Erroll Wood

Gonville & Caius College

eww23

Computer Science Tripos Part II

Three Dimensional Fractal Exploration using a Scale-Adaptive Sparse Voxel Octree

May 16, 2012

Proforma

Name:	Erroll Wood
College:	Gonville & Caius College
Project Title:	Three Dimensional Fractal Exploration using a Scale-Adaptive Sparse Voxel Octree
Examination:	Computer Science Tripos, July 2012
Word Count:	11968
Project Originator:	Dr Alex Benton, Erroll Wood
Supervisor:	Dr Alex Benton

Original Aims of the Project

To implement a system capable of procedurally generating volumetric three dimensional fractal data and rendering it in real time. A sparse voxel octree data structure should be used which intelligently adapts during exploration to store only the detail required. The user should be able to explore fractal detail at different scales without experiencing decreases in system performance.

Work Completed

The project's core success criteria have been fulfilled. Fractal data is procedurally generated at run-time and unloaded from memory when no longer visible. A modified version of the sparse voxel octree traveral algorithm was implemented capable of rendering fractals with performance independent of scale. A constant average of 18 frames per second was achieved during a zoom bringing the user 10^{11} times closer to a fractal.

Special Difficulties

None.

Declaration

I, Erroll Wood of Gonville & Caius College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date **May 16, 2012**

Contents

Chapter 1

Introduction

Benoit Mandelbrot claimed that "patterns of Nature are so irregular and fragmented" that they exhibit a degree of complexity beyond Euclidean geometry [1], requiring the fractal "geometry of nature" for representation. This chapter explains the motivation behind the development of a 3D fractal explorer, and describes some of the key concepts and related work in Computer Science expanded on by the project.

1.1 Motivation

Current typical volumetric data-sets sampled by a 3D scanner or created by an artist will have detail limited by the scanner's resolution or artist's ability. This means that when rendering at high magnification, individual volume elements will eventually become visible to the user, limiting the amount of information a rendering provides. If instead the amount of detail in the data-set can be indefinitely increased the user will not experience this loss in information.

By providing an extension to the sparse voxel octree data-structure and related algorithms that supports efficient procedural content generation and rendering, complex and feature-rich geometry could be injected into volumetric models in the form of fractally generated programmable 3D textures and entities. An example could be fractally generated terrain and foliage over the surface of planets in a solar system simulator, allowing user exploration at nigh-infinite[1] magnifications.

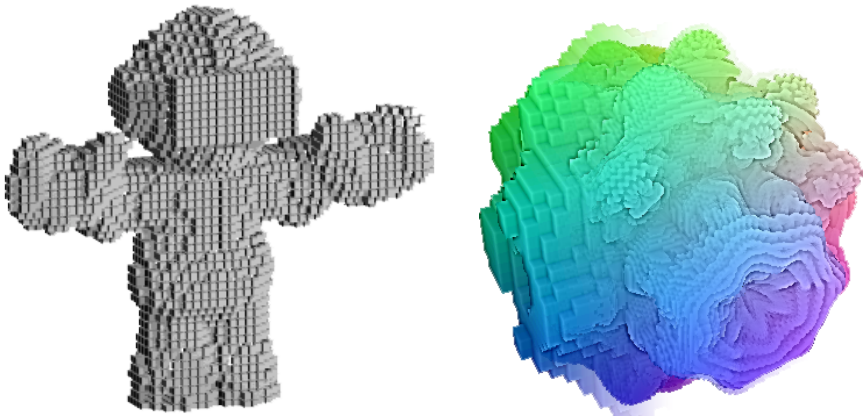
[1] Limitations of floating-point arithmetic mean rounding errors will eventually affect results. The terms "infinite" and "indefinite" refer to the limit of double-precision floats, any smaller detail is beyond the scope of this project.

While current state of the art voxel rendering engines achieve "real-time rendering performance for several billion voxels,"[2] using such an extension would allow rendering of procedural datasets of infinite size.

3D fractals were chosen for this project as they dramatically express the concept of procedural generation through their property of self-similarity; meaning that it is possible to indefinitely deconstruct them into smaller pieces "geometrically similar to the whole" [1].

1.2 Voxels

Voxels are volumetric pixels (picture elements) that represent a value in 3D space in the same way that pixels represent 2D image data in a bitmap. Each voxel is a cube and 3D models can be approximated using many little cubes as an alternative to polygon meshes (see Figure 1.1).



(a) Anthropomorphic character from [3] (b) Mandelbulb from this project

Figure 1.1: Figure 1.1(a) shows a 3D model approximated with voxel cubes. Figure 1.1(b) shows how if enough voxels are used, the shape's surface appears completely smooth.

Benefits of Using Voxels

Unlike polygons, each voxel does not need to encode its own position in space as this can be inferred from its position relative to other voxels. While simple objects containing mostly flat surfaces can be efficiently represented with polygon meshes, more complex, organic-looking structures such as mountain ranges, foliage, and

fractals are better suited for voxel representation. For these types of structures, as many polygons as voxels would be needed to approximate them to the same degree of accuracy, and polygons require more memory.

The Sparse Voxel Octree

An efficient voxel storage data-structure is the commonly used *sparse voxel octree* (SVO). The SVO is a tree data-structure where every node can have up to 8 child nodes, and is sparse in the sense that it does not store subtrees for empty sections. This makes it very memory efficient, the state of the art Atomontage[2] voxel engine can compress each voxel's memory footprint down to one bit [3].

To create a SVO from a 3D modeled object we start with one cube enclosing the shape and then repeatedly *subdivide* each non-empty cube through its centre along its axis-aligned planes into 8 smaller cubes until the leaf nodes sufficiently approximate the object's shape. Figure 1.2 demonstrates this concept in 2D using a sparse pixel quadtree[3].

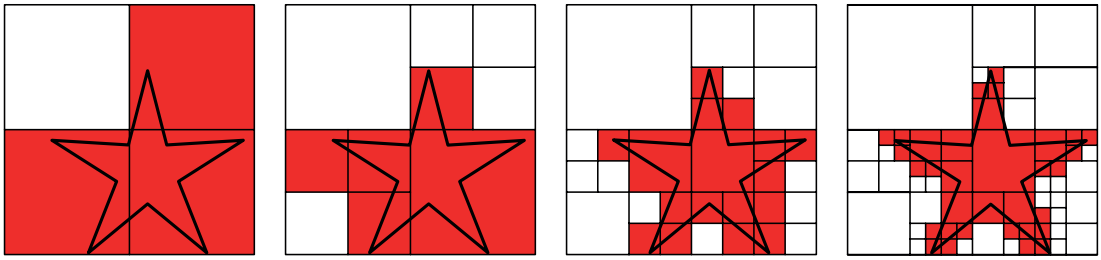


Figure 1.2: Quadtree of a star at 4 different subdivision levels. The rightmost quadtree uses 64 nodes to store the shape while a bitmap array would use 256 pixels

[2]http://www.atomontage.com/

[3]The 2D analogue of the SVO. Reasoning and visualizing in three dimensions can be difficult at times and many SVO concepts can be more easily explained in two dimensions

As well as natively compressing data, the SVO's hierarchical nature benefits many computer graphics techniques including:

- Efficient rendering using *ray-tracing* (where one ray is sent out per-pixel to intersect with an object) as a result of the SVO's uniform geometry.
- Implicit *level of detail* (LOD) management, decreasing the complexity of the rendered 3D model as the viewer moves further away from it.
- Out-of-core techniques to stream SVO data from complex objects into memory only when they are needed using feedback mechanisms.

1.3 Available Fractal Renderers

Software currently available for visualizing 3D Fractals[4] are focussed on being able to render attractive images implementing many computer graphics features at the expense of performance, meaning real-time exploration is not ideal. They utilize a different rendering method where no volumetric data data describing the fractals is stored. While this project will choose a different novel approach for rendering the 3D fractals, some techniques that were developed for these renderers will still be useful.

1.4 Project Overview

The chapters that follow present the preparation, implementation and evaluation of a 3D fractal exploration system based on extensions to SVO technology. In Chapter 4 the system is shown to be capable of providing a constant average of 18 frames rendered per second during an zoom into procedurally generated fractal data where the user is brought 10^{11} times closer to the fractal surface, thus fulfilling the project's requirements.

[4]For example: Fragmentarium (http://syntopia.github.com/Fragmentarium/) and Mandelbulber (http://sourceforge.net/projects/mandelbulber/)

Chapter 2

Preparation

This chapter outlines the necessary work carried out in order for implementation to begin. This planning includes familiarization with new concepts such as the SVO data-structure, related algorithms, and 3D fractal generation; and how they influenced software engineering and design choices. An overview is then given explaining the choice of development tools and practical measures to increase the project's resilience to failure.

2.1 Prerequisite Knowledge

Whilst many concepts involved in the project had to be learnt from scratch, others drew upon topics covered or introduced in the Computer Science Tripos.

Computer Graphics and Image Processing introduced many of the fundamental techniques used when rendering 3D objects, including ray tracing.

Concurrent and Distributed Systems is necessary to achieve the highest possible performance by using threads and writing thread-safe code.

Floating-Point Computation to understand the perils involved when working at very small scales during calculation of fractal data and ray-traversal.

Further Java covered the language's advanced programming features including concurrency control.

Software Engineering to ensure a professional approach was taken.

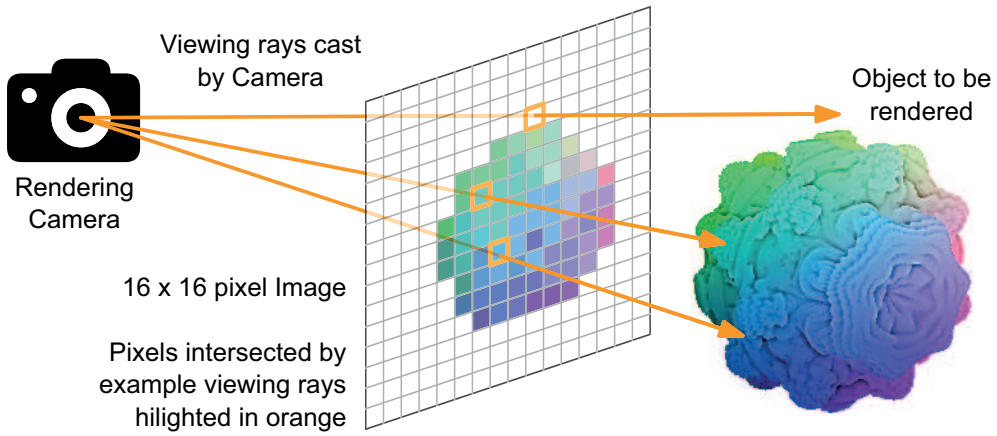


Figure 2.1: The concept of rendering an image by ray casting. The pixels corresponding to the 3 cast rays are highlighted.

2.2 Rendering a Sparse Voxel Octree

This section briefly describes the standard computer graphics techniques and rendering algorithms used for SVOs which had to be learnt before the modifications proposed by this project could be made. It does not exhaustively describe the different methods used for rendering SVOs but instead outlines the basic structure they all follow.

2.2.1 Ray Casting

SVOs are well suited for *ray tracing* because of their regular hierarchical nature. Ray tracing is a computer graphics technique for rendering an image by tracing the path of light through each pixel in the image plane and accounting for intersections with objects. *Ray casting* is a variant of ray tracing that does not cast additional rays following the first ray intersection[1] (Figure 2.1). The fact that child nodes in a SVO lie geometrically inside their parent node, so all child nodes boundaries are within the parent's boundaries, means SVOs are a very good acceleration structure for ray intersection tests.

[1]Ray tracing often involves secondary rays for calculating shadow or reflection effects

2.2.2 The Octree Traversal Algorithm

Octree traversal is an algorithm that works by iteratively stepping through the set of SVO nodes intersected by the ray in first-to-last order, terminating when the ray reaches a voxel leaf node or misses the object. As no position data is encoded by voxels themselves, it is necessary for the algorithm to keep track of the position and size of each octree node currently being tested, as determined by its hierarchical path from the root node.

To determine which ray-intersecting SVO leaf node is being tested at any time, a point P is kept track of along the ray's direction vector, initialized to the point where the ray first intersects the SVO root node. If an empty leaf node (representing empty space) is encountered at that point, the point P is moved to P' - the point where the ray leaves that leaf, and the next node the ray intersects at P' is determined and tested (Figure 2.2). If a ray leaves the opposite side of the root node from where it entered, it has missed the object.

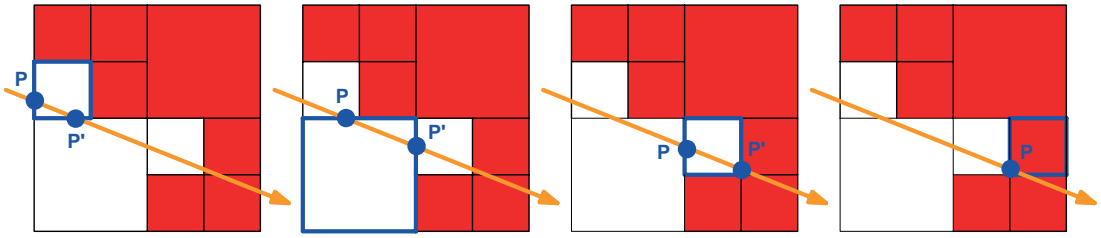


Figure 2.2: The sequence of quadtree leaf nodes tested and progression of point P for each iteration. The algorithm terminates when it intersects the full leaf node in the rightmost quadtree.

