School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Improving the Speed of Rendering Three-Dimensional Fractals using Precalculated Information

Nell Mills

Submitted in accordance with the requirements for the degree of MSc High Performance Graphics and Games Engineering

2021-2022

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Project Report	Report	SSO (19/08/22)
Project Code	Link to Repository	SSO (19/08/22)

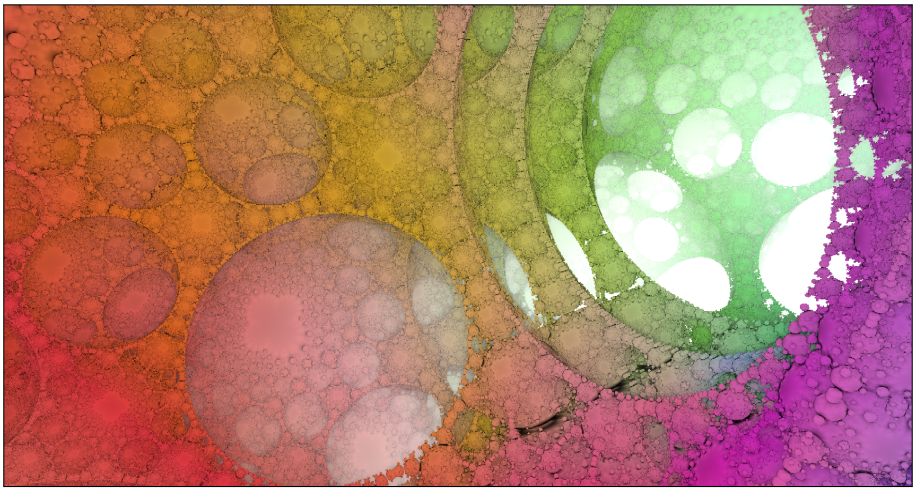
Type of project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) 

Summary



Fractals have long captured the imagination with their beautiful self-similar patterns and infinite detail. Rendering 3D fractals in real time using sphere tracing (a form of ray tracing) is the focus of this project. The aim is to speed up the rendering process using precalculated information, saving computation time each frame. Two methods were chosen for this purpose. One of these aims to generate information offline, and make it available to the shaders during rendering. The other aims to save information from previous frames, to be sampled in the current one.

During the course of the project, a piece of software will be produced that is capable of rendering 3D fractals in real time. It will be configurable with options to render different fractal formulae, to change the optimization method, to choose animations of the scene, and to take performance measurements to test the optimization methods.

There will be an extensive evaluation of the different performance methods implemented, to see which one, if any, is suitable for use in speeding up the rendering of 3D fractals.

Acknowledgements

I would like to thank my project supervisor, Dr. Markus Billeter, for his support and guidance throughout the project, and for agreeing to supervise this project in the first place.

I would also like to thank my assessor, Professor Hamish Carr, for taking the time to assess this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Fractals are self-similar patterns generated using iterative formulae, and are famous for their beauty and infinite complexity. Generation of three-dimensional fractals has applications in generative artwork, games and even films. This project will focus on improving the performance of real time rendering of 3D fractals, with the eventual motivation to incorporate them into terrain generation for games.

1.1 Project Aim

The aim of this project is to produce software to render 3D fractals in real time using sphere tracing, and implement two methods of improving the speed of doing so. Success will be measured in terms of program features, measurements of render pass times, and visual evidence of performance differences.

Sphere tracing is a form of ray marching, that relies on calculating the approximate difference to the nearest surface in order to suggest a marching distance in each step.

The first optimization method to be investigated is a Signed Distance Field (referred to as an SDF from this point on), which calculates the approximate distance to the surface offline, and stores it in a grid structure. The second optimization method will be a form of temporal caching, which stores the calculated distance from four of the previous frames, and uses it to speed up calculation in the next.

1.2 Objectives

This project will involve four main stages:

- Write a renderer that is capable of sphere tracing three-dimensional fractals.
- Implement two optimization methods that attempt to improve the rendering speed.
- Implement a performance measuring system to capture performance differences.
- Measure the performance differences between the optimized and unoptimized rendering in a variety of scenarios.

1.3 Deliverables

The main deliverables of this project are:

- A piece of software capable of rendering 3D fractals, with two optimization methods to choose from.
- The source code for the software, in the form of a GitHub link (in Appendix C).
- The report for the MSc project.

1.4 Ethical, Legal and Social Issues

The only ethical or legal issue that this project may encounter is the use of third-party software to implement the project aims. To deal with this, the licensing of each piece of third-party software used will be made clear.

Chapter 2

Background

This project aims to improve the efficiency of real-time rendering of 3D fractals. This chapter will focus on background theory for fractals, signed distance functions and sphere tracing. Afterwards, two possible methods of improving efficiency will be discussed, namely SDFs and temporal caching. Finally, some motivation for choosing these methods will be given.

2.1 Rendering of Fractals

2.1.1 2D Fractals - The Mandelbrot Set

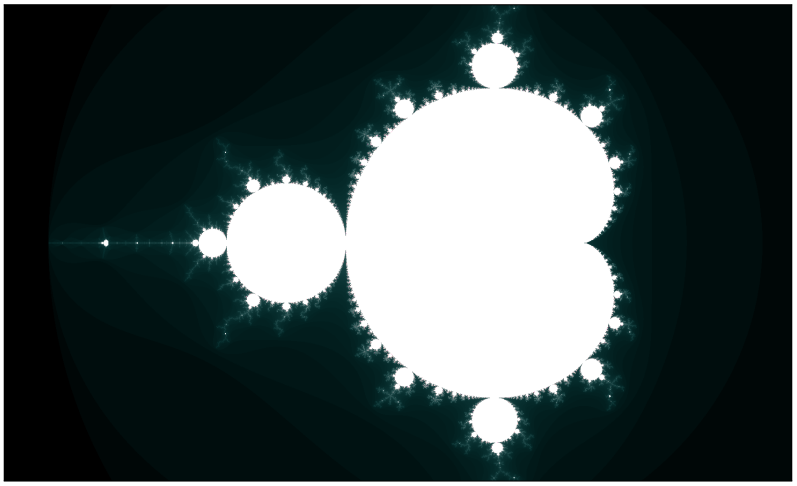


Figure 2.1: The Mandelbrot set. The white points in the centre are inside the set.

The Mandelbrot set is the set of two-dimensional points that satisfy a certain constraint on the following complex quadratic equation:

$$Z = Z^2 + C \tag{2.1}$$

where Z and C are complex numbers. The constraint on the points is that their orbit must be bounded. The value of Z is initialized to 0 and equation 2.1 is iterated over, each new value of Z being placed back in to the equation in the next iteration. If the length of the point Z does not exceed a threshold, then the point (represented by C) is in the Mandelbrot set [4].

Figure 2.1 shows a generated Mandelbrot set. The real part of the point C is represented by the x-axis, and the imaginary part by the y-axis. Equation 2.1 is iterated

over a maximum of five hundred times, and the threshold value is two. The pixels are coloured according to how many iterations are achieved before the length of Z exceeds the threshold.

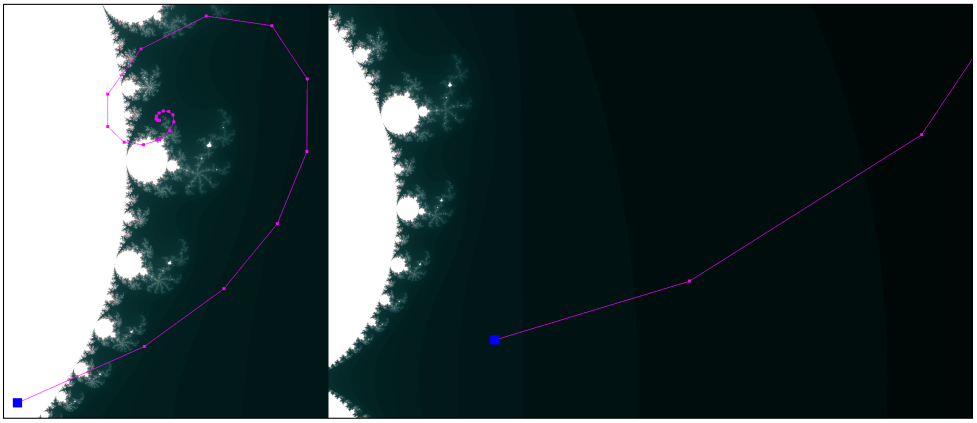


Figure 2.2: Visualization of the first twenty five iterations of equation 2.1 on the initial points [0.3, 0.05] (left) and [0.5, 0.04] (right). The initial points are shown in blue.

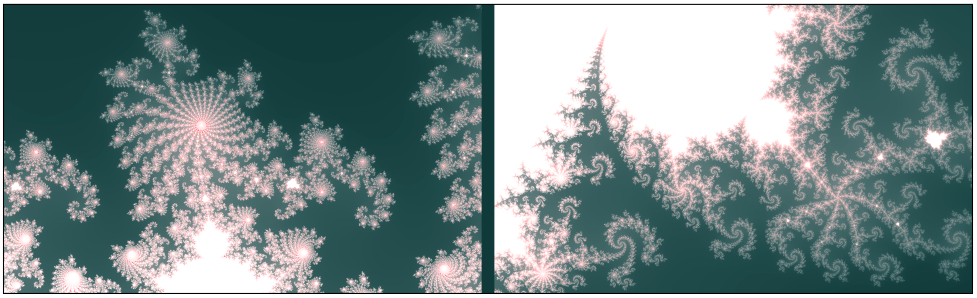


Figure 2.3: Two different views of the Mandelbrot set, zoomed in.

Figure 2.2 illustrates the first twenty five iterations on two different points. For the first point, the iterations converge in a spiral shape and the length of Z never exceeds the threshold of two, therefore the point is in the Mandelbrot set and is coloured white. For the second point, the iterations diverge and exceed the threshold of two within a few iterations, so this point is not in the Mandelbrot set and is coloured dark.

Figure 2.3 shows two zoomed-in views at the edge of the original shape. New patterns can be seen, as well as repeated ones, and even new instances of the original shape. This is because the Mandelbrot set has infinite detail, so if one decreases the range of the axes, new patterns will emerge [5].

2.1.2 3D Fractals - The Mandelbulb

Since the Mandelbrot set is in two dimensions, and since complex numbers have only two components (real and imaginary), obtaining a true three-dimensional Mandelbrot set is a challenging task, and a mathematically rigorous three-dimensional Mandelbrot

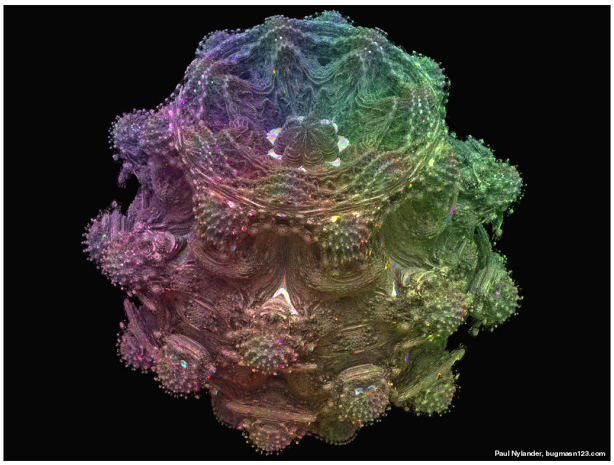


Figure 2.4: First look at the Mandelbulb, from Paul Nylander's website [1].

has not yet been found [6].

One attempt that seems to have come close was originated by Rudy Rucker in 1987. Rucker thought that expressing the three-dimensional points in spherical coordinates would allow the manipulation of the points in a similar way to the complex-number operations performed on the points in the two-dimensional Mandelbrot set [7].

Rucker did not have the computational power to accomplish a rendering of this idea, so it was put aside for twenty years, until Daniel White independently published a formula in 2007, which took the approach proposed by Rucker. White decided to approach the problem by considering the geometrical consequences of multiplying numbers in the complex plane, which amounts to rotating them [6].

White's formula produced images that looked promising, but they didn't have the level of detail that was expected from a true three-dimensional equivalent of the Mandelbrot set. A mathematician, Paul Nylander, raised White's formula to a higher power (eight), which would be equivalent to increasing the number of rotations of the point. The resulting image is shown in figure 2.4. The shape maintains excellent detail, even at high levels of magnification [6].

The new shape, known as the Mandelbulb, has roughly the same formula as the Mandelbrot set (equation 2.1):

$$Z = Z^k + C \tag{2.2}$$

where Z is raised to an arbitrary power like so:

$$Z^k = r^k(sin[k\theta]cos[k\phi], sin[k\theta]sin[k\phi], cos[k\theta]). \tag{2.3}$$

The variable r is the norm of Z (|Z|), θ is equal to $\arctan(Z_y/Z_x)$ and ϕ is equal to $|(Z_x, Z_y)|/Z_z$. The spherical coordinates of the point Z/|Z| are represented by θ and ϕ. Equations 2.2 and 2.3 are sourced from Chapter 33 of the book Ray Tracing Gems II [8].

2.1.3 Signed Distance Functions

Signed distance functions provide an estimate of how close a point is to the surface of a shape. If the result of the function is positive, then the point is outside the surface. If the result is negative, then the point is inside the surface. If the result is zero, then of course the point is exactly on the surface of the shape described by the function [9].

A signed distance function can be derived using the Böttcher map for the fractal formula, which is a deformation of the space. Closer to the surface of the fractal, the space is deformed to a greater degree than parts further away from the surface, as shown in figure 2.5. The deformations of the space occur in such a way as to map the exterior of the fractal to the exterior of a unit disk [2].

A rigorous mathematical explanation of the Böttcher map will not be given in this paper, but a brief introduction is helpful to understand the derivation of the distance function for the Mandelbulb. The map can be calculated as follows:

$$\phi_C[Z] = lim_{n\to\infty}[f^n[Z]]^{k^{-n}} \tag{2.4}$$

where the value of f[Z] is the same as in equation 2.2 and n refers to the current iteration of the equation [8].

