Utilising ray marching and signed distance functions to render a scene of primitives

By Anthony Sturdy (18015368)

BSc (Hons) Computer Games Programming

Supervisors: Dr. David White, Craig Weightman

May 2022



Contents

[Abstract 2](#_Toc101868492)

[Introduction 3](#_Toc101868493)

[Aim 3](#_Toc101868494)

[Objectives 3](#_Toc101868495)

[Literature Review 4](#_Toc101868496)

[3D Wireframe Rendering 4](#_Toc101868497)

[Raycast Rendering 5](#_Toc101868498)

[Rasterised Rendering 6](#_Toc101868499)

[Shaders 6](#_Toc101868500)

[Ray Traced Rendering 7](#_Toc101868501)

[Ray Marching 8](#_Toc101868502)

[Algorithm 8](#_Toc101868503)

[Lighting and Shading 9](#_Toc101868504)

[Rendering Volumes 10](#_Toc101868505)

[Point-cloud Rendering 11](#_Toc101868506)

[Constructive Solid Geometry 11](#_Toc101868507)

[Fractals 12](#_Toc101868508)

[Research Methodology 13](#_Toc101868509)

[Analysis 13](#_Toc101868510)

[Project Management 13](#_Toc101868511)

[Project Planning 14](#_Toc101868512)

[Platform 14](#_Toc101868513)

[IDE 16](#_Toc101868514)

[Version Control 16](#_Toc101868515)

[Design 17](#_Toc101868516)

[Project Management 17](#_Toc101868517)

[Project Planning 17](#_Toc101868518)

[Platform 19](#_Toc101868519)

[IDE 19](#_Toc101868520)

[Version Control 19](#_Toc101868521)

[Additional Software 19](#_Toc101868522)

[Implementation 21](#_Toc101868523)

[Development 21](#_Toc101868524)

[Application Framework 21](#_Toc101868525)

[Basic Ray Marcher 21](#_Toc101868526)

[Lighting / Shading 24](#_Toc101868527)

[Constructive Solid Geometry 27](#_Toc101868528)

[Materials 28](#_Toc101868529)

[Fractals 30](#_Toc101868530)

[Ambient Occlusion 32](#_Toc101868531)

[Skybox 33](#_Toc101868532)

[Testing 33](#_Toc101868533)

[Results 34](#_Toc101868534)

[Critical Evaluation 37](#_Toc101868535)

[Bibliography 39](#_Toc101868536)

[Image Bibliography 45](#_Toc101868537)

# Abstract

Ray marching is a method of rendering which uses only mathematics to determine the shapes of objects within a scene. This allows for visual effects which are very difficult to achieve using traditional methods of rendering such as rasterisation.

Along with the rendering of primitive objects, which are defined by their specified signed distance functions, ray marching enables the rendering of dynamic fractal objects, constructive solid geometry, and a mathematically infinite number of objects whilst remaining performant and real-time, due to the nature of using mathematics to determine the shape, position and visual effects applied to objects.

In the games industry, ray marching and SDFs are commonly used for effects such as volume rendering, as the algorithm allows for the sampling of a volume to render photorealistic clouds or other volumetric objects and is commonly paired with traditional rasterised rendering to create a hybrid renderer.

# Introduction

Recent advances in graphics processing hardware have brought ray tracing to the forefront of real-time rendering (Akenine-Möller et al., 2018). Ray tracing can be used to accurately simulate how light travels which can result in a higher fidelity render than traditional methods of rendering.

Whereas a ray tracing algorithm will perform a test for an intersection between a ray segment and objects in the scene to find the nearest intersection (Whitted & Foley, 1980), ray marching differs by stepping along a ray incrementally until an intersection is found.

A common ray marching technique is known as sphere tracing. Sphere tracing utilises signed distance functions (SDFs) to calculate the distance of each step along the ray. At each ray step, all objects are sampled for their distance to the current point, the ray can then be stepped forward by the shortest distance which ensures the ray will not travel inside, nor skip over, any object in the scene (Hart, 1996). Due to the type of SDF defining the shape of that object, it is possible to render fractals or modify the output distance to manipulate the shape or appearance of the object.

## Aim

This research aims to explore the areas needed to produce a rendering engine that utilises ray marching as a method of rendering a 3D scene and to take advantage of the characteristics of ray marching to achieve various visual effects.

## Objectives

To achieve this aim, the research will enable the development of:

* A performant, real-time ray marching renderer.
* Implementation of rendering effects only viable in real-time with ray marching.
* Various visual effects typically associated with ray-based rendering.
* An engine framework surrounding the renderer, providing a graphical user interface for users to control objects within the scene.

# Literature Review

## 3D Wireframe Rendering

Original real-time 3D video games, such as Elite (Frontier Developments, 1984), for example, used a wireframe rendering technique to render polygons. This resulted in the outline of polygons (triangles) to be rendered without being filled, contrary to modern rasterised rendering which does fill the rendered polygons (see *Rasterised Rendering* below).



Fig 1: Elite, Meijer (2016)

Elite’s non-horizontal lines were rendered using Bresenham’s line algorithm (Bresenham, 1965). Bresenham’s algorithm works by calculating the delta (difference) between positions *x1* and *x2* (*deltaX*), and the delta between positions *y1* and *y2* (*deltaY*), then for each position between *x1* and *x2* (*currentX*), the y position of the pixel (*currentY*) is incremented by *deltaY ÷ deltaX*. A pixel is plotted at the position *(currentX, currentY)* on each loop iteration. This technique would typically use real/floating-point numbers, but to avoid this, the algorithm used in Elite would multiply all real numbers by 256 so that 256 would be equivalent to 1.0 (Moxon, 2020).

After polygons were rendered to the screen, Elite would ‘undraw’ the polygons to clear the screen for the next frame. This is due to a full screen clear being too slow for the hardware of 1984, and double buffering was not feasible due to memory constraints. The implemented solution was for the line rendering algorithm to invert each pixel drawn, which allowed for the same routine to be used for drawing and un-drawing, and for the algorithm to be re-run after the frame had been rendered to minimise the number of pixels being cleared (Braben, 2011).

Convex shapes are shapes which will only return a maximum of two intersections when a straight line is passed through any part of the shape (Underground Mathematics, 2022), a ball is a convex shape, for example. However, a bowl is not. The hull of each ship in Elite is convex, as convex shapes cannot occlude themselves partially. This means that lines would not need to be half-rendered if part of the hull is partly occluding itself (Braben, 2011), thus increasing the render speed. A line draw call could be completely skipped if drawing the back-face of a polygon, which allows for the ship to appear to occlude parts of itself (as the polygons facing away from the camera would not be seen) and increase performance when rendering.

Elite also optimised the matrix multiplications required to project vertices into world and screen space by avoiding using multiplication operations, as CPUs in 1984 lacked the hardware for multiplication and division (Retro64, 2017), which meant multiplication and division operations were computationally slow. The method used for multiplication, without using a multiplication operation, was to use logarithms - a method of approximately multiplying and dividing two numbers with only a simple, and computationally fast, addition or subtraction and looking up a pre-defined log and antilog table (Sears, 2020). Ships in Elite were also symmetrical, sometimes in more than one axis, which allowed for values of matrices to be negated, resulting in the same vertex position as if the matrix had been calculated using the real negative position. This resulted in almost-computationally-free matrix multiplication for many vertices of the ships (Braben, 2011).

## Raycast Rendering

Wolfenstein 3D (id Software, 1992) was one of the original first-person shooter games but is not a true 3D game. Ray casting is a technique which renders a 3D projection of a 2D grid (Vandevenne, 2020). The ray casting algorithm casts a ray for each column of the screen from the player's perspective and is stepped across a 2D grid until an intersection is found. The distance in which the ray has travelled determines how many pixels are filled for that column of the screen (Sanglard, 2017).

|  |  |
| --- | --- |
| Fig 2: Rays being cast from the camera, Sanglard (2017) | Fig 3: Resulting rendered image, Sanglard (2017) |

Unlike its predecessors such as Hovertank 3D (id Software, 1991), Wolfenstein 3D’s environments featured textured walls which increased the technical complexity of the engine. As the game world is a stored as a 2D grid, the mantissa of the ray intersection point can be used for a horizontal texture coordinate, and the vertical texture coordinate can be calculated by dividing 1 by the current column height, for each vertical pixel drawn, the current vertical texture coordinate can then be incremented by that amount (Vandevenne, 2020).

An optimisation id Software used for Wolfenstein 3D to increase texturing performance was to buffer pixel columns if they were similar – columns are considered similar if they share the same horizontal texture coordinate - and batch render them later at the same height regardless of their actual distance from the player, which resulted in slight distortion, but less write operations (Sanglard, 2017).

|  |
| --- |
| comanche-maximum-overkill screenshot for dos  Fig 4: Comanche: Maximum Overdrive, abandonwaredos.com (2022) |

Other engines utilised similar ray casting techniques for different styles of rendering. The Voxel Space engine (Novalogic, 1992) rendered a 3D projection of a 2D heightmap, which allowed for the rendering of a high detail textured 3D terrain, featuring valleys, shadows, and man-made structures. The algorithm used, often referred to as floor casting or heightmap ray casting (Deschaseaux, 2018), begins by sampling the heightmap and colour map for each horizontal pixel, at the furthest point in the view frustum, and draws a vertical column of pixels of the sampled colour of a height determined by the sampled heightmap value (Macke, 2020). This algorithm repeats sampling increasingly closer to the camera, overriding previous pixels to guarantee occlusion, this type of algorithm is referred to as a painter’s algorithm (Hughes et al., 2013). An optimisation for this technique is to store a y-buffer for each screen column and sample the height and colour maps increasingly further from the camera, referencing the y-buffer to determine whether the current column is entirely or partly occluded, and drawing only the non-occluded portion of each column.

## Rasterised Rendering

Over time, graphics pipelines and hardware have improved to meet the market demand for higher visual fidelity in real-time applications. Dedicated graphics processing hardware, also known as graphics processing units (GPUs), gained popularity in the 1990s and has since evolved to be programmable (Verschelde, 2021) and include various hardware acceleration chips (Walton, 2020) alongside the ever-improving hardware for efficiently rendering triangles – for rasterisation. The rasterisation rendering technique is the most common rendering technique for 3D real-time applications (Scratchapixel, 2015).

The rasterisation process begins by projecting vertices into screen space via the vertex shader. The vertex shader multiplies each vertex position by the World and View transformation matrices, which brings each vertex into world space and then into space relative to the camera respectively. Finally, the camera space vertex position is multiplied by a Projection matrix which brings the coordinate into projection space (Akenine-Möller et al., 2008). The Projection matrix is where a perspective or orthographic camera effect can be applied, depending on how the Projection matrix is constructed.

The graphics pipeline goes through other programmable and fixed-function stages before eventually reaching the pixel shader. The pixel shader is another programmable stage, similarly to the vertex shader stage, that is run in parallel once for each pixel which is occupied by a triangle. The pixel shader, also referred to as the fragment shader, processes the data provided by the vertex shader (or an intermediate pipeline stage) and calculates a colour for that pixel (Caulfield, 2018).



Fig 4: Diagram demonstrating rasterisation of a triangle, raywenderlich.com (n.d.)

## Shaders

Graphics Processing Units are used for rendering, over CPUs, due to the focus on a parallel architecture which allows for much faster processing on specialised tasks such as rendering. A CPU has fewer, but more powerful cores, whereas a GPU has many smaller cores which allows for many simultaneous programs, known as Shaders, to be executed at once (Vivo & Lowe, 2015). The ability to execute many shaders at once is beneficial when the application requires a Shader to be executed once per pixel and/or per vertex.