Algorithm 2.1 Ray Casting an SVO

```
P ← point where ray first intersects the root SVO node
while P lies inside root SVO node do
    N ← SVO leaf node intersected by P
    if N is not empty then
        return colour of N
    else
        P ← P', the point where ray leaves N
    end if
end while
```

Rendering an entire image using the octree traversal algorithm is *embarrassingly parallel* [4] in the sense that little or no effort is required to decompose the

problem up into independent sub-tasks as long as they safely share the SVO data-structure. This provides opportunity for performance gains.

Design Consequences

In order to be able to output these rendered images to the user in real-time, it was necessary to choose a high-performance language with good multi-threading support. In this case Java was chosen with its Swing API for graphical user interfaces and its built-in high-level concurrency APIs. For the maths behind ray-traversal itself to be computed the `vecmath` package[2] for vector manipulation was deemed sufficient. Eclipse[3] was chosen as integrated development environment for the project as it provides sufficient support for coding and debugging.

2.3 Fractal Generation Methods

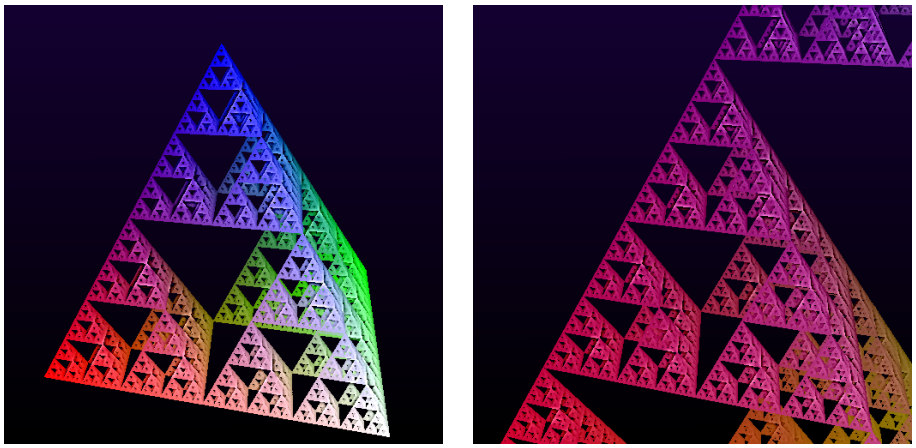
This section describes the two fractal archetypes required for the project which were studied in order to determine how best to represent them in SVO data-structure form. A technique used to render fractals known as *Distance Estimation* that was adapted for use in this project is then described.

2.3.1 Iterated Function System Fractals

Iterated function system (IFS) fractals are constructed through the union of multiple self-copies, with each copy being transformed in scale, position and/or rotation by a contractive function, scaling points closer together and making shapes smaller. This means that IFS fractals are constructed from ever-smaller copies, which are also constructed from smaller copies, ad infinitum. It is the fact that they are constructed from copies of themselves that give them Falconer's fractal characteristic of *strict-self-similarity*, where they are identical at all scales [5], as can be seen in Figures 2.4 and 2.3.

The canonical example of an IFS fractal is the *Sierpinski Triangle*; its 3D analogue is the *Sierpinski Gasket*. The Sierpinski Triangle can be constructed as follows:

[2]http://java.net/projects/vecmath
[3]http://www.eclipse.org/



(a) Entire Sierpinski Gasket (b) 1/12 of the Sierpinski Gasket

Figure 2.3: Strict-self-similarity displayed in the Sierpinski Gasket.

1. Begin with an equilateral triangle with base aligned to the horizontal axis.
2. Scale the triangle to 1/2 its height and 1/2 its width.
3. Replicate the triangle twice for a total of 3 copies.
4. Translate the copies so each triangle meets the other two with a vertex. A hole in the centre of the 3 triangles will now be apparent.
5. Repeat steps 2, 3 and 4 as many times as desired.

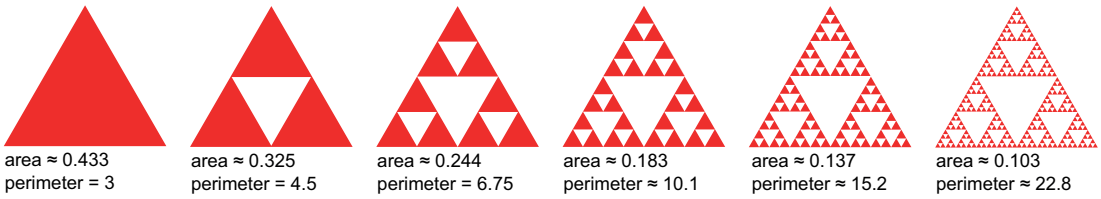
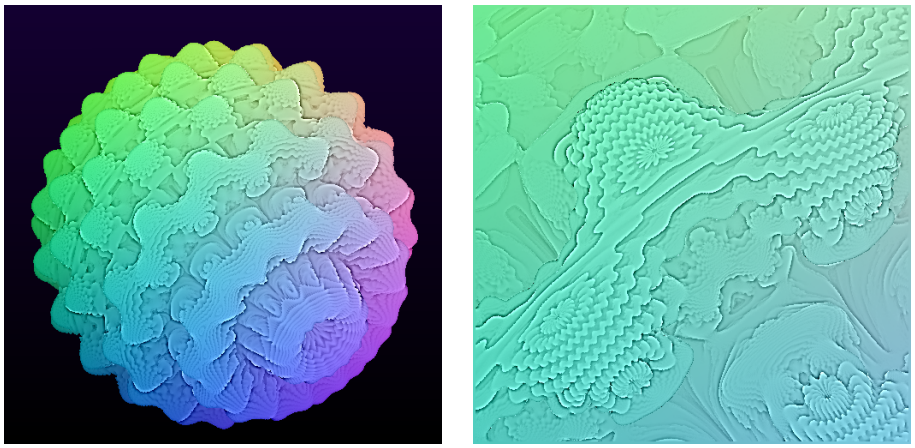


Figure 2.4: The iterative creation of a Sierpinski Triangle tending to zero area and infinite perimeter.

While this method for generating IFS fractals explains their nature well, it is not suited for storing a fractal as volumetric data in a SVO as there is no obvious indication of whether a set of arbitrary coordinates should be determined to be part of the fractal or not. Techniques using *Distance Estimation* (Section 2.3.3) are more appropriate.



(a) Entire Mandelbulb (b) External bulb detail

Figure 2.5: Quasi-self-similarity displayed in a Power-16 Mandelbulb. Similar ripples can be seen on the surface of the entire bulb and extruding bulbs.

2.3.2 Escape Time Fractals

Escape-time fractals are defined using an iterative formula or recurrence relation over every point in a domain (such as the complex plane or three dimensional space) and determining if the function's output for a that point is bounded or not. The set of points in the fractal are the points for which the function does not diverge. This method for generating fractals is better suited for the procedural subdivision of voxel nodes in 3D space as the iterative formula can be applied at a node's position without any previous construction required.

The canonical escape-time fractal (and perhaps most well-known fractal of all) is the *Mandelbrot Set*. Its closest 3D analogue is the *Mandelbulb* (Figure 2.5). The Mandelbrot Set is a two dimensional fractal based on the convergence of the sequence generated by the iterative formula $z_{n+1} = z_n^2 + c$ over each complex number. To determine if a complex number c is part of it, this iteration is repeatedly applied to the starting value of $z_0 = 0 + 0i$. If the absolute value of z_n remains bounded during the iterations *orbit* as $n \to \infty$, c is a member of the Mandelbrot set.

Displaying these results in the complex plane results in the shape of the Mandelbrot Set which, according to Falconer's fractal characteristics[5], exhibits *quasi-self-similarity*: the fractal approximates the same pattern at different scales and may contain small copies of the entire fractal in distorted and degenerate forms.

2.3.3 Distance Estimation

A common way to render fractals is through *raymarching*[4] accelerated using a *Distance Estimation* (DE) function. While raymarching is not ideal for use with a SVO, the DE functions proved integral in designing a system to store fractals as volume data. A DE function returns a lower bound on the distance between a point in space and an object being checked against, e.g. the DE from a point P to a sphere of radius r around the origin is simply $|P| - r$.

Different methods for obtaining DE functions exist for different types of fractal. In the case of escape-time fractals where the iterated orbit of z is compared against the bounds of an $EscapeRadius$, a running spatial-derivative dz can be calculated and thereby the DE estimated from how long it would take to reach the $EscapeRadius$ from z:

$$DE = \frac{z - EscapeRadius}{dz}$$

DE functions for IFS fractals can be obtained by iteratively scaling a point in space towards a fractal vertex and returning the distance moved by that point. The symmetrical nature of IFS fractals can be exploited to reflect or *fold* every point checked into one section of the fractal, so only one vertex need be considered. As a result of the folding, this fractal generation method is known as Kaleidoscope IFS [6].

DE fractal functions were first discussed in Hart's 1989 paper [7] for rendering a 3D Quaternion Julia set, but only in recent years have DE functions for a wider range of fractals been discovered [6]. Pre-existing DE code for the whole range of fractals required for this project can be found online in Mikael Hvidtfeldt Christensen's blog [6] and in his 3D Fractal rendering package, Fragmentarium[5].

Design Consequences

As the scale-adaptation system is a modification of the SVO, the functions for procedurally calculating new data should be easily interchangeable to allow a range of uses. It was decided that the scale-adaptation system should rely only on a simple interface to determine the object being generated. The fact DE

[4]Raymarching involves stepping along a ray incrementally until signalled to stop, without needing volume data.

[5]http://syntopia.github.com/Fragmentarium/

functions exist for fractals supported this and prevented the need for a more complicated fractal mathematics system as each fractal could be generated in a similar way.

DE functions were also chosen as they return distance values based on points in space rather than booleans, meaning they can be used to generate volumetric fractal data which varies depending on the scale at which the DE function is invoked at. Traditional escape-time systems only depend on a single point in space, so are not immediately adaptable for use with voxels which have size.

2.4 Software Engineering

This section details the work taken at the beginning of the project unrelated to learning new concepts, but which was nonetheless necessary for employing a professional approach and increasing the project's resilience to failure.

Requirements Analysis

The purpose of the extensions to the SVO data structure and algorithms was to procedurally and indefinitely increase detail in a volumetric model whilst keeping performance independent of the scale the volume data is being rendered at, using 3D fractals as an example. Such a system should therefore fulfil these requirements:

1. It must be possible to generate volumetric data described by a range of 3D fractals and store this information in a scale-adaptive SVO.
2. Further generation of fractal detail must be computed procedurally to allow infinite exploration.
3. The fractal must be rendered in real-time with performance independent of scale during navigation around and into the fractal.

Software Development Process

The *spiral development model* was chosen as it suited the fact the project was based on extending an existing concept and that not all detailed requirements

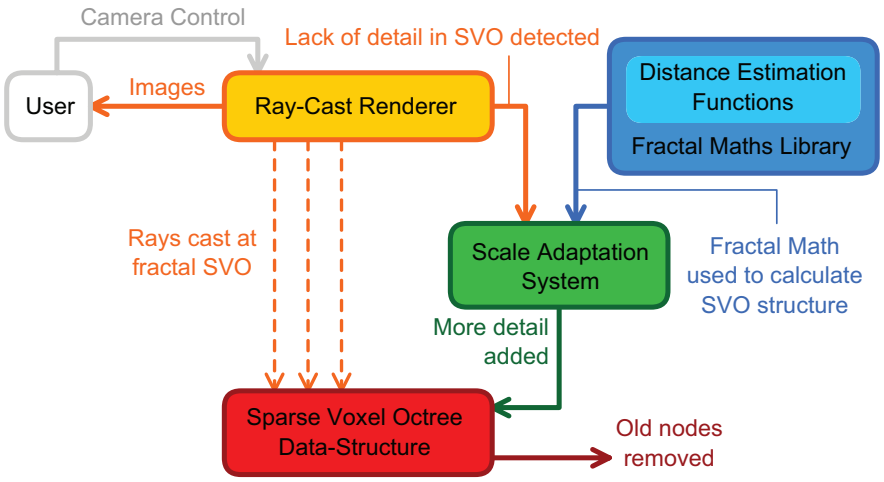


Figure 2.6: Rough system design showing different components to be implemented.

of the scale-adaptation system could be initially known. It allowed for implementing the core functionality of the SVO first and incrementally refining the project by introducing fractal math and scale-adaptation modules. This was ideal as designing the scale-adaptation system took many properties of the SVO and rendering algorithms into account which were only learnt during their implementation.

System Design

By studying these new topics enough had been learnt about the SVO data-structure and 3D fractal generation to design a basic overview of the system which would fulfil the requirements gathered (Figure 2.6).

Preparation for Testing

Testing for this project occurred both during implementation, in the form of unit tests, and after the system's completion as part of the project's evaluation. The JUnit[6] framework was used in both cases, chosen for its integration with Eclipse. Unit testing during development was beneficial in ensuring the system was robust at all times by quickly detecting bugs and facilitated refinement of the system by ensuring modules continued to work correctly. Automated test suites

[6]http://www.junit.org/

carried out evaluation testing to avoid unpredictable user interaction affecting the results.