Looking at equation 2.4, take a point Z that is not in the fractal, so that the function $f^n[Z]$ grows as the number of iterations increases, tending towards infinity. For a large enough value of n, the term $f^n[Z]$ will become vastly larger than the value of C (as the point is far from the fractal), meaning that C can reasonably be discarded [8].

From this casting away of C we obtain:

$$f^{n+1}[Z] \approx (f^n[Z])^k \tag{2.5}$$

for points that are sufficiently far away from the surface of the fractal. Since the point at iteration n is far from the fractal, if we were to undo all of the iterations to get back to the original point at $f_0[Z]$, we would obtain this expression:

$$f_0^n[Z]^{k^{-n}} = Z \tag{2.6}$$

and by extension, returning to the Böttcher map:

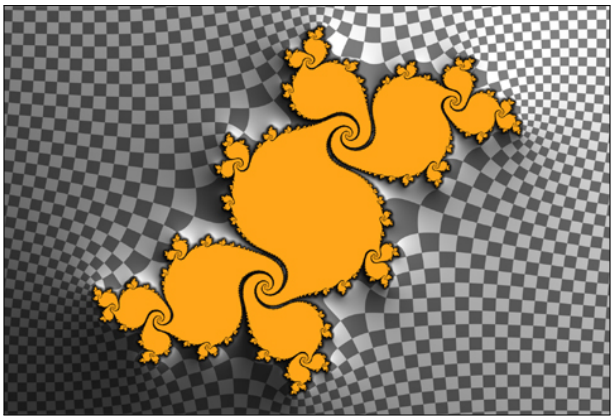


Figure 2.5: The Böttcher map generated for the Julia set (another two-dimensional complex fractal related to the Mandelbrot set), from the website of Inigo Quilez [2].

$$\phi_0[Z] = Z \tag{2.7}$$

which is the approximate result that equation 2.4 gives when the function $f^n[Z]$ ultimately diverges [8].

Next, we will use something called the Hubbard-Douady potential, equal to the logarithm of the modulo of the Böttcher map. This is a map of points onto a unit disk. Recall that the Böttcher map maps the exterior of the fractal to the exterior of a unit disk. This now becomes important, as it enables us to use the Hubbard-Douady potential for all of our complex fractals. We now have a function that tends towards zero as the points approach the boundary of the fractal [2]:

$$G[Z] = lim_{n\to\infty}\frac{log|f^n[Z]|}{k^n}. \tag{2.8}$$

Equation 2.8 is not ready to be used as a distance measurement yet. However, it's possible to make it so. More detail is given in Ray Tracing Gems II, but a brief explanation will be given here. If we divide G[Z] by its gradient (obtaining this from the first-order Taylor expansion of the function), then we obtain an upper bound on the distance to the surface. The final equation for distance estimation therefore is [8]:

$$d(Z) = lim_{n\to\infty}\frac{|f^n(Z)|log|f^n(Z)|}{|(f^n)'(Z)|}. \tag{2.9}$$

2.1.4 Ray and Sphere Tracing

Ray tracing is a technique used for rendering scenes. Generally, a ray is a line that is cast from the camera or eye through each pixel of an image. Tests are performed to see which object (if any) is encountered by the ray first. By bouncing the ray from object to object, the program can gather the various light contributions from objects that reflect,

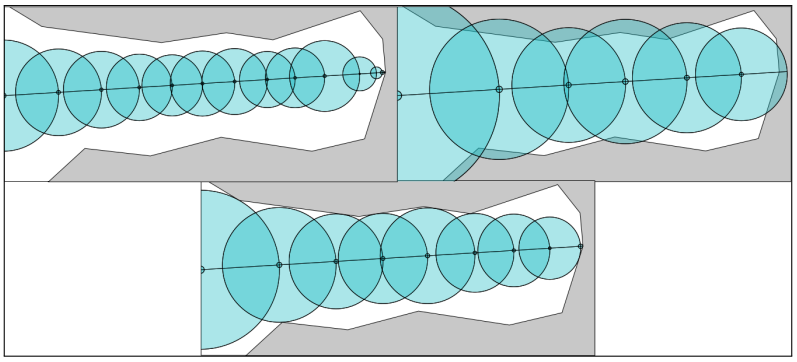


Figure 2.6: Three scenarios in sphere tracing. Top left: Distance function underestimates distance, resulting in a loss of performance. Top right: Distance function overestimates distance, resulting in a loss of accuracy. Bottom: Distance function is exact.

emit or refract light [10].

Distance estimators can be used for ray tracing. If the approximate distance to the nearest surface can be calculated then the current point can be moved along the ray safely, until it is close enough to the surface to stop, according to this formula:

$$P_{n+1} = P_n + \mathbf{v}d(P_n) \tag{2.10}$$

where P_{n+1} is the point at the next step, P_n is the point at the current step, $\mathbf{v}$ is the unit vector representing the ray direction and $d(P_n)$ is the distance function acting on the current point [8].

Using a distance function in this way is known as sphere tracing. The magnitude of the result of the distance function can be considered as the radius of a sphere. This sphere is guaranteed not to go through any part of the surface, making it safe to step according to the distance function result in any direction. For this guarantee to hold, the distance function must either be an exact calculation of the distance, or an underestimate [11].

See figure 2.6. This shows three scenarios for tracing a ray with the sphere tracing technique. At the bottom, the distance function is an exact measure of the distance to the nearest point on the surface. This is the ideal scenario, for correctness and performance. The top left image shows the result of a distance function which underestimates the distance each time. As a result, significantly more steps are taken along the ray, which will result in decreased performance. Lastly, the top right image shows the result of overestimating the distance. The ray ends up stepping past the boundary of the shape.

2.2 Optimization Methods

2.2.1 Signed Distance Fields

An SDF is the result of discretizing a signed distance function over a finite space. The aim of the SDF is to reduce the need for evaluating potentially expensive signed distance functions for use in, for example, collision detection or ray tracing. Samples of the distance are taken at each point of the field (for example, at every voxel in a three-dimensional grid, as the case will be in this project) and stored for later use. Often, these samples are taken at the vertices of the grid, and when they are read, they are interpolated to get an approximation of the value at the specific point in the voxel [12].

Memory usage is a concern when using an SDF, especially if the shape requires high accuracy, as it will require a greater concentration of samples, especially near the surface. Some methods try to deal with this by increasing the amount of subdivision close to the surface or in areas with more complex geometry, and reducing it elswhere [13].

A common representation scheme for an SDF, and the one chosen for this project, is an eight-dimensional binary tree, also known as an octree. The nodes of the tree represent areas of the space, and at each level of the tree, these areas get exponentially smaller. Leaf nodes contain the information necessary for construction of the scene. Other nodes point to eight child nodes each, so that leaf nodes can be reached by searching the tree, starting with the root [14].

This structure, being constructed out of cubes, is simple to manipulate and search. All of the nodes of the tree can be kept in a specific order, which makes indexing very simple, since the tree can just be traversed in a specific order, rather than searching the space itself for the correct cube [14].

2.2.2 Temporal Caching

Temporal caching methods rely on the idea that the contents of an image don't change much from frame to frame; they are temporally coherent. With this in mind, a possible performance booster can be implemented for ray tracing, which uses information (such as colours, lighting data or positions) from previous states to speed up or provide a hint for calculations in the current frame [15].

Taking advantage of temporal coherence in ray tracing is of special interest in the case of global illumination data, which is very expensive to calculate. In order to maintain spatial coherence between frames, methods often perform temporal reprojection, which

maps the location of the point on the surface from the current frame to the previous frame, so that it samples from similar areas in the scene [16].

Consider the scenario in which a ray travels through a room, hitting the back wall. In the next frame, the camera has moved, and the same ray now intersects a closer object, say a chair. However, the sampled distance from the previous frame already places the point behind the chair, rendering the chair invisible. The method used in this project needs to take this problem into account, since the camera position and rotation will change.

Determining whether a pixel value is invalid can be done in a few ways. For example, testing the angle between the normal of the previous frame and the current ray, and recasting the ray if the angle exceeds some threshold. It could also be done by comparing the depth of previous frames with the depth of the current frame, though this would likely work better for mesh geometry, as it requires knowing the exact depth in the current frame, the calculation of which this project is trying to avoid [17].

2.3 Implementation of Optimization Methods

2.3.1 Signed Distance Field

The use of an SDF is intended to provide a cheaper way of evaluating the distance at any point around the fractal shape than calculating the signed distance using the given function. If this is possible, then performance boosts will be expected when rendering the fractals in real time.

The octree data structure has been chosen for this project for its simplicity, and for the ability to be selective about when to perform a memory read, which is expensive compared to calculations on the GPU. Memory usage will need to be considered, as a very high resolution representation of the space will be costly. For this project, very fine detail is not needed, so a relatively coarse field can be used. However, accuracy is still a concern. Chapter 4 on implementation will discuss decisions made to tackle this problem.

The octree representation chosen will not be sparse. This is because there is a need to reduce the frequency of memory reads as much as possible, and having a sparse structure would introduce the need to store additional information, used for traversing the tree. If the octree is not sparse, the required voxel index can be calculated, rather than read from memory.

2.3.2 Temporal Cache

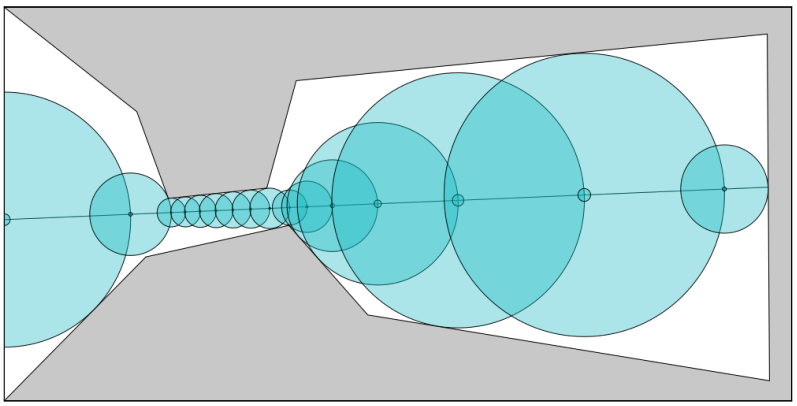


Figure 2.7: A particular scenario in sphere tracing, where the ray passes close to some geometry, but never intersects. This slows the ray down and results in a loss of performance.

The goal of using temporal caching for rendering three-dimensional fractals is to reduce the number of iterations required to get to the surface. This is a different approach than using an SDF, which is intended to reduce the cost of the iterations themselves.

The information required to render the scenes in this project is the distance travelled by each ray. For a still image (no camera movement), the result of the last frame can be sampled and used again exactly the same in the next frame, skipping all sphere tracing iterations. Of course for real time rendering, an image is likely to change, either because of moving geometry, or camera changes. To tackle this, a test will be performed to see if the camera has moved too much, and an effort will be made to have a consistent, but preferably small, underestimate of the true distance, so that the rays don't overshoot.

The test chosen is to sample the depth stored for the last few frames. The difference between the maximum and minimum of these values will then be calculated. If the difference between them exceeds some threshold, then the ray will be recast. I chose not to use the angle between the normal and current ray because that would require storage of the normal, and therefore more sampling from the cache every frame.

One scenario of interest in this project is illustrated in figure 2.7. The ray passes very close to the surface of the shape along its path, but never intersects that part of the shape. This results in smaller steps for the sphere tracing algorithm, and a loss of performance, as a significant number of iterations are expended getting past the bottleneck in the scene. This paper proposes that a temporal caching method could be employed that provides a significant performance boost by bypassing the bottlenecks in these scenarios.

Chapter 3

System Design

This chapter will go through the design of the project software. First, the system requirements will be laid out, followed by the general system design. The system design is simple, as the focus is on improving the performance of rendering.

3.1 System Requirements

3.1.1 Functional Requirements

- The software will use fragment shaders to render 3D fractals using sphere tracing.
- The software will provide user controls for camera motion.
- The software will implement a Signed Distance Field for optimization.
- The software will implement a temporal cache for optimization.
- The software will be capable of measuring the performance of the rendering process.
- The software will be capable of writing these measurements to file in an easily readable format.
- The software will be configurable through program arguments.

3.1.2 Non-Functional Requirements

- The software should be easy to use, with helpful instructions in the README file and printed to the terminal on load.
- The software user controls should be intuitive.
- The software arguments should be clearly explained.
- The software should be stable (it should not crash or have obvious bugs).
- The software should have a sensible layout in the code repository.

3.2 System Design

The project must include several components. One of these is a renderer, capable of rendering 3D fractals to the screen. This must be configurable through the program's arguments, as the different optimization methods will have different requirements. The next component is the implementation of the two optimization techniques. These will affect the configuration of the renderer as well. Lastly, performance measurements must be taken. These should not affect the rendering process.

With these components, the overall system design is simple. There is no GUI to implement, so no overall design or consideration for the user experience. The main design consideration is in the configuration of the various states of the program. A design for the arguments that the program will take, and how they will affect the program at run time, is considered now.

3.2.1 Program Arguments

The various configurations of the system at run time will be decided via the program's arguments. These will be as follows:

The plan is to render more than one type of fractal, so these will need to be selected via the first argument. The type of fractal affects which signed distance function to use, so is needed to select the shaders to load.

The focus of the project is the improvement of performance, so the various methods will need to be compared to each other. The second input to the program will decide which optimization method to use; in this case, the options are no optimization, the SDF and the temporal cache. This will further affect which shaders will get loaded, as well as some other settings in the renderer.

A real time rendering project would not be very useful if it did not allow animation or movement of some kind. With that in mind, it might be of interest to see how the different optimization methods perform when running animations. The third argument will select whether to animate the scene. This will not affect which shaders are loaded, only which data is passed into them.

The last arguments control the measurement of performance. Argument 4 will control whether performance measurements are taken at all, and argument 5 will contain the name of the file to be written out to. Data will be written to the file specified in a readable format.

Chapter 4

Implementation

This chapter will go through the implementation of the project software. First, the system information and general program structure will be looked at. Then, the rendering of fractals will be covered, followed by the attempts at improving performance, and the method of measuring performance. Finally, there will be a section on debugging.