Modern GPUs can be programmed by executing Shaders, which are created using C-Style languages such as HLSL or GLSL which are then compiled to an intermediate language to be executed (Akenine-Möller et al., 2008). GPUs were originally not programmable and were entirely fixed function (Tamasi, 2008), but over time have become more flexible and programmable, leading to the GPUs of today which have evolved to become more than hardware for rendering games. Although real-time rendering is their primary use, modern GPUs can be utilised as general-purpose parallel processors (Intel, 2021) and can be applied to many different applications outside of gaming/real-time rendering.

Since the dawn of programmable GPUs, other shader types have been introduced which provides developers with more control over the GPU and how it is used. After the introduction of programmable vertex and pixel/fragment shaders, a geometry shader stage was introduced which is responsible for creating new primitives from the output of the vertex shader. Compute shaders were also introduced but are not part of the graphics pipeline, they allow for general computation to be performed on the GPU (Rodríguez, 2013) and the highly parallelised architecture to be taken advantage of outside of graphics rendering.

## Ray Traced Rendering

Ray tracing is a rendering technique which attempts to emulate the basic principle of vision. Our eyes see the environment around us because of light rays reflecting from surfaces and hitting our eyes. Ray tracing flips the direction of these rays, projecting them from the camera (eye) and shading the pixel based on the lighting and material at the intersection point. This ray reversal is an optimisation so light rays which would not hit the camera (eye) are not considered (Deng et al., 2017).

Finding the intersection between a ray and an object is the fundamental operation of ray tracing (Glassner et al., 1989), the intersection point paired with the surface normal allows for various visual effects to be simulated by bouncing the ray from surface to surface, similarly to how light bounces from surface to surface in reality. Ray tracing algorithms often use a recursive trace function which runs for a specified number of bounces (Buck, 1999). This allows the colour of the pixel to accumulate every time the ray bounces from one object to another. The accumulation of the colour through the recursive function allows for effects such as the reflections of a surface also having their own reflections (Szirmay-Kalos et al., 1995).

A picture containing plate, several, surrounded

Description automatically generated

Fig 5: Recursive ray traced reflections, Kuri (2018)

Due to the general computational expense (Rinker, 1991) required to utilise ray tracing in real-time applications, ray tracing was not a viable technique for mainstream video games to integrate for an improvement in fidelity until NVIDIA released their 20 series GPUs, which used the Turing architecture containing new ray tracing (RT) cores for hardware-accelerated ray tracing (Kilgarif et al., 2018). Many games utilise these cores to implement a hybrid-rendering approach, which renders the scene with traditional rasterised techniques and utilises the RT cores for visual effects which provide the most benefit, such as reflections or diffuse global illumination (Sjoholm, 2018).

## Ray Marching

### Algorithm

The fundamental concept of ray marching is to step along a ray in a specified direction until an intersection has been found. The naïve approach to implementing this would be to step along the ray linearly using a fixed step size (Biagioli, 2016), this algorithm is simple but introduces potential problems. The accuracy of this algorithm is tied to the step size – a shorter step is more accurate but computationally more expensive – which means a perfectly accurate step size is unachievable.

One of the potential problems with using a linear step size is that an intersection can be found inside the geometry, as seen in the following visual representation:

Fig 6: Linear ray overstepping into an object

Another problem is that geometry can be entirely stepped over, meaning it would not be rendered. This would occur when trying to render, for example, a thin wall or an object with a high amount of fine detail (such as a fractal).

Fig 7: Linear ray stepping through narrow object

An improved algorithm is to ray march across distance fields, where the step size at each increment is determined by the shortest distance from the current position to any object in the scene. As the ray approaches a surface, the distance travelled becomes increasingly shorter (Hart et al., 1989), when the distance becomes lower than a specified threshold, the ray is considered to have intersected with the geometry. This method provides increased accuracy as it is impossible for the ray to pass inside, or through, any geometry. This method is referred to as sphere tracing.

The following diagram is a 2D visual representation of sphere tracing, each circle’s radius represents the shortest distance to a surface at each step, which is how far the ray is stepped at each iteration along the direction of the ray.

Fig 8: 2D representation of sphere marching

### Lighting and Shading

Shadow mapping with a rasterised render pipeline requires extra render passes to render scene depth from each light’s point-of-view. The depth maps are then bound to the pixel shader and used to determine whether the current pixel is in shadow. When rendering using rays, rendering shadows is a much simpler process: Upon the detection of an intersection, rays are cast toward each light from the intersection point; if there is an intersection between the light and the intersection position, the pixel is in shadow (Benton, n.d.).

This method can be expanded upon to emulate soft shadowing by dividing the shortest distance to the scene by the distance the shadow ray has travelled when calculating the shadow strength, which attenuates the shadow as its source becomes farther from the illuminated point (Benton, n.d.).

|  |
| --- |
| A picture containing black, white, dark, tableware  Description automatically generated  Fig 9: Fast AO vs. Real AO, Zucconi (2016) |

One of the advantages of ray marched rendering is the amount of information returned for every ray which is cast. This data can be utilised for a computationally fast ambient occlusion (AO) approximation. The number of ‘steps’ taken by the ray marching algorithm, upon intersecting the scene, can be divided by the pre-determined ‘maximum steps’ value, the resulting value can be used to determine how much AO the current pixel has. Alternatively, real ambient occlusion can be applied by linearly ray marching (i.e., not sphere tracing) in the direction of the surface normal for a pre-determined number of steps, with the distance from the scene being summed in a separate variable for each step. The ‘sum’ can then divide by a value of the maximum number of AO steps multiplied by the AO step size, the resulting value represents the AO strength of the current pixel (Zucconi, 2016).

The Phong illumination model is a popular model used with rasterised rendering to determine how much light each pixel has. The result of the Phong model is a diffuse shaded (self-shadow) object with a specular highlight. The Phong model can be represented using the following mathematical equations (Trebino, 2017):

The ambient value is calculated using an ambient intensity () and an ambient colour ().

Fig 10: Ambient lighting equation

The diffuse value creates the sum of each light, using the light’s intensity (), the direction from the pixel position to the light (), the surface normal () and the diffuse light colour ().

Fig 11: Diffuse lighting equation

The specular light value is similarly a sum of each light, using the light’s intensity (), the view direction (), the reflected vector between the surface-to-light direction against the surface normal (), the specular power or shininess () and the specular colour ().

Fig 12: Specular lighting equation

The final light value can then be calculated using the previous values and the pixel colour ().

Fig 13: The final lighting value equation

Other Bi-directional Reflectance Distribution Functions (BRDF) can improve surface shading by more accurately modelling how much light is reflected from a surface in a given direction based on how much light is emitted from another direction (Boksansky, 2021). BRDFs can be grouped with both a BTDF (Bi-directional Transmittance Distribution Function) and a BSSDF (Bi-directional Surface Scattering Distribution Function, also known as a Bi-directional Absorption Distribution Function) to further increase visual accuracy and fidelity (Papas, 2010).

A picture containing text, antenna

Description automatically generated

Fig 14: Light reflectance distribution of diffuse (left) and specular (right) surfaces. BRDF values in blue, cosine weighted BRDF contribution (scattered energy/exiting radiance contribution) in red, Boksansky (2021)

### Rendering Volumes

Ray marching is commonly used for the real-time rendering of volumetric objects, such as clouds. Volumes can be simulated using traditional rasterised rendering techniques by layering multiple flat surfaces, but this lacks visual fidelity especially when up-close or inside the volume.

Using ray marching for volume rendering allows for a much more realistic representation of the volume, as the ray can sample density multiple times from within the volume and calculate the colour for the current pixel (Häggström, 2018). The technique of rendering volumetric objects with ray marching has been used alongside traditional rasterised rendering for games (Schneider & Vos, 2015).

### Point-cloud Rendering

Point-cloud rendering is where a collection of points in space are represented visually, instead of in an interconnected topology (Shahrabi, 2019). There are various reasons why a point-cloud rendering technique would be used, in the case of this research it could be used for increased performance – per-pixel rendering requires each pixel to be ray marched, whereas point-cloud rendering does not require a ray for each pixel – and aesthetic purposes. Media Molecule utilises SDFs for rendering in their game “Dreams”. The game allows the user to sculpt objects and apply effects to them. A point-cloud rendering technique is used to achieve a unique painterly effect by rendering the points as paint splats (Evans, 2015). The source of the following images can be found in appendix 3.

|  |  |
| --- | --- |
| A picture containing indoor, decorated  Description automatically generated  Fig 15: High-density point-cloud, Evans (2015) | A map of the world  Description automatically generated with medium confidence  Fig 16: Low-density point-cloud, Evans (2015) |

### Constructive Solid Geometry

Constructive solid geometry is a method of creating complex objects from simple primitives using Boolean operations. A Boolean operation takes two objects – or in the case of ray marching SDFs, the distance to two objects – and returns a new result, a new distance value (Wong, 2016).

There are three primitive Boolean operations: Intersection, union, and difference/subtract. Intersection returns only the space in which both objects occupy. Union returns both objects combined. Difference returns a subtraction of the second object from the first object. The following image only uses three primitive types of objects, but results in a relatively complex object:



Fig 17: CSG demonstration, Jacobson (2016)

### Fractals

Fractals are commonly found in nature, such as in snowflakes, plants, coastlines, and galaxies. Benoit Mandelbrot, the person who coined the name ‘Fractal’ and pioneered early research into fractals, explored the occurrence of fractals in nature in his book ‘The Fractal Geometry of Nature’ where he states *“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. More generally, I claim that many patterns of Nature are so irregular and fragmented, that, compared with [standard geometry,] Nature exhibits not simply a higher degree but an altogether different level of complexity.”* (Mandelbrot, 1982). An example of a computer-generated fractal which mimics nature is the Barnsley Fern, which was first described by Michael Barnsley in his book ‘Fractals Everywhere’ (Barnsley, 1993), where the drawn fractal closely resembles a black spleenwort fern.

As described by Della-Bosca et al. (2014), virtual game worlds are dominated by the Cartesian concept of space. Built rectilinearly, polygon-by-polygon, into a collective, familiar experience. The natural world contrasts this as it is filled with rough edges, asymmetries, and complex interactions. Fractals can provide rich and interesting visuals, not only for the real natural world but also in virtual worlds, particularly to provide the ability for shapes to appear more natural.

Ray marching enables the rendering of 3D deterministic fractals, which can provide unique, complex, and visually appealing geometry in games as explored by the open-source physics video game ‘Marble Marcher’ (HackerPoet, 2019). The goal of the video game is to traverse various fractal terrains to reach the flag in the shortest time possible. Many of the terrains featured in Marble Marcher appear organic, whether in a familiar, earth-like manner or alien in appearance. Others appear industrial, with straight edges and grid patterns.

# Research Methodology

## Analysis

### Project Management

For this research, three project management frameworks have been considered. The reasonings for choosing between Waterfall, Kanban, and Scrum, are that each framework offers advantages and disadvantages, and they are each unique. Kanban and Scrum are both Agile frameworks, yet they are both very different in how they operate, which is why both are considered.

#### Waterfall

The Waterfall methodology is a linear approach to project management. Each stage of development must be complete before the next one begins as each stage depends on the previous one. Once a stage of development is complete, it should not be returned to.

The advantages of using the Waterfall methodology are that the structure is clear and easy to understand. The following phase of development cannot be started until the current one is finished. Another advantage is that the end goal is defined early in the project’s life cycle, which helps to keep the team aligned to the goal and prevents the project from growing out of proportion.

The disadvantages of using the Waterfall methodology are due to the linearity. It is difficult to make changes as it would deviate from the planned goal. Another disadvantage is that testing does not start until after completion. (Lucid Content Team, 2018)

#### Kanban

Kanban is an agile methodology that focuses on flexibility and adaptability. Agile frameworks allow for multiple stages of development to be in progress simultaneously, as the project is broken down into small segments which allow for iteration. Kanban uses cards to represent tasks, which can be moved from stage to stage of development, the ‘work-in-progress’ stage of development must have limits to how many tasks can be assigned simultaneously.

The advantages of using Kanban come from its flexibility. Due to each task having its own development cycle, there is constant progress being made even when other features may still be in an early phase.

Flexibility is also what gives Kanban disadvantages too, as an improperly maintained board can result in issues with development and the board can become overcomplicated. Kanban also does not have specified time-frames alike Scrum, which can more easily lead to delays or roadblocks. (PLANETTOGETHER, 2019)

#### Scrum

Scrum is also an agile methodology, which allows for tasks to be completed incrementally and simultaneously. Scrum differs by using sprints of a specified time (which varies based on the project) to complete tasks. Tasks are then reviewed as a team following the sprint, where they can be evaluated, and team members can sync to avoid any interference with development progress. A scrum master is assigned to ensure the team is following the correct processes and to remove obstacles.

Scrum offers more transparency and visibility across the entire workflow than other frameworks, which is an advantage as changes are easy, and the team is always in sync. This leads to improved productivity for a team.

The disadvantages of using Scrum for this research project stem from Scrum having many different roles. Scrum can only be taken full advantage of when working in a team, and due to this research being a solo project, Scrum would be ineffective (PLANETTOGETHER, 2019).

#### Scrumban

Scrumban is a project management framework which combines features of two agile methodologies, Scrum and Kanban, by merging the structure and predictability of Scrum with the flexibility and continuous workflow of Kanban (ProductPlan, 2022).

Scrumban works by keeping iteration periods (sprints) shorter, allowing for the team to adapt quickly to a changing environment, removing the disadvantage of being unable to quickly adapt introduced by using Scrum. Scrumban does this by using the work-in-progress limit present in Kanban workflows, when the number of work-in-progress tasks falls below a pre-determined amount, a planning event is held where tasks for the next iteration are added to the to-do section of the board, starting the next iteration (Germanov, 2022).

### Project Planning

#### Trello

Trello is a web-based project management service which is free to use. Trello has an array of features to assist with project management, with a minimal and easy-to-use GUI, making the service simple to begin working with. Trello also provides a mobile app which allows for a Kanban board to be easily updated if away from a computer, and boards can be shared with users outside of the team for viewing-only.

The disadvantages of Trello are that the free version does not include time-tracking, so each task is required to be manually timed outside of Trello. This is a feature which other alternatives such as Microsoft Planner offer (nTask, 2021), which is useful for sprints if following a project management framework such as Scrum.

#### Microsoft Planner

Microsoft Planner is a web-based project management service which is included with Office 365. The project management tool offers a variety of features, including time-tracking and an embedded calendar which is useful for keeping the project on schedule.

The disadvantages of Microsoft Planner are the Office 365 subscription fee required for using the software, and the inability for users outside of the team to access and view the board (nTask, 2021).

#### GitHub Projects

GitHub Projects is a beta web app which is functionally similar to Trello, also having a Kanban card-based interface which is simple and intuitive. GitHub Projects have the advantage of being integrated into GitHub, meaning if GitHub is being used to host a repository, it is convenient to keep the entire project on one service and can seamlessly integrate with managing repository issues (Pollard, 2017).

The disadvantage to using GitHub projects is that the software is in beta, meaning bugs are expected to occur and the software may change during use. Being closely integrated with GitHub can also be seen as a disadvantage to some, as Trello can be broader and accommodate non-code tasks (Lawlor, 2017) due to its separation from the repository.

### Platform

Three platforms have been considered for the development of this project. Each platform is considerably unique and offers advantages and disadvantages. DirectX11 and Vulkan are both graphics APIs that would allow for the creation of a rendering engine or other applications which need to utilise graphics hardware. Unity differs, as it is already a complete real-time development platform (Unity Technologies, 2021).

#### Unity

Unity is a real-time development platform that includes tools to assist the creation of games or other real-time rendered media. Ray marching in Unity would be possible, as it allows for user-made shaders and control over the rendering pipeline via the scriptable render pipeline.

The advantage of using Unity for the development of this project would be the collection tools already built into the engine, which could be utilised with the ray marching renderer to provide the user with a familiar environment.

These tools and extra and unused features could lead to overhead on performance, which would be one of the disadvantages of using Unity for this project. This would not likely be a problem if using DirectX or Vulkan as the engine would be built from the ground up to only include what is necessary.

An alternative to Unity would be Unreal Engine, which offers similar functionality being a complete development platform, but lacks features provided by Unity which would provide useful for this research project. The first limitation would be the lack of support for scriptable render pipelines, which could be a useful feature for writing the ray marching renderer. Another feature difference between Unity and Unreal Engine is the handling of shader code, all shader code in Unreal Engine is implemented through the provided visual scripting tool (to which a custom shader code module is available but with a limited development environment), whereas Unity offers support to use standard shader files, which allows for the use of fully-featured development environments and debugging tools when writing shader code.

#### DirectX11

DirectX11 is an API that can be used by an application to utilise graphics hardware, it supports multiple shader types, such as pixel and compute shaders (Microsoft, 2020), which could be utilised for ray marching. An alternative to DirectX11 would be OpenGL, which is comparable in terms of functionality and design.

The advantages of using DirectX11 are that there would not be unused feature overhead that could be present when using Unity. DirectX is relatively (to Vulkan) high level, which means development progress would be faster than using an API such as Vulkan.

The disadvantages of using DirectX11 are that there would be more work than if using Unity, as some of the existing tools would have to be made from scratch. The performance also may not be as fast as if the project was developed using a lower-level API such as Vulkan, performance advantages of Vulkan will be further covered in the following Vulkan consideration.

#### Vulkan

Vulkan is also an API to utilise graphics hardware, it differs from DirectX11 as it is much lower-level and verbose (NVIDIA, 2021), giving the developer more control over the entire rendering process. An alternative to Vulkan would be DirectX12, which is similarly low-level and too offers fine control to the developer.

The advantage of using Vulkan, over another API such as DirectX11, is that the finer control which the API provides can lead to improved performance, as driver overhead is mitigated due to driver software having to make fewer assumptions on what the application is doing. Another performance advantage Vulkan provides is the batch processing feature, which can help to reduce CPU load (Furmaniak, 2018).

This control is also one of Vulkan’s disadvantages. Having more control over the API could cause the program, if improperly utilised, to operate slower than a higher-level API. The verbosity of the code also makes using Vulkan much more complex than a higher-level API, which is likely to slow the development progress.

### IDE

#### Visual Studio 2022

Visual Studio 2022 is the most recent version of the software which includes new features over previous versions, such as an upgrade to 64-bit architecture which provides increased IDE performance, better cross-platform development tools and the latest C++ build tools which support C++20 (Microsoft, 2021c), allowing for the use of modern C++ features.

Visual Studio also supports additional extensions which increase the number of features the IDE can provide, most notably extensions such as ReSharper by JetBrains (JetBrains, 2022), which provides a vast array of new features and overall improves the development workflow when using Visual Studio, and HLSL Tools which improves the experience of writing shaders in Visual Studio by enabling code auto-completion and syntax error highlighting.

#### CLion

CLion (JetBrains, 2022) is an IDE which is developed by JetBrains, developers of the aforementioned ReSharper. CLion is considered due to the known quality of other products developed by JetBrains and its direct focus on being a C/C++ IDE.

CLion, being developed by JetBrains, offers many of the advantages which ReSharper brings to Visual Studio, such as improved refactoring and many smaller quality of life improvements. CLion is also cross-platform, which can be useful if developing for multiple platforms for not having to adjust to a new workflow/IDE.

#### Code::Blocks

Code::Blocks is a cross-platform C/C++ and Fortran IDE which is designed to be extensible and fully configurable (CodeBlocks, 2022).

The advantage of using Code::Blocks is that the software is open-source, allowing modifications to the IDE if desired. IDEs such as Visual Studio are fully closed source and do not allow for the same level of modification.

### Version Control

#### Git

Git is the most popular version control system with an 80% market share (Johnson, 2015), and is a distributed system meaning a copy of the repository and full commit history is cloned to each device.

Git is popular due to its fast speeds, offline compatibility, and ability to easily undo any mistakes which are made due to data rarely being deleted (Günther, 2020).

Due to the popularity of Git, many repository hosting services are available, most notably GitHub, but other popular hosting services include GitLab and BitBucket. Another benefit of Git’s popularity is the number of online resources for learning Git or solving any problems which may occur.

#### Subversion (SVN)

Subversion is a centralised version control system, which means it is connected to a central repository compared to distributed systems which can be used offline. The advantage of a centralised system is that upon a computer crashing, code would still be available on another machine (SlikSVN, 2022). SVN also allows for partial checkouts, so it is not required for the entire repository to be cloned at once (Perforce, 2018) which has the benefit of SVN handling large files better than distributed systems such as Git and Mercurial as the entire history of the repository is not required to be checked out.

However, Git’s branching system is considered to be better due to it only referencing a specific commit, making the branch lightweight and powerful (Perforce, 2018).

#### Mercurial

Mercurial is a distributed version control system similarly to Git, and thus shares many similarities to Git. However, Mercurial is considered to be easier to learn than Git and is faster on Windows than Git (Johnson, 2015), but Git being the more popular VCS has a higher number of resources available (80% market share compared to Mercurial’s 2% (Johnson, 2015)), and Git has a wider selection of repository hosting options available because of the higher market share. Git also has an improved branching system over Mercurial which allows for editing the history of commits, which prevents the user from merging code incorrectly (Johnson, 2015).

## Design

### Project Management

Kanban will be the methodology used for research and development throughout this project.

Waterfall will be too restrictive for this project due to its linearity. There would be a risk of planning and researching more than is viable to implement within the time constraint of the project, which is not a problem that is present for other, agile, methodologies.

The Scrum methodology is well-suited for use in a team, as it allows for iteration and simultaneous stages of development whilst keeping all members in sync. Due to the nature of this research project being a solo effort, Scrum would not be suitable.

Scrumban would be unsuitable for this project for the same reason as Scrum. As it is a solo research and development project, Scrumban would be ineffective due to the inability to assign different roles to separate people.

Kanban is a flexible methodology that doesn’t require a team to be fully taken advantage of. Tasks can be researched and developed simultaneously, which mitigates the risk of planning more than can be implemented within the time constraint.

### Project Planning

#### Kanban Board

The project will be planned and managed using Trello due to Trello’s simplicity and wide selection of extensions. A Kanban board can be created with Trello to break the project into individual tasks and subtasks which gives an overview of the status of the project. The task limit, per stage, will be two tasks. This will prevent new features from being worked on before other features are finished but allows for a degree of flexibility and simultaneous development.

The Trello project has four lists to categorise the stage of each task: To Do; Pending; Blocked; Complete. There is a fifth list to contain extra project resources.

The ‘To Do’ stage will be for features which have not yet started and are where all tasks begin. Once a task has begun development, it will be moved into the ‘Pending’ stage. The ‘Pending’ stage is where the task limit is applied. The blocked stage will contain tasks which are not yet finished and cannot be finished yet. For example, if Task A could not progress any further without the completion of Task C, Task A would then become blocked until Task C is complete. Finally, tasks are moved into the ‘Complete’ stage once each subtask is finished and there is nothing more which is required.

Graphical user interface, application, Teams

Description automatically generated

Fig 18: Initial Kanban board

#### Gantt Chart

Fig 19: Gantt chart for Semester 1

Fig 20: Gantt chart for Semester 2

### Platform

The platform on which this project will be built is DirectX11, as it strikes a balance between being lightweight to avoid unnecessary overhead, but high-level enough that the project will be able to progress at a reasonable rate.

Using DirectX11 would also enable research into implementing other standard features of a game engine, for example, Gizmos, Scene Management, Camera Controls, etc. which Unity would have otherwise handled.

### IDE

Visual Studio will be used for the development of this project because it supports multiple programming languages (should they be required for additional tooling) and build tools, is widely used within the industry, and, like the other considerations, offers a free non-commercial license (or student license) which can be used for this research project.

Additional benefits offered by other IDEs, such as cross-platform support and open-source are not required for this research project, as Windows will only be used for development and Visual Studio paired with ReSharper and other plug-ins offers the desired functionality without needing to modify the source to extend the development environment.

### Version Control

The software used for version control will be Git, paired with GitHub for hosting the code repository. Git has been chosen due to its wide industry use over Mercurial, meaning there are many resources for learning to use Git and a variety of services which host Git repositories for free. Git was chosen over SVN due to the ability to work offline, thus improving the workflow of the project and ensuring minimal interruption to research and development.

### Additional Software

#### C++

The language of choice for the implementation of this research project is C++. The C++ language is cross-platform and allows low-level interaction with the hardware. An alternative to C++ would be the C language, but C++ is preferred as it allows for object-oriented design (DataFlair, 2020) which would be advantageous for structuring the codebase.

C++ is natively supported by major graphics APIs (KhronosGroup, 2022; Microsoft, 2021a; Apple, 2022) which gives an advantage over many other languages which offer similar low-level access and performance but do not offer a direct implementation of graphics APIs and other useful libraries.

#### ImGui

ImGui (Cornut, 2021) is an open-source C++ graphical user interface library, which is suited for use in real-time applications. The library has many advantages over alternatives such as Nuklear (Mettke, 2022) by providing an extensive number of features, wide use adoption across the games industry, and is trivial to integrate and use. As ImGui is open source, many add-ons are available which integrate seamlessly into the library and provide extra functionality. For example, ImGuizmo is an add-on which will be used later in the project, for the implementation of game object and camera control gizmos.

#### DirectXTK

DirectX Tool Kit (Microsoft, 2022) will be used for the base template code which handles window instantiation, DirectX resource creation and basic application structure. DirectXTK also offers a collection of helper classes for writing DirectX code, such as SimpleMath, which is a wrapper for DirectX XM Math; SpriteBatch, for simplified text rendering; Keyboard and Mouse input tracking classes; and more, all of which streamline the development process when working with DirectX 11 or 12.

Another popular alternative to DirectXTK is bgfx (bkaradzic, 2022), a multi-platform library which supports multiple backends, including DirectX 11, OpenGL 3.1+, and lower-level APIs such as Vulkan. DirectXTK was chosen over bgfx because the base boilerplate code is implemented using DirectXTK and the time investment to port the existing code to a new framework would not provide enough benefit to the development of the project.

# Implementation

## Development

### Application Framework

To begin developing the ray march renderer, a DirectX 11 framework was needed as a basis. The minimum requirements of this framework were: a component system, mesh rendering, shader compilation, camera matrix calculations, and a GUI. These features had previously been researched for the module ‘Advanced Graphics and Realtime Rendering’, so the base code for this engine was referenced from the existing codebase with only the minimum required features implemented. The project uses DirectXTK (Microsoft, 2021b) for the codebase template, which provides a general structure for the engine and fundamental features such as window handling.

Before beginning the ray marching implementation, a quad was rendered over the entire window. A pixel shader, containing the ray marching code, can then be executed for each pixel. This can be achieved by creating a quad with vertex positions ranging from minus one to one. In the Vertex shader, matrix multiplications would usually be performed to move the vertex into world, view, and projection space, if none of these calculations are performed, the quad will cover the screen. This quad is then rendered to an ImGui window which allows for it to be docked and moved to different regions of the screen.

Background pattern

Description automatically generated

Fig 21: Quad rendered to the viewport window. Texture coordinates output as a colour via the pixel shader.

A component system is used for keeping functionality and data modular by allowing objects to take ownership of classes and call common methods which each component can override and provide individual functionality for.

A *RayMarchObject* component is attached to all GameObjects which should be used by the *RayMarchingManager* component (which is attached to a separate GameObject) upon the render method being called. The *RayMarchingManager* component can then access and gather the relevant data from the objects which will be ray marched, such as its object type and parameters, and bind that to a constant buffer for usage in the pixel shader. The same method can be used for other object types, such as lights, which will then be sent to a separate constant buffer. The *RayMarchingManager* component also handles global render settings, such as maximum draw distance, and also binds the values to a separate constant buffer.

### Basic Ray Marcher

To ray march the scene, a sphere tracing algorithm is used for its improved accuracy over a linear step algorithm. For each pixel rendered, a ray direction needs to be calculated for the initial ray. This direction will determine visual aspects such as a perspective effect, the camera’s field of view, and the direction in which the camera is facing. The ray direction vector can be calculated using the following equations:

Fig 22: Ray direction calculation

is the transposed view matrix, which is calculated using the DirectX XMMatrixLookAtLH() function. and are the current floating-point screen coordinates ranging from -1 to 1. is the camera’s field of view value in radians (Scratchapixel, 2014). is the normalised ray.

Once the ray direction had been calculated, the ray marching algorithm could begin. An SDF is needed to test the algorithm, so a sphere was used as it is one of the simplest distance functions and would be immediately clear if there were any problems. The distance to a sphere (Quilez, n.d.) can be calculated using the following equation:

Fig 23: Signed distance function of a sphere

The ray marching loop can then begin. The ray is stepped along by the distance to the sphere at each point. On each iteration of the loop, before the ray has been stepped forward, a check takes place to see if the distance to the sphere is less than a specified threshold value, if it is, the sphere is considered to have been intersected and the loop can exit. The following pseudocode summarised the algorithm:

1. depth **=** 0
3. **for** (i **=** 0, i < MAX\_STEPS, i**++**):
4. distance **=** SphereDist(origin **+** direction **\*** depth)
5. **if** (distance < THRESHOLD):
6. **return** true
8. depth **+=** distance
10. **return** false

Fig 24: Sphere tracing algorithm

Once this algorithm had been implemented in the pixel shader, a colour could be selected based on the Boolean return value and resulted in the following image:

Shape

Description automatically generated

Fig 25: Sphere with a radius of 1

The image is outputting the ray direction as a colour if no intersection is found, and a bright yellow colour if an intersection is found.

Only having a Boolean return value is very limiting, and there is much more intersection data to be utilised, so the ray marching function has been modified to return a struct containing other useful information such as position, depth, and the number of steps taken.

The following image is the output when the number of steps is used for the colour. It results in a Fresnel effect around the sphere as the algorithm must take more, shorter, steps to reach the sphere on the edges.



Fig 26: Sphere with a radius of 2, using number of ray steps as colour

Objects can be transformed, rotated, and uniformly scaled by modifying the position passed to a signed distance function (SDF).

Translation is performed by negating the desired object position from the position from which the ray marching algorithm is currently sampling the object distance (Quilez, n.d.). In the following equation, is the object SDF, is the current ray marching position, and is the translation vector.

Fig 27: Object translation calculation

Rotation is achieved by combining three two-dimensional rotations along each axis. The following equation represents a two-dimensional rotation (Agu, 2007). and are the two axis perpendicular to the axis being rotated around, and theta () is equal to the desired rotation amount for that axis.

Fig 28: 2D (single axis) rotation calculation

Non-uniform scaling distorts the Euclidean space (Quilez, n.d.), so only uniform scaling has been implemented. Scaling is applied using the following equation where is the object SDF, is the current ray marching position, and is the scaling value.

Fig 29: Object uniform scaling calculation

When ray marching multiple objects, every object in the scene must have its SDF sampled, and the shortest distance used in the ray marching loop. The nature of constant buffers requiring a fixed size caused performance issues when implementing support for multiple objects, which was able to be mitigated over several iterations.

The first implementation looped for the maximum number of objects in the scene, for each iteration checking if the object is active (and exiting the loop if not), then used a list of else-if statements to check which SDF the current object being sampled was using, sampled the distance, and finally checked whether that distance was shorter than the current shortest distance. The problems this method faced were the limited amount of SDF types available to use, as the functions had to be pre-determined and did not allow for the addition of new functions easily, and the slow performance that branching code brings when used in a shader. The many branches which determined which SDF to use for each object slowed the performance to an unusable amount with only a few objects in the scene.

The next iteration removed the branching code and instead sampled all SDFs for every object and selected the SDF which this object used – for example, if the object was a cube, there would be a sample taken for a sphere, cube, torus, and cone, but only the cube sample would be used – which increased performance greatly. This method performed well with a small number of objects, but performance deteriorated when the maximum object count was increased and would not scale well when more SDF types were added, especially if a more-complex SDF was added. This problem of not having the ability to easily add new SDFs also persisted with this method.

The final iteration of this system was to remove the loop through all objects on the C++ side and instead generate part of the ray marching shader at runtime (which returns the distance to the scene). The pixel shader now includes a separate *hlsli* file which is regenerated at runtime if a new object is added, the SDF type of an object is changed, or the Boolean operator of an object is changed (which will be implemented later). The generated shader includes a method which returns the closest distance to the scene, which contains a sequential list of SDF calls for each object in the scene rather than the previous loop through all objects. The advantage to using this method is that the SDF used for each object can be inserted directly into the shader where it is required, rather than needing to sample all SDFs and choosing the correct value, which removes the performance issue when scaling the number of SDF types and active objects. Generating the shader at runtime also means the SDFs can be inserted into the shader upon regeneration, allowing the user to define custom functions via the GUI, and also allowing for unused SDFs to be excluded from the shader to reduce recompile time.

Following the improvement of this system, the performance now scales better when increasing the number of objects and is generally much more stable. Adding more objects to the scene does lower the performance, which is expected, but performance now appears to be linked more toward how many objects are on-screen and how much of the screen the objects occupy. The *SDFManager* component (which generates the shader code) also now provides a GUI for the user to add, remove, or modify the SDFs available to add to the shader.

### Lighting / Shading

To implement effects such as lighting/shading, the surface normal of the intersection would need to be calculated. The gradient of the surface can be approximated by sampling points around the given intersection point and then returning the surface normal. This calculation can be represented by the following pseudocode snippet and equation (Wong, 2016), where is a defined epsilon value and is the distance function:

Fig 30: Approximate normal at a given point

1. float3 CalculateNormal(float3 p):
2. float2 e **=** { 0.001f, 0.0f }
4. float3 n **=** { SphereDist(p **+** e.xyy) **-** SphereDist(p **-** e.xyy),
5. SphereDist(p **+** e.yxy) **-** SphereDist(p **-** e.yxy),
6. SphereDist(p **+** e.yyx) **-** SphereDist(p **-** e.yyx) }
8. **return** normalise(n)

Fig 31: Pseudocode function to approximate surface normal at a given point

Using the calculated normal, Lambertian shading can be implemented by calculating the dot product between the surface normal and the normalised direction from the surface position to the light position (Steven M., 2020). The returned scalar value is then multiplied by the object colour, resulting in a shaded effect on the object.

The following equation represents the Lambertian shading calculation being performed in the shader where is the surface normal, is the position of the light, and is the position of the surface.

A picture containing shape

Description automatically generated

Fig 32, 33: Diffuse equation and lambert shading applied to sphere coloured by surface normal

The diffuse Lambertian shading algorithm can then be expanded upon by adding specular highlighting and ambient lighting by applying the aforementioned Phong shading model algorithms (Trebino, 2017), which gives the sphere a more realistic approximation of lighting.



Fig 34: Phong (diffuse, specular, and ambient) shading model applied to a sphere

The implementation of shadow casting/mapping is much simpler when ray marching compared to traditional shadow mapping techniques used while rendering with a rasterised pipeline. With rasterised rendering, typically the scene is rendered to a depth buffer from a light’s point-of-view, which is then sampled in a later render pass to determine whether the current pixel is visible from the light’s view, thus determining whether the pixel is in shadow or not.

Ray marching allows for the use of rays to determine whether the current pixel is in shadow (Benton, n.d.), as the following diagram demonstrates, when an intersection is found, another (shadow) ray is created which travels toward the light, if the ray intersects with an object before intersecting with the light source, the pixel is in shadow.

­

Fig 35: 2D representation of two example camera rays – one ray in shadow, the other illuminated by a light source

With this implementation, the results are comparable to the aforementioned technique commonly used with rasterised rendering pipelines. The following two images are scenes of similar composition, one using ray marching for rendering and the other using a rasterised rendering framework previously developed for the module ‘Advanced Graphics and Realtime Rendering’, which utilises the light depth buffer shadow mapping technique.

|  |  |
| --- | --- |
| Fig 36: Shadows rendered using rasterisation | Fig 37: Shadows rendered using ray marching |

The ray marching shadowing technique can be expanded upon to add a penumbra to the edge of the shadows, resulting in a soft-shadow effect and increasing the visual fidelity of the scene. Soft shadows can be a challenge to implement using rasterised rendering, especially when real-time performance is considered (Uralsky, 2005). The soft-shadowing effect is nearly computationally free for ray marching, adding only a single multiplication and division operation to the shadow ray marching loop.

The following equation represents the calculation used for implementing soft shadows (Quilez, 2010), this equation is performed for each iteration of the shadow ray marching loop and the result is stored in variable if the value is lower than the current value stored in , which is finally returned if the shadow ray does not intersect with the scene. The variable represents the shadow penumbra value which determines how sharp or soft the shadow is, the variable represents the distance from the scene, and the variable represents the total distance travelled by the shadow ray.

A picture containing logo

Description automatically generated

Fig 38, 39: Soft shadow equation and soft shadowing demonstrated on two ray marched objects

### Constructive Solid Geometry

Ray marching allows for the real-time creation of complex objects from multiple primitive objects using Boolean operations, the use of Boolean operations to create complex objects is called constructive solid geometry (CSG) (Segura et al., 2013).

Performing Boolean operations on a triangular mesh requires complex algorithms, which often fail to produce a closed, self-intersection free output, have limitations, and/or are a relatively expensive operation to perform in real-time (Zhou et al., 2016). However, Boolean operations are computationally free when ray marching (excluding the computational cost of sampling another SDF for the secondary object).

Referencing Quilez’s (n.d.) SDF article, when sampling distances for a scene, by default objects use the “Add” Boolean operator, where overlapping objects occlude each other based on the closest distance from the camera. Therefore, the pseudocode for sampling the scene distance would be as so:

1. float SceneDistance(float3 p):
2. float sphere **=** SphereSDF(p)
3. float cube **-** CubeSDF(p)
5. **return** min(sphere, cube) # Add

Line 5 of this function is what determines which Boolean operation is being applied between these objects. Changing the “min” to instead be “max” and negating one of the values will result in the “Subtract” Boolean operator being applied, where one object’s shape is subtracted from another, leaving an empty space in the first object of the shape and position of the second object.

1. float SceneDistance(float3 p):
2. float sphere **=** SphereSDF(p)
3. float cube **-** CubeSDF(p)
5. **return** max(sphere, -cube) # Subtract

Finally, to achieve the “Intersect” Boolean operator effect, the function can be changed so both parameters are passed into the “max” operator are left unmodified. Resulting in only the overlapping geometry between two objects remaining.

1. float SceneDistance(float3 p):
2. float sphere **=** SphereSDF(p)
3. float cube **-** CubeSDF(p)
5. **return** max(sphere, cube) # Intersect

Complex objects can then be created from simple primitive objects by combining these operators, the following image is an example of a complex object created from a sphere using the “Add” operator and three cylinders which are using “Subtract”.

Logo

Description automatically generated

Fig 40: A complex object composed of four primitive objects

### Materials

To allow for the appearance of object surfaces to be modified, a Material component is introduced which stores information regarding the object, which can be used in the shader to determine the colour of an object’s surface at the current pixel. The Material component contains three values: Colour, Metallic, and Roughness. Colour will determine the object’s base colour, metallic will determine how strong reflections are in the object, and roughness determines how glossy or matte the surface is.

Object colour was implemented, to begin with, which required a modification of the generated shader to return the index of the object that was intersected alongside the scene distance, this was so the correct material values could be referenced upon a ray intersection. The index was determined by comparing the shortest distance before and after each object SDF sample and updating the index to the most recent object’s index if the distance had changed. Once the index had been determined, the object colour could be retrieved and used in place of the existing code to determine the colour of the surface, which was previously representing the surface normal as a colour.

A picture containing background pattern

Description automatically generated

Fig 41: Three objects of different colours

Reflections were implemented by creating an additional ray in the reflected direction of the original ray direction against the surface normal (Fortin, 2021). The same ray marching code can be used for the reflection that is used for the initial ray, although a modification was made to lower the fidelity and accuracy of the reflection ray to increase the performance of reflections.

The result is mirror reflections on objects, which are interpolated between the reflection colour and object colour by the metallic amount of the object.

A picture containing shape

Description automatically generated

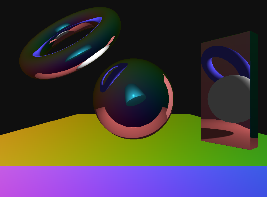
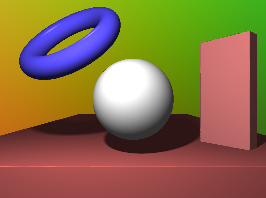
Fig 42: Three objects, with varying levels of metallic

The implementation of roughness determines how glossy or matte the surface of an object is. The roughness would affect the appearance of reflections, blurring them as the surface becomes less glossy, and affect the specular highlighting of the Phong shading, which was previously a pre-determined value.

Implementing a physically accurate directional diffuse reflection model would require creating multiple rays upon the intersection of the initial ray and determining their direction using a BRDF model (Bala, 2015), then averaging the result of each ray colour. However, this method would drastically increase the number of rays being created leading to a heavy performance decrease.

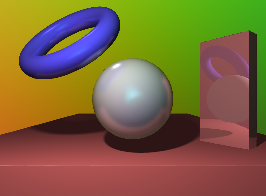
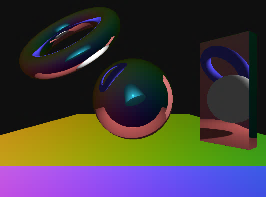
To avoid creating more rays than necessary, the implementation developed for this application only uses one ray for the reflection (mirror reflection) and renders reflections to a separate render target, alongside another render target for other data such as the metallic and roughness values of the object.

In a separate render pass, a compute shader blurs (Jon, 2015) the reflection layer by the roughness value and interpolates the base colour and blurred reflection layer by the metallic value, resulting in one image including diffuse reflections. The following diagram demonstrates the render pipeline:



First Pass (Pixel Shader)

Second Pass (Compute Shader)



*Colour Render*

*Reflections Render*

*Gaussian Blur*

*Combine*

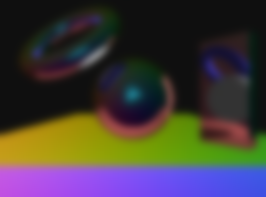
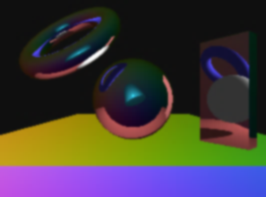


Fig 43: Reflections render pipeline diagram

The resulting render shows multiple objects each reflecting other objects in the scene with varying colours and surface types. Ray marching also allows for reflections of objects which are off-screen (as can be seen in the following image, a blue object behind the camera), something which is not possible with traditional rasterised rendering, using a common technique called Screen Space Reflections, which only allows for reflections of what is visible to the camera (Virtix Edge, 2018).

Icon

Description automatically generated with medium confidence

Fig 44: Scene demonstrating different materials

### Fractals

*Ray tracing deterministic 3-D fractals* (Hart et al., 1989) was the first paper to introduce distance-estimated 3D fractals, describing how distance estimation is used to render a Julia 3D Fractal, and how they are rendered using what is now known as ray marching (however the paper refers to the method as ray tracing).

A 3D Sierpiński triangle was implemented, to begin with – a fractal composed of a recursive triangle, or in 3D, a recursive tetrahedron – due to its visual simplicity compared to other fractals. Using the distance function provided in Christensen’s (2011) series of articles about ray marching fractals, the following result was achieved.

Shape

Description automatically generated

Fig 45: Sierpiński tetrahedron fractal

This distance to this fractal is calculated by iterating several times, each iteration determining the closest vertex of the tetrahedron to the ray position, scaling the system from the selected vertex, and then finally returning the distance to the closest vertex once all iterations are complete. The iterations result in the recursive tetrahedron effect, and although the tetrahedrons do not repeat infinitely, the number of objects grows exponentially as the number of iterations increases, resulting in over one million objects when iterating ten times (, or 1,048,576), which can be rendered in real-time.

An improvement on this algorithm would be to use symmetry, which can be achieved when ray marching by using the absolute value of the position passed to the SDF along the axis being mirrored (Quilez, n.d.). This is a common technique used with fractal distance functions (Christensen, 2011).

The following diagram demonstrates how the planes of symmetry of a tetrahedron are used to create the fractal, by mirroring across the symmetry planes of a tetrahedron. The object is then scaled with each iteration (like the previous algorithm) to achieve the recursive effect, before reducing the overall size to back a regular scale.

|  |  |
| --- | --- |
| Fig 46: Diagram demonstrating symmetry planes used in the updated Sierpiński algorithm, Christensen (2011) | Fig 47: Fractal rendered using updated Sierpiński algorithm |

Similar iterative algorithms have been developed (Costa, 2018) for other fractal types, such as the Mandelbulb fractal and Quaternion Julia fractal. Both of which were inserted into the shader using the SDFManager component GUI at runtime and then rendered. The following screenshots were taken after ambient occlusion was implemented, as it allows for more detail to be seen in the fractals.

|  |  |
| --- | --- |
| Fig 48: Mandelbulb fractal | Fig 49: Quaternion Julia fractal |

As fractals are rendered with the same method as other primitive objects, using an SDF, they are also compatible with other ray marching features such as CSG/Boolean operators. The following image is a Mandelbulb fractal subtracting from a cube.

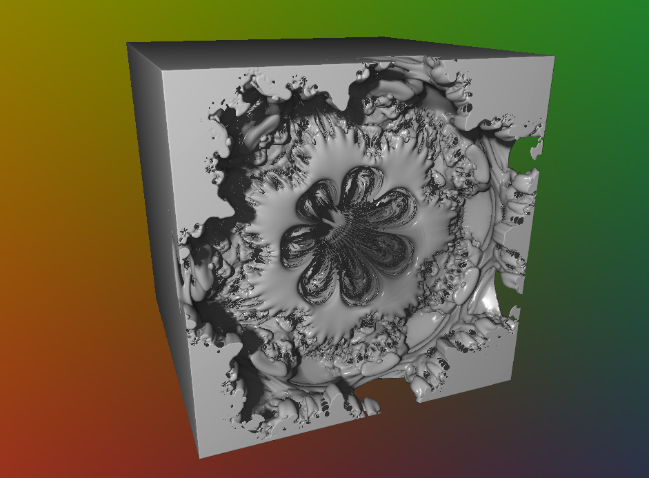


Fig 50: Mandelbulb fractal subtracted from cube object

### Ambient Occlusion

There are two common methods of implementing ambient occlusion (AO) when ray marching. One method is a computationally fast approximation of AO using the number of steps the ray has taken, whereas the other is a method that calculates the amount of occlusion by sampling the distance of nearby geometry, which is more computationally expensive due to the increased number of scene distance samples but is, physically, more accurate than the faster method.

A common AO technique when rendering using a rasterised pipeline is called Screen-Space Ambient Occlusion (SSAO), which uses the depth buffer to calculate the occlusion factor of the current pixel (Chapman, 2013).

The method implemented in this project is the fast ambient occlusion technique, the choice to use this calculation over the physically accurate ambient occlusion technique was made because the results between both methods do not differ dramatically, and to avoid adding unnecessary complexity to the pixel shader, which would negatively affect the frame rate.

The equation to calculate the occlusion factor for the current pixel is as follows (Zucconi, 2016), where is the number of steps the ray has taken, is the maximum number of steps the ray can take, and is the desired AO strength value.

Fig 51: Fast ambient occlusion equation

When rendering objects using only the calculated occlusion factor as a colour, the object details can be seen to appear with pronounced detail despite having no other shading applied.

|  |  |
| --- | --- |
| Fig 52: Object rendered with no lighting or AO | Fig 53: Object rendered only using the occlusion factor |

Ambient occlusion can be applied to the main render by multiplying the final colour by the calculated occlusion factor, the render using ambient occlusion provides much more fine detail and depth to objects, especially in shadowed areas, which can be seen in the following images:

|  |  |
| --- | --- |
| Fig 54: Shaded object without AO | Fig 55: Shaded object with AO |

### Skybox

The implementation of a skybox in this project is similar to the implementation of a rasterised render pipeline, due to how cubemap textures are sampled within a shader. Whereas textures are usually sampled using a UV coordinate, cubemap textures feature a third W value (UVW) for sampling. This is because cubemap textures contain six textures, one for each cube face, which requires a third value for the third dimension.

A feature of HLSL is to allow for the sampler to take a direction vector as the UVW value, used in rasterised renderers for reflecting a cubemap using an object normal (or skybox, by calculating the view direction). In the ray marcher, the ray direction can be used to sample the skybox in the same way when an object is not intersected, and the vector calculated for reflections can be used for sampling the reflection colour if an object's reflection ray does not intersect with another object.

A picture containing ball

Description automatically generated

Fig 56: Objects of varied materials with skybox

## Testing

This research project will be tested by visually comparing images rendered using the ray marching application with various sourced images, which will be used to determine whether the implemented features behave correctly and produce the desired results.

The visual nature of the project makes data-oriented testing ineffective due to the lack of valid data to compare. Using the frame rate of the application as a data point for testing performance was considered, but excluded due to the lack of eligible comparisons, as many features implemented with ray marching are not possible with rasterised rendering, therefore cannot be compared. A group of varied source images will be used to ensure all features and as many visual edge-cases are tested as possible.

To perform an individual test, each source image will be recreated within the ray marching engine as accurately as possible to ensure the test is accurate and to reduce false positives. Each image will be analysed, and any differences considered when concluding whether a test has passed, partially passed, or failed. Should a source image utilise a feature not implemented in the ray marching engine (e.g., Bloom, Depth of Field, etc.), the difference will be noted but not considered when deciding the final result of a test.

The tests use a Cornell Box as a testing environment. A Cornell Box is typically a simple, controlled environment used for computer graphics testing, commonly featuring two coloured walls and a light source on the ceiling of the box. A photo can be taken from inside the physical box and compared with a rendered image to identify any artefacts which do not occur in a real, physical environment.

A physical Cornell Box was constructed for the first graphics test, other tests then use a virtual pre-rendered Cornell Box to compare with rendered images from the ray marching engine. The physical box features three white surfaces, a green surface, and a red surface. The ceiling has a square opening which allows for a light source. A felt curtain is draped in front of the opening which prevents light, that is not from the light source, from interfering with the scene.

A picture containing indoor

Description automatically generated

Fig 57: Physical Cornell Box

## Results

#### Test 1

|  |  |
| --- | --- |
| Fig 58: Original photograph | Fig 59: Recreation in ray marching renderer |

The first test is a recreation of the physical Cornell Box built for graphics testing. Three objects were placed in the box of different shapes and surface types. The red sphere has a slightly reflective, but matte surface, which gives some highly diffused/blurred reflections. The white box is a completely matte surface and is not reflective. The torus is a slightly glossy surface which is not reflective. After comparing both images, it is clear that the ray marching render succeeds in some aspects but fails in others.

|  |  |
| --- | --- |
| A picture containing indoor, red  Description automatically generated  Reference | A picture containing indoor, light  Description automatically generated  Render |

An example of where the renderer succeeds would be the shading red ball. Though the lighting in the rendered scene is dimmer than in the photograph, the ball features accurate diffuse lighting, the specular highlight is positioned correctly, and the ball is reflecting some light from the white floor beneath it.

|  |  |
| --- | --- |
| A picture containing indoor, red  Description automatically generated  Reference | A picture containing indoor, light  Description automatically generated  Render |

Shadowing is a partial pass, as the original photograph has softer shadows than the render. The shadow softness can be modified but increasing the value too high would introduce shadow artefacts which occur as the light ray uses the maximum number of steps, which was common in this scene due to the light being located very close to the room’s ceiling. However, although the shadows are not as soft as they should be, the shadows do become softer in the correct areas, which is why shadowing is considered to be a partial pass.

|  |  |
| --- | --- |
| A picture containing indoor, red  Description automatically generated  Reference | A picture containing indoor, light  Description automatically generated  Render |

Ambient occlusion (AO) is an area where the renderer did not perform well, the renderer is using the ‘fast’ AO technique in this image, which uses the number of steps taken to darken the image and result in an ambient occlusion effect. However, the physically-based AO technique would be much more effective in this scene and provide a much more realistic effect. The lack of ambient occlusion is most evident when looking beneath the red ball but can also be noticed in the top corners of the room and beneath the torus.

#### Test 2

|  |  |
| --- | --- |
| Fig 60: Source render, Leeuwen (2020) | Fig 61: Recreation in ray marching renderer |

The second test uses more features of the ray marcher, most notably, mirror reflections. The scene features another Cornell Box, the reference image is a rendered image instead of a physical photograph. The box contains two spheres, one reflective and the other with a matte surface, and a cube which also has a matte surface. The light is an area light which spans the majority of the ceiling, an area light effect was attempted to be reproduced by using five lights spaced across the ceiling, as the ray marcher does not support actual area lighting.

|  |  |
| --- | --- |
| Reference | Render |

The ray marcher succeeded at reproducing accurate reflections, being almost identical to the reference image except for the area light. The object placement and shading within the reflections are accurate and the visual warping due to the rounded surface of the sphere is also identical to the reference image.

|  |  |
| --- | --- |
| A picture containing wall, indoor, red  Description automatically generated  Reference | A picture containing wall, indoor, desk, red  Description automatically generated  Render |

Shadowing is a partial pass, this is for the same reason as the previous test, as the shadows are not soft enough. The shadows could not be made softer without introducing artefacts; however, the shadows are somewhat soft and are cast in the correct areas.

|  |  |
| --- | --- |
| A picture containing wall, indoor, red  Description automatically generated  Reference | Render |

Indirect lighting is the primary difference between the two renders. Bounce lighting is not implemented within the ray marcher due to the expense of computing bounce light in real-time. However, bounce light makes the scene considerably softer and adds another degree of photorealism, the lack of bounce light is most predominant when analysing the purple sphere and red cube. An attempt to simulate the effect of bounce light can be performed using a slight reflection which is highly diffused, but it is not as effective nor a physically accurate model of bounce lighting. Therefore, indirect lighting is considered to have failed this test.

#### Test 3

|  |  |
| --- | --- |
| Fig 62: Source render, Wolfe (2020) | Fig 63: Recreation in ray marching renderer |

This test features reflective objects with a coloured surface and different amounts of surface roughness. The reference image also uses a skybox, which can be seen in the reflections of the objects. The original skybox could not be sourced, so a different skybox was used instead. Although the difference in skybox images will affect the appearance of reflections, it will not be considered when determining whether the areas of the test have passed or failed, as it does not affect the scene in a significant way. To avoid repeating what has been said in previous tests, the accuracy of reflections, shadow softness, and ambient occlusion will not be analysed in this test.

|  |  |
| --- | --- |
| A picture containing indoor, green  Description automatically generated  Reference | A group of colorful balls  Description automatically generated with low confidence  Render |

The ray marcher allows for a varying degree of roughness and reflections are effectively blurred based on the roughness values to give the effect of a matte surface.

|  |  |
| --- | --- |
| A picture containing indoor, green  Description automatically generated  Reference | A group of colorful balls  Description automatically generated with low confidence  Render |

The ray marcher allows for a varying level of metallic surface, which allows the surface to transition between the surface colour and the reflection colour. This allows objects to be reflective but also maintains a coloured tint. However, this test demonstrated a problem with the method used for this effect. Currently, the surface colour is interpolated between the original surface colour, and the reflection colour, based on the metallic value of the surface. The correct way of doing this should be to interpolate from the surface colour to the reflection colour multiplied by the material colour. Multiplying the reflection colour by the material colour creates a ‘rich’ tint, as seen in the reference image, instead of the ‘cloudy’ tint seen in the ray marched render, but still allows for the cloudy effect via the interpolation, if desired. This difference can most predominantly be seen in the green spheres and the large pink sphere.

|  |  |
| --- | --- |
| A picture containing indoor, green  Description automatically generated  Reference | A group of colorful balls  Description automatically generated with low confidence  Render |

A feature which the reference image appears to have, but not the ray marcher, is the ability to tint the reflected light to a different colour from the surface colour, as seen on the large blue sphere. The reflected light has a red tint whereas the sphere itself is blue.

# Critical Evaluation

This research and development project aimed to explore ray marching as a method of rendering primitive objects in real-time using signed distance functions. Ray marching was chosen because of the unique features that it enables for rendering - due to the algorithm solely using mathematics for rendering objects - when compared with other rendering techniques such as rasterisation or ray tracing.

The project succeeded in its initial goal of rendering a scene of primitives using ray marching and a variety of features are implemented which showcases where ray marching excels, such as through real-time constructive solid geometry and fractal rendering, though the results do have room for improvement when considering the photorealism of the images produced by the application, most notably the absence of physically-based ambient occlusion and indirect/bounce lighting. However, other graphical features such as reflections, and soft shadowing have been shown to produce visually appealing results.

Since beginning the project, I have learned a considerable amount about rendering, mathematics, and shader optimisation. Researching into the history of 3D rendering introduced me to many techniques I had not explored previously, as my previous knowledge was primarily regarding rasterised rendering, and introduced me to what was possible with ray marching and how it is applied in industry. The research and implementation of the ray marching algorithm has brought me to learn more about 3D mathematics. The implementation of features such as AO has led me to understand why such effects are implemented and the reason for their existence, in reality, I now have a much greater understanding of how light interacts with objects, which then assists with understanding the implementation of such features. Ensuring the application can run in real-time proved to be a great challenge during the development stage, so effective shader optimisation was crucial to ensure the renderer remained performant. Due to the much of development of this project being within shader files, I was able to gain a greater understanding of what to do, and what-not-to-do when writing shaders, and exploring methods of increasing the renderer’s performance was an interesting and educational process.

Were the project to be repeated, I would likely spend less time ensuring the application interface is user-friendly and instead opt for an approach which supplies the developer with an API that allows for the quick implementation of ray marching features via shader code. I believe that attempting to keep the program GUI-based, though an interesting and educational process, slowed the development of the project and used valuable time which could have been spent exploring more features of ray march rendering.

Other features which I would like to have included are transparent objects and refraction, which simulates the change in direction of light rays through an object such as glass. A common use for ray marching is to render volumetric objects such as clouds, which would have been an interesting topic to research as existing research on the topic has led to incredibly visually impressive results. Ray marching also allows for objects to be manipulated in ways such as bending, twisting, symmetry, infinite repetition, etc. by modifying the input or output of an SDF, the implementation of this feature would have been a relatively quick process, had there not been such a focus on the application being GUI-based, but ensuring the application stayed performant while providing the control for users to add these features in the GUI meant the feature would have taken considerably longer due to the number of changes required to the shader generation code, therefore, other features were prioritised.

The development process of this project has been an insightful experience and will prove valuable for future academic studies and my career overall. Not only for the specific skills learnt regarding rendering, mathematics, and shader optimisation; but also, for time-keeping and project management. This project has demonstrated the importance of a properly planned project and keeping to a schedule, should this project had been planned improperly, the result may have been much less than it currently is. If I would have had more prior experience in project management, the renderer could have contained more features or have been developed as an API rather than heavily focussing on a user interface.

In conclusion, I would consider this project to be an overall success, although the inclusion of more features would have been ideal, the project has allowed me to gain a solid understanding of the ray marching algorithm and its possibilities, as well allow me to improve many skillsets which will be useful for future projects. This research and development will serve as a basis which will enable me to conduct further research into more advanced features of ray marching and rendering with mathematics.

# Bibliography

Agu, E. (2007). *CS 543: Computer Graphics Lecture 4 (Part I): 3D Affine transforms*. [Online]. Available from: https://web.cs.wpi.edu/~emmanuel/courses/cs543/slides/lecture04\_p1.pdf. [Accessed: 30 March 2022].

Akenine-Möller, T., Haines, E. & Hoffman, N. (2008). *Real-Time Rendering: Third Edition*. Third. Taylor & Francis.

Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Hillaire, S. & Iwanicki, M. (2018). *Real-Time Rendering: Fourth Edition*. Fourth. [Online]. A K Peters/CRC Press. Available from: https://www.realtimerendering.com/Real-Time\_Rendering\_4th-Real-Time\_Ray\_Tracing.pdf. [Accessed: 25 October 2021].

Apple (2022). *Getting started with Metal-cpp*. [Online]. Available from: https://developer.apple.com/metal/cpp/. [Accessed: 20 January 2022].

Bala, K. (2015). *Ray Tracing Light Reflection, Illumination Hierarchies, Transforms, Advanced Rendering*. [Online]. Available from: https://www.cs.cornell.edu/courses/cs4620/2015fa/lectures/36ReflectionBVHAdvRTWeb.pdf. [Accessed: 6 April 2022].

Barnsley, M. (1993). *Fractals Everywhere*. 2nd Ed. AP Professional.

Benton, A. (n.d.). *Ray Marching and Signed Distance Fields*. [Online]. Available from: https://www.cl.cam.ac.uk/teaching/1718/FGraphics/1.%20Ray%20Marching%20and%20Signed%20Distance%20Fields.pdf. [Accessed: 8 February 2022].

Biagioli, A. (2016). *Raymarching Distance Fields: Concepts and Implementation in Unity*. [Online]. 1 October 2016. Available from: https://adrianb.io/2016/10/01/raymarching.html. [Accessed: 30 October 2021].

bkaradzic (2022). bgfx. *GitHub Repository*. [Online]. Available from: https://github.com/bkaradzic/bgfx. [Accessed: 18 March 2022].

Boksansky, J. (2021). *Crash Course in BRDF Implementation*. [Online]. Available from: https://boksajak.github.io/files/CrashCourseBRDF.pdf. [Accessed: 10 February 2022].

Braben, D. (2011). *Classic Game Postmortem - ELITE*. [Online Video]. 2011. Available from: https://www.gdcvault.com/play/1014628/Classic-Game-Postmortem. [Accessed: 30 January 2022].

Bresenham, J.E. (1965). Algorithm for computer control of a digital plotter. *IBM Systems Journal*. 4 (1). p.pp. 25–30.

Buck, J. (1999). *The Recursive Ray Tracing Algorithm*. [Online]. Available from: http://www.geocities.ws/jamisbuck/raytracing.html#RayTracingOverview. [Accessed: 20 January 2022].

Caulfield, B. (2018). *What’s the Difference Between Ray Tracing and Rasterization?*

Chapman, J. (2013). *SSAO Tutorial*. [Online]. Available from: http://john-chapman-graphics.blogspot.com/2013/01/ssao-tutorial.html. [Accessed: 8 April 2022].

Christensen, M.H. (2011). *Distance Estimated 3D Fractals*. [Online]. Available from: http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i/. [Accessed: 9 April 2022].

CodeBlocks (2022). *Code::Blocks*. [Online]. Available from: https://www.codeblocks.org/. [Accessed: 19 March 2022].

Cornut, O. (2021). Dear ImGui. *GitHub repository*. [Online]. Available from: https://github.com/ocornut/imgui.

Costa, J. (2018). Raymarched-Fractals. *GitHub Repository*. [Online]. Available from: https://github.com/JPBotelho/Raymarched-Fractals/blob/master/Content/DistanceFunc.cginc. [Accessed: 9 April 2022].

DataFlair (2020). *Advantages and Disadvantages of C++*.

Della-Bosca, D., Patterson, D. & Costain, S. (2014). *Fractal Complexity in Built and Game Environments*. In: [Online]. pp. 167–172. Available from: https://hal.inria.fr/hal-01408518. [Accessed: 12 December 2021].

Deng, Y., Ni, Y., Li, Z., Mu, S. & Zhang, W. (2017). Toward Real-Time Ray Tracing. *ACM Computing Surveys*. 50 (4). p.pp. 1–41.

Deschaseaux, P. (2018). *Heightmap rendering using a floorcasting algorithm*. [Online]. Available from: https://observablehq.com/@ehouais/heightmap-rendering-using-a-floorcasting-algorithm. [Accessed: 3 February 2022].

Evans, A. (2015). *Learning from failure: A Survey of Promising, Unconventional and Mostly Abandoned Renderers for ‘Dreams PS4’, a Geometrically Dense, Painterly UGC Game*. [Online]. Available from: http://advances.realtimerendering.com/s2015/AlexEvans\_SIGGRAPH-2015-sml.pdf. [Accessed: 2 November 2021].

Fortin, A. (2021). *Simple ray-marching reflection*. [Online]. Available from: https://antoinefortin.ca/uncategorized/simple-ray-marching-reflection/. [Accessed: 5 April 2022].

Frontier Developments (1984). *Elite*.

Furmaniak, L. (2018). *Stuck on OpenGL ES? Time to move on! Why Vulkan is the future of graphics*. [Online]. Available from: https://www.imaginationtech.com/blog/stuck-on-opengl-es-time-to-move-on-why-vulkan-is-the-future-of-graphics/. [Accessed: 24 November 2021].

Germanov, I. (2022). *Kanban vs Scrum vs Scrumban: What Are The Differences?* [Online]. Available from: https://ora.pm/blog/scrum-vs-kanban-vs-scrumban/. [Accessed: 17 April 2022].

Glassner, A., Haines, E., Hanrahan, P., Arvo, J., Cook, R., Heckbert, P. & Kirk, D. (1989). *An Introduction to Ray Tracing*. [Online]. Available from: https://www.realtimerendering.com/raytracing/An-Introduction-to-Ray-Tracing-The-Morgan-Kaufmann-Series-in-Computer-Graphics-.pdf. [Accessed: 19 January 2022].

Günther, T. (2020). *8 Reasons Why Git is so Popular*. [Online]. Available from: https://dev.to/gittower/8-reasons-why-git-is-so-popular-3mj4. [Accessed: 18 March 2022].

HackerPoet (2019). MarbleMarcher. *GitHub repository*. [Online]. Available from: https://github.com/HackerPoet/MarbleMarcher. [Accessed: 12 December 2021].

Häggström, F. (2018). *Real-time rendering of volumetric clouds*. [Online]. Umeå. Available from: http://www.diva-portal.org/smash/get/diva2:1223894/FULLTEXT01.pdf. [Accessed: 2 November 2021].

Hart, J.C. (1996). Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*. [Online]. 12 (10). Available from: https://www.researchgate.net/publication/2792108\_Sphere\_Tracing\_A\_Geometric\_Method\_for\_the\_Antialiased\_Ray\_Tracing\_of\_Implicit\_Surfaces. [Accessed: 26 October 2021].

Hart, J.C., Sandin, D.J. & Kauffman, L.H. (1989). Ray tracing deterministic 3-D fractals. *ACM SIGGRAPH Computer Graphics*. [Online]. 23 (3). Available from: https://www.cs.drexel.edu/~david/Classes/Papers/rtqjs.pdf. [Accessed: 30 October 2021].

Hughes, J.F., van Dam, A., Mcguire, M., Sklar, D.F., Foley, J.D., Feiner, S.K. & Akeley, K. (2013). *Computer Graphics: Principles and Practice*. Third Edition. Addison-Wesley Professional.

id Software (1991). *Hovertank 3D*.

id Software (1992). *Wolfenstein 3D*.

Intel (2021). *CPU vs GPU: What’s the Difference*. [Online]. Available from: https://www.intel.co.uk/content/www/uk/en/products/docs/processors/cpu-vs-gpu.html. [Accessed: 19 January 2022].

JetBrains (2022). *CLion: A cross-platform IDE for C and C++*. [Online]. Available from: Jetbrains.com/clion/. [Accessed: 20 January 2022].

Johnson, E. (2015). *Pros and Cons: Is Git Better Than Mercurial?* [Online]. Available from: https://content.intland.com/blog/sdlc/why-is-git-better-than-mercurial. [Accessed: 18 March 2022].

Jon (2015). *#5 Blur Shaders Part 2*. [Online]. Available from: https://xorshaders.weebly.com/tutorials/blur-shaders-5-part-2. [Accessed: 29 December 2021].

KhronosGroup (2022). Vulkan-Hpp. *GitHub repository*. [Online]. Available from: https://github.com/KhronosGroup/Vulkan-Hpp.

Kilgarif, E., Moreton, H., Stam, N. & Bell, B. (2018). *NVIDIA Turing Architecture In-Depth*. [Online]. Available from: https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/#:~:text=Turing%20GPUs%20introduce%20new%20RT,tracing%20approaches%20of%20the%20past. [Accessed: 19 January 2022].

Lawlor, M. (2017). *Off Topic: Github Projects? Trello? What do you use?* [Online]. 2017. Available from: https://discourse.roots.io/t/off-topic-github-projects-trello-what-do-you-use/10457/6. [Accessed: 19 March 2022].

Lucid Content Team (2018). *The Pros and Cons of Waterfall Methodology*. [Online]. 10 August 2018. Available from: https://www.lucidchart.com/blog/pros-and-cons-of-waterfall-methodology. [Accessed: 7 November 2021].

Macke, S. (2020). *Voxel Space - Terrain rendering algorithm in less than 20 lines of code*. [Online]. Available from: https://github.com/s-macke/VoxelSpace. [Accessed: 3 February 2022].

Mandelbrot, B. (1982). *The Fractal Geometry of Nature*. W. H. Freeman and Co.

Mettke, M. (2022). Nuklear. *GitHub repository*. [Online]. Available from: https://github.com/Immediate-Mode-UI/Nuklear.

Microsoft (2021a). *Direct3D 12 programming environment setup*. [Online]. Available from: https://docs.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-environment-set-up. [Accessed: 20 January 2022].

Microsoft (2022). DirectXTK. *GitHub Repository*. [Online]. Available from: https://github.com/microsoft/DirectXTK. [Accessed: 18 March 2022].

Microsoft (2021b). DirectXTK. *GitHub repository*. [Online]. Available from: https://github.com/microsoft/DirectXTK.

Microsoft (2020). *Graphics Pipeline*. [Online]. 4 November 2020. Available from: https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline. [Accessed: 8 November 2021].

Microsoft (2021c). *What’s new in Visual Studio 2022*. [Online]. Available from: https://docs.microsoft.com/en-us/visualstudio/ide/whats-new-visual-studio-2022?view=vs-2022. [Accessed: 20 January 2022].

Moxon, M. (2020). *Bresenham’s line algorithm*. [Online]. Available from: https://www.bbcelite.com/deep\_dives/bresenhams\_line\_algorithm.html. [Accessed: 30 January 2022].

Novalogic (1992). *Voxel Space*.

nTask (2021). *Microsoft Planner vs Trello: Which One is Better?* [Online]. Available from: https://www.ntaskmanager.com/blog/microsoft-planner-vs-trello/. [Accessed: 19 March 2022].

NVIDIA (2021). vk\_mini\_path\_tracer. *GitHub repository*. [Online]. Available from: https://github.com/nvpro-samples/vk\_mini\_path\_tracer. [Accessed: 8 November 2021].

Papas, M. (2010). *A physically-based BSDF for modeling the appearance of paper*. [Online]. Available from: https://escholarship.org/content/qt2mw373wg/qt2mw373wg.pdf?t=ml528h. [Accessed: 10 February 2022].

Perforce (2018). *Git vs. SVN – What Is The Difference?* [Online]. Available from: https://www.perforce.com/blog/vcs/git-vs-svn-what-difference. [Accessed: 18 March 2022].

PLANETTOGETHER (2019). *Agile Methodologies: Kanban Vs Scrum - Advantages and Disadvantages*. [Online]. 31 December 2019. Available from: https://www.planettogether.com/blog/agile-methodologies-kanban-vs-scrum-advantages-and-disadvantages. [Accessed: 7 November 2021].

Pollard, J. (2017). *Trello vs GitHub Projects. Power vs Simplicity*. [Online]. Available from: https://yeltzland.medium.com/trello-vs-github-projects-power-vs-simplicity-d237c0322c9b. [Accessed: 19 March 2022].

ProductPlan (2022). *Scrumban*. [Online]. Available from: https://www.productplan.com/glossary/scrumban/. [Accessed: 16 April 2022].

Quilez, I. (n.d.). *Article on Distance Functions*. [Online]. Available from: https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm. [Accessed: 24 November 2021].

Quilez, I. (2010). *soft shadows in raymarched SDFs*. [Online]. Available from: https://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm. [Accessed: 31 March 2022].

Retro64 (2017). *An introduction to 6502 math: addition, subtraction and more*. [Online]. Available from: https://retro64.altervista.org/blog/an-introduction-to-6502-math-addiction-subtraction-and-more/. [Accessed: 1 February 2022].

Rinker, R. (1991). *Reducing Computational Expense of Ray-Tracing Using Surface Oriented Pre-Computation*.

Rodríguez, J. (2013). *GLSL Essentials*.

Sanglard, F. (2017). *Game Engine Black Book: Wolfenstein 3D*.

Schneider, A. & Vos, N. (2015). The Real-time Volumetric Cloudscapes of Horizon: Zero Dawn. *Siggraph 2015*. [Online]. Available from: https://www.guerrilla-games.com/read/the-real-time-volumetric-cloudscapes-of-horizon-zero-dawn. [Accessed: 15 January 2022].

Scratchapixel (2015). *Rasterization: a Practical Implementation*. [Online]. Available from: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation. [Accessed: 18 January 2022].

Scratchapixel (2014). *Ray-Tracing: Generating Camera Rays*. [Online]. Available from: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays. [Accessed: 23 November 2021].

Sears, D.W. (2020). *Introduction to Multiplication and Division by Logarithmic Algebra*. [Online]. Available from: https://biosci.mcdb.ucsb.edu/biochemistry/tw-lig/logarithms/logarithmic-algebra.htm. [Accessed: 2 February 2022].

Segura, C., Stine, T. & Yang, J. (2013). *Constructive Solid Geometry Using BSP Tree*. [Online]. Available from: https://static1.squarespace.com/static/51bb9790e4b0510af19ea9c4/t/51bf7c34e4b0a897bf550945/1371503668974/CSG\_report.pdf. [Accessed: 1 April 2022].

Shahrabi, S. (2019). *Point cloud rendering*. [Online]. Available from: https://medium.com/realities-io/point-cloud-rendering-7bd83c6220c8. [Accessed: 27 November 2021].

Sjoholm, J. (2018). *Effectively Integrating RTX Ray Tracing into a Real-Time Rendering Engine*. [Online]. Available from: https://developer.nvidia.com/blog/effectively-integrating-rtx-ray-tracing-real-time-rendering-engine/#:~:text=Effectively%20Integrating%20RTX%20Ray%20Tracing%20into%20a%20Real%2DTime%20Rendering%20Engine,-By%20Juha%20Sjoholm&text=RTX%20is%20NVIDIA’s%20new%20platform,number%20of%20games%20and%20engines. [Accessed: 20 January 2022].

SlikSVN (2022). *What Is Subversion?* [Online]. Available from: https://sliksvn.com/support/what-is-subversion/. [Accessed: 19 March 2022].

Steven M., L. (2020). *Lambertian shading*. [Online]. Available from: http://vr.cs.uiuc.edu/node197.html. [Accessed: 30 March 2022].

Szirmay-Kalos, L., Márton, G., Dobos, B., Horváth, T., Risztics, P. & Kovács, E. (1995). *Theory of Three Dimensional Computer Graphics*.

Tamasi, T. (2008). The Evolution of Computer Graphics. *NVISION 08*. [Online]. Available from: https://www.nvidia.com/content/nvision2008/tech\_presentations/Technology\_Keynotes/NVISION08-Tech\_Keynote-GPU.pdf. [Accessed: 19 January 2022].

Trebino, M. (2017). *The Phong illumination model*. [Online]. Available from: https://mtrebi.github.io/2017/01/25/phong-illumination.html. [Accessed: 10 February 2022].

Underground Mathematics (2022). *Convex shape*. [Online]. Available from: https://undergroundmathematics.org/glossary/convex-shape. [Accessed: 1 February 2022].

Unity Technologies (2021). *Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine*. [Online]. 2021. Available from: https://unity.com/. [Accessed: 8 November 2021].

Uralsky, Y. (2005). *Chapter 17. Efficient Soft-Edged Shadows Using Pixel Shader Branching*. [Online]. Available from: https://developer.nvidia.com/gpugems/gpugems2/part-ii-shading-lighting-and-shadows/chapter-17-efficient-soft-edged-shadows-using. [Accessed: 31 March 2022].

Vandevenne, L. (2020). *Raycasting*. [Online]. Available from: https://lodev.org/cgtutor/raycasting.html. [Accessed: 2 February 2022].

Verschelde, J. (2021). Evolution of Graphics Pipelines. *Introduction to Supercomputing*. [Online]. Available from: http://homepages.math.uic.edu/~jan/mcs572/gpu\_evolution.pdf. [Accessed: 18 January 2022].

Virtix Edge (2018). *SHADER SERIES – SCREEN SPACE REFLECTIONS*. [Online]. Available from: https://virtexedge.design/shader-series-basic-screen-space-reflections/. [Accessed: 6 April 2022].

Vivo, P.G. & Lowe, J. (2015). *The Book of Shaders*. [Online]. Available from: https://thebookofshaders.com/. [Accessed: 19 January 2022].

Walton, J. (2020). *Nvidia RTX 30-Series Ampere Architecture Deep Dive: Everything We Know*.

Whitted, T. & Foley, J.D. (1980). An Improved Illumination Model for Shaded Display. *Graphics and Image Processing*. [Online]. 23. p.pp. 343–349. Available from: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.156.1534. [Accessed: 26 October 2021].

Wong, J. (2016). *Ray Marching and Signed Distance Functions*. [Online]. Available from: http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions. [Accessed: 26 November 2021].

Zhou, Q., Grinspun, E., Zorin, D. & Jacobson, A. (2016). *Mesh Arrangements for Solid Geometry*. [Online]. Available from: https://cims.nyu.edu/gcl/papers/zhou2016mas.pdf. [Accessed: 1 April 2022].

Zucconi, A. (2016). *Volumetric Rendering: Ambient Occlusion*. [Online]. Available from: https://www.alanzucconi.com/2016/07/01/ambient-occlusion/. [Accessed: 8 February 2022].

# Image Bibliography

Anon (2022). *Comanche: Maximum overkill screenshots - Abandonware DOS*. [Online]. 2022. abandonwaredos.com. Available from: https://www.abandonwaredos.com/abandonware-screenshot.php?gid=1409&idi=YWJhbl9pbWdfc2NyZWVucy9jb21hbmNoZW1heGltdW0tMy5qcGc=&tit=comanche-maximum-overkill.

Anon (n.d.). *Metal by Tutorials, Chapter 3: The Rendering Pipeline*. [Online]. raywenderlich.com. Available from: https://www.raywenderlich.com/books/metal-by-tutorials/v2.0/chapters/3-the-rendering-pipeline.

Evans, A. (2015). *Learning from failure: A Survey of Promising, Unconventional and Mostly Abandoned Renderers for ‘Dreams PS4’, a Geometrically Dense, Painterly UGC Game*. [Online]. 2015. Advances.realtimerendering.com. Available from: http://advances.realtimerendering.com/s2015/AlexEvans\_SIGGRAPH-2015-sml.pdf.

Jacobson, A. (2016). *constructive solid geometry*. [Online]. 2016. Alecjacobson.com. Available from: http://www.alecjacobson.com/weblog/?tag=constructive-solid-geometry.

Kuri, D. (2018). *GPU Ray Tracing in Unity – Part 1 – Three Eyed Games*. [Online]. 2018. Blog.three-eyed-games.com. Available from: http://blog.three-eyed-games.com/2018/05/03/gpu-ray-tracing-in-unity-part-1/.

Leeuwen, N. (2020). *Ray Tracer | Niek van Leeuwen | Computer Science and Engineering student*. [Online]. 2020. Niekvanleeuwen.nl. Available from: https://niekvanleeuwen.nl/2020/08/ray-tracer/.

Meijer, R. (2016). *Elite (1984)*. [Online]. 2016. When Games Were King. Available from: https://whengameswereking.wordpress.com/2016/12/08/elite-1984/.

Sanglard, F. (2017). *Game Engine Black Book: Wolfenstein 3D*.

Wolfe, A. (2020). *Casual Shadertoy Path Tracing 2: Image Improvement and Glossy Reflections*. [Online]. 2020. The blog at the bottom of the sea. Available from: https://blog.demofox.org/2020/06/06/casual-shadertoy-path-tracing-2-image-improvement-and-glossy-reflections/.