Backup Provision

Git[7] was used as a version control system to both keep track of changes and provide backup support using off-site servers provided by Google Code[8]. Workspace and dissertation directories were synchronized to the cloud using Microsft Skydrive[9]. Local backups of previous versions were also kept on the development machine. Git allowed reversions to a previous working state after changes in design and the online availability of the code itself allowed remote examination and discussion with the project supervisor.

[7] http://git-scm.com/
[8] http://code.google.com/
[9] https://skydrive.live.com/

Chapter 3

Implementation

This chapter first describes the implementation of core components necessary for a system capable of ray-casting a 3D fractal, including a simple SVO data-structure, scale-adaption system, fractal maths library, and ray-caster using the simple kd-restart algorithm. These were implemented to better understand the challenges involved with creating a scale-adaptive SVO and produce baseline data against which to compare the performance of the finished system.

The chapter will then describe the implementation of a more advanced SVO structure, scale-independent rendering algorithm, and memory management system which fulfil the project's requirements. Concurrency issues between components are described to explain how the system takes advantage of parellelism. Post-processing effects which produce more detailed images will also be described.

3.1 Simple SVO Data-Structure

As it is not feasible to efficiently test for intersection between every ray and every voxel, the SVO is used as an accelerative data-structure allowing efficient ray casting of volume data. This section describes the implementation of a basic SVO which minimizes the number of required intersection tests during ray traversal by partitioning volume data into a hierarchical uniform grid.

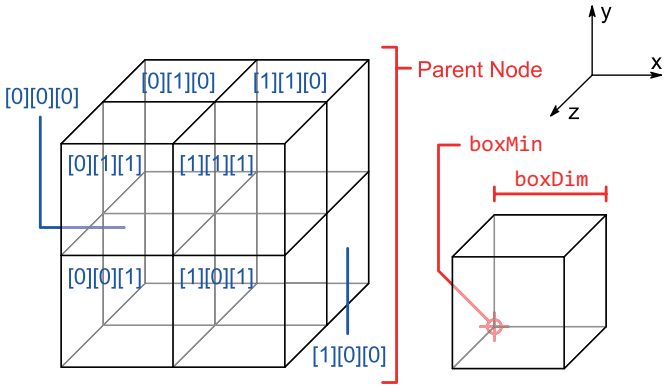


Figure 3.1: The structure of an `OctreeNode` showing parent and child boundaries, indexes into the array of children, and explaining `boxMin` and `boxDim`.

Tree Structure

A class `OctreeNode` was implemented for the SVO nodes. The root node represents the enclosing volume for the entire volumetric data and is an axis-aligned cube with side length 2 and centred at the origin, spanning the space from $(-1, -1, -1)$ to $(1, 1, 1)$[1]. As each SVO node is axis-aligned, they can be described using only their minimum position in space (`boxMin`), and side length (`boxDim`).

Each non-leaf node has an array of 8 SVO node children which may or may not be leaf nodes (OctreeNodes without children). Its structure is described in Figure 3.1.

If a leaf node is empty then it contains no volumetric data within its boundaries and is leveraged during traversal algorithms to skip through large empty sections of the model. If a leaf node is full then it describes some part of the model and has an associated 32-bit colour. To support implicit LOD, each non-leaf inner node also has a colour which is the average of all of its children's colours. This is so that rays can terminate early on non-leaf nodes and return a colour if that node's detail is adequate.

Memory Usage

The basic SVO node implemented was therefore described by:

- `boolean leaf`: `true` if the node has no children, `false` otherwise

[1] These positional and size values for the root SVO node were chosen to simplify mathematics

- `boolean empty`: `true` if the node is an empty leaf, `false` otherwise
- `OctreeNode[][][] children`: 3D array of references to 8 `OctreeNode` children if node is a parent, `null` otherwise
- `int color`: 32-bit colour value for non-empty nodes (alpha channel unused)

This gives an upper bound of 11 bytes of data for each non-leaf SVO node (not including JVM data)[2]. This is much lower than the 72 bytes stored for 12 double-precision triangles to represent voxel cube as a triangle mesh. The memory consumption is $O(n)$ where n is the total number of nodes in the octree.

3.2 Scale-Adaptative Fractal Generation

To explore a fractal without scale-adaptation, SVO generation need only be carried out once before run-time. However, for detail to be procedurally generated it is necessary to be able to calculate more detailed fractal data during exploration as the user navigates at smaller scales. This is done using *subdivision*: splitting a SVO voxel leaf node into up to 8 new smaller voxels, as seen in Figure 3.2.

In this way volume data is generated by starting with one full root SVO node cube and repeatedly *subdividing* it and its children as necessary until the rendered image is resolution-bound, so each visible leaf voxel's projection onto the viewing window is smaller than one pixel in size. This is to ensure users interpret the SVO as a smooth shape rather than seeing individual cubes. If a voxel is ever detected as being too large, it is queued for subdivision. Calculating fractal data procedurally amortizes the time taken to generate the volume data so there is no delay on system start-up.

3.2.1 Approximating Fractals with Voxels

This section describes the implementation of a method to approximate fractals with volume data so that they can be stored in an SVO. A mapping from voxel cubes in 3D space to boolean values was necessary. This is sufficient as while an exact evaluation of the fractal may technically have zero volume and infinite surface area, the system is only concerned with a voxel approximation of the shape.

[2]boolean values are manipulated as 32-bit entities by the JVM

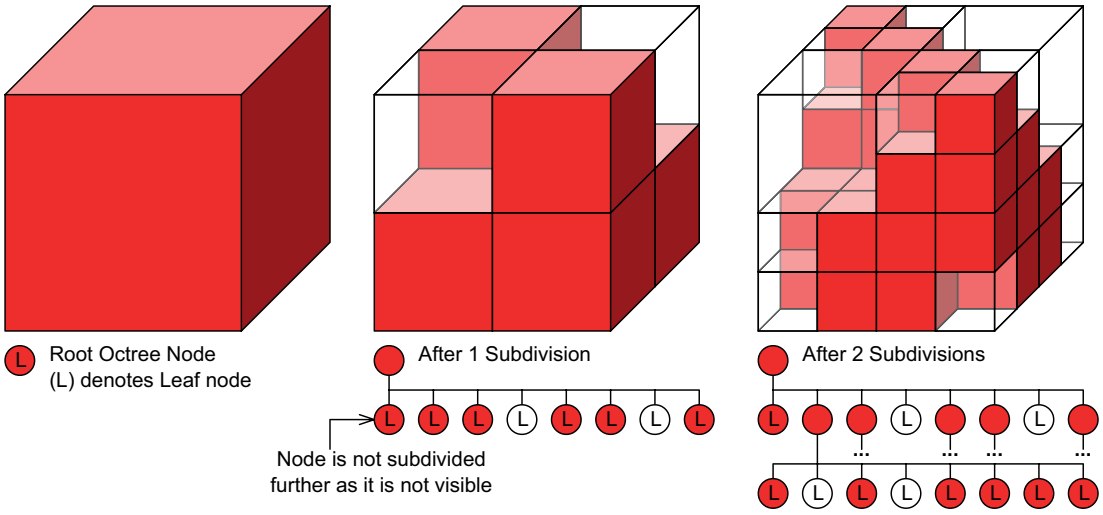


Figure 3.2: Subdivision progression from a single SVO root node to a more detailed voxel approximation. The octree hierarchy is shown under each subdivision level.

Voxel Semantics

For each fractal required, a corresponding class was implemented with the simple `boolean isInFractal(x, y, z, r)` which returns `true` (generating a voxel) if its centre coordinates (x, y, z) are less than distance r away from the fractal and `false` otherwise. This was done to simplify implementation as each fractal could be generated in a similar way using pre-existing Distance Estimation functions. A fractal library was thus created with each fractal implementing this method signature using an interface.

However, this `isInFractal` detects if the fractal exists in a bounding **sphere** of radius r around a point but voxels are **cubes**. This means a certain amount of inaccuracy and extra effort will have to be taken into account depending on how it is chosen to allow spheres to approximate cubes. On the one hand false positives will generate a voxel where the fractal does not exist, but on the other false negatives will mark a SVO node as empty even though the fractal exists within its boundaries. 2 obvious choices for r are available:

- Let $r =$ `boxDim/2` so the bounding sphere exists entirely within the voxel, eliminating false positives.
- Let $r =$ half the voxel's diagonal (`boxDim` $\times\sqrt{3}$) so the voxel does not exist outside bounding sphere, eliminating false negatives.

The second option was implemented as this simplified the system by reducing the number of necessary fractal calculations. If it is not certain whether an empty SVO node contains fractal data, it must be repeatedly checked at lower and lower scales when its neighbours are subdivided, potentially resulting in multiple subdivisions to generate the necessary hierarchy to rectify a past error. Therefore it was decided to eliminate the possibility of false negatives entirely; albeit at the expense of dealing with false positives.

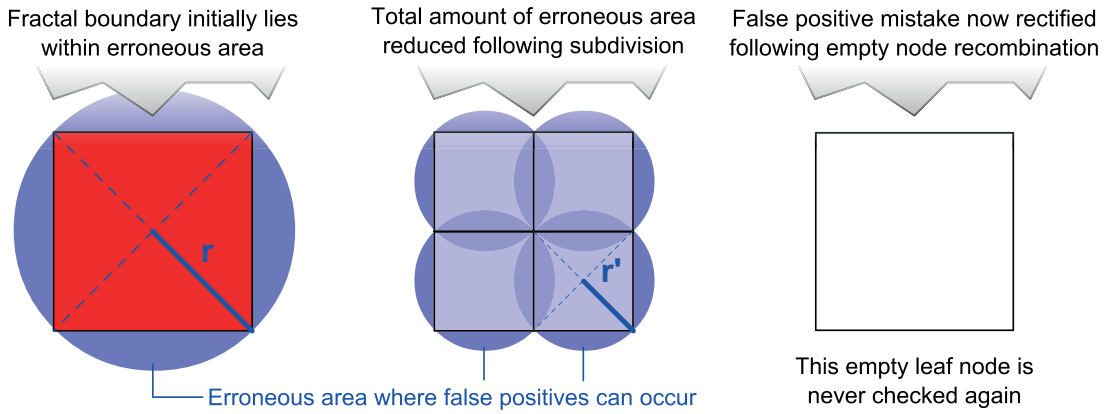


Figure 3.3: The automatic correction a false positive erroneously generated voxel during subdivision.

False positives have minimal effect as if a voxel is incorrectly generated the mistake will be automatically rectified during a future subdivision stage. When the incorrect the cube is split up into 8 smaller cubes, the 8 spheres that represent them have a smaller total erroneous false positive volume. While the fractal may lie inside the erroneous volume for the parent node, the child nodes will test as empty and can be *recombined* into an empty SVO node, keeping the octree's sparse property, as seen in Figure 3.3. Recombination is recursive to ensure no SVO parent nodes needlessly contain 8 empty children.

As long as it is certain that empty nodes can never contain fractal data, coordinates inside empty nodes never need to be re-checked so the subdivision process is sufficient to keep an accurate approximation of the 3D fractal.

Using Distance Estimation

Pre-existing DE code was studied and re-implemented to provide the maths behind the `isInFractal()` function. DE functions for Mandelbulb and Mandelbox fractals were ported from the Syntopia Blog [6] and functions for Menger Sponge

and various other Sierpinski-esque polyhedra fractals were ported from a thread from Fractal Forums detailing Kaleidoscope IFS fractal construction[3]. DE function examples can be found in Appendix A.

3.2.2 Subdivision

Leaf nodes are stored in a priority queue for subdivision and removed in largest-first order, so large discrepencies in the fractal approximation are refined before small details. Each subdivision consists of calculating 8 new `isInFractal()` values corresponding to the nodes 8 new children. If every child is determined to actually be empty, a recursive `checkParentNowEmpty()` procedure is started to recombine any unnecessary empty octree leaf nodes. On the other hand if at least one of the children will be full then 8 new leaf `OctreeNode` children are created. The final act is to flip the new parent node's `leaf` flag to `false`, allowing octree down-traversal.

3.2.3 Concurrency

Adding a large amount of detail to the SVO can be broken broken down into separate node subdivision subtasks, providing the opportunity for improved performance. As no data outside the leaf node is accessed during subdivision, the subtasks are independent and can be carried out in parallel without worrying about thread-safety.

To implement this a thread pool using the Java `Executor` framework was implemented to create a work queue of tasks while also controlling the number of threads in use, preventing the CPU from over context-switching between tasks, reducing performance.

3.3 Rendering using Kd-Restart Algorithm

This section describes the basics behind rendering and ray casting, and the simplest octree traversal algorithm (*kd-restart* from [8]) that provided data against which to compare the performance of the finished system. It also introduces the

[3] http://www.fractalforums.com/3d-fractal-generation/kaleidoscopic-(escape-time-ifs)/

issues with the standard SVO and rendering algorihm that were overcome by this project.

Basics: Camera

During ray casting at least one ray must be sent out per pixel to be rendered. To determine where these rays are cast from and what direction they travel in, a *Camera* object is implemented. Each ray's origin is the Camera's position c in space and its direction vector d points in the direction of the pixel on the viewing window, a rectangular section of the image plane.