In terms of general optimizations, there were no specific efforts to optimize the code or the basic algorithms (except perhaps to implement a view distance limit), since the most important thing was to remain consistent across the different scenarios, and the relevant measurements were the performance differences (if any), not the raw performance.

4.1 Operating System and Hardware

The operating system used was Linux Mint 20.1. No other operating system was tested.

The CPU used was an Intel©Core^{TM} i7-9750H with 6 cores.
The GPU used was an NVIDIA TU106M (GeForce RTX 2060 Mobile).
The Vulkan version being used was 1.3.205.

4.2 Program Structure and Libraries

4.2.1 Build System

The build system chosen was Make. The project is small in terms of the code base so writing a Makefile for compilation was simple. A shell script was also written, and run from Make, that used GLSLC to compile the shaders and place them in the appropriate directories. GLSLC is licensed under the Apache License Version 2.0 [18].

4.2.2 Language and Libraries Used

The programming language of choice for this project was C, because it is the language that the libraries used were written in, and it's simple and efficient.

Vulkan

Vulkan was chosen because it is performant and modern, and because there is a new feature on the way that will, I think, help with the efficiency of rendering using one of

the optimization methods chosen (this will be mentioned in Chapter 6). It is licensed under the Apache License 2.0 [19].

Volk

The third-party library, Volk, is a meta-loader for Vulkan. It allows one to load the Vulkan API without needing to link to Vulkan. This makes setup much easier to manage. It is licensed under the MIT license [20].

GLFW

GLFW is a library for creating windows and surfaces, which can be used for Vulkan development. Additionally, it is easy to use and multi-platform, so was the logical choice for this project. It is licensed under the zlib/libpng license [21].

4.2.3 Vulkan Setup

The rendering process was split into two passes, one for geometry and one for colour. This was done because the speed at which the geometry of the scene was obtained was of interest, and the speed of colouring the fractals was not, so splitting them enabled measurement of the geometry render pass on its own. No vertices are passed in to the shaders; the vertex shaders conjure up fullscreen triangles using the vertex indices, which are then worked on by the fragment shaders, where all the calculations and colouring happen.

Specific setups for different optimization methods will be discussed in the relevant sections.

4.2.4 Program Arguments and Controls

The program takes arguments which specify the setup when the program is loaded. The user can change which fractal to display (including the 2D Mandelbrot set, the Mandelbulb and the Hall of Pillars), which optimization method to use, whether to vary any fractal parameters (which can be used to animate the fractal) or run a predefined animation, and whether to take any performance measurements during runtime. Some of the arguments affect which shaders are loaded, and the Vulkan setup.

The user is able to control the camera with the mouse and keyboard, speed up and slow down, and print the current position and camera front vector. Input is handled by GLFW.

4.3 Rendering 3D Fractals

The project makes use of two fractal formulae. One is for the Mandelbulb, as described in the background section. Another was chosen to provide variety, specifically with regards to the depth of the image. The Mandelbulb is neatly contained within a box, and the program renders its exterior, but not any 'interior'. The alternative fractal (named 'Hall of Pillars' by me given its lack of another name) is reminiscent of a room with many archways, which I thought might give a greater variety of results in terms of the performance measurements when using the optimizations developed in the project.

The calculations for the fractals are done entirely within shaders, except for the generation of the 3D signed distance field, which is calculated on the CPU upon loading the program, before any rendering occurs. Single-precision floating point numbers are used for all calculations within shaders. These were chosen over double-precision numbers, as the scale of the fractals is reasonable, and high-detail zooms into the fractal are not necessary for the project.

4.3.1 Basic Sphere Tracing Implementation

The basic sphere tracing algorithm is implemented in GLSL and is largely the same across the fractal types and optimization methods. Figure 4.1 shows the implementation.

```
vec4 sphere_trace(vec3 origin, vec3 ray)
{
        vec4 current_position = vec4(origin, 1.f);
        int max_steps = 999;
        float distance_estimate;
        float distance_travelled = 0.f;
        float distance_threshold = 0.0001f;

        for (int steps_taken = 0; steps_taken <= max_steps; steps_taken++)
        {
                // Get distance estimate and update total distance travelled:
                distance_estimate = distance_estimator_mandelbulb(current_position.xyz);
                distance_travelled += distance_estimate;

                // Get current position. Encode iterations in w-coordinate:
                current_position = vec4(origin + (ray * distance_travelled),
                        1.f - (float(steps_taken) / float(max_steps)));

                // Check how close the point is to the surface:
                if (distance_estimate < distance_threshold) { break; }

                // Check the view distance:
                if (abs(distance_travelled) >= u_scene.view_distance) { break; }
        }

        // Return current position along with iterations achieved:
        return current_position;
}
```

Figure 4.1: GLSL code snippet of the sphere tracing algorithm.

The most important things that must remain consistent between different optimizations on the same fractal are the termination conditions, which are as follows:

- The maximum number of iterations allowed.
- The distance threshold.

- The view distance, contained in the scene uniform.

4.3.2 Mandelbulb Fractal

```
float distance_estimator_mandelbulb(vec3 position)
{
        int max_iterations = 4;
        float escape_radius = 2.f;
        float parameter = u_scene.fractal_parameter;

        vec3 z = position;      // Z = Z^2 + C.
        float dr = 1.f;
        float r = 0.0;          // Radius.

        for (int i = 0; i < max_iterations; i++)
        {
                r = length(z);
                if (r > escape_radius) { break; }

                // Convert position to spherical coordinates:
                float theta = acos(z.z / r);
                float phi = atan(z.y, z.x);
                dr = (pow(r, parameter - 1.f) * parameter * dr) + 1.f;

                // Scale and rotate position:
                float zr = pow(r, parameter);
                theta *= parameter;
                phi *= parameter;

                // Convert position back to Cartesian coordinates:
                z = (zr * vec3(sin(theta) * cos(phi), sin(phi) * sin(theta),
                                                cos(theta))) + position;
        }

        // Calculate distance:
        return 0.5f * log(r) * (r / dr);
}
```

Figure 4.2: GLSL code snippet of the distance estimator function for the Mandelbulb fractal.

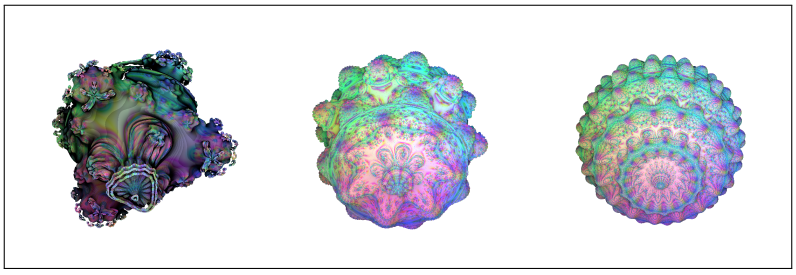


Figure 4.3: The Mandelbulb, raised to different powers. Left to right: four, eight, sixteen.

Figure 4.2 shows the implementation of the distance estimator for the Mandelbulb fractal. Recall equations 2.2 and 2.3. Equation 2.2 is iterated over four times maximum. During the iterations, the current point is converted to spherical coordinates, scaled, and then converted back to Cartesian coordinates using equation 2.3. The distance formula based on equation 2.9 is used to get the final result. The gradient of |f(Z)| is computed during the iterations (represented by dr in the shader).

The fractal parameter in the scene uniform is the power that Z is raised to in the Mandelbulb formula. Varying this parameter results in different forms of the

Mandelbulb. By default this parameter is eight, to match the original discovery by Paul Nylander. Figure 4.3 illustrates the effect of varying this parameter.

4.3.3 Alternative Fractal

```
float distance_estimator_hall_of_pillars(vec3 position)
{
        vec3 z = position.xzy;
        float scale = max(0.1f, u_scene.fractal_parameter - 1.f);
        vec3 size_clamp = vec3(1.f, 1.f, 1.3f);

        for (int i = 0; i < 12; i++)
        {
                z = (u_scene.fractal_parameter * clamp(z, -size_clamp, size_clamp)) - z;
                float r2 = dot(z, z);
                float k = max(u_scene.fractal_parameter / r2, 0.027f);
                z *= k;
                scale *= k;
        }

        float l = length(z.xy);
        float rxy = l - 4.f;
        float n = l * z.z;
        rxy = max(rxy, -n / 4.f);

        return rxy / abs(scale);
}
```

Figure 4.4: GLSL code snippet of the distance estimator function for the Hall of Pillars fractal. Full credit goes to Dave Hoskins for the formula [3].

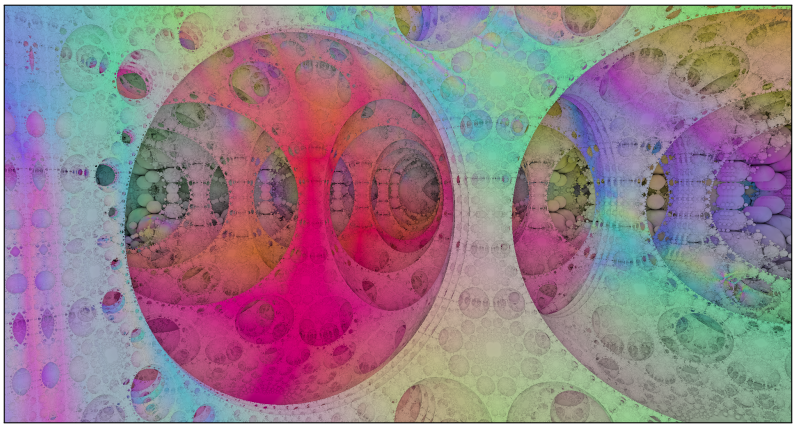


Figure 4.5: Rendering of the Hall of Pillars fractal.

Figure 4.4 shows the implementation of the distance estimator for the alternative 'Hall of Pillars' fractal. This fractal has different mathematical origins to the Mandelbulb, and is known as a Pseudo Kleinian fractal, but the mathematical background will not be explored here [22].

As you can see from figure 4.5, this fractal varies a lot in depth, and in particular has a lot of holes in it, so will result in the bottlenecks discussed in section 2.2.2. The fractal formula was chosen for this reason, and to give variety when collecting results. Exploring the fractal also reveals large flat sections, and even larger 'halls' much like the one in figure 4.5, as well as more intricate sections with great depth variety, but less overall depth. This fractal provides a good set of representative views for results collection.

4.3.4 Colour

Colour was not of the utmost importance in the project, so this section will be very brief. Colouring of the mandelbulb is done based on the closest distance from each point in the image to some fixed point. The Mandelbulb equation is iterated through, as in the distance estimation function, but in each iteration, the distance of the current point to a fixed point is measured, and the minimum of these is taken as a colour value.

For the Hall of Pillars fractal, the colour is obtained in each iteration by getting the difference between elements of the current point and the previous point, and accumulating these differences.

4.3.5 Ambient Occlusion

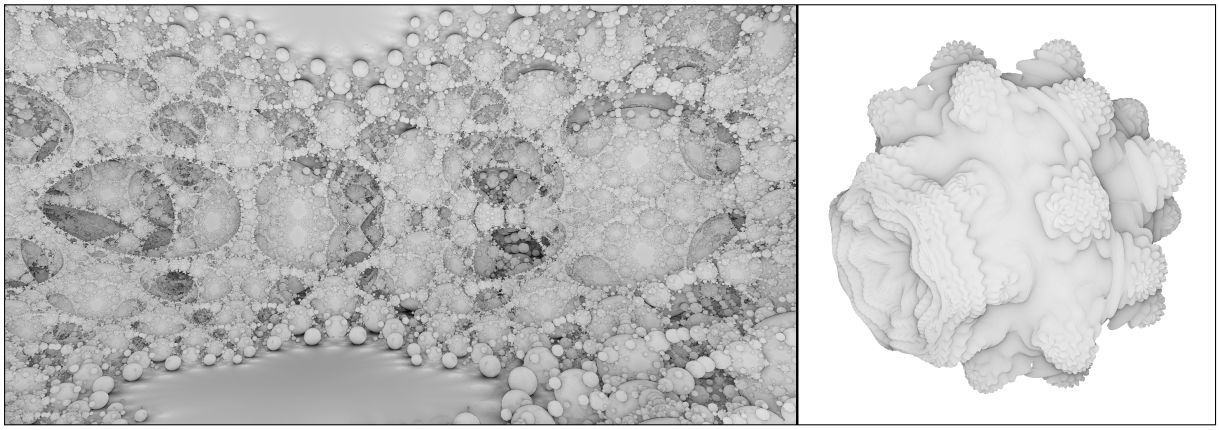


Figure 4.6: Rendering of the Hall of Pillars fractal (left) and Mandelbulb (right), coloured based on the number of iterations achieved before reaching the surface.

One of the consequences of using sphere tracing of signed distance functions is that it gives an estimate of the complexity of the surface. If a point takes more iterations to reach, then it's likely in a more complex area, and will be occluded. For both fractals in the project, the final colour is multiplied by a value between zero and one, based on the number of iterations achieved before reaching the surface. Figure 4.6 shows renderings of the fractals used in this project, coloured purely based on the number of iterations.

This free ambient occlusion effect also gives an informal measure of performance. Brighter areas mean less iterations, so two identical views can be compared which are using different optimization methods, to see if one performs better in terms of the number of iterations. Since this is the aim of using temporal caching, this is used as a measure of performance, in addition to render pass timings.

