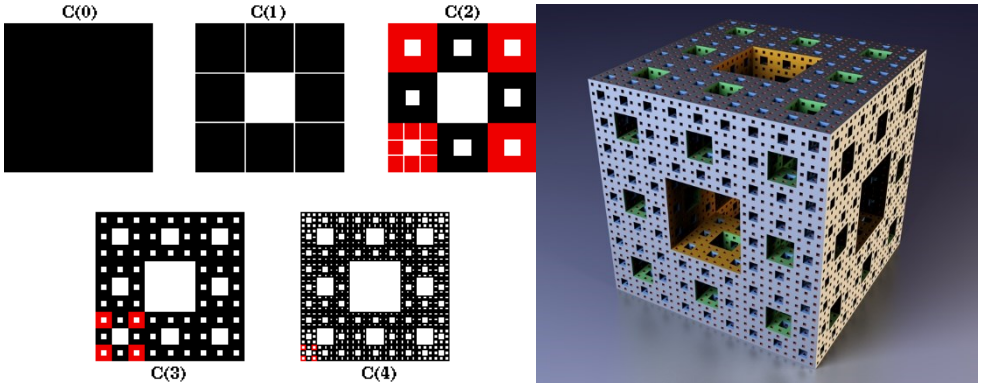

Figure 2.2.ii Sierpiński carpet (left) [11] *and Manger sponge (right)* [12] *both of recursive depth 4*

The number of objects required to create these fractals at various recursive depths increases similarly to the Sierpiński triangle and tetrahedron, but at a different rate. The Sierpiński carpet increases by a factor of 8 each iteration and the Menger sponge by a factor of 20. A Menger cube at recursive depth n is made up of 20^n smaller cubes, each with a side length of $(\frac{1}{3})^n$ [13].

2.2.3 Mandel Bulb

The Mandel bulb is a 3D fractal, created by Daniel White and Paul Nylander and is now the commonly used representation of the 2D Mandelbrot fractal. For many years, it was thought that a true 3D representation of the Mandelbrot fractal did not exist, since there is no 3D representation of the 2D space of complex numbers, on which the Mandelbrot fractal is built upon [14].

The equation $f(z) = z^n + c$ leads us

White and Nylander's formula for the nth power of a point $p = (x, y, z)$ is

$$p^n = r^n(\sin(n)\cos(n), \sin(\quad)\sin(\quad), \cos())$$

Where $r = |p|$ is the norm of p

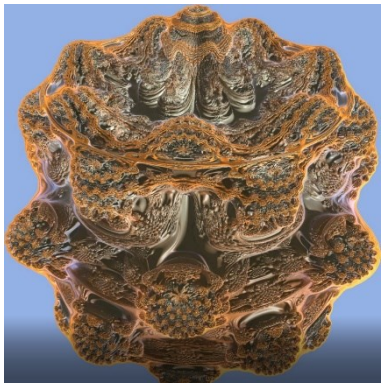

Figure 2.2.iii Render of Mandel bulb fractal created using DXR shaders [14]

2.2.4 Julia Set

The Julia set is a set of fractals named after the French mathematician Gaston Julia [15]. Fractals in the Julia set come from the convergence of the system given by the quadratic function $f(z) = z^2 + c$, where different values of c produce different fractals.

Similar to the Mandelbrot fractal, 2D Julia sets use the complex number plane as the domain of the quadratic function f. Unlike the Mandel bulb, the Julia set can be extended into 4D using quaternions as the domain of f [16], and then 3D slices can be taken of the quaternions to view the fractal in 3D space.

Some recent work, such as Quilez [17], uses pixel shaders to render a 3D Julia set.

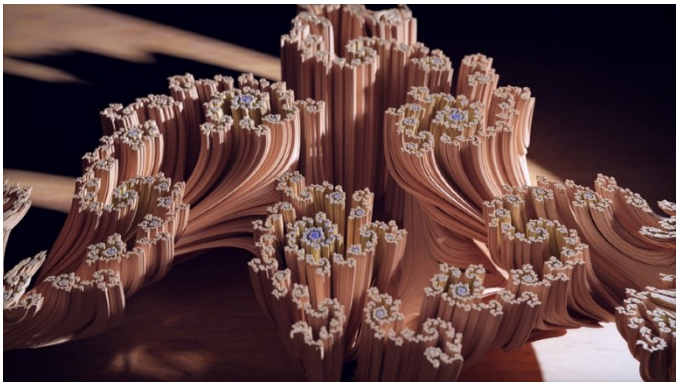

Figure 2.2.iv Ray marched Julia set, cut in half to expose the fractal interior [17]

The recent work da Silva et al in 2021 [14] took this a step further, using Nvidia DirectX Raytracing (DXR) shaders to render the visualisations of the 3D Julia set and Mandel bulb fractal.