The Camera can be sufficiently described using its 3D positional coordinates, 2 orthogonal 3D vectors describing its *viewing window*'s shape, and the perpendicular distance from the Camera's position to the image plane to determine the viewing window's position in space, as shown in Figure 3.4). The vector from the position of the camera to the centre of the viewing window, the `lookVector`, must be orthogonal to the image plane to prevent the image appearing skewed. The `Vector3d` (tuple of 3 double-precision floating point numbers) data type from the `vecmath` package was suitable.

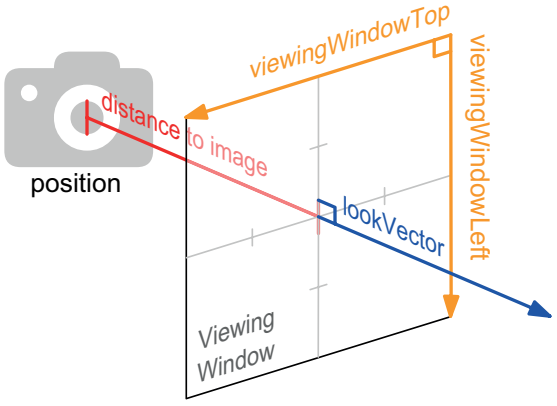


Figure 3.4: The components used to define the Camera and its viewing window.

Controlling the Camera

Methods were implemented so the Camera could be moved and rotated in a number of different ways allowing user and automated navigation. Each method involves transforming camera vectors to move the position of the Camera itself and/or change the direction it's looking in. Examples include:

- `moveCameraBy(Vector3d offset)` to translate the position camera by the given `offset`, modelling a viewer stepping forwards, backwards, or side-stepping without turning their head.
- `rotate(Axis axis, double angle)` to rotate the `Camera`'s viewing window and `lookVector` around a given `axis` about its own position - this is equivalent to a viewer turning their head while standing still.

The `vecmath` package provided sufficient support for creating and applying transformation matrices, allowing a comprehensive range of camera control.

Casting Rays Through Pixels

Once the camera is in position, it is used to calculate the direction vector of each viewing ray which can then traverses the SVO. To do this the function `getVectorToPixel(x, y, resolutionWidth, resolutionHeight)` was implemented which finds the position of the pixel (x, y) in the viewing window through linear interpolation between the corner coordinates of the viewing window, and then calculates a ray direction vector d by subtracting the position of the pixel from the ray's start, c.

Basics: Ray-Cube Intersection

Each ray is described by its start position c and direction vector d, and each SVO node is described by its minimum position $boxMin$ and its size $boxDim$. Let each ray be defined parametrically as $p_t(t) = c + td$. To solve for t against an x-axis-aligned plane:

$$t_x(xPlane) = \frac{(xPlane - c_x)}{d_x}$$

Similar equations exist for y and z-axis-aligned planes. Using this equation, 2 corners of the cube $(x0, y0, z0)$ and $(x1, y1, z1)$ can be chosen so that:

$$t_x(x0) \leq t_x(x1),\ t_y(y0) \leq t_y(y1),\ t_z(z0) \leq t_z(z1)$$

To calculate these t-values $boxMin$ and its opposite corner are used, and $t_x(x0)$ and $t_x(x1)$ are swapped as necessary (along with y and z t-values) to ensure the condition holds[9]. Once these 6 axial t-values have been calculated, the minimum and maximum t-values that correspond to the ray being inside the voxel can be found:

$$t_{min} = max(t_x(x0), t_y(y0), t_z(z0)),\ t_{max} = min(t_x(x1), t_y(y1), t_z(z1))$$

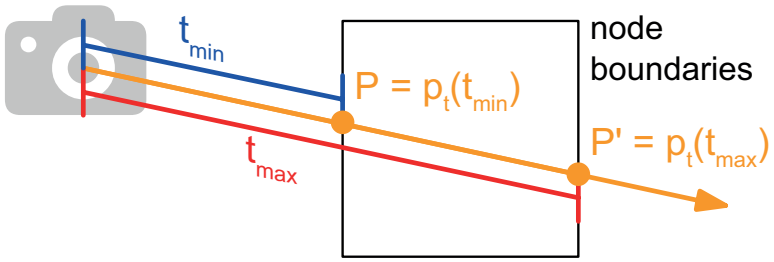


Figure 3.5: Clipping a ray against a quadtree node, showing the t-values corresponding to the ray entering (t_{min}) and exiting (t_{max}) the node boundaries.

These parameters represent the points where the ray enters and exits the bounds of the voxel cube, as shown in Figure 3.5. If $t_{max} < t_{min}$ or if returned values are negative then the ray has missed the bounding cube.

3.3.1 Octree Traversal Using Kd-Restart

Once P is initialized to the point where the ray enters the root SVO node's boundaries, the next step is to find the corresponding leaf node at that point. Then, until a solid leaf node is reached, each *neighbouring* leaf node the ray the ray intersects next must be checked. In kd-restart both of these steps use the same technique of descending the entire octree hierarchy from the root node at each iteration. This pattern of traversal is visualised in Figure 3.6.

Each node visited during traversal is kept track of using a reference to its `OctreeNode` object (`node`), `boxMin`, and `boxDim`. Until a leaf node is reached (assuming no implicit LOD) down-traversal can be executed by halving `boxDim` and updating `boxMin` to the `boxMin` of the child node containing P at each iteration. By comparing P against the centre of the voxel ($boxMid$) a binary 3-tuple is calculated:

$$step = (P_x > boxMid_x \; ? \; 1 \; : \; 0, \; P_y > boxMid_y \; ? \; 1 \; : \; 0, \; P_z > boxMid_z \; ? \; 1 \; : \; 0)$$

It is then possible to determine the `boxMin` of the correct child node by executing `boxMin += boxDim/2*step` and `node` can be updated to reference the correct child at `node.getChild[step.x][step.y][step.z]`.

If an empty leaf node is reached then `node`, `boxMin`, and `boxDim` are reset to root node values and P is updated to P' where it exits the empty leaf node before the next iteration. If a non-empty leaf node is reached the algorithm terminates, returning the node's colour. If P is ever determined to be outside the

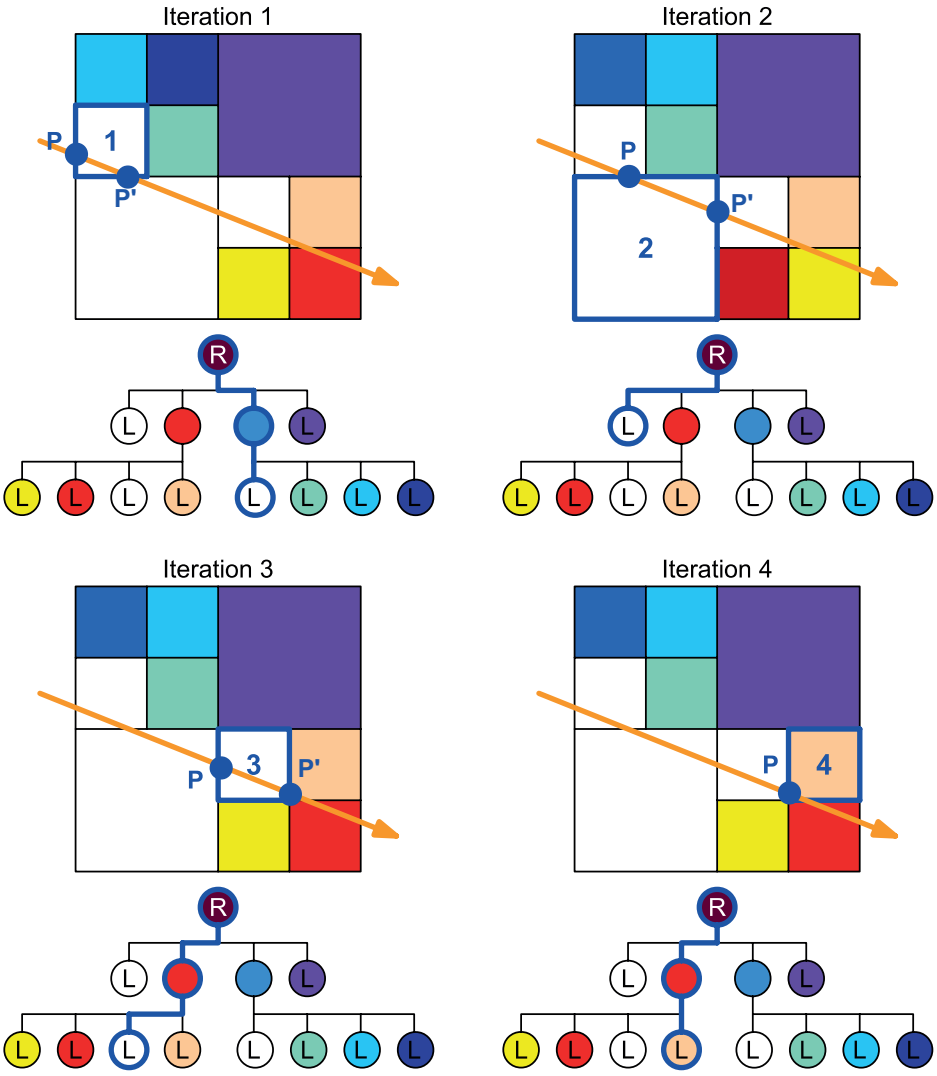


Figure 3.6: The sequence of nodes visited using kd-restart. The down-traversal path from root node to leaf in each iteration is highlighted for each time a new ray-intersecting neighbour leaf node must be found.

root node boundaries then the ray has passed through the octree without hitting anything.

This algorithms complexity is $O(m * log_x(n))$ where n is the number of nodes in the tree, x is the degree of the tree being traversed, and m is the number of neighbors that are visited before the algorithm terminates [8].

Problems with Kd-Restart

Kd-restart is inappropriate for delivering performance independent of scale as the octree develops a deeper hierarchy as smaller voxels are added. The root node's position and size does not change but each subdivision halves voxel size, meaning tall trees are needed for detailed volume data.

At a reasonable fixed level of detail this time taken to down-traverse the height of the octree is low enough not to be an issue, but as voxel cube sizes $\rightarrow 0$ this time significantly impacts performance, especially if a full traversal is necessary for each neighbor node traversal. In order to attain performance indepdendent of scale it will be necessary to find the sequence of leaf nodes the ray intersects without starting traversal from the root node.

Implicit Level Of Detail

The SVO hierarchy makes implementing implicit LOD optimizations straightforward. For each non-empty octree node visited during down-traversal, its projected size onto the viewing window is calculated. If this size is smaller than a pixel then traversal can terminate early before a leaf node is reached by returning the parent node's colour. This is because any further down-traversal will not enhance the returned image any further as smaller leaf voxels will not be discernable at the rendered resolution. This same size projection calculation is used to determine if a node is big enough to be queued for subdivision.

By changing this voxel projection size limit we can adjust the fidelity of the renderer and scale-adaptive SVO for better performance, allowing a voxel's projection to be up to 4 pixels in size and rays to terminate even earlier.

3.3.2 Ray Cast Algorithm

Algorithm 3.1 returns the colour of a ray cast through a viewing window pixel. It uses the standard kd-restart octree traversal method and is modified only to include scale-adaptation using subdivision so fractal data can be generated.

Algorithm 3.1 Ray Casting using Kd-Restart

```
1: Calculate t_min for where ray first enters rootNode
2: while P lies inside root SVO boundaries do
3:     P ← ray.start + t_min * ray.dir
4:     node ← rootNode                    ▷ Initialize root values for each iteration
5:     boxMin ← (−1, −1, −1)
6:     boxDim ← 2
7:     while !node.isLeaf() do            ▷ Down-traverse until leaf found
8:         if node is smaller than a pixel then
9:             return node.colour         ▷ Terminate early with implicit LOD
10:        end if
11:        boxDim /= 2
12:        s ← step tuple for point P
13:        boxMin ← boxMin + s*boxDim
14:        node ← node.getChild[s.x,s.y,s.z]
15:    end while
16:    if !node.isEmpty() then
17:        if node is larger than a pixel then
18:            Queue node for subdivision  ▷ Adapt SVO to changes in scale
19:        end if
20:        return node.colour             ▷ Terminate on reaching full leaf node
21:    else
22:        Calculate t_max for point P' where ray exits node
23:        t_min ← t_max                  ▷ Assign t_min for node's neighbour
24:    end if
25: end while
26: return background colour               ▷ Ray has missed SVO
```

3.3.3 Concurrency

Rendering an entire image can be broken down into parallel subtasks of ray casting individual pixels, improving performance. Though each ray cast thread

shares the same data, the SVO will never be visible in an inconsistent state so thread synchronization is not needed. Subdivisions modify SVO data but the node appears as a leaf until the process is complete and the `leaf` flag flipped; for this Java uses an atomic operation based on low-level hardware primitives so thread synchronization is not needed and the subdivision algorithm is non-blocking with the rendering algorithm. When writing the calculated colour into an image buffer, atomic operations are also used. As a result the algorithm is non-blocking, avoiding significant concurrency control overheads.