4.4 3D Signed Distance Field

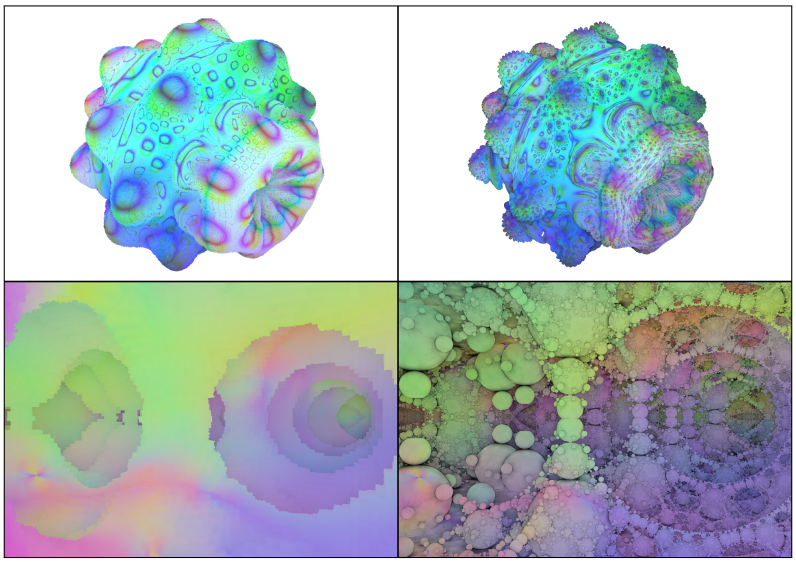


Figure 4.7: Rendering of the Mandelbulb (top) and Hall of Pillars fractal (bottom), using the signed distance field alone (left) and the hybrid approach (right). The Mandelbulb is rendered with 8 levels of subdivision (256^3 voxels), and the Hall of Pillars is rendered with 9 levels (512^3 voxels).

4.4.1 Structure and Usage

The SDF is represented a three-dimensional grid. Samples are taken at the centre of each voxel in the grid. To obtain a guaranteed underestimate here, the length of the line between the centre and corner of the voxel is taken away from this distance (or added, if the distance is negative). The more common approach is to sample the distance at the corners of the voxels and interpolate based on the location inside. The decision to only take the centre value was made to reduce memory reads, as the signed distance functions for the fractals are cheap in comparison to, say, the signed distance function for an arbitrary polyhedral shape.

Another design decision that was made was to take a hybrid approach. Rather than try to obtain as much detail as possible (and use a lot of memory) in the SDF, the program performs sphere tracing through the field, until the sampled distance is smaller than the size of the diagonal between the voxel centre and corner; the largest distance possible from the centre before leaving the voxel. Obtaining a value such as this means that the surface is likely within the voxel. When this happens, the program reverts to using the signed distance function to obtain the finer detail. This provides a balance between speed, memory use and accuracy. Figure 4.7 shows a comparison of visual quality for both fractals used in the project when using the SDF alone, and using the hybrid

approach.

For the Hall of Pillars fractal particularly, the signed distance field does not have nearly the required level of detail, despite it having one more level of subdivision than the Mandelbulb. This is because this fractal has a much larger scale, so in order to cover a reasonable area of the fractal with the field, each voxel must be much larger. For the Mandelbulb, the whole shape can be contained in a small box, approximately 2.2 units along each edge. The Hall of Pillars fractal seems to be infinitely large. This difference makes the signed distance field very limited for use in large fractals.

4.4.2 Octree Storage

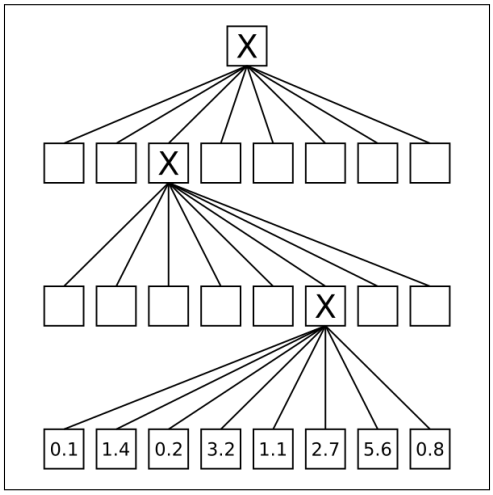


Figure 4.8: Structure of the SDF octree. Only children of traversed nodes are shown.

```
// Ordering of sub-cubes (looking top-down, upper/lower is y-coordinate):
vec3 sub_cubes[] = {
        vec3(-1.f,  1.f,  1.f), // Upper, top-left.
        vec3( 1.f,  1.f,  1.f), // Upper, top-right.
        vec3(-1.f,  1.f, -1.f), // Upper, bottom-left.
        vec3( 1.f,  1.f, -1.f), // Upper, bottom-right.
        vec3(-1.f, -1.f,  1.f), // Lower, top-left.
        vec3( 1.f, -1.f,  1.f), // Lower, top-right.
        vec3(-1.f, -1.f, -1.f), // Lower, bottom-left.
        vec3( 1.f, -1.f, -1.f)  // Lower, bottom-right.
};
```

```
for (uint i = 0; i < 8; i++)
{
        vec3 new_centre = centre + (sub_cubes[i] * size);
        if (in_cube(new_centre, size, position))
        {
                if (level == max_level) {
                        return index + i;
                }

                // Move to sub-cubes:
                centre = new_centre;
                index += uint(pow(8.f, float(max_level) - float(level))) * i;
                break;
        }
}
```

Figure 4.9: Code snippets from the octree traversal function. The voxel index is incremented by the number of leaf nodes that must appear when traversing the entire tree that comes before the current node.

The SDF is represented as an octree. Figure 4.8 shows the layout. Only the leaf nodes hold data, and the higher level nodes aren't represented at all.

The child nodes of every parent node are stored in the same order, which is what makes it possible to calculate the index without explicitly storing information about non-leaf voxels. In memory, the whole structure is represented as an array of floating point numbers; the calculated distances at the centre of each voxel. Additional information is given in the scene uniform. This includes the size and centre of the root voxel, and the total number of subdivisions of the SDF. The values in the array are ordered left to right, as the leaf nodes would be if the whole tree was expanded.

Figure 4.9 shows a snippet of shader code for traversing the tree. No memory reads are required until the index of the leaf voxel is found (this function returns said index, which is used to fetch the data in the main sphere tracing function), so most of the effort is spent traversing the tree and calculating the index. This is in line with the goal of introducing as few memory reads as possible, since they are comparatively slow. As mentioned, for an arbitrary, complex polyhedral shape, this might not be a problem, but the fractals are fairly efficient to calculate, so the SDF traversal needs to be competitive.

The calculation of the voxel index is based on its current level in the tree, compared to the maximum level, and its current position in the group of eight child voxels (see the lower image in figure 4.9). The centre positions of each voxel are calculated based on the position in the group of child voxels (the index of the loop is used to get a value from the array of positions, seen in the top of figure 4.9), and the current voxel size, which begins equal to the total SDF size, and is halved upon entering each new level.

4.5 Temporal Caching

4.5.1 Data to Cache

The data chosen to persist across frames is the true distance travelled by the ray. The aim of this optimization method is to reduce the number of iterations in certain areas of the image, especially edges and holes, so just storing the first distance estimate is not the solution. In order to preserve depth data for multiple frames, a texture image with red, green, blue and alpha floating-point components was chosen. With this, the data for the last four frames is stored. The reasoning behind only using one texture image was that sampling a texture requires an overhead that quickly builds up, so storing too much data may offset any performance benefit.

4.5.2 Image Sampling

The distance values are written to the G-Buffer for the first render pass. Every frame, the values are shifted one place to the right, discarding the last (alpha) value each time, and the new value is written to the first slot. After the render pass, the G-Buffer image is copied over to a texture image, ready for sampling from during the next frame. Each pixel only samples from its own position in the texture. It could be possible to do a kind of edge detection based on the depth values in neighbouring pixels, but this was deemed too expensive.

4.5.3 Camera Movement

```
// Step according to written distance:
vec4 distance_sample = texture(u_distance_sampler, in_tex_coord).rgba;

float distance_min = min(min(min(distance_sample.r, distance_sample.g),
                             distance_sample.b), distance_sample.a);

float distance_max = max(max(max(distance_sample.r, distance_sample.g),
                             distance_sample.b), distance_sample.a);

// Check for excess camera movement, scaled by how far away the point is:
float invalidation_threshold = 0.0001f;
if (abs(distance_max - distance_min) > (distance_min * invalidation_threshold))
{ distance_travelled = 0.f; }
else { distance_travelled = distance_min * 0.9f; }

current_position = vec4(origin + (ray * distance_travelled), 1.f);
```

Figure 4.10: Test to determine if a pixel is invalid and the ray needs recasting.

The test to see if the camera had moved too much is shown in figure 4.10. It relies on the difference between the largest distance sample and the smallest. Generally speaking, the 'landscape' of the fractals chosen is not smooth, so this proved to be a reliable test for invalidated geometry, without triggering at the smallest movement. If the test passes, and the pixel is still valid, the minimum sampled distance estimate is chosen and reduced a little, to provide a reliable underestimate of the true distance for the current frame. The usual sphere tracing algorithm is then performed, using this head-start.

Alternative data for testing for camera movement was considered as well, such as end position, end distance estimate and the lowest distance estimate along the ray, but in the end, storing the true distance travelled for the last four frames was the most effective in terms of performance and artefacts.

4.5.4 Artefacts

The method for determining pixel invalidity is, unfortunately, not perfect. A balance had to be struck between maintaining a performance boost during camera movement, and reducing artefacts associated with overshooting the true distance. Particularly, since the method chosen relies on the fractal landscape changing in depth, when there are

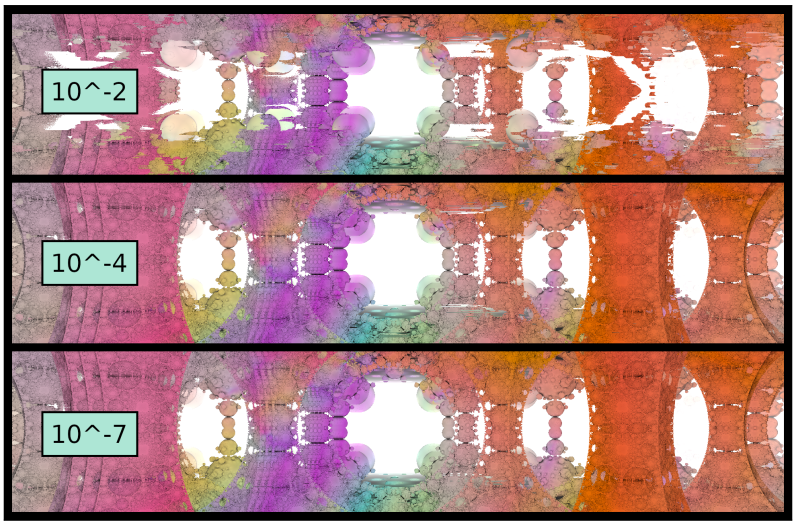


Figure 4.11: Artefacts generated in a render of the Hall of Pillars fractal, by slow sideways movement, at different pixel invalidation thresholds.

particularly smooth sections (such as when rays are hitting the view distance threshold), more artefacts appear. Changing the threshold at which pixels are declared invalid can be done to balance artefacts and performance. Figure 4.11 shows how the severity of the artefacts varies with this threshold. It ranges from near-perfect (bottom image) to unacceptable degeneration of the scene (top image).

The artefacts in figure 4.11 were triggered by slow, sideways movement and no camera rotation. The scene was chosen because of the large portion that was outside the allowed view distance, the presence of flat areas within view range, and the large number of edges at different view distances. The value for the threshold settled on was 0.0001, as shown in the centre image. This image represents a worst-case scenario for this particular scene at the different thresholds. The movement employed to get these artefacts to appear was rather specific, and unnatural for a lot of use cases, such as a game or fractal exploration animation. This is why the threshold chosen was deemed acceptable for use in this project. Any more relaxed and artefacts begin to appear during more 'natural' movement. Any less relaxed, and the performance benefits are lost during any sort of movement.

4.6 Performance Measurement

This section will go over how performance differences are measured, and how different scenarios are captured. In order to make sure the data collected is consistent, the following measures are taken:

- Wait 1000 frames before starting any measurements, to let the GPU 'warm up'.

- For static images, take measurements for 1000 frames, then take the median of the whole set, to cut out outliers.
- For animated images, run the animation 25 times, then take the median over sets of matching frames.
- Close all other programs to minimize interrupts, and turn off the screensaver.
- Make all measurements at the same image resolution: 1280 by 720.

4.6.1 Data Types Collected

The main value taken to measure performance is the execution time of the geometry render pass. To accomplish this, a Vulkan query pool is used to collect timestamps at two different stages each frame. One stage is at the beginning of the render pass and one is at the end, in order to make sure the timestamps are taken at the correct times (Vulkan commands in a command buffer may execute in any order unless specific synchronization is added), and consistently, with flags specifying the pipeline stages at which to take the measurements.

The time it takes to copy the image after the render pass is not included in the measurements, because copying the image could have been avoided by swapping the G-Buffer image and texture image after the render pass.

4.6.2 Representative Views

In order to obtain a useful set of data for performance differences, a set of representative views was chosen. They were selected to provide variety in the complexity of the scene, to see if either of the optimization methods chosen performed better under certain circumstances than in others. For example, the temporal caching method was theorized to give the most benefit in situations where there are a lot of holes and edges in the scene, so views were chosen to represent best and worst case scenarios for this theory.

The views will be displayed in chapter 5, next to the data collected for each one, so the two can be looked at together.

4.6.3 Animation

As well as static scenes, animated scenes were created. For the Hall of Pillars fractal, a fly-through of the scene was created, varying the speed of the camera. This was intended to test the performance of the temporal caching method during movement of the camera, to see if there was still a benefit to using it while moving under different

circumstances. Camera movement does not affect the SDF, so an animation was not created. Additionally since the range of the SDF is so small (especially for the Hall of Pillars fractal), an animation would have been rather limited.

The other type of animation was accomplished by varying a parameter used in the calculation of the distance function for each fractal, which resulted in smooth changes in geometry. This was done for both fractals. This type of animation was not done for the SDF optimization method, since any change in the geometry invalides the data that has been calculated.

4.7 Debugging

Debugging for this project was done manually by printing information out to the terminal. There is a function to print out all values, including handles, for the program's Vulkan structures, to check that they have been initialized properly and to track any Vulkan errors.

To debug the SDF, there is a function to print a subset of the voxels, to check that the search process works properly and that the voxels are stored in the correct order and with the expected range of values.

The key to print the current position and camera front came in useful for checking camera movement, as well as creating animations.

