

by

Solomon Baarda

Meng Software Engineering

Heriot-Watt University

Supervisor: Dr Benjamin Kenwright

Second Reader: Ali Muzaffar

Real-time Rendering of 3D “Fractal-like” Geometry

Deliverable 1: Final Year Dissertation

03/11/2021

Abstract

TODO

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

[1 Introduction 5](#_Toc86833138)

[1.1 Aims & Objectives 5](#_Toc86833139)

[1.2 Project Description 5](#_Toc86833140)

[1.3 Scope 6](#_Toc86833141)

[2 Literature Review 6](#_Toc86833142)

[2.1 Realtime Rendering Methods 6](#_Toc86833143)

[2.2 Ray Tracing 6](#_Toc86833144)

[2.3 Ray Marching 7](#_Toc86833145)

[2.3.1 Benefits of ray marching 8](#_Toc86833146)

[2.3.2 Signed Distance Function 8](#_Toc86833147)

[2.3.3 Primitives 9](#_Toc86833148)

[2.3.4 Alterations & Combinations 9](#_Toc86833149)

[2.3.5 Surface Normal 11](#_Toc86833150)

[2.3.6 Fractals 12](#_Toc86833151)

[2.3.7 Collision detection 13](#_Toc86833152)

[2.4 Existing Projects 13](#_Toc86833153)

[2.4.1 Fragmentarium 13](#_Toc86833154)

[2.4.2 Smallpt 13](#_Toc86833155)

[2.4.3 Ray Tracing in One Weekend 13](#_Toc86833156)

[3 Requirements Analysis 13](#_Toc86833157)

[3.1 Use Cases 13](#_Toc86833158)

[3.2 Requirements Specification 14](#_Toc86833159)

[3.3 Testing Strategy 14](#_Toc86833160)

[3.4 Evaluation Strategy 14](#_Toc86833161)

[4 Software Design 15](#_Toc86833162)

[4.1 Technologies 15](#_Toc86833163)

[4.2 Class Structure 15](#_Toc86833164)

[5 Project Plan 17](#_Toc86833165)

[5.1 Design Methodology 17](#_Toc86833166)

[5.2 Legal, Ethical & Social Issues 17](#_Toc86833167)

[5.3 Risk Analysis 17](#_Toc86833168)

[5.4 Timetable 17](#_Toc86833169)

[6 References 19](#_Toc86833170)

[7 Appendices 20](#_Toc86833171)

Table of Figures

[Figure 1 - Ray Marching 7](#_Toc86833172)

[Figure 2 - DF and SDF 9](#_Toc86833173)

[Figure 3 - Union of Sphere and Box 10](#_Toc86833174)

[Figure 4 – Intersection of Sphere and Box 10](#_Toc86833175)

[Figure 5 - Smooth Union of Sphere and Box 11](#_Toc86833176)

[Figure 6 - Surface Normal of Sphere and Box Scene 12](#_Toc86833177)

Table of Tables

[Table 1 – Common Definitions 4](#_Toc86833187)

[Table 2 – Common Abbreviations 4](#_Toc86833188)

[Table 3 - Functional Requirement Specification 14](#_Toc86833189)

[Table 4 - Non-functional Requirement Specification 14](#_Toc86833190)

[Table 5 – Application Technologies 15](#_Toc86833191)

[Table 6 – Development Technologies 15](#_Toc86833192)

[Table 7 – Class Responsibilities 15](#_Toc86833193)

[Table 8 – Kernel Method Reusability Matrix 16](#_Toc86833194)

[Table 9 – Kernel Constant Reusability Matrix 16](#_Toc86833195)

Common Definitions

Table 1 – Common Definitions

|  |  |
| --- | --- |
| Word | Definition |
| Frame |  |
| Geometry |  |
| Ray |  |
| Render |  |

Common Abbreviations

Table 2 – Common Abbreviations

|  |  |
| --- | --- |
| Word | Abbreviation |
| FPS | Frames per second |
| PC | Personal computer |
| SDF | Signed distance function |

# Introduction

<https://github.com/SolomonBaarda/dissertation>

## Aims & Objectives

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

Want to create a rendering engine capable of rendering non-euclidian geometry

For which a ray-surface intersection function does not exist

There are lots of shader code to do this

Not many compiled applications though

Needs to be easily extendable to allow for custom scenes

Must have good performance to be realtime

Find a good balance between looking good and performance

Need approximations

## Project Description

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

The benchmark scene has yet to be fully defined, but it must be non-trivial to render. This means it should contain multiple geometries (both fractal and primitive) and multiple lights while also making use of advanced rendering features like ambient occlusion, soft shadows, and reflections. The camera should 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.

## Scope

The scope of the project has been carefully considered, and several stretch goals have been included in the requirements specification if good progress is made. Some initial experiments with a prototype have been made (renders can be viewed on the GitHub repository) and good progress has been made.

A basic Mandel bulb

# Literature Review

Renderer takes 3d scenes, and maps them to a 2d screen

Rasterizer

## Realtime Rendering Methods

Computing intersections is not cheap

## 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 its effects on geometry by taking into consideration reflection, refraction, reflections of reflections [1].

In ray racing, 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 strengths (but also 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. Ray tracing, however, is not suitable for any geometries for which a ray-surface intersection function does not exist [2], such as fractal geometries.

## Ray Marching

Ray marching is a variation of ray tracing, which differs in the method of detecting collisions 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, the distance estimator 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. Each points distance estimation is marked as the circle centred on that point.

Diagram, engineering drawing

Description automatically generated

Figure 1 - Ray Marching

A distance estimation 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 too large. If it is too small, then the ray marching algorithm becomes very inefficient so a fine balance must be found between accuracy and efficiency.

### Benefits of ray marching

Ray marching may sound more computationally complex than ray tracing, however, it does provide several benefits. Most notably, ray marching does not require a surface collision 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 later in the report. Many effects such as reflections, hard shadows and depth of field can be implemented almost identically to how they are in ray tracing. However, there are several effects that the ray marching algorithm gets essentially for free.

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 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. 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. 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.

### Signed Distance Function

A signed distance function (SDF) for an object, is a function which given any point in 3D space, will return the distance to the surface of the object. 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 calls to corresponding SDF for every geometry in the scene, and will return the smallest value.

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.

In the scene below

![Shape

Description automatically generated with medium confidence](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAmgAAAFaCAMAAACda1FAAAAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAAAGUExURbq6ugAAADaON5cAAAACdFJOU/8A5bcwSgAAAAlwSFlzAAAh1QAAIdUBBJy0nQAAAOZJREFUeF7twTEBAAAAwqD1T20LLyAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAOCuBkIXAAFtyuL2AAAAAElFTkSuQmCC)

Figure 2 - DF and SDF

### Primitives

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

Where:

Indigo Quilez maintains a list of SDFs for most 3D primitives [4].

### Alterations & Combinations

Signed distance functions can be translated, rotated, and scaled.

Signed distance functions can be combined using union, subtraction, and intersection operations.

A picture containing dishware

Description automatically generated

Figure 3 - Union of Sphere and Box

Icon

Description automatically generated with low confidence

Figure 4 – Intersection of Sphere and Box

Signed distance functions can also be combined using a version that uses smoothing.



Figure 5 - Smooth Union of Sphere and Box

There are several 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 very cheap.

Signed distance functions can also be repeated, twisted, bent, and surfaces displaced using an equation e.g., a noise function or sin wave.

### Surface Normal

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

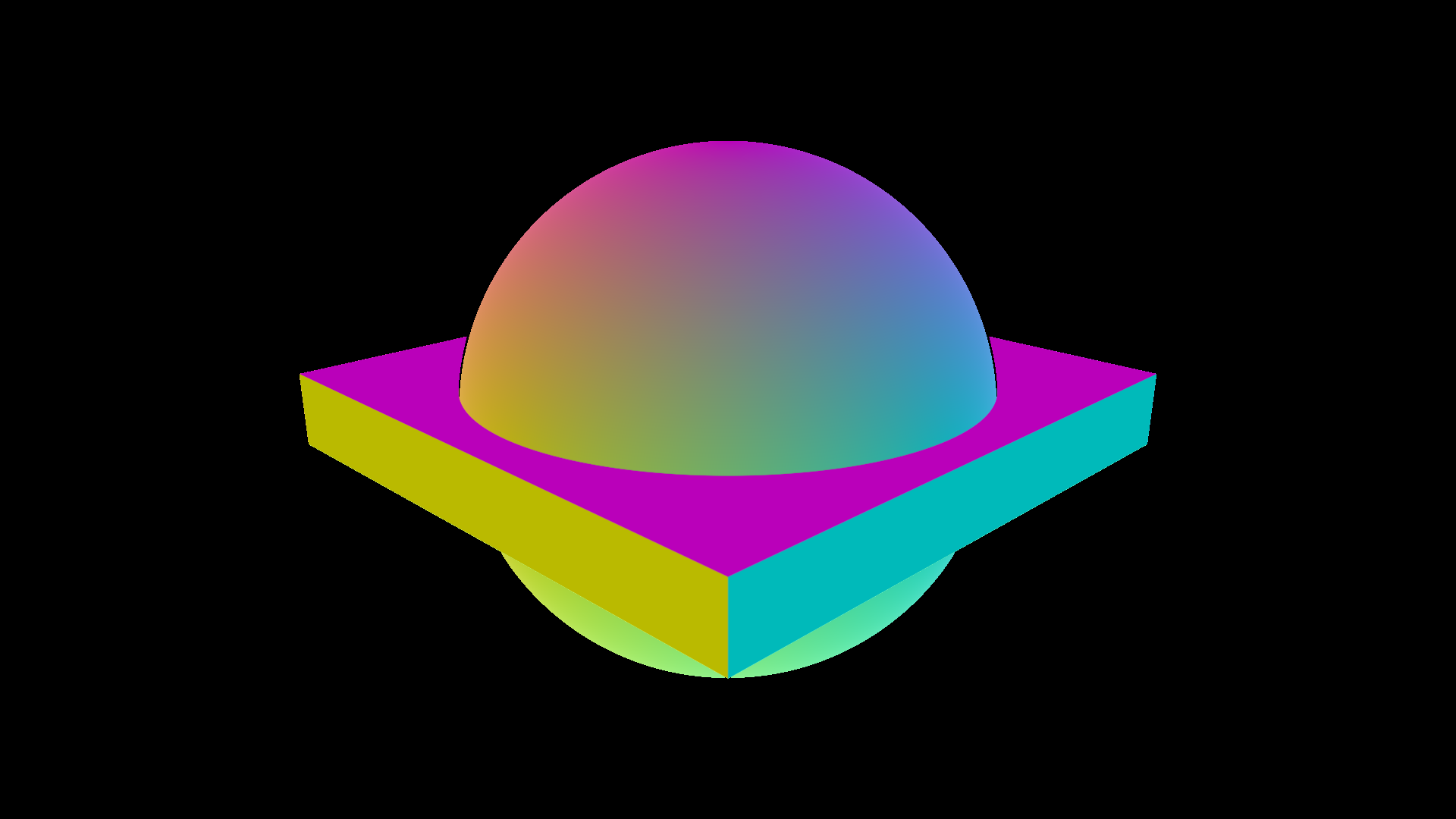


Figure 6 - Surface Normal of Sphere and Box Scene

### Fractals

In mathematics, a fractal is a complicated pattern built from simple repeated shapes, which are reduced in size every time they are repeated [5]. These shapes are self-similar, though not often symmetrical.

The idea of fractal geometry appeared in the late 1970s, inspired through the work of Benua Mandelbrot and his book “Fractals: form, chance and dimension”, released 1977. This book introduced the concept of a “fractal dimension”, a measure of the complexity of how the detail in a pattern will change in respect to the scale at which it is measured [6].

How they are calculated – running sum etc

3D fractals

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

Serpinski tetrahedron

Menger sponge

Mandelbulb

Julia sets

### Collision detection

Collision detection is possible

Marble marcher

## Existing Projects

### Fragmentarium

<https://github.com/Syntopia/Fragmentarium>

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

### Smallpt

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

### Ray Tracing in One Weekend

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

# Requirements Analysis

## Use Cases

## Requirements Specification

Table 3 - Functional Requirement Specification

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Name | Description | Priority | Testing strategy |
| F-1 | Real-time | The application must be capable of rendering scenes in real time | MUST |  |
| F-2 |  | The application must |  |  |
| F-3 |  |  |  |  |
| F-4 |  |  |  |  |
| F-5 |  |  |  |  |
|  |  |  |  |  |

Table 4 - Non-functional Requirement Specification

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Name | Description | Priority | Testing strategy |
| NF-1 | Executable | The application must run from a compiled executable | MUST |  |
| NF-2 | Display resolutions | The application must support the following common display resolutions: 1366x768, 1920x1080, 2560x1440 and 3840x2160 |  |  |
| NF-3 |  |  |  |  |
| NF-4 |  |  |  |  |
| NF-5 |  |  |  |  |
| NF-6 |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

## Testing Strategy

## Evaluation Strategy

# Software Design

## Technologies

The application will be created using the following technologies:

Table 5 – Application Technologies

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

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

Table 6 – Development Technologies

|  |  |  |
| --- | --- | --- |
| Technology | Description | Justification |
| GitHub | Version control software | * TODO |
| Microsoft Word | Word processing software |  |
| Mendeley | Reference manager |  |
| Microsoft Teams |  |  |

## Class Structure

The application will be structured using several key classes:

Table 7 – Class Responsibilities

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

CLASS DIAGRAM TODO

The Display and Controller classes are basic and only provide an interface to some SDL2\_Event, SDL2\_Renderer, SDL2\_Window and SDL2\_Texture instances. The Renderer class, however, is much more complex and requires discussion.

The Renderer class provides an interface to 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. However, this makes it hard to reuse code as the implementations of several methods, and the values of several constants will differ between scenes. The tables below show the main methods and constants used in the kernel and their reusability status between scenes.

Table 8 – Kernel Method Reusability Matrix

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

Table 9 – Kernel Constant Reusability Matrix

|  |  |  |
| --- | --- | --- |
| Constant name | Purpose | Reusable across scenes? |
| MAXIMUM\_MARCH\_STEPS | Maximum number of iterations the ray marching algorithm can make | NO |
| MAXIMUM\_MARCH\_DISTANCE | Maximum distance the ray can be marched in the scene | NO |
| SURFACE\_INTERSECTION\_EPSILON | A very small value used to determine when the DE has converged to zero | YES |
| SURFACE\_NORMAL\_EPSILON | Arbitrary distance to probe the DE function when calculating the surface normal | YES |

Solution to code reusability:

Have main kernel file with empty implementation of DE and full implementation of all other methods. Use OpenCL C++ kernel language to overload the DE function? Available in OpenCL 3.0. Each scene would have its own kernel file, and would include the main file, overloading the things it needs to.

Other solutions?

# Project Plan

## Design Methodology

## Legal, Ethical & Social Issues

A well-researched consideration of any Professional, Legal, Ethical, and Social Issues pertinent to the project. (e.g. codes of conduct (BCS), codes of practice, standards, computer law, ethical decision making, intellectual property, social aspects, copyright, data protection, and so on)

## Risk Analysis

Table 10 – Risk Analysis

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

## Timetable

Chart, waterfall chart

Description automatically generated

# References

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

[2] 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

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

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

[5] Cambridge English Dictionary, “Cambridge English Dictionary.” https://dictionary.cambridge.org/dictionary/english/fractal (accessed Oct. 18, 2021).

[6] M. S. Longuet-Higgins, “Review of Fractals: Form, Chance and Dimension by Beniot B. Mandelbrot,” *Journal of Fluid Mechanics*, vol. 92, no. 1, pp. 206–208, May 1979, doi: 10.1017/s0022112079210586.

# Appendices