As multiple rays can now terminate on the same voxel, a `queuedForSubdivision` flag is set and checked in the `OctreeNode` to prevent a node being queued for subdivision more than once.

Basics: Displaying the Image

The colour of every pixel calculated is stored in a bitmap image. To display this to the user Javas Swing API was chosen as it provided a simple straightforward method for displaying graphics in a `JFrame` with support for double buffering.

Double buffering is a computer graphics technique to reduce flickering in animated images where multiple changes to the on-screen image are made at once, resulting in parts of more than one image being visible on the screen. To remedy this the renderer draws into a non-visible back buffer while the previous image stored in the front buffer is shown on the screen. When the drawing process is complete the buffers are flipped showing the now complete back buffer's image. This was implemented in the JFrame by creating a `BufferStrategy` to retrieve `Graphics` objects to draw into and display.

3.4 Scale-Independent Octree Traversal

This section describes the implementation of an algrotihm capable of traversing an octree with performance independent of the scale of the voxels being rendered, so the height of the octree no longer factors in the rendering time. To achieve this, the neighbour-traversal algorithm was modified.

3.4.1 Octree Traversal Using Neighbor-References

By including references in each node directly to its neighbours, entire hierarchical down-traversals are no longer necessary for each neighbour traversal. Therefore each `OctreeNode` needs 6 neighbour references, 1 for each cube face. Neighbour references to outside the root node are `null`. In this way each ray cast starts with one down-traversal from the root node finding the first leaf intersected, and then follows neighbor references, as shown in Figure 3.7.

Determining Neighbour Direction

The direction of the neighbor node to visit next can be calculated using the t-values measured during ray-cube intersection (section 3.3). Whichever of $t_x(x1)$, $t_y(y1)$ or $t_z(z1) = t_{max}$ corresponds to the axis the neighbor lies on, and the sign of the ray's direction vector then determines which neighbor of the two it is. If the neighbouring nodes are the same size then `boxDim` remains the same after traversal and `boxMin` can be shifted `boxDim` amount in the neighbor's direction.

However not all neighboring nodes will be the same size so further down traversals may be necessary after following a neighbor reference to a node at a coarser level. Complications therefore arise in keeping track of `boxMin` and `boxDim` which are no longer recalculated during down-traversal. This is solved by adding a `depth` field to each node which describes `boxDim` as $2^{-(depth+1)}$. To calculate the neighbour's position, `boxMin` is shifted as if the neighbor is of the same size and then "snapped" onto the grid described by its `depth` using `boxMin = boxMin/boxDim * boxDim` [10].

Issues with Neighbour Traversal

This algorithm's execution time therefore has a lower bound of $O(log_x(n) + m)$ where x is the degree of the tree, n is the number of nodes, and m is the number of neighbour-traversals [8]. This is more efficient than kd-restart, though is still not capable of providing performance independent of scale as each ray must make the entire octree heirarchy down-traversal once when initializing the first leaf node.

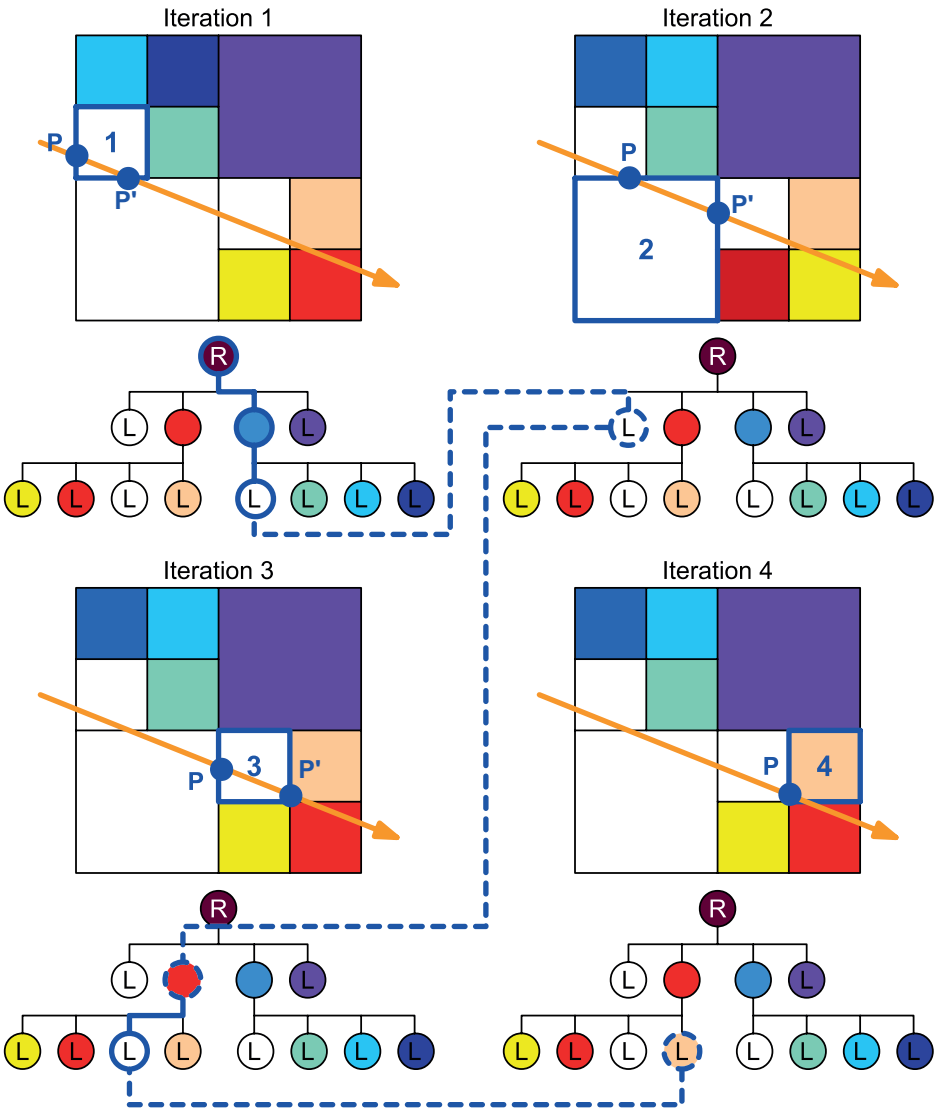


Figure 3.7: The sequence of nodes visited using neighbour traversal; down-traversals are highlighed with a solid line, neighbour-traversals with a dashed line. The down-traversal from root node to first leaf can be seen in Iteration 1. Transitioning from Iteration 2 to Iteration 3 is an example of down-traversals following a neighbour-traversal.

3.4.2 Skipping Down-Traversal

This section explains how the entire SVO hierarchy down-traversal is eliminated under certain circumstances, providing performance that does not depend on octree height. If the ray cast algorithm can initialize starting `node`, `boxMin` and `boxDim` values to the first leaf the ray insterects, a *skip-node* rather than the root, the initial down-traversal can be skipped. As a result ray casting will be bounded by $O(m)$ where m is the number of neighbours visited; the SVO height is no longer involved. If m remains constant during exploration, then the rendering time will be also constant.

Choosing a Skip-Node

There are two possible ways a user might magnify part of a fractal:

1. Decreasing the camera's *field of view* (FOV), equivalent to staying stationary but twisting a camera lens to examine distant objects.
2. Moving the camera closer to the fractal, equivalent to leaning closer to examine object detail.

FOV adjustment results in m increasing during zooming as the skip-node remains the same though more neighbour traversals are needed to examine smaller detail. This would mean the traversal algorithm's performance still relates to the scale of detail being examined. Therefore the FOV is kept constant and the camera moves closer to the fractal during exploration.

When the camera lies inside the root node's boundaries, it will be certain that all rays first intersect one skip-node: the empty leaf node containing the camera's position. Recording only one skip-node is sufficient as when the camera moves closer to the fractal shape, it will almost always enter the root node.

Updating the Skip-Node

If a skip-node ever becomes invalid it must be updated to ensure optimal traversal. When the camera is navigated outside its boundaries this can be done by comparing new and previous camera positions and following the old skip-node's neighbour reference. If the new skip-node is incorrect (after the user moves the camera a long distance) then the system resorts to the default hierarchy traversal from root to find a correct one. So even when the skip-node fails to update, only

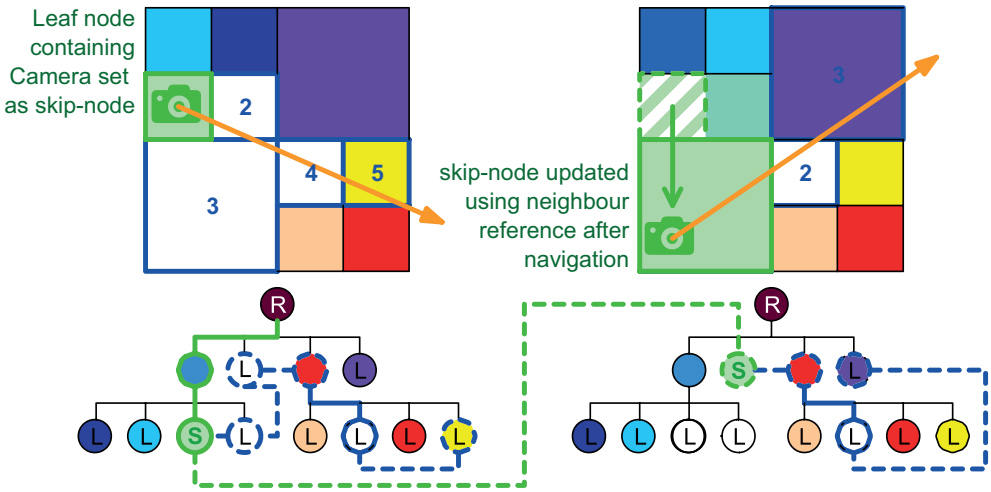


Figure 3.8: Each quadtree shows the path of all nodes visited by a ray cast. The skip-node is set using down-traversal only once and all traversals for rays and skip-nodes can be carried out using neighbour-traversals.

one entire down-traversal is executed for the entire image rather than one for each pixel.

3.4.3 Concurrency

The subdivision procedure now has to also assign new leaf nodes neighbour references to allow neighbour-traversal. Child neighbor references between sibling leaf nodes are safe to assign but child neighbour references outside the parent may be dangerous as they may reference neighbour node children which can be simultaneously subdivided. These therefore copy the parent's neighbour reference so data is still only accessed within the leaf node being subdivided, keeping the algorithm thread-safe. These neighbour references may now be sub-optimal if a lower level neighbouring leaf node exists, but these inaccuracies do not leave the SVO in an inconsistent state and they can still be traversed by rays correctly.

To remedy this neighbor references are optimized on-the-fly during neighbor-traversals to coarser, non-leaf nodes. When a possible optimization is possible the node's neighbour reference is updated so future rays will take the shorter route, as seen in Figure 3.9. This is non-blocking thread-safe as the SVO is in a consistent state before the neighbour reference is optimized and the update is an atomic action. Subdivision can thus still be executed in parallel with rendering.

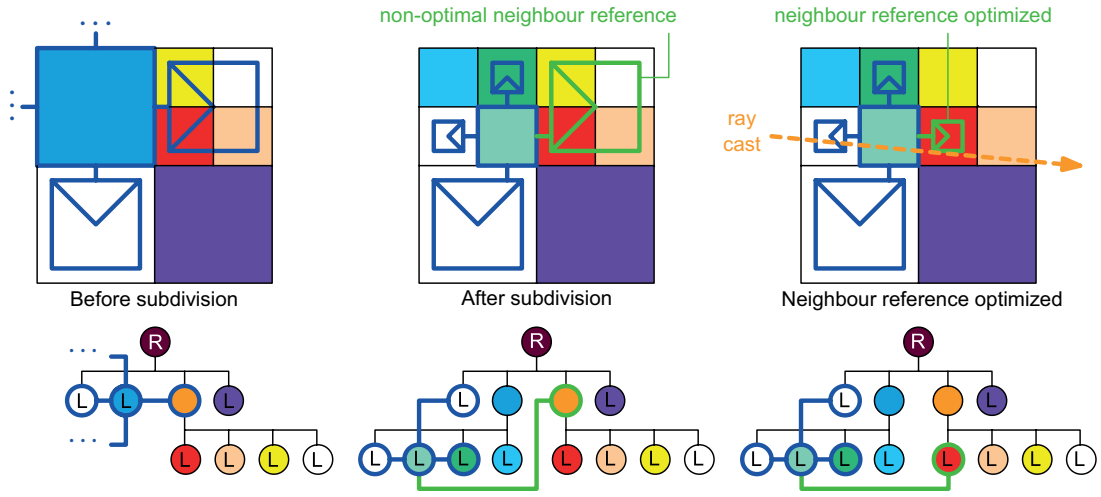


Figure 3.9: Neighbour references of a quadtree leaf node highlighted before and after subdivision. A sub-optimal child neighbour reference is updated during ray casting.

3.5 Memory Management System

By adding more detail through subdivision to the SVO structure, system memory usage increases exponentially as $O(x^s)$, where x is the degree of the tree and s is the number of subdivision levels; this is obviously unsustainable. This section describes the implementation of a *dynamic memory management system* inspired by static out-of-core techniques used in [10] to unload SVO data when it is no longer visible.

The octree nodes stored in memory at any one time is known as the *working set*. The memory management system's goal is to arrange a working set of voxels which adequately represent the section of the fractal being explored whilst storing as little data as possible describing non-visible sections. The number of nodes in the octree makes it unfeasible to keep track of every individual node's visibility, but the regular SVO structure can be exploited by tracking the visibility of sets of nodes.

3.5.1 Brick Storage Structure

Bricks are cubical sets of voxels representing a section of the whole model - sub-trees of the entire octree. As child boundaries lie inside their parent's boundaries, when a ray passes through a child node it must also have passed through the brick

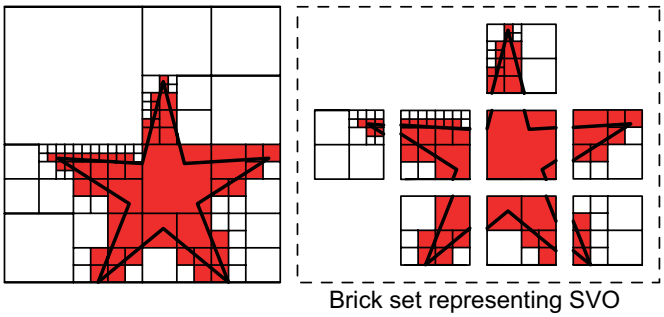


Figure 3.10: Quadtree statically decomposed into bricks once after 2 subdivision levels. Every brick is the same 3D size and there is no upper bound on how many voxels a brick can represent.

that child belongs to. In this way a working set is constructed out of bricks by a `BrickManager` and a recently `visited` flag is tracked for each brick rather than each node.

Dynamic Brick Creation

In previous out-of-core systems such as [10] the SVO model is decomposed into bricks of equal size before run-time (as in Figure 3.10), but in a scale-adaptive SVO this would not be sufficient as volumetric data is not available before subdivision and equal-sized bricks limit the granularity of memory management. As the scale of detail decreases, the scale of the smallest bricks stored in the working set should also decrease so the number of voxels they represent remains constant.

This was implemented by dynamically assigning nodes on generation as *brick representatives* (the root of the brick subtree) and adding a reference in each node to the nearest brick up the hierarchy it is a member of. Bricks are created every fixed cycle of subdivision levels so the number of bricks in the SVO remains relatively constant as scale decreases, as can be seen in Figure 3.11. A reference to each brick is stored by a `BrickManager` in a `brickSet`. The root node is the primary brick and each brick will contain smaller bricks within themselves following subdivision, so nodes can be members of multiple bricks.

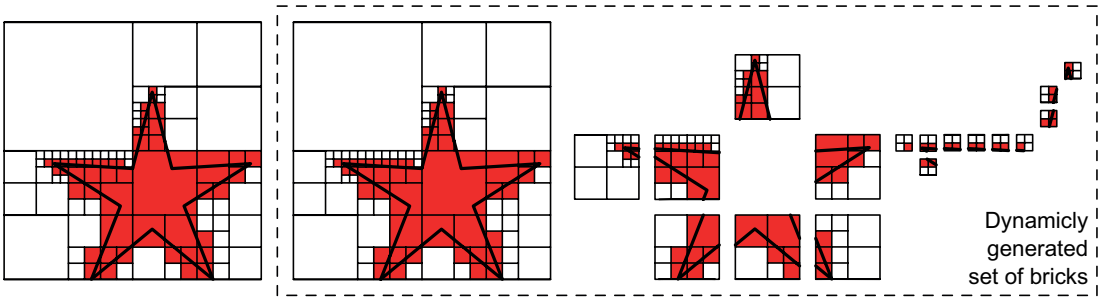


Figure 3.11: Quadtree dynamically decomposed into bricks every 2 subdivision levels. Bricks generated become progressively smaller so the smallest bricks will never represent more than 4^2 voxels.

Unifying Unvisited Bricks

When a ray passes through a node, it marks every brick that node is a member of as `visited` by following brick references up the SVO. This up-traversal can terminate early if a brick has already been marked as visited as it will be certain every brick it is a member of has already been marked as visited. To free memory a procedure `unifyBricks()` is occasionally called which unloads all unvisited bricks including their child nodes, implementing a *Not Recently Used* policy, as shown in Figure 3.12. Allowances can be made for non-visited bricks near the top of the SVO hierarchy as they will be few in number and give a rough approximation of the fractal which is always immediately avaiable for further subdivision if the user rotates to an un-refined area.

The `unifyBricks` procedure combines every node in a non-visible brick into a single leaf node by setting that brick's `leaf` flag. The SVO in this state is no longer consistent however as neighbor references from outside into the brick may still reference nodes that should be unloaded. To rectify this each member of the brick has a `deleted` flag set and its fields cleared. If a `deleted` node is encountered during traversal then the neighbor reference is reset during a `node.getNeighbour()` procedure to its corresponding unified brick, providing a consistent SVO state.

As memory cannot be explicitly freed using Java, clearing the references to unvisited nodes will flag them for garbage collection as they can no longer ever be accessed. In this way the JVM automatically unloads non-visible node data.

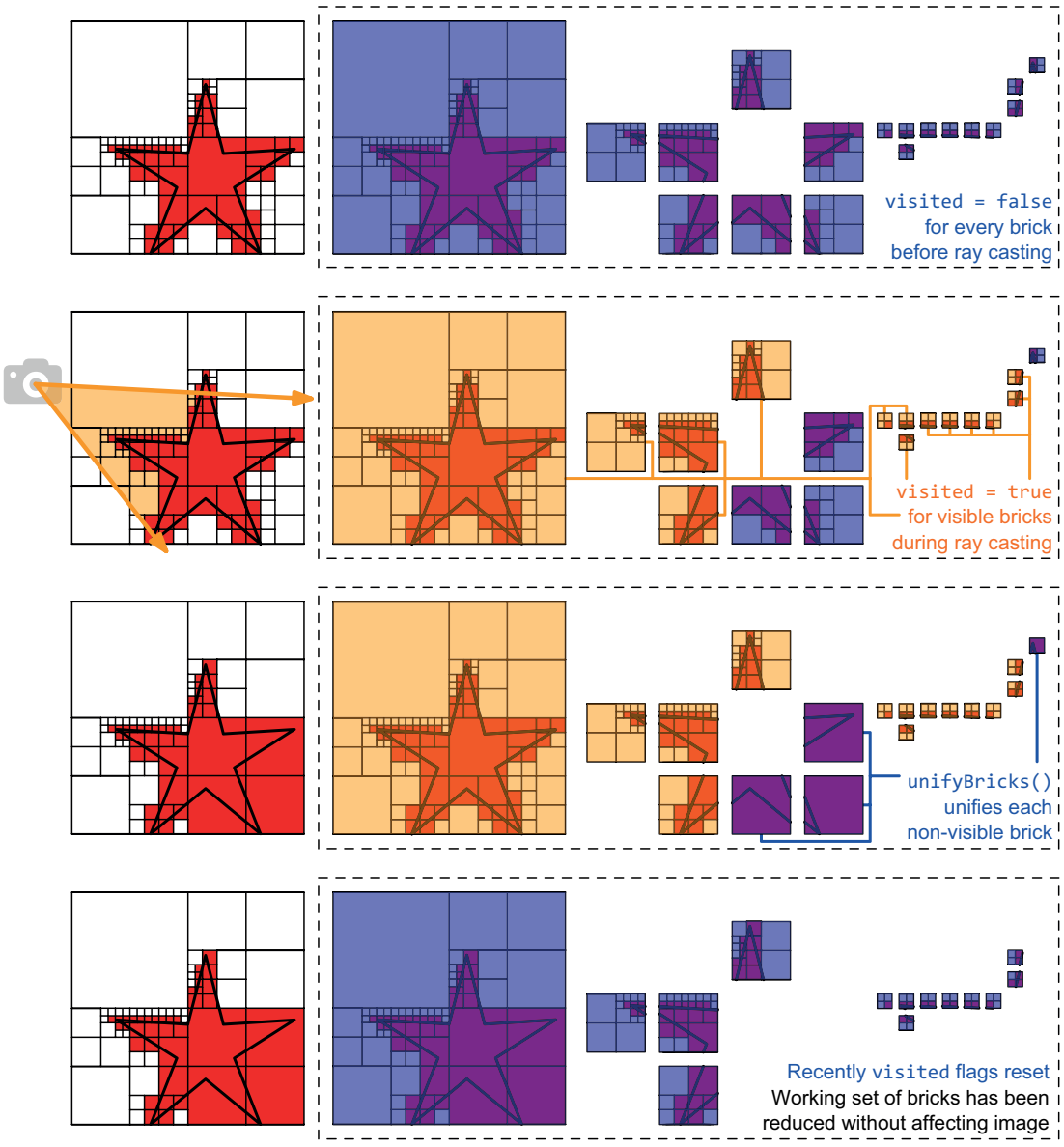


Figure 3.12: Unloading Not Recently Used bricks from the `brickSet` to simplify the SVO and free memory using `unifyBricks()` after `visited` flags are set during rendering.

3.5.2 Concurrency

Unloading a large number of brick nodes has scope for multi-threading performance improvements as many bricks can be unified simultaneously. However a safety issue must be avoided where two bricks can be unified simultaneously though one is a member of another. To avoid this any unification or deletion action is carried out in a critical section provided by thread synchronization. If unification is started on a brick that has already been deleted then the process immediately terminates as it is no longer necessary.

Unification and subdivision can be executed in parallel by synchronizing them over the node they are carried out on to avoid concurrent modification. Ray-casting and unification are not carried out in parallel however as there is no simple procedure to follow if a ray is traversing through a brick simultaneously being deleted and the necessary locks to make it safe would severely impact performance.

As a result node subdivision threads run constantly while the system multiplexes rendering and node unification over time.

3.6 Post-Processing Effects

This section describes how fractal details are made more discernible after rendering without using surface normal calculations or casting additional shadow rays which would both require extra memory usage and processing time.

Screen-Space Ambient Occlusion

Screen-space ambient occlusion (SSAO) is a graphics lighting technique to approximate ambient occlusion (AO) - shadows formed by light radiating off lambertian surfaces, e.g. corners being darker than the middle of a wall in a room. This is implemented using only the depth buffer which stores the distance of each pixel from the camera. A kernel is iterated over the depth buffer, calculating the average ratio between each pixel's depth and the surrounding depths. Large differences are ignored as they represent standard occlusion and should not affect results. The average ratio reveals whether a pixel is extruding or intruding compared to its surroundings. Extruding pixels are highlighted to extenuate edges

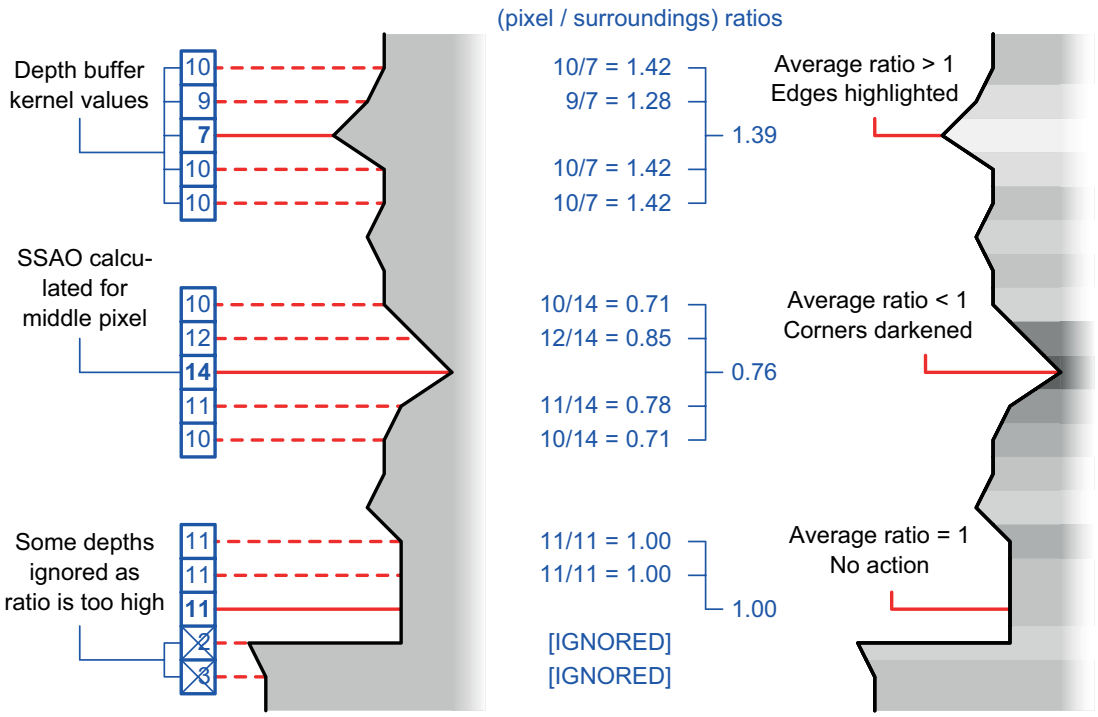


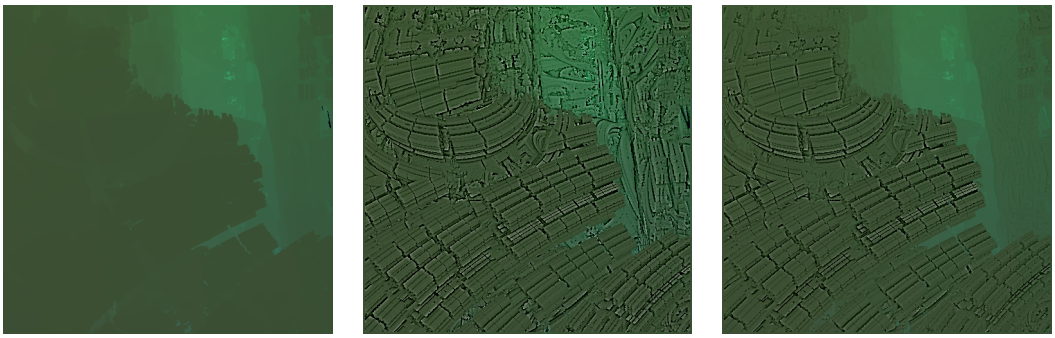
Figure 3.13: 2D SSAO calculation with a 5-wide kernel over a 2D depth buffer. The rightmost 2D outline shows SSAO lighting results.

while intruding pixels are darkened to represent AO shadow, as shown in Figure 3.13.

As ratios between depths are used this effect scales during user exploration so strictly-self-similar fractal sections will look exactly the same even though they are of different sizes, and would realistically have different AO.

Fog effects

A SSAO modification provides fog effects which draw the user's attention away from background detail to exaggerate foreground detail. This is done by alleviating SSAO shadows over a certain distance away from the fractal rendered in the centre of the image. This understates background detail without removing the fractal shape, as seen in Figure 3.14.



(a) No post-processing (b) SSAO without fog (c) SSAO with fog

Figure 3.14: The difference between Mandelbox images direct from the renderer and enhanced with various post-rendering techniques.

3.7 System Summary

This section summarizes the modifications made to the basic SVO data structure to allow faster octree traversal, scale-adaptation, and dynamic memory management. It also outlines the complete scale-independent rendering algorithm using the additional techniques described above. Together, they produce a system capable of fulfilling the project requirements.

3.7.1 Scale-Adaptive SVO Data-Structure

Fields added for direct neighbour node traversal during rendering:

- `OctreeNode[] neighbors`: array of references to neighbouring nodes.
- `int depth`: node's distance in down-traversals from the root.

For scale adaptation and thread-safety:

- `OctreeNode parent`: reference to the node's parent; used in recursive recombination of empty nodes.
- `boolean queuedSubdiv`: true if node is queued for subdivision, false otherwise.

For memory management:

- `OctreeNode brick`: reference to the node's brick representative.

- `boolean visited`: `true` if a node in its brick has been recently visited, `false` otherwise; only used by brick representatives.
- `boolean deleted`: `true` if node member of unified brick, `false` otherwise.

With these extra fields each `OctreeNode` consumes 26 bytes (without JVM data). While it is not possible to retain gigavoxel data-sets in memory, enough voxels can be stored in the working set to provide a smooth approximation of the shape to the user.

3.7.2 Scale-Independent Ray Cast Algorithm

Algorithm 3.2 calculates the result of a ray-cast accelerated with neighbour-traversals between nodes and skip-node initiation. It is also adapted to implement memory management with a NRU policy. Some sections from the standard algorithm which were not changed have been summarized for legibility. Full ray cast code can be found in Appendix B.

Algorithm 3.2 Ray Casting using Kd-Restart

```
1: node ← skipNode                                   ▷ Initialize skipNode values
2: boxMin ← skipNode.boxMin
3: boxDim ← skipNode.boxDim
4: Calculate t_min for where ray first enters node
5: while node ≠ null do
6:     P ← ray.start + t_min * ray.dir
7:     while !node.isLeaf() do
8:         Potentially terminate early with implicit LOD
9:         Traverse to child node
10:    end while
11:    node.visit()                                  ▷ Sets brick visited flags
12:    if !node.isEmpty() then
13:        Scale-adapt SVO to changes in scale if necessary
14:        return node.colour
15:    else
16:        Calculate t_x(x1), t_y(y1), t_z(z1), and t_max of ray-node intersection
17:        neighbourDir ← neighbour direction from t-values
18:        neighbour ← node.getNeighbour(neighbourDir)
19:        boxMin += boxDim * neighbourDir
20:        if neighbour.depth < node.depth then
21:            boxDim = ← 2^-(neighbour.depth+1)
22:            Snap boxMin onto grid described by neighbour.depth
23:            Optimize node.neighbour reference if possible
24:        end if
25:        node ← neighbour
26:        t_min ← t_max
27:    end if
28: end while
29: return background colour                         ▷ Ray has missed SVO
```

Chapter 4

Evaluation

This chapter describes the ways in which the project was evaluated against its requirements. The memory management and procedural generation systems were benchmarked to show that new fractal data can be generated indefinitely, and the rendering system was benchmarked to show that fractals can be rendered in real-time with performance independent of scale. Fractal data errors and image artifacts[1] were also analysed.

4.1 Evaluation Preparation

This section desribes the work done before carring out experiments to ensure evaluations were accurate and meaningful.

Experiment Objectives

The main questions this evaluation attempts to answer are:

1. **Is the procedurally generated fractal detail correct?** The voxels presented to the user should accurately represent the fractal shape. Rendered images can be compared against the output of other fractal rendering systems, and strict-self-similarity in IFS fractals should be apparent.
2. **Does memory usage remain bounded as smaller fractal details are generated?** To allow long explorations, the system must cope with

[1] Error in the presentation of visual information

continuous fractal data generation. Memory usage can be sampled during exploration to determine if it is bounded or not.

3. **Is rendering time per image independent of scale?** Users should not experience a decrease in performance as a result of viewing smaller scale detail. Frames rendered per second can be sampled during exploration to determine whether performance is affected by scale.

4. **Can the fractal be rendered in real-time?** The renderer must produce images at interactive framerates where movement appears fluid.

Test Suite and Harness

To assess the system's memory usage and performance a small test suite was composed containing an IFS and Escape-Time fractal, representing both archetypes. The project's entire range of fractals is displayed in Appendix C and screenshots from the automated tests can be seen in Appendix D. User exploration was simulated and included a cinematic journey into and around the fractal and a continuous zoom into one fractal section[2].

- The IFS Menger Sponge was used as it contains strictly-self-similar detail so the voxel representation should be near-identical at different scales when exploring points of recursion.

- The Escape Time Mandelbulb was used as it contains quasi-self-similar detail so the system's ability to continuously produce and render unpredictable fractal shapes can be determined.

Images were rendered into a square viewing window with size described by the length of one of its sides in pixels.

User input during experiments can affect results so all tests were automated with JUnit and repeated multiple times to reduce noise in the results caused by background processes. Measurements were taken every fixed cycle of rendered images to provide a controlled independent variable representing an amount of work that must be done by the system. Once the automated explorations are complete, results are calculated using the Apache Commons statistics library[3] and written to disk.

[2]Videos of tour explorations can be found online: www.youtu.be/BU5UXhPGvU4 (Menger Sponge), www.youtu.be/oFAiiiJTo50 (Mandelbulb)

[3]http://commons.apache.org/math/

Evaluation Hardware

The evaluation was carried out on a desktop computer with 2nd generation Intel i5 2500K CPU (4 cores clocked at 4.7Ghz) and 12GB of RAM running Windows 7. As the system does not use any GPU or out-of-core mechanisms, these specifications entirely govern performance.

4.2 Fractal Correctness

This section describes the system's ability to generate data accurately representing a fractal. A comparison can be made between the output of this voxel-based system and an alternate raymarch-based system using the same DE algorithms, Fragmentarium[4]. The implemented IFS fractals should also correctly display the property of strict-self-similarity, where sections of them appear identical at different scales.

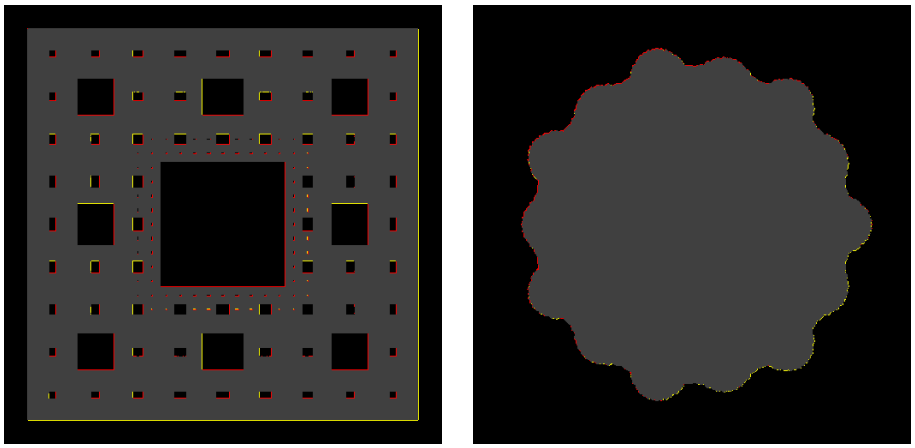
Comparison with Fragmentarium

Hypothesis: Fragmentarium utilizes the same pre-existing DE functions as this project, so fractals rendered by both systems should produce similar images.

Experiment: Ideally both systems would be set up to render identical images with identical checksums, but Fragmentarium implements different graphics techniques for shadows and fractal colouring so this was not possible. To verify the shape of the fractals generated, both systems were set up to render thresholded images of the Menger Sponge and Mandelbulb under specified camera settings and pixel differences were used to measure fractal outline similarity. Verification of fractal detail was carried out by eye, with outputs from both systems overlaid to facilitate comparison.

Results & Analysis: Images of fractal outlines (Figure 4.1) measured as 99% identical with small differences resulting from difficulties in matching the configuration of the ray-marching and ray-casting systems. The fractal shapes and detail (Figure 4.2) visible however is similar enough to show that the DE functions used in Fragmentarium were correctly implemented in this project. Though DE is only an approximation so results generated will not correctly represent the fractal for

[4]http://syntopia.github.com/Fragmentarium/



(a) Menger Sponge outline differences (b) Mandelbulb outline differences

Figure 4.1: Differences between thresholded fractal outlines produced by both systems were minimal. Pixels only in Fragmenterium's output: red, pixels only in this project's output: yellow.

all evaluations[5], this project can generate fractal images as accurately as other systems so the fractal SVO data can be deemed sufficiently correct.

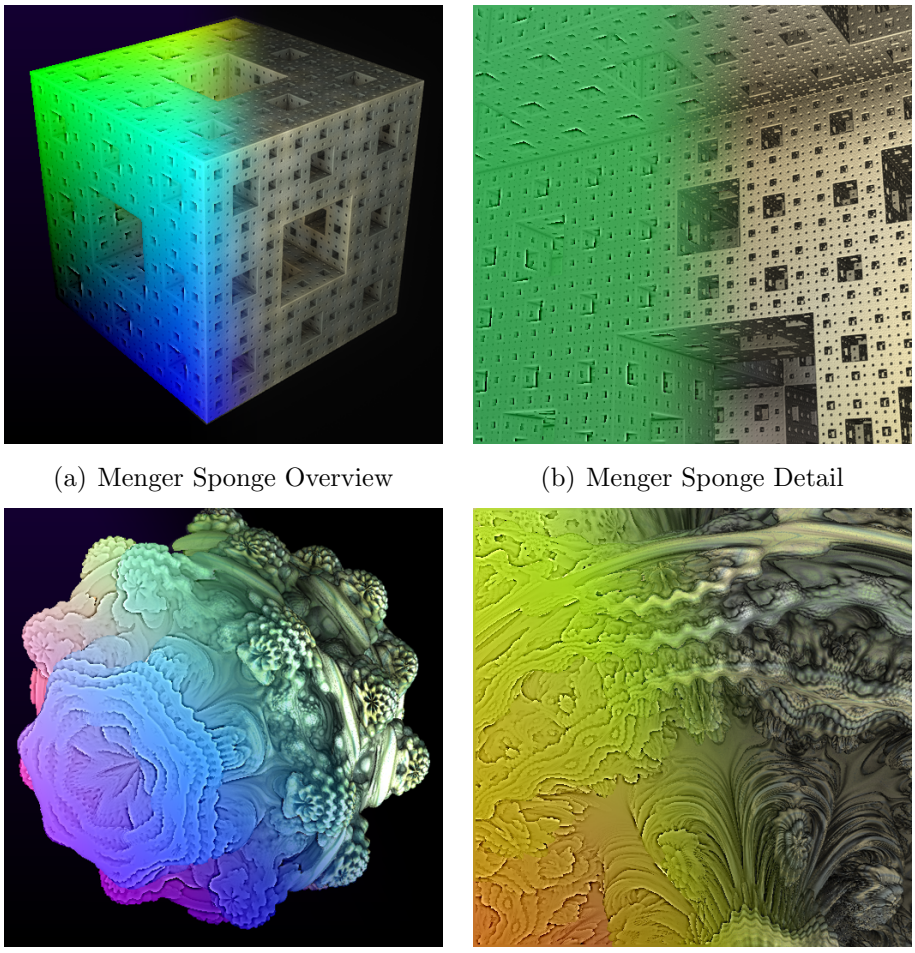
Strict-Self-Similarity Evaluation

Hypothesis: As IFS fractals are strictly-self-similar, renders focussed on a point of recursion should output the same image at certain different scales.

Experiment: Images of points of recursion in the Menger Sponge can be saved at distances of 1/3, 1/9, and 1/27; and in the Sierpinski Gasket at distances of 1/2, 1/4, and 1/8. The similarity of these images can be measured by comparing their difference in pixels.

Results & Analysis: The 3 raw Sierpinski Gasket images were 99% identical, but the 3 Menger Sponge images in Figure 4.3 were only 28% identical. These discrepancies between images are result of node boundaries not aligning with fractal boundaries, producing small differences in the image. Once a 10% difference in pixel colour was allowed, the Menger Sponge images measured as 99% similar. This sufficiently shows that the IFS fractals generated by this project display the strict-self-similarity property.

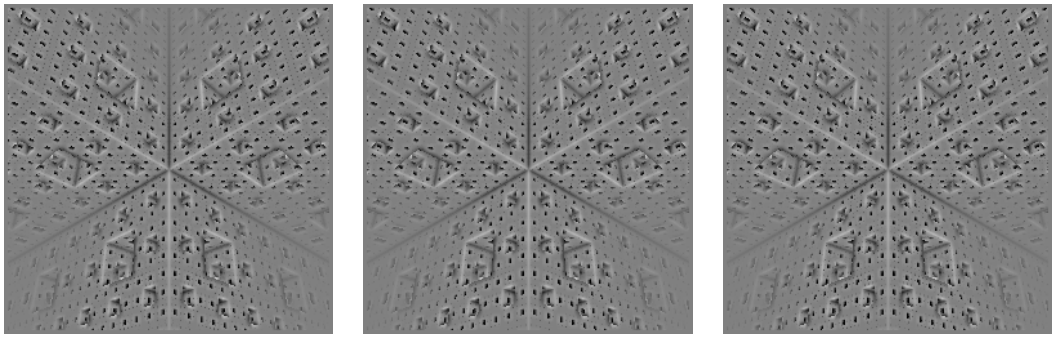
[5]Calculating the generalized accuracy of DE is beyond the scope of this project



(a) Menger Sponge Overview (b) Menger Sponge Detail

(c) Mandelbulb Overview (d) Mandelbulb Detail

Figure 4.2: These overlaid images show the similarity between the test suite fractals generated by this project (on the left), and the fractals rendered in Fragmenterium (on the right).



(a) Distance: 1/3 (b) Distance: 1/9 (c) Distance: 1/27

Figure 4.3: Near-identical images from different scales inside the Menger Sponge.

Erroneous Fractal Data

During fractal exploration the system occasionally generated false-negative errors, marking a SVO node as an empty leaf when it should contain fractal data. These errors were only recorded during Menger Sponge explorations, perhaps because they are most noticeable against a cubical fractal shape. The nodes affected were always the same. It is likely that this is a floating point computation error as it can be rectified by slightly increasing the bounding sphere checked by `isInFractal()`.

4.3 Memory Management

This section evaluates the system's ability to indefinitely generate new fractal data without running out of memory. For this to be the case, memory usage (JVM heap size - JVM used memory) and the number of bricks in the working-set should be bounded during each exploration.

JVM Memory Usage

Hypothesis: A suite of repeated explorations can be run on the same Java application but the JVM will never run out of memory despite new fractal data being constantly generated. This is because the memory management system will correctly allow non-visible voxel data to be garbage collected.

Experiment: Menger Sponge and Mandelbulb zoom explorations were repeated 20 times and JVM memory usage periodically sampled. The JVM was allowed 2Gb of memory to provide a reasonable cap on the size of the working-set.

Results & Analysis: Average JVM usage was 1200Mb, under half of the maximum 2048Mb the JVM had available (Figure 4.4). This shows that the memory management system worked correctly and garbage collection freed memory used by old SVO nodes. Memory usage was similar for both fractals. The standard deviations are large as it is not possible to predict when garbage collection is initiated during the repeated explorations as each iteration shares the same heap, containing old node data from the previous iteration. As a result memory usage is not an accurate measurement of the number of nodes in the working set.

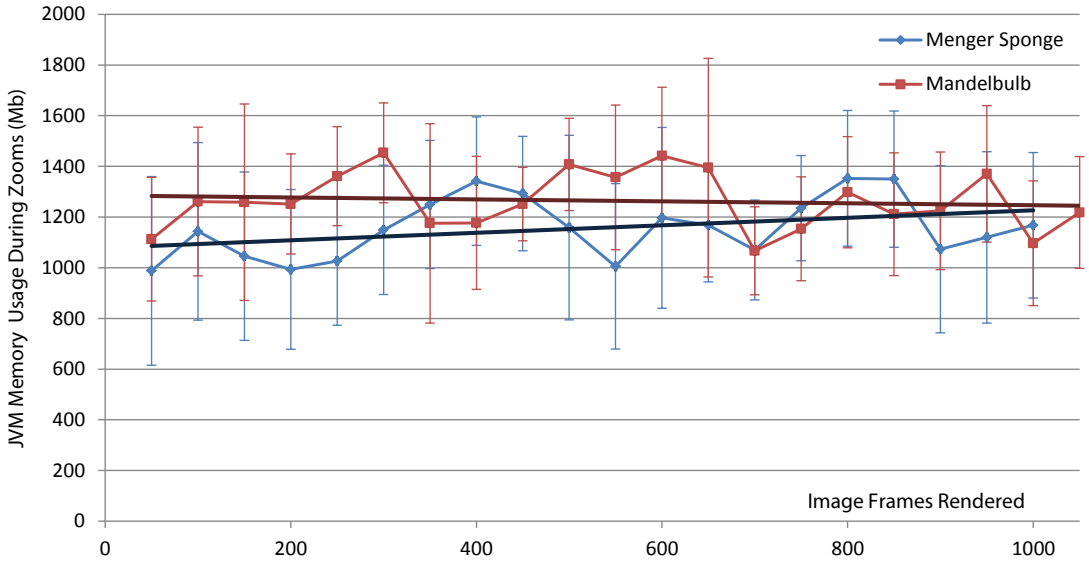


Figure 4.4: JVM memory usage over repeated Menger Sponge and fractal explorations.

Size of Brick Set

Hypothesis: During a zoom into a point of recursion, the number of nodes subdivided each frame should stay roughly constant. Therefore the number of bricks in the working set should follow a cycle as new bricks are generated and added every few subdivision levels, and unloaded when no longer visible.

Experiment: A long Menger Sponge zoom exploration was executed with the size of the brick set being periodically measured. A linear trend is fitted to assess how the average number of bricks changes over time.

Results & Analysis: A brick set size cycle is visible in Figure 4.5, increasing dramatically each time a new level of bricks is generated, and dropping as old bricks are unloaded. Error bars are not visible as the standard deviation is very low (averaging 0.1% for only 5 repetitions). This is because the working set is nearly exactly determined by what voxels are visible at each image frame.

The average number of bricks follows a slightly increasing linear trend. This is because an octree hierarchy from root node must be kept in memory for each visible brick, but this is acceptable considering the distance from fractal decreases from by a factor of 10^{-11} during the exploration whilst the average brick set size does not significantly change. There is only a 1.6% difference between the size of the working set at the first cycle and the maximum working set size.

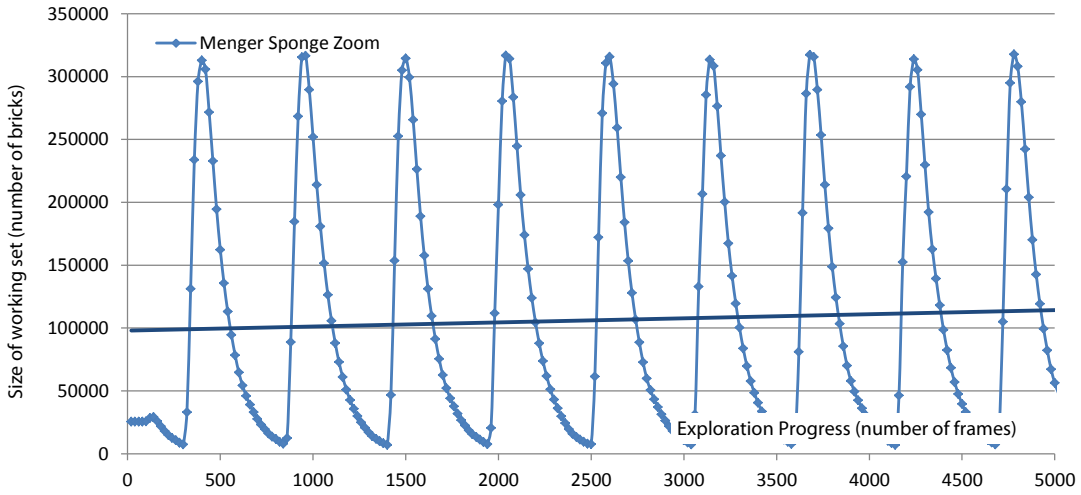


Figure 4.5: Brick set size cycle as a new level of bricks is periodically generated during a Menger Sponge zoom.

4.4 System Performance

This section evaluates the scale-independent rendering algorithm and compares its performance against kd-restart. It also analyses the relationship between the image size, fidelity of voxel approximation, and performance. To measure of performance, an average number of image frames rendered per second (fps) is calculated by dividing a fixed cycle length of frames by the time taken to render them.

Scale-Independence during Explorations

Hypothesis: Throughout a zoom exploration into a point of recursion, fractal data being generated and rendered will be identical except differences in scale. The scale-independent rendering algorithm should therefore provide constant performance whereas the kd-restart algorithm suffers.

Experiment: Both scale-independent and kd-restart rendering algorithms were used during Menger Sponge zooms where the distance from the camera to the fractal is reduced from 1×10^{-11}, though detail presented remains the same. A linear trend line was calculated for both algorithms to determine the relationship between performance and scale.

Results & Analysis: The scale-independent algorithm performed 80% better than kd-restart at the beginning of the exploration and 500% better at the end

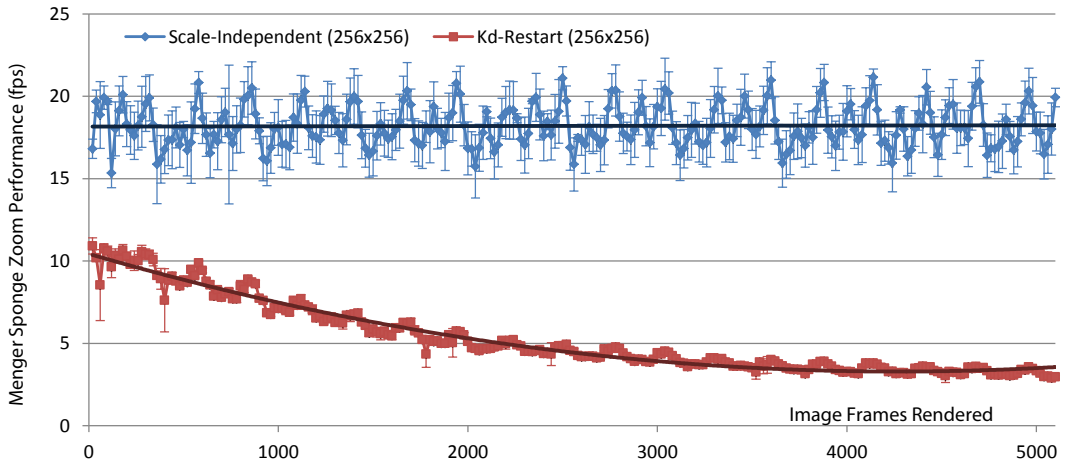


Figure 4.6: Average fps during a Menger Sponge zoom to compare kd-restart with the scale-independent rendering algorithm.

(Figure 4.6). Kd-restart performs worse in general as it does not use a skip-node or neighbour traversals, but while the trending fps for the scale-independent algorithm remained constant throughout the experiment (gradient of 10^{-5}), kd-restarts performance was inversely proportional to the distance away from the fractal (distance is halved every 138 frames). This is because the each kd-restart ray traverses the entire octree height for each node visited and this cost is proportional to the height of the octree which grows as detail is added.

The scale-independent algorithm was not a close match for its trend line (R^2 = 0.0004). Though the images rendered may appear identical for each point of recursion iteration, the octree hierarchy will not be. At times the skip-node may be large, allowing traversal to jump large distances; while at other times it may be small, meaning more neighbour traversals are needed. This difference in octree hierarchy has the same effect on kd-restart, but to a lesser extent as it does not use neighbour traversals (R-squared 0.96).

Other Experiments: Similar experiments were carried out on the Mandelbulb, producing similar results (Figure 4.7). The slight downward trend is a result of the nature of the Mandelbulb zoom where previously obscured sections of the Mandelbulb suddenly become visible, requiring a large number of subdivisions. Sections of the fractal relatively far from the foreground fractal detail remain visible during exploration, requiring more neighbour traversals to render. Larger variations in performance over the course of the exploration were apparent as the octree hierarchy differs vastly for unpredictable fractal shapes.