I also used Valgrind to search for and fix any errors with memory management. Unfortunately, it seems that either Vulkan or Volk causes some memory leaks, so debugging this was tricky. Even the function that initializes Volk introduced memory leaks.

I debugged the shaders by adding special return values at various locations, so that if there was a problem or a statement was not running when it should have been, the rendered image would not be as expected (for example, it may be entirely black). This was used in combination with RenderDoc to view the different rendering stages.

Chapter 5

Results and Evaluation

This chapter will focus on the data obtained in the project. For clarity, there is a separate section on each fractal, as they are quite different and had their own sets of representative views. A small explanation of the expected results will be given for each fractal. For the temporal caching, there were some tests that could be performed, that could not be performed with the signed distance field, because they involved animations that either changed the geometry or travelled far outside acceptable range for the signed distance field. Therefore, static image tests will be separated from animation tests. After each set of results is presented, discussion and evaluation of those results will be done.

The unoptimized time given is the time taken to execute the entire geometry render pass, in nanoseconds. The performance gain (or loss) is measured by obtaining the percentage difference between the unoptimized method, and the optimized one, like so:

$$diff = \frac{unopt - opt}{unopt} * 100 \tag{5.1}$$

where unopt and opt are the unoptimized time and optimized time, respectively. This equation will produce negative results if the time taken in the optimized render pass is greater than in the unoptimized one, so negative numbers mean a loss of performance compared to the unoptimized version.

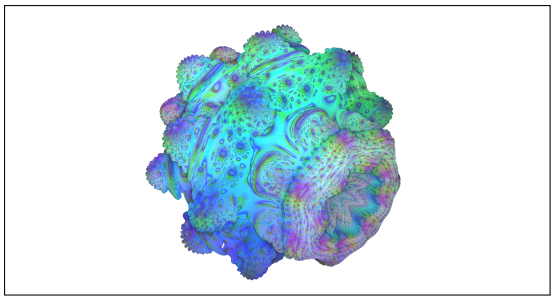
5.1 Mandelbulb Static Image Tests

5.1.1 Representative Views

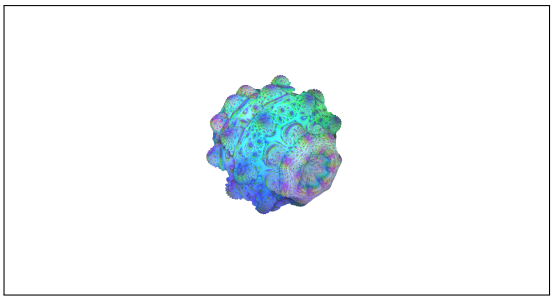
Figure 5.1 shows the four representative views used for performance measurements. Figure 5.1a shows the default view, which is not expected to be particularly well suited to any performance measurement, to get a general-case overview of the performance differences between the methods used. Figure 5.1b is the view intended to include the most empty space (short of simply turning around and not looking at the fractal at all), and by contrast figure 5.1c is intended to include the least. Lastly, figure 5.1d includes a mix of depth and smoothness, and some bottlenecks.

5.1.2 Expected Results

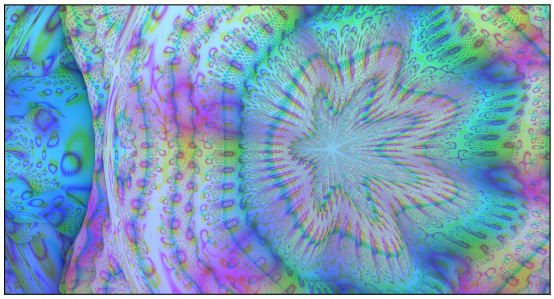
The view of the Mandelbulb fractal involves quite a bit of empty space, unlike the Hall of Pillars fractal, which contains none. The expected result is that the Temporal caching



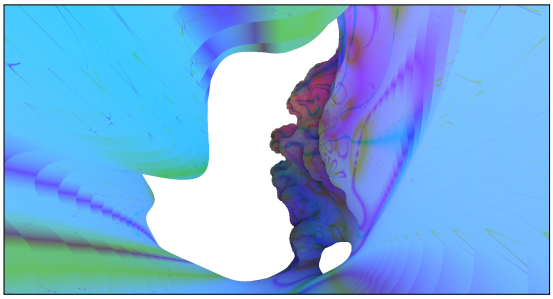
(a) Mandelbulb: Default View.



(b) Mandelbulb: Zoomed Out View.



(c) Mandelbulb: Zoomed In View.



(d) Mandelbulb: Bottleneck View.

Figure 5.1: Four representative views of the Mandelbulb fractal, used in performance tests.

method will give a slight performance boost to views with empty space, especially around the edges of the fractal, where more iterations would normally occur, but huge boosts are not expected since the Mandelbulb does not contain so many bottlenecks, and the depth variance is small. For still images, though, the temporal caching should give a boost no matter what is being looked at, as long as the camera does not move.

For the signed distance field, it entirely depends on whether searching through the three-dimensional grid is less expensive than calculating the signed distance function or not, as the signed distance field is intended to provide a cheaper alternative every iteration, not to reduce the number of iterations as the temporal caching is supposed to do. The prediction is that the more costly the function, the more benefit will be seen, but that, since the Mandelbulb function is relatively cheap, there won't be much benefit, if any, with the distance function used here.

5.1.3 Results

Table 5.1 shows the results obtained for the four representative views of the Mandelbulb fractal. It shows the unoptimized time and the performance differences with the SDF and Temporal Caching (TC).

Figure 5.2 shows the difference in the number of iterations using the unoptimized rendering and temporal cache methods, by colouring the fractal based on this number.

View	Unoptimized Time (ns)	SDF Difference (%)	TC Difference (%)
Default	1272832	-154.38	2.06
Zoomed Out	1083328	-60.08	8.67
Zoomed In	1241088	-33.74	-2.30
Bottleneck	1593888	-6.8	7.01

Table 5.1: A table showing the performance differences between the two optimization methods and the base, unoptimized rendering, for four different views of the Mandelbulb.

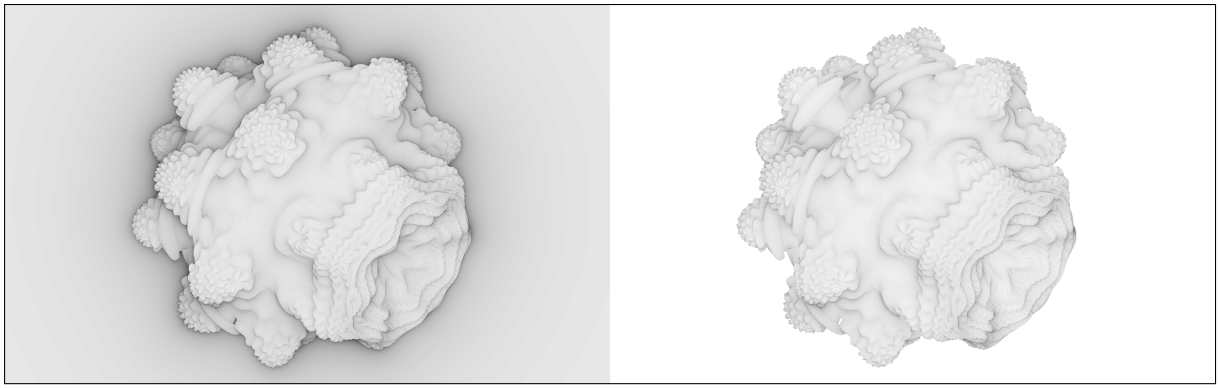


Figure 5.2: The Mandelbulb, coloured based on the number of iterations achieved, rendered using the unoptimized method (left) and the temporal caching method (right).

5.1.4 Evaluation

Following the gathering of results, I decided to do a one-iteration performance measurement. In the shaders, I allowed one iteration of either the SDF search process, or the distance estimation, for the SDF shader and the unoptimized shader, respectively. The result was as expected; the render pass took 0.446ms with no optimizations, and 0.556ms with the SDF.

Signed Distance Field

Clearly, the signed distance field did not do well for this fractal, even though at first glance it seems a perfect fit for it, seeing as the whole fractal fits inside a reasonable boundary. I think overall, this is because the distance estimator for the Mandelbulb is so cheap, so the search through the SDF is outpaced by the distance function.

The worst performance is seen with the default view. This may be because the whole fractal is visible, and fairly close to the camera, so fine detail is visible. This means that more iterations are needed, and most of them will fall within the cube, at least at first, making this the worst performer. Contrast that with the zoomed out view, where most of the rays fall outside the boundary of the SDF, which saves the performance a little, as the shader will revert to using the regular signed distance function if this happens. Additionally, less of the fractal is visible here, and in less detail. It seems that the

performance measurements for the Mandelbulb simply show which of the views the SDF fights against the least.

The zoomed in view is a little better. Considering the performance of the SDF so far, I think this view is better simply because it is closer to the fractal. This means that less iterations are done before the ray gets close enough to the surface of the fractal to begin using the regular signed distance function. I think the bottleneck view performed the best for a similar reason, with the addition that the surface is even simpler in this view, and the view out to the background is quite close to the edge of the SDF boundary, so it wouldn't be very many iterations once past the bottleneck to be free of the SDF cube.

The SDF is not a suitable optimization for this fractal, in its current state. The process of searching through the SDF is clearly more expensive than the signed distance function calculation. If I were to use a more expensive fractal then maybe using the SDF would be worthwhile.

Temporal Cache

The temporal cache performed much better than the SDF, to the point where it actually gives a performance boost. The scenarios are also a little easier to explain. Starting with the zoomed in view, which decreased the performance. The surface in this view is quite detailed, and close to the camera. I think the performance difference was negative because of the deliberate underestimate that happens when using the saved distance. It is multiplied by 0.9, meaning that any iterations in that last 10% are lost every frame. In this case, I think that most of the iterations happen in this last 10%, where the surface becomes more complex, so the temporal cache provides no benefit here. The negative difference most likely comes from the overhead involved in sampling from the texture image, and the extra comparisons that happen for the camera movement test, for example.

The largest performance difference came from the zoomed out view. I was expecting it to come from the bottleneck, but I suppose that in terms of edges, the zoomed out view contains just as many. In addition, the empty space will also receive a boost, and not just around the edges, so a large part of the performance increase likely comes from here. As well as this, less of the fractal surface is visible, so although the problem present in the zoomed in view applies, and most of the iterations for the surface come from that last 10% that is cut out, the portion of the image to which this applies is very small in this view.

The performance for the default view lines up with the explanations for the zoomed out

and zoomed in views. There is more empty space, and there are more edges, in the default view than in the zoomed in view, so the performance will be better, but there is also more surface space, which this optimization method does not perform well on, so that will cause a hit to overall performance.

This drawback does not apply so much to the bottleneck scenario, despite it being close to the surface. There is not a lot of complicated geometry in this view, so the cut to iteration progress doesn't hit the performance so hard. Most of the performance boost probably comes from the avoidance of the obvious bottleneck (hole) in the middle, and the smaller one on the right.

Figure 5.2 shows a distinct reduction in the number of iterations using the temporal caching method, especially in the background and at the edges of the Mandelbulb. However, during testing the image did flicker, especially in the background. I think this is because of the view distance limit. The rays will travel a little further each frame, which triggers the ray recasting. If this was fixed, then one might expect better performance from the temporal cache. Overall, the temporal cache is worth using for the Mandelbulb fractal, even though there are some improvements to be made.

5.2 Mandelbulb Animation Test

This section will examine the performance while animating the Mandelbulb by changing its parameter. Recall figure 4.3 in chapter 4. It shows the Mandelbulb, raised to different powers. This power is the parameter that is varied in the animation.

5.2.1 Animation

The animation of the Mandelbulb parameter takes a total of 8000 frames. The parameter starts at 10 and gets raised to 16 by the end of the first 2000 frames. Then, it decreases to 4 over 4000 frames. Finally, over 2000 frames it returns to 10. Refer back to figure 4.3 in chapter 4 for an overview of how this looks.

5.2.2 Expected Results

I don't have solid predictions for this animation. I expect to see some gains in some sections, and perhaps some losses in others. Any section of the animation that is visually slower will perhaps trigger the ray recast mechanism less, so there would be gains here. Any faster animation, and the performance may stay the same or decrease. Any part of the animation that involves more empty space will likely see gains, based on the results for the static images.

5.2.3 Results

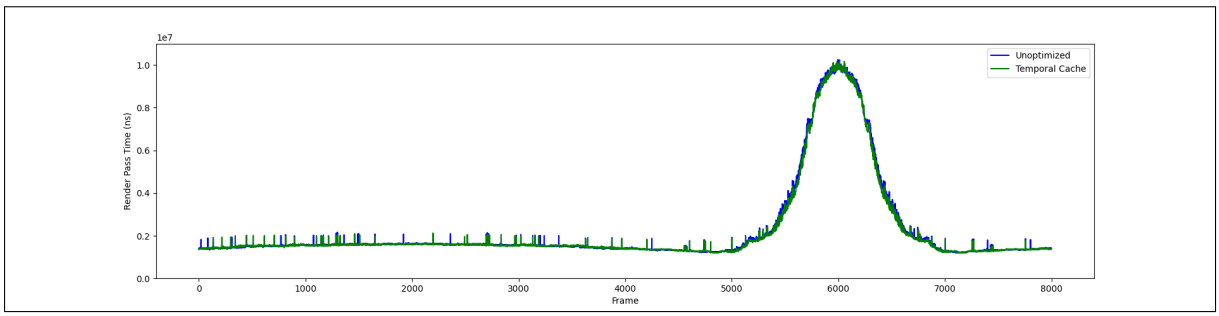


Figure 5.3: A graph showing the render pass time during an animation of the Mandelbulb parameter, for both unoptimized rendering and rendering using the temporal cache.

Figure 5.3 shows the graph of the render pass time over the animation. It is very difficult to see if there are performance differences anywhere in the animation from this, as the two lines overlap almost exactly, but it's useful to know where the performance changes overall during the animation. Figure 5.4 shows a plot of the performance difference over the animation, with a line of best fit plotted to make it clearer (the line of best fit ignores the large spikes in the data).

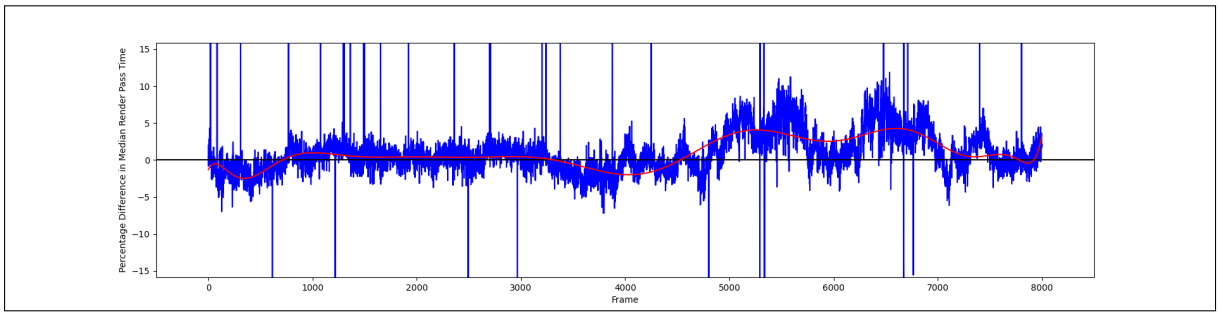


Figure 5.4: A graph showing the performance difference during an animation of the Mandelbulb parameter, between unoptimized rendering and rendering using the temporal cache. Positive values mean that the temporal cache performs better.

5.2.4 Evaluation

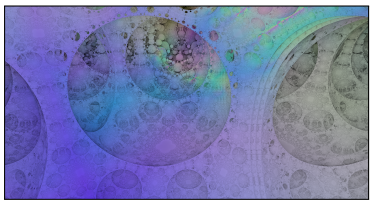
The main surge in render pass time, in figure 5.3, comes about when the parameter is approaching its lowest value. The number of iterations required to reach the surface increases because the surface becomes more intricate. This difference can actually be seen in the image from chapter 4, figure 4.3. The image on the left is noticeably darker. This is because the image colouring is done in combination with the ambient occlusion effect described. There is a greater portion of smooth surface, as well as empty space, at this point.

Around the same time as the surge in render pass time, there is a general increase in the performance benefit received from the temporal cache. I believe this is because of the extra smooth surface and empty space, as described. The animation is moving at the same speed, but the changes in the landscape are less from frame to frame. Contrast this with the times around frames 0/8000 and 4000. This is where the parameter reaches 10, which is the starting value. At this point, the landscape is changing the quickest, with new geometry occluding the old at high rates, and less empty space. When the parameter reaches its peak, 16, at frame 2000, these occlusions are still happening, but at a lower rate, so the performance does not dip in the same way as when the animation is at its fastest in terms of occlusion.

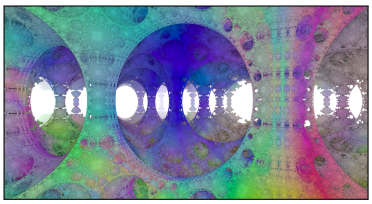
Overall the temporal cache performs quite well during the animation, despite, at times, rapid changes in geometry. Additionally, there are more gains than losses, and the losses are smaller than the gains.

5.3 Hall of Pillars Static Image Tests

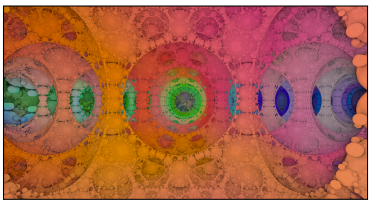
5.3.1 Representative Views



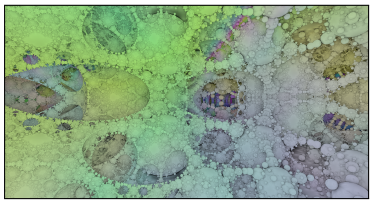
(a) Hall of Pillars: Default View.



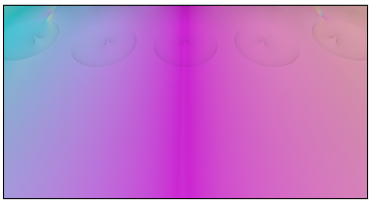
(b) Hall of Pillars: Large Hall View.



(c) Hall of Pillars: Bottleneck Corridor View.



(d) Hall of Pillars: Intricate Geometry View.



(e) Hall of Pillars: Simple Geometry View.

Figure 5.5: Five representative views of the Hall of Pillars fractal, used in performance tests.

Figure 5.5 shows the representative views chosen for the Hall of Pillars fractal. These were chosen to vary greatly in depth and complexity. Given the theory on why the temporal cache should give a performance boost (bottlenecks), images 5.5a and 5.5b were chosen to provide an average case, images 5.5c and 5.5d were chosen as best cases, and finally 5.5e was chosen as a worst case. These views also provide good variety for

testing the SDF. All but the 'Large Hall' view are mostly encompassed by the SDF; the Hall is just too large in scale to be covered adequately.

5.3.2 Expected Results

This fractal contains many bottlenecks (the ray has to pass close to lots of geometry before reaching any surface), so the temporal caching method is expected to perform better here than with the Mandelbulb, in terms of performance difference. A view with very simple geometry was also chosen because of this, as the prediction is that the temporal caching method will provide minimal benefit where these bottlenecks don't exist or the surface is very smooth.

The signed distance field is being used in a somewhat different scenario compared to the Mandelbulb. This fractal doesn't seem to be finite, so only a relatively small area can be covered by the SDF, which makes testing tricky. A ray culling step was tried, to stop any ray which ventures outside the main SDF cube (including in shaders that aren't using the 3D SDF, for consistency) but this did not yield significantly different results to using it as normal, and just allowing the rays to travel outside the box. It was also not representative of the kind of uses for this fractal. Especially in the 'Large Hall' view, in which the SDF can only cover a segment of one pillar. I predict that, like with the Mandelbulb, this will not perform well, as the Hall of Pillars distance estimator is not significantly more expensive than the Mandelbulb one.

5.3.3 Results

View	Unoptimized Time (ns)	SDF Difference (%)	TC Difference (%)
Default	7524352	-2.08	24.02
Large Hall	12879840	-1.39	12.43
Bottleneck Corridor	10053440	-10.76	26.07
Intricate Geometry	10091680	-8.04	25.62
Simple Geometry	1176352	-4.76	12.39

Table 5.2: A table showing the performance differences between the two optimization methods and the base, unoptimized rendering, for five different views of the Hall of Pillars fractal.

As before, table 5.2 shows the raw time for the render pass in the unoptimized rendering, as well as performance differences for the SDF and temporal caching (TC) methods, and figure 5.6 shows the differences in the number of iterations achieved.

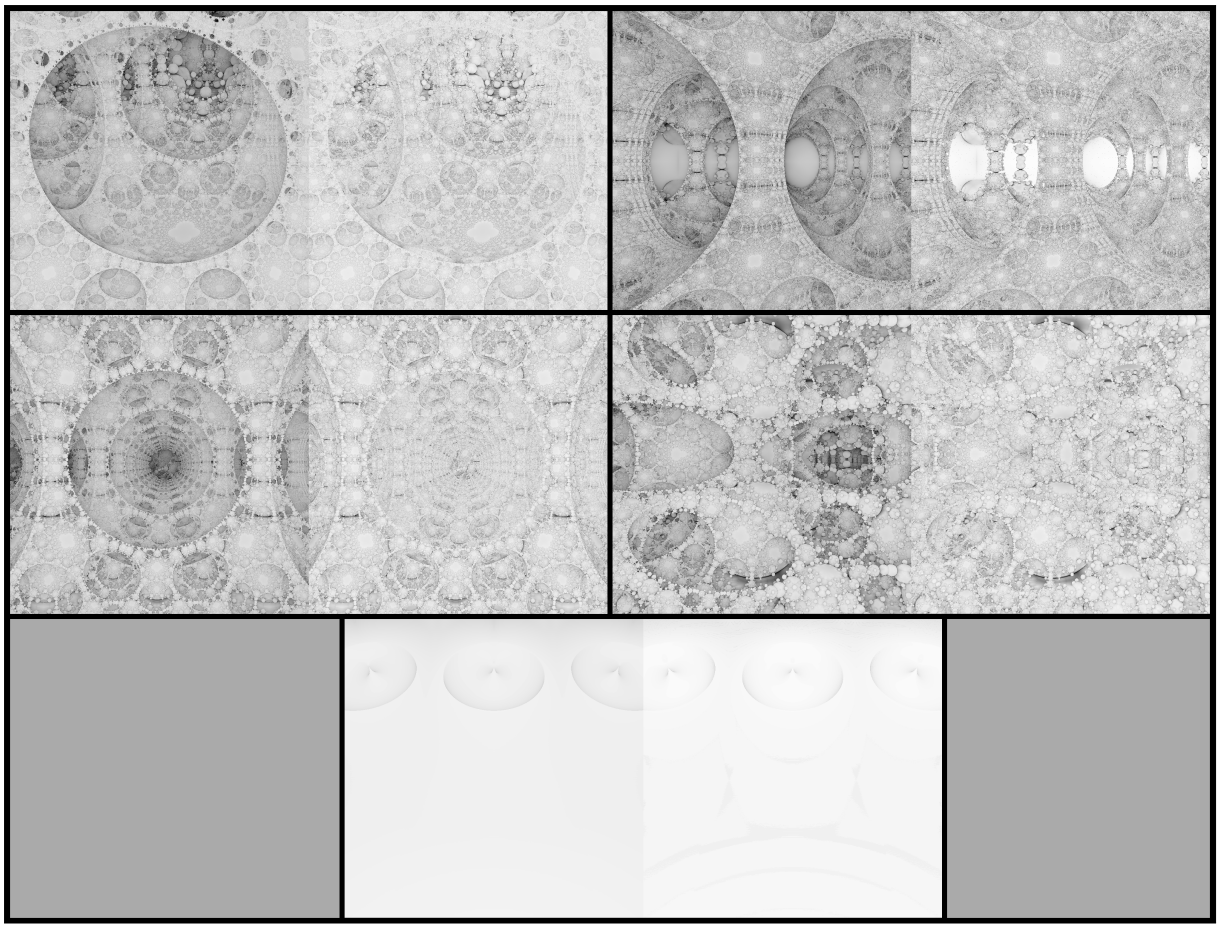


Figure 5.6: The Hall of Pillars fractal, coloured based on the number of iterations achieved. Each image is split in two; the left half is rendered using the unoptimized method, and the right half uses the temporal cache. The lighter the image, the better the performance.

5.3.4 Evaluation

Signed Distance Field

Compared to the SDF performance for the Mandelbulb, the SDF here performed extremely well, though it still reduced performance in all cases. I think that if the Hall of Pillars distance estimator was more expensive to evaluate, there may have been some positive results. That being said, the SDF was comparatively a lot more coarse than for the Mandelbulb, though it did benefit from an extra subdivision level. I did try changing the subdivision level from 9 to 8, but this did not result in a performance difference.

I think that perhaps for scenes that cross a great deal of the SDF, the shaders are likely to encounter a lot more cache misses, as the total size of the SDF in these tests was 512MB. This will lower performance for scenes such as this. If the subdivision level is too low, and the SDF stretched across too much space, then the voxels will be large, and it will be easier to get past the SDF search stage to fall back on the regular signed distance function. Recall figure 4.7 from chapter 4. The SDF is clearly a lot more coarse

for the Hall of Pillars, in order to cover any kind of decent distance. Another level of subdivision was not possible due to memory constraints. I think this is one reason for the 'increase' in performance compared to the Mandelbulb; it simply did not get in the way as much. This is evidenced by the fact that in the view in which it covers the least ground, the Large Hall, we have the smallest performance loss.

Further evidence for this claim is the result for the Bottleneck Corridor view. This view contains a great deal of tunnels and holes in the scene. This view was chosen because I thought it might slow the rays down the most, and therefore result in the greatest performance gain for the temporal cache. It has also resulted in the greatest performance loss for the SDF. The scale of this view means that most of it is contained inside the SDF. This is the worst case out of the views. Lots of bottlenecks, almost all of which are inside the SDF. The Intricate Geometry view has a very similar problem; almost all of that is contained inside the SDF as well (this goes for all views except the Large Hall).

Overall, although the SDF performed better than with the Mandelbulb, I believe this to be an illusion. It is not suitable for use in its current state.

Temporal Cache

In contrast to the SDF, the results for the temporal cache were much improved over those it achieved on the Mandelbulb. Even the worst case view resulted in a 12.39% performance gain. However, this is for a static image, and much of the gain is likely to be lost during movement (more on that in the next section). There isn't much to explain with these results. They are as expected, and likely for the reasons predicted; the presence of a large number of bottlenecks in the scene, and complex geometry. The worst performers were the view with the simplest geometry, as expected, but also the one with the largest view distance ('Large Hall'). This was surprising, as the scene does contain a lot of bottlenecks.

Figure 5.6 shows the comparison between the number of iterations achieved with unoptimized rendering and the temporal cache, for each of the representative views. The differences seem to match up with the data obtained. The difference is particularly stark for the Bottleneck Corridor view and the Intricate Geometry view. During rendering, the same flickering happened in the background of the Large Hall view as it did for the Mandelbulb, which may be why, despite there being a lot of bottlenecks in the scene, the performance wasn't better. I did expect this view to yield more of a benefit than it did.

The temporal caching method is particularly well suited to this fractal. The next section will explore how much of the performance gain can be preserved with movement.

5.4 Hall of Pillars Animation Tests

5.4.1 Animations

Two animations were constructed for the Hall of Pillars fractal. One is a flythrough of several scenes, and the other is an animation resulting from varying a parameter in the distance function. Refer back to figure 4.4 in chapter 4 to see the use of the fractal parameter in the distance estimation.

Flythrough

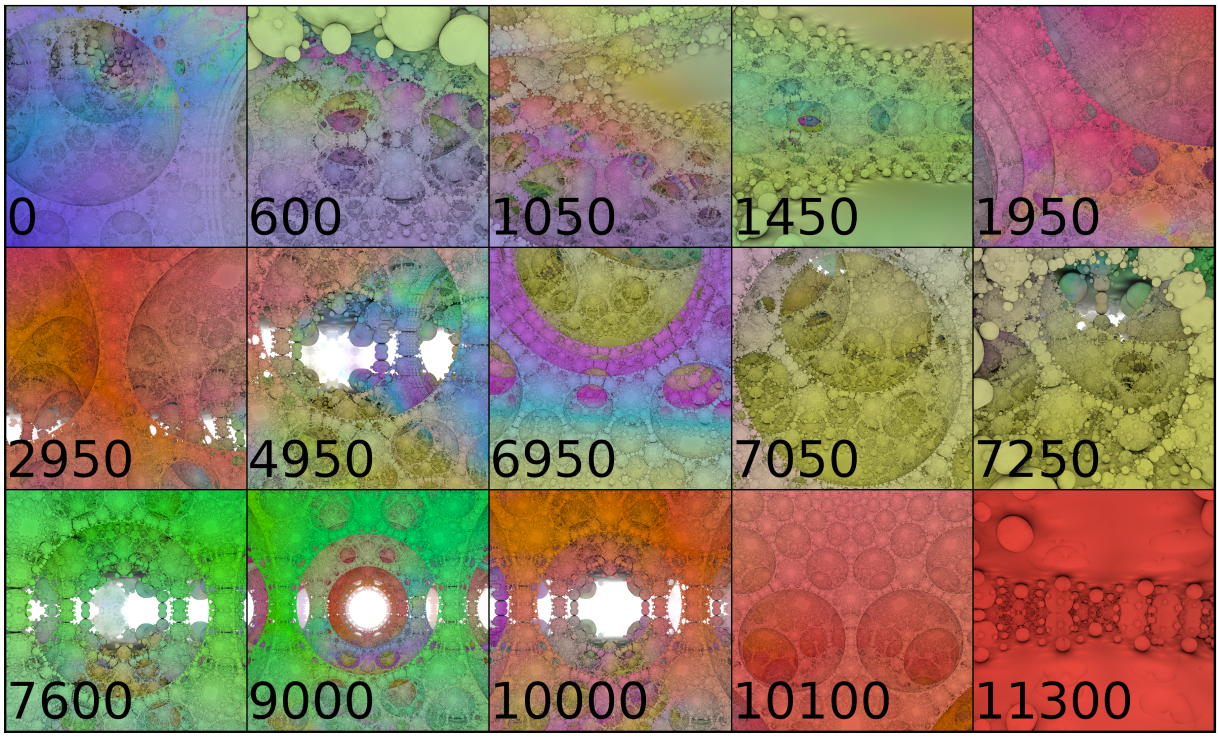


Figure 5.7: Key frames in the Hall of Pillars flythrough animation. They are in order, left to right, top to bottom. The frames at which they occur are written on each image.

Figure 5.7 provides a guide to the progress of the animation at key points. To give a sense of scale, any image in which you can see white is a scene where the view distance limit (which is just over sixteen thousand) is exceeded. The first four images have a range of approximately five hundred. The penultimate image is of the top of a pillar; a corresponding structure can be seen at the top of the image preceding it (frame 10,000). The last image is inside the top of this structure. The images shown are key images in relation to the graph of performance that will be shown in the results section. These will be explained there, but refer back to this image to aid in understanding what is happening during the animation to cause fluctuations in performance, and performance gain.

Parameter Animation

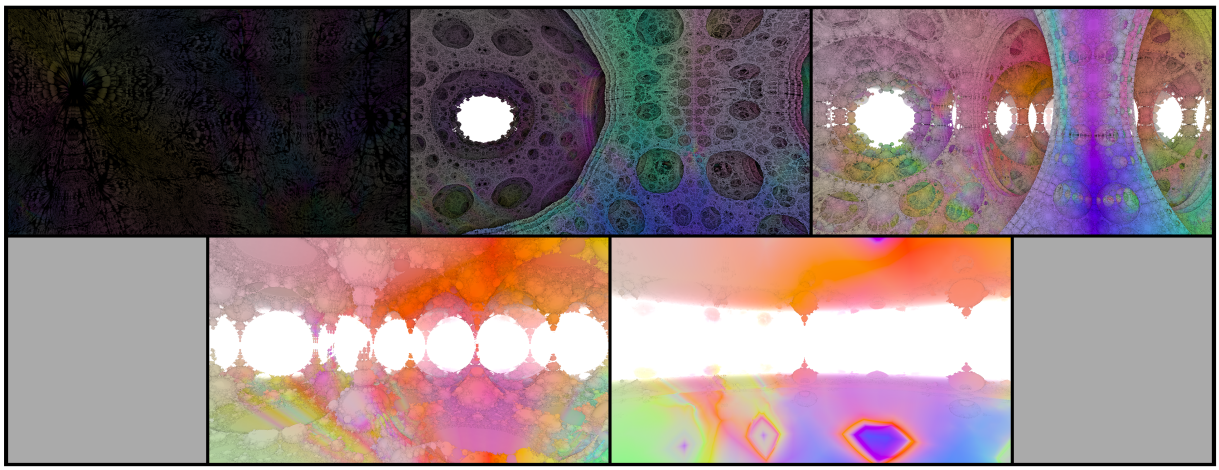


Figure 5.8: Key stages in the Hall of Pillars parameter animation. The parameter values are as follows, left to right, top to bottom: 1.6, 1.8, 2.0, 2.2 and 2.4.

Figure 5.8 shows a guide for the stages of the parameter animation. The sequence starts in the middle (the third image) and moves towards the scene in the last image. Then, the animation reverses and morphs in to the first image. Finally, it moves back to the start. The first image is so dark because the scene takes progressively more iterations to render as the parameter decreases, and these images are rendered with the 'ambient occlusion' effect previously mentioned.

5.4.2 Expected Results

I expect to see a performance gain over some of the flythrough animation - particularly sequences in which movement is slow, and there is no camera movement. I don't expect to see a gain where there is movement, combined with complex geometry, or a lot of camera movement; recipes for lots of occlusion and ray recasting.

For the parameter animation, the view is much more stable, given that the camera doesn't move at all, but the geometry changes. With this in mind, I expect performance somewhere between the static images and the flythrough animation.

5.4.3 Results - Flythrough

Figure 5.9 shows the graph of the render pass time over the animation. A small performance difference can be seen here (where the blue line is visible above the green one), despite the general noisiness due to the changing landscape. The sharp drop in time just before frame 2000 corresponds with the camera moving through the 'ceiling' of one of the rooms in the fractal. During this time, the number of iterations will drop to 1 or close to 1, as the camera is inside the shape. The time drops close to zero, but this

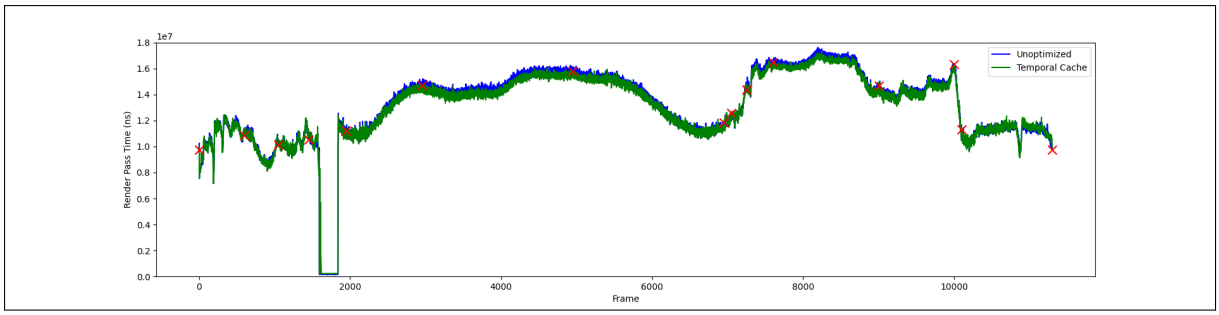


Figure 5.9: A graph showing the render pass time during a flythrough animation of the Hall of Pillars, for both unoptimized rendering and rendering using the temporal cache. The red X marks indicate key frames for the animation.

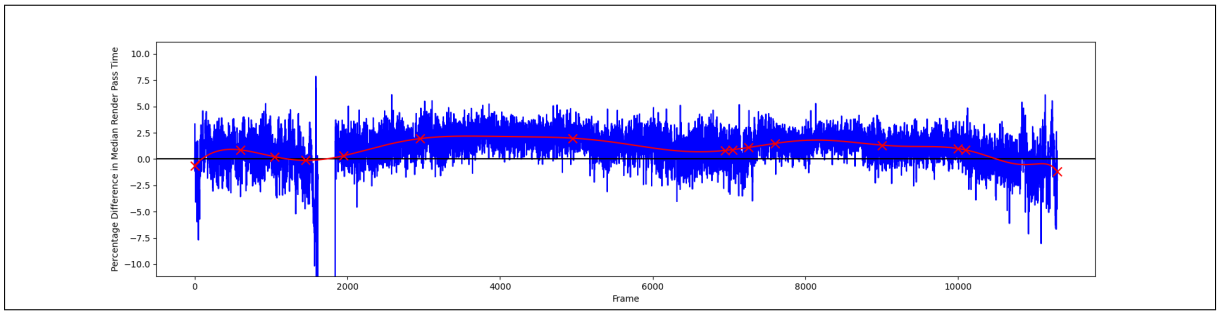


Figure 5.10: A graph showing the performance difference during a flythrough animation of the Hall of Pillars, between unoptimized rendering and rendering using the temporal cache. Positive values mean that the temporal cache performs better. The red X marks indicate key frames for the animation.

was not considered an interesting part of the animation, so it is cut off a bit in these graphs. Perhaps it is interesting to note, however, the drop in performance around that time (see figure 5.10). This is likely because of the overhead involved in sampling the texture image, and the extra calculations, for no reduction in iteration count.
Figure 5.10 confirms the observation of a performance boost. A positive difference in performance can be observed over most of the animation, meaning that even for scenes with more rapid movement, the temporal cache still maintains its advantage.

5.4.4 Evaluation - Flythrough

Surprisingly, the temporal cache managed to provide a performance benefit throughout the animation, though it was much reduced. It seems to fluctuate mostly between approximately 1% and 2.5%. This is a huge reduction from the numbers seen for the static images, but was to be expected.

The sequence between frames 0 and 600 involves fairly quick movement, but slow camera rotation. There is also a lot of complex geometry in this sequence, mixed with some flat surfaces. I think the performance increases during this sequence because the flat areas can be seen through bottlenecks. The combination of these things means that

there is a bottleneck-related performance gain, and the frame-to-frame distance doesn't change too much.

After this sequence, the camera dive into one of the holes near the ground. The geometry becomes less complicated during this time, and the distance from the camera to any surface decreases. Not much performance benefit can be seen here as a result. After this, as mentioned, the camera goes through the ceiling, which results in a dramatic decrease in performance. Upon emerging on the other side, the image in frame 1950 (top-right in figure 5.7) can be seen. The camera is still moving, and a large portion of the bottlenecks in this particular scene are near the end of the ray's journey, so some of the performance benefit will be lost due to the deliberate underestimate.

The camera then pans across the ceiling of this large hall to end up at the view in frame 2950. During this sequence the performance increases to its highest level. The camera movement is very slow, as is the rotation, and the movement brings into focus a scene more suited to the strengths of the temporal cache. This scene slowly evolves for 2000 frames until the view in frame 4950. The performance benefit is maintained for this entire duration. This sequence has the best performance for the temporal cache. I think this is due to the very slow movement and rotation, and the temporal cache-friendly scenery.

The performance dips after this, as the camera takes another dive into the base of the right-hand pillar. The geometry contains less bottlenecks and complexity for this sequence, but the camera rotation and movement is still slow, so performance stays in the positive. Here, there is a sequence where the camera moves quickly through a couple of 'tunnels', to emerge in another area with a large amount of complex geometry and high view distance, resulting in the second best performance at around frame 7600.

After this, the movement is quite slow, but there are a couple of fast, 90 degree turns of the camera between frames 7600 and 10000. Still, the temporal cache manages to maintain overall positive performance. The performance really suffers when zooming in to the top of a pillar near the end of the sequence. As you can see from the last image, the geometry is simple, and since the camera is moving and rotating at a moderate rate, the ray casting mechanism will be triggered quite often.

Overall the temporal cache holds up well with movement and doesn't completely deteriorate, but much more work is needed to boost the performance during movement. I think a better, more reliable way to detect occlusion would allow for a closer estimate of the true distance, reduce the need for such a harsh underestimate during sampling, and help to trigger ray recasting a lot less often.

5.4.5 Results - Parameter Animation

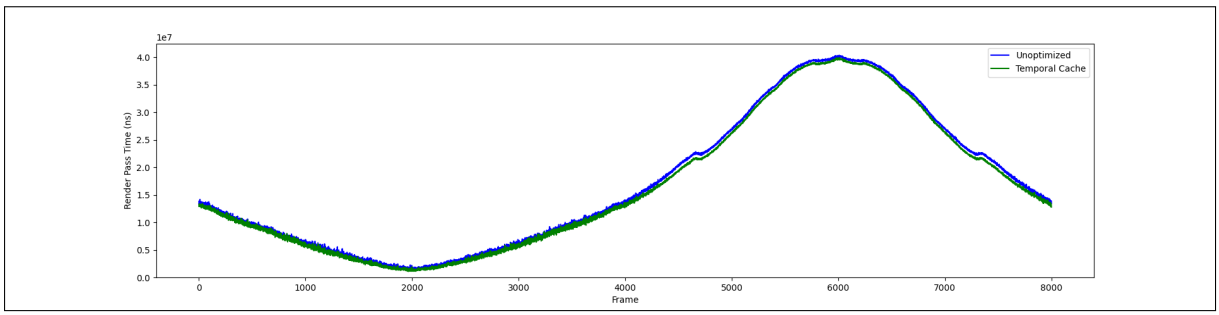


Figure 5.11: A graph showing the render pass time during an animation of the Hall of Pillars parameter, for both unoptimized rendering and rendering using the temporal cache.

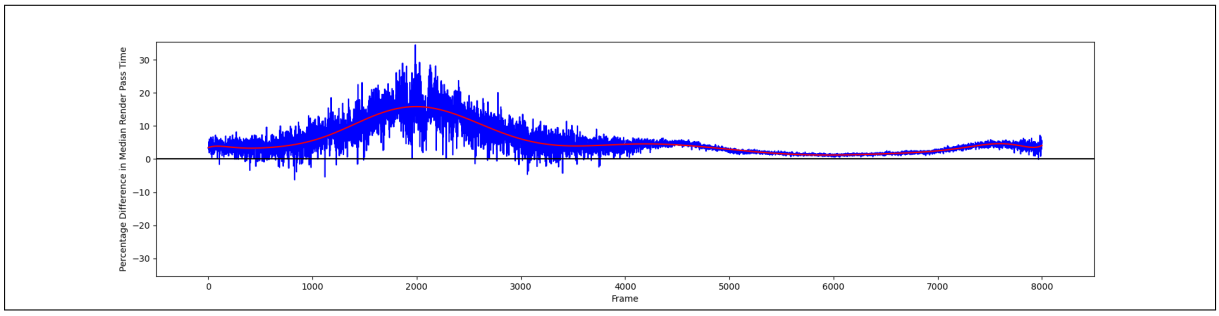


Figure 5.12: A graph showing the performance difference during an animation of the Hall of Pillars parameter, between unoptimized rendering and rendering using the temporal cache. Positive values mean that the temporal cache performs better.

Figures 5.11 and 5.12 show the render pass time and performance difference for this animation, respectively. The performance difference is much easier to see this time, even in figure 5.11. Still, this view is a bit misleading, as the performance gain is higher in the noisier areas near the start of the animation.

5.4.6 Evaluation - Parameter Animation

The temporal cache performs very well in this animation. Overall, I think this is because the camera never moves, only the geometry, resulting in a much more temporally stable image.

As mentioned, the number of iterations required to render the image increases as the parameter decreases. This is why the overall render time changes. The parameter increases first, reaching the peak at 2000 frames, then it decreases, reaching its lowest at frame 6000. It then returns to the start. An interesting feature is the small bump in the graph in figure 5.11. Refer back to the animation guide image, figure 5.8 and compare the second and third images. As the pillar on the right becomes larger, it blocks out the

light, so to speak. I think this is what causes this small plateau. However, there is no corresponding feature of interest in figure 5.12.

The greatest performance gain occurs when the parameter hits its highest value, 2.4, and the gain is lowest when the parameter hits its lowest value, 1.6. I think this is partly because the actual movement of geometry is slowest at the peak, and partly because the image at the peak has geometry that resembles one huge bottleneck. I would have expected the image before it to yield greater performance benefits, though. I think it does not because the geometry is changing too rapidly and is triggering ray recasts.

At the lowest parameter level, the geometry becomes very complex (hence the dark image). It is also moving, which will trigger ray recasts a lot of the time, which I think it why the performance is lower here.

Overall, this gave levels of performance somewhere between that for the flythrough and static images, as expected. In all three scenarios, the temporal cache has proved to be worth using, rarely giving performance losses, and only giving small ones when is did. Mostly it seems to be a positive influence on performance.

Chapter 6

Conclusion

This chapter will summarize the paper by giving an overall impression of the two optimization techniques implemented, comparing the project's end product with the initial goals, discussing any difficulties encountered, and describing plans for further work.

6.1 Conclusions

The results indicate a clear winner in terms of the optimization techniques; the temporal cache. It gave performance benefits in almost all situations it was tested in, even during movement, which was its biggest weakness. The SDF, on the other hand, performed poorly in all situations it was tested in. I think this was because the signed distance functions used were simply not computationally expensive enough to justify the use of an SDF. Additionally, the range of the SDF was so limited in the case of the Hall of Pillars fractal, it wasn't worth using at all. I do think, however, that there are some things I could do to try to make the method more viable. These will be discussed in the section 6.4, below.

6.2 Project Goals

The aim of this project was to attempt to improve the performance when rendering 3D fractals. Two methods were employed in order to explore this idea; a 3D Signed Distance Field, and a temporal caching method. The software is configurable through its arguments, which were decided as part of the software design and aided in the construction of the project's layout. The software is capable of measuring the performance of the rendering, by means of the render pass time, and writing data out to file to be processed. Overall, the original goals of the project have been met. It was unfortunate that one of the optimization methods turned out to be inadequate, but the other one turned out to have a lot of potential.

6.3 Difficulties

There were a couple of significant difficulties that arose. First, the Vulkan setup took much longer than expected, leaving less time for implementation of the optimization methods. It would have been nice to have more time for experimentation (I had lots of

ideas in mind), and time to try to improve the results with the SDF, but overall this hiccup did not derail the project.

The second difficulty was, of course, the performance of the SDF. While this is not necessarily a terrible thing, since I did implement a method that worked in the end, I did spend quite a bit of time trying to improve the SDF and taking performance measurements with different configurations, to no avail. This also left me with less time to try to improve the temporal cache method, which would have been more fruitful, I think.

6.4 Further Work

This section will be split into subsections, each tackling a different area of the project I would like to work on in future.

6.4.1 Rendering

One drawback of the temporal cache is that it ruins the free ambient occlusion effect that comes with the sphere tracing algorithm. Implementing some kind of method to reduce this would be an interesting task. Perhaps keeping track of the number of iterations during the last complete raycast would work.

Lighting is another thing that could be implemented. It's possible to calculate normals based on the gradient of the signed distance function along the ray. I should think that using the temporal cache would ruin this as well, but there may be another acceptable way to calculate normals.

6.4.2 Project Application: Game

Eventually, I would like to use fractals as terrain in a game. This will require me to implement a collision system for the fractals; this could be done simply by evaluating the signed distance function for the fractal, and declaring that a collision has occurred if the player is within some distance. Being able to influence the terrain would be excellent as well. During the Hall of Pillars flythrough, the camera moves through the ceiling, which could be considered immersion-breaking. Instead, it would be better to be able to, for example, drill tunnels through the fractal. I would implement this by storing a set of primitives that represent these holes, and if the ray is travelling through one of them, the fractal will not be calculated. I'm particularly excited about the possibility of morphing the terrain during the game, and implementing physics-based puzzles using this mechanic.

6.4.3 SDF

First and foremost, making the SDF more memory efficient would be very important. During development, the first SDF implemented did actually attempt (successfully, I might add; the number of voxels was reduced by a factor of 10) to construct a sparse octree, based on the distance calculated at the centre of the cube. If the fractal surface was not likely to be inside the cube, the cube was not subdivided. However, the structure I came up with for traversing the octree was extremely inefficient, to the point that it made the SDF less memory efficient than it turned out to be in the end, with the regular octree. I would be interested in rolling back to this attempt and implementing it properly. This may allow the SDF to extend far beyond its current range.

Additionally, finding a more computationally expensive fractal to use would be interesting, to see if the SDF actually has promise in some circumstances. It would be excellent, as the SDF is theoretically immune to the pitfalls of the temporal cache (movement).

Lastly, finding a more efficient way to construct the SDF would be preferable. At the moment, it's generated offline on the CPU, but if I could move this work to a compute shader (which would be easier to parallelize with a dense octree than with a sparse one), it may be a lot quicker. I also had a plan to move the SDF with the camera, so that the camera is always surrounded by it, up to a certain view distance. I was going to achieve this by having a grid of SDF cubes. If the camera moved out of range of the end row or column, then the out of range one would be discarded and a new one would be generated in the direction of movement. If the SDF cubes were small enough, this could potentially be done in real time in the shaders.

If an efficient method of generating the SDF in real time can be implemented, then the SDF would be able to handle changes in geometry, which it cannot currently do.

6.4.4 Temporal Cache

The most important piece of work to do here is to find a way to more accurately determine if a closer object has occluded the current ray. Improving this would result in less ray recasts, a reduced need for such a large underestimate of the true distance, less artefacts and better performance during movement.

As mentioned in chapter 4, the image output from the geometry render pass is currently copied to the texture image to be sampled in the next frame, after the render pass. This could be avoided completely, by simply swapping the G-Buffer image and texture image

after each frame. Another way this could be avoided is hopefully coming soon. There is to be a new Vulkan extension, 'VK_EXT_attachment_feedback_loop_layout', that will allow a single image to be both sampled from and used as a framebuffer image, in the same render pass. This sounds perfect for my purposes, as each pixel only writes to, and reads from, the same pixel each frame, so there would be no interference or data dependencies between pixels.

References

[1] P. Nylander, "Hypercomplex fractals." `http://www.bugman123.com/Hypercomplex`, 2009. accessed: 31/07/22.

[2] I. Quilez, "Distance to fractals." `https://iquilezles.org/articles/distancefractals`, 2004. accessed: 5/08/22.

[3] D. Hoskins, "Fractal explorer." `https://www.shadertoy.com/view/4s3GW2`, 2016. accessed: 9/08/22.

[4] R. L. Devaney, "The mandelbrot set, the farey tree, and the fibonacci sequence," *The American Mathematical Monthly*, vol. 106, no. 4, pp. 289–302, 1999.

[5] D. Ashlock, "Evolutionary exploration of the mandelbrot set," in *2006 IEEE International Conference on Evolutionary Computation*, pp. 2079–2086, IEEE, 2006.

[6] J. Aron, "The mandelbulb: first 'true'3d image of famous fractal," *New Scientist*, vol. 204, no. 3736, pp. 54–55, 2009.

[7] R. Rucker, "In search of a beautiful 3d mandelbrot set." `https://www.rudyrucker.com/blog/notebooks/rucker_mandelbulb_ver7_sept24_2009.pdf`, 2009. accessed: 31/07/22.

[8] A. Marrs, P. Shirley, and I. Wald, *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Springer Nature, 2021.

[9] C. Robles, H. Lee, S. Yin, and V. Dhar, "Procedural rendering w/ray marching,"

[10] E. Haines and T. Akenine-Möller, *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Springer, 2019.

[11] J. C. Hart, "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, vol. 12, no. 10, pp. 527–545, 1996.

[12] D. Koschier, C. Deul, and J. Bender, "Hierarchical hp-adaptive signed distance fields.," in *Symposium on Computer Animation*, pp. 189–198, 2016.

[13] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones, "Adaptively sampled distance fields: A general representation of shape for computer graphics," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 249–254, 2000.

[14] D. Meagher, "Geometric modeling using octree encoding," *Computer graphics and image processing*, vol. 19, no. 2, pp. 129–147, 1982.

[15] B. Cosenza, G. Cordasco, R. De Chiara, U. Erra, and V. Scarano, "On estimating the effectiveness of temporal and spatial coherence in parallel ray tracing.," in *Eurographics Italian Chapter Conference*, pp. 97–104, 2008.

[16] D. Scherzer, L. Yang, O. Mattausch, D. Nehab, P. V. Sander, M. Wimmer, and E. Eisemann, "Temporal coherence methods in real-time rendering," in *Computer Graphics Forum*, vol. 31, pp. 2378–2408, Wiley Online Library, 2012.

[17] M. Weier, T. Roth, E. Kruijff, A. Hinkenjann, A. Pérard-Gayot, P. Slusallek, and Y. Li, "Foveated real-time ray tracing for head-mounted displays," in *Computer Graphics Forum*, vol. 35, pp. 289–298, Wiley Online Library, 2016.

[18] T. A. S. Foundation, "Apache license, version 2.0." `https://www.apache.org/licenses/LICENSE-2.0`, 2004. accessed: 12/08/22.

[19] K. Group, "Vulkan github repository license." `https://github.com/KhronosGroup/Vulkan-Headers/blob/main/LICENSE.txt`, 2022. accessed: 7/08/22.

[20] Zeux, "Volk github repository license." `https://github.com/zeux/volk/blob/master/LICENSE.md`, 2022. accessed: 7/08/22.

[21] GLFW, "Glfw license." `https://www.glfw.org/license.html`, 2022. accessed: 7/08/22.

[22] M. H. Christensen, "Distance estimated 3d fractals (part viii): Epilogue." `http://blog.hvidtfeldts.net/index.php/2012/05/distance-estimated-3d-fractals-part-viii-epilogue/`, 2012. accessed: 9/08/22.

Appendices

Appendix A

External Material

A.1 Third-Party libraries

The third-party libraries used in the project, and their licenses, are:

A.1.1 Vulkan

`https://www.vulkan.org/`

Apache License 2.0:
`https://github.com/KhronosGroup/Vulkan-Headers/blob/main/LICENSE.txt`

A.1.2 Volk

`https://github.com/zeux/volk`

MIT license: `https://github.com/zeux/volk/blob/master/LICENSE.md`

A.1.3 GLFW

`https://www.glfw.org/`

Zlib/libpng license: `https://www.glfw.org/license.html`

A.2 Other Tools Used

Other tools used to aid in the development of the project are:

A.2.1 Make

`https://www.gnu.org/software/make/`

A.2.2 RenderDoc

`https://renderdoc.org/`

A.2.3 GLSLC

`https://www.apache.org/licenses/LICENSE-2.0`

A.3 Algorithms

The implemention for the distance estimator for the Mandelbulb fractal was influenced by the content in these two sites:

https://iquilezles.org/articles/mandelbulb/
http://blog.hvidtfeldts.net/index.php/2011/09/distance-estimated-3d-fractals-v-the-mandelbulb-different-de-approximations/

The formula for the Hall of Pillars fractal was obtained from:

https://www.shadertoy.com/view/4s3GW2

The algorithm for calculating the intersection point of a ray with a cube was obtained from:

https://iquilezles.org/articles/intersectors/

Appendix B

Ethical Issues Addressed

No particular ethical issues were encountered. The software did not require testing by other people, and the exploration of the topic chosen is not controversial in any way. Where materials have been taken from external sources, this has been stated clearly.

Appendix C

Code Repository

The link to the project's code repository is:

`https://github.com/Nell-Mills/SDF-Fractal-Renderer`