TODO talk about pros/cons of existing work

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

TODO talk about my experiments?

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

2.3 Ray Tracing

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

When rendering an image using ray tracing, for each pixel in the camera, a ray (simply a line in 3D space) is extended or traced forwards from the camera position until it intersects with the surface of an object. From there, the ray can be absorbed or reflected by the surface and more rays can be sent out recursively, which can be used to take into consideration light absorption, reflection, refraction, and fluorescence.

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

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

2.4 Ray Marching

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

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

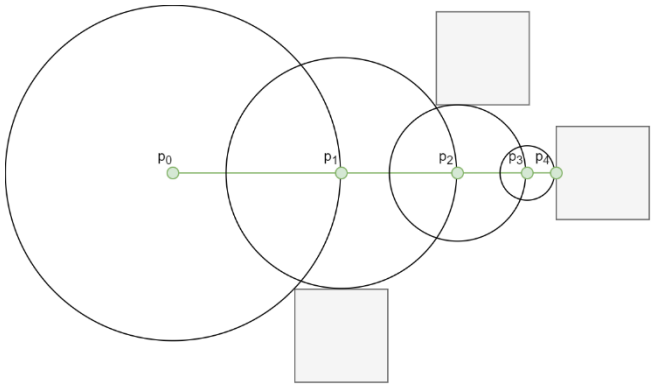

Figure 2.4.i Ray marching diagram

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

2.4.1 Benefits of Ray Marching

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

Ambient occlusion is a technique used to determine how exposed each point in a scene is to ambient lighting [3]. This means that the more complex the surface of the geometry is (with creases, holes etc), the less ways ambient light can get into it those places and so the darker they should be. With ray marching, the surface complexity of geometry is usually proportional to the number of steps taken by the algorithm [4]. This approximation works well in practice and comes with no extra computational cost at all.

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

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

2.4.2 Signed Distance Functions

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

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

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

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

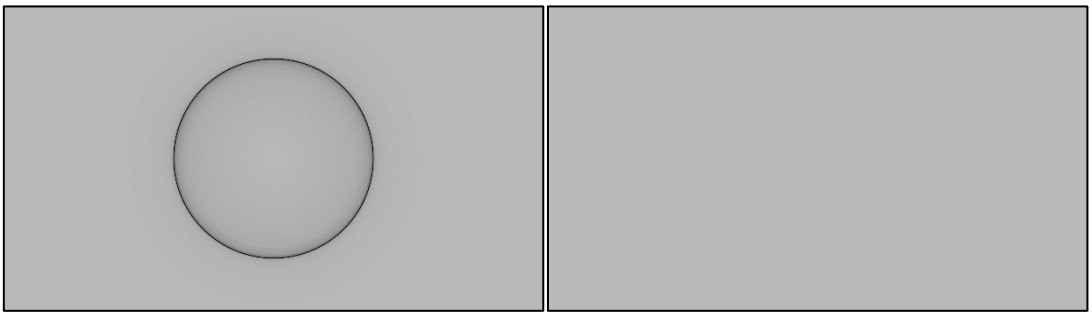

Figure 2.4.ii Hollow vs solid interior geometry experiment

2.4.3 Primitives

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

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

Where:

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

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

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

2.4.4 Alterations & Combinations

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

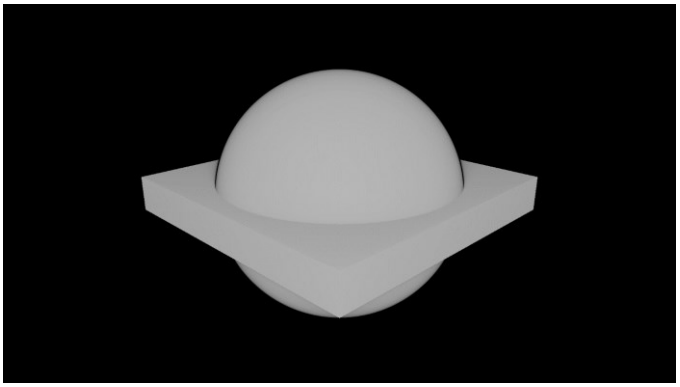

Figure 2.4.iii Ray marched union of sphere and box experiment

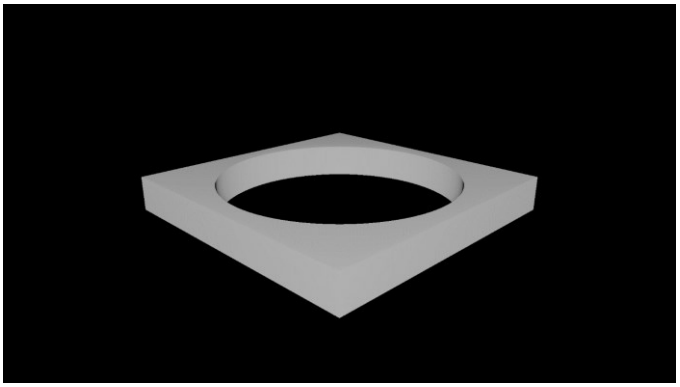

Figure 2.4.iv Ray marched intersection of sphere and box experiment

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

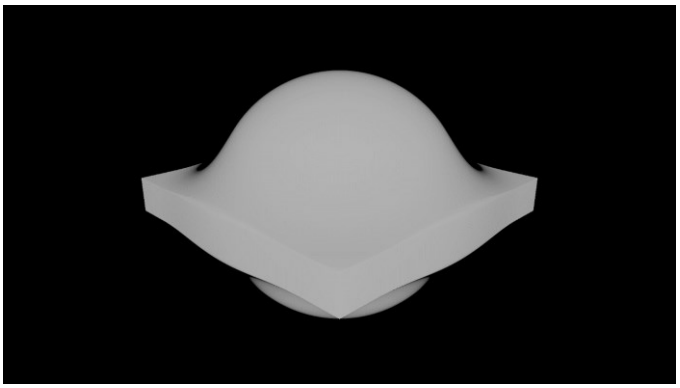

Figure 2.4.v Ray marched smooth union of sphere and box experiment

There are several additional alterations that can be applied to primitives once we have their signed distance function. A primitive can be elongated along any axis, its edges can be rounded, it can be extruded, and it can be "onioned" – a process of adding concentric layers to a shape. All these operations are relatively cheap. Signed distance functions can also be repeated, twisted, bent, and surfaces displaced using an equation e.g., a noise function or sin wave, though these alterations are more expensive.

2.4.5 Surface Normal

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

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

where

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

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

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

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

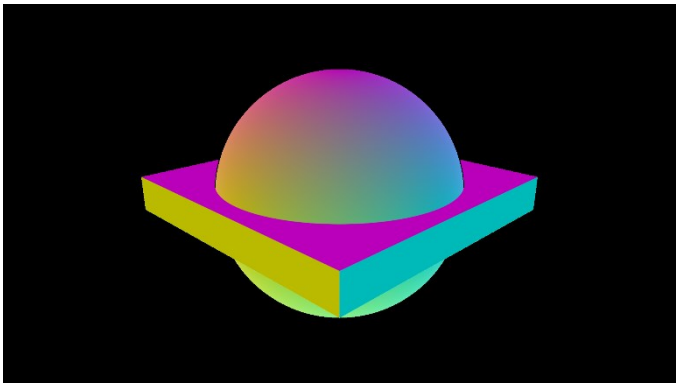

Figure 2.4.vi Surface normal of ray marched sphere and box scene experiment

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

2.5 Existing Projects

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

2.5.1 Fragmentarium

https://github.com/Syntopia/Fragmentarium

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

2.5.2 Smallpt

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

2.5.3 Ray Tracing in One Weekend

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

3 Requirements Analysis

3.1 Use Cases

TODO

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

3.2 Requirements Specification

Table 3.2.i Functional Requirement Specification

ID	Name	Description	Objective	Priority	Testing strategy
F-1	Real-time	The application must be capable of rendering scenes in real-time, running at 1920x1080 with a minimum fps of 60	7	MUST	Benchmark
F-2	Scene requirements	A scene must contain: • Geometry • Lights • Camera		MUST	Unit test
F-3	Example scenes	The application must contain multiple example scenes, some of which could include: • Julia set fractal • Mandel bulb fractal • Sierpinski tetrahedron fractal • Menger sponge fractal		MUST	Unit test
F-4	Mandatory optical effects	The application must support the following optical effects: • Ambient occlusion • Hard and soft shadows • Glow		MUST	

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

Table 3.2.ii Non-functional Requirement Specification

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

3.3 Testing Strategy

TODO

Benchmark – to test for real-time project requirement

Unit tests – several requirements + code correctness

Other type of test for other requirements? User test?

4 SOFTWARE DESIGN

4.1 TECHNOLOGIES

The application will be developed using the following technologies:

Table 4.1.i Application Technologies

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

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

Table 4.1.ii Development Technologies

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

4.2 Class Structure

The application will be structured using several key classes:

Table 4.2.i Class Responsibilities

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

CLASS DIAGRAM TODO

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

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

Table 4.2.ii Kernel Method Reusability Matrix

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

Table 4.2.iii Kernel Constant Reusability Matrix

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

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

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

5 Project Plan

5.1 Design & Evaluation Methodology

TODO

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

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

emphasise visualising

pros and cons

5.2 Legal, Ethical & Social Issues

TODO

open source standards

BCS

license

5.3 Risk Analysis

Table 5.3.i Risk Analysis

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

5.4 Timetable

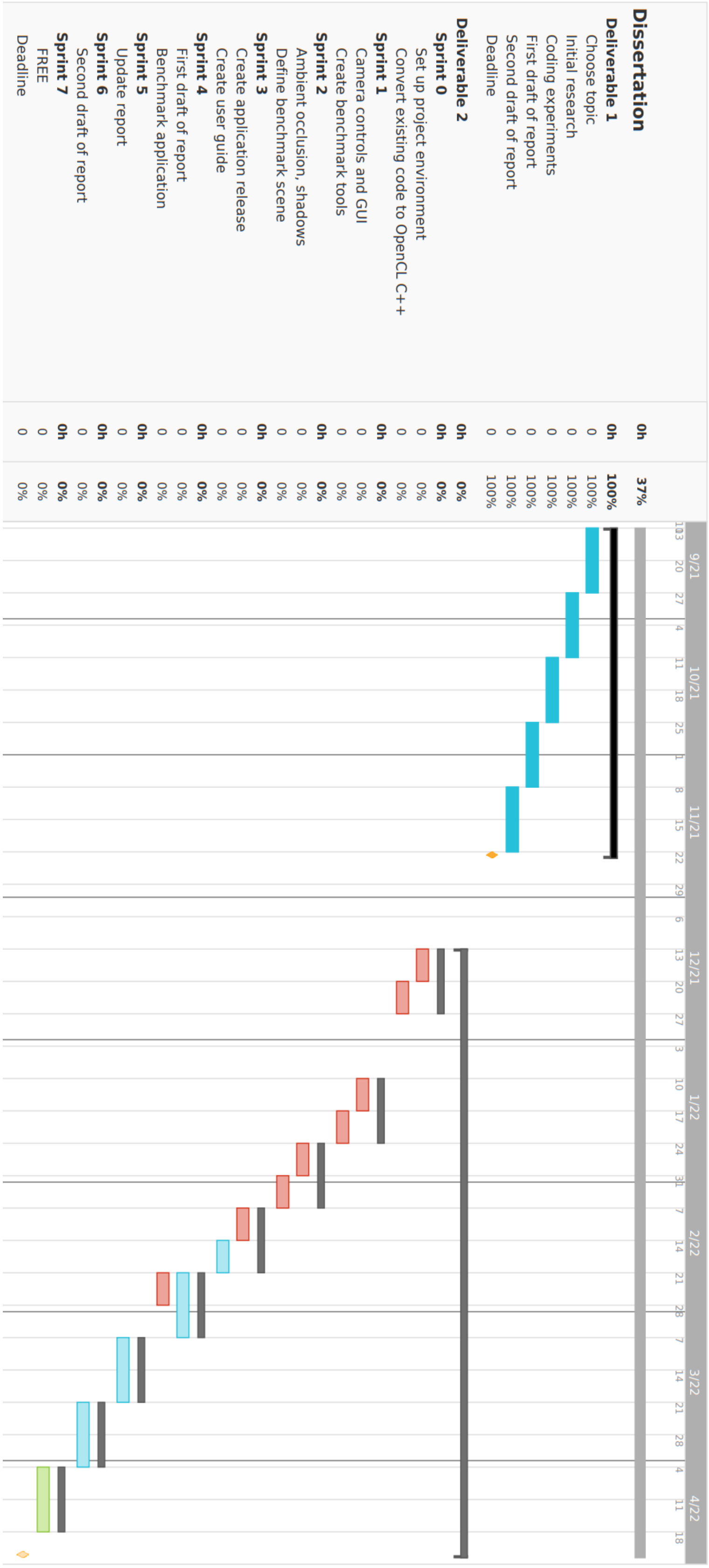

Figure 5.4.i Project timeline Gantt chart

6 CONCLUSION

TODO

7 REFERENCES

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

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

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

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

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

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

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

[8] "aloe | In UC Berkeley's botanical garden. Added to Cream of … | Flickr." https://www.flickr.com/photos/genista/2447322/in/photolist-dxvd (accessed Nov. 10, 2021).

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

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

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

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

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

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

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

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

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

[18] J. Peddie, "Ray Tracing: A Tool for All," 2019.

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

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

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

[23] 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).

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

8 APPENDICES

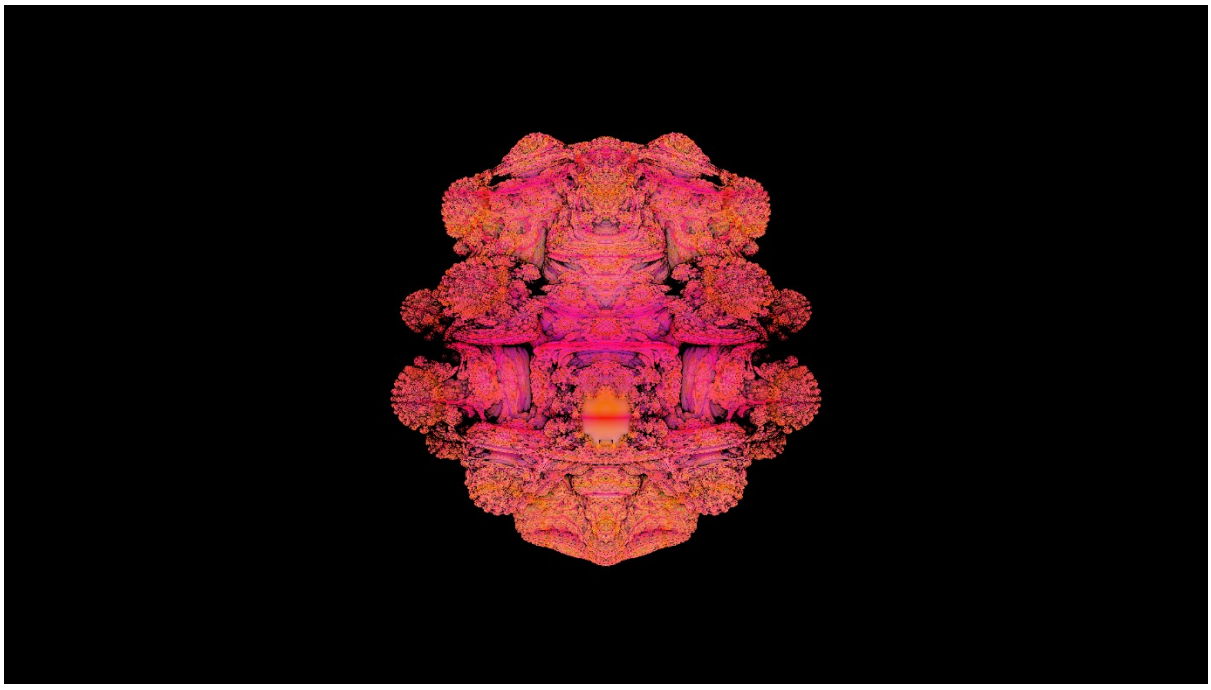

Figure 5.4.i Render of the Mandel bulb fractal, created using fractal equation from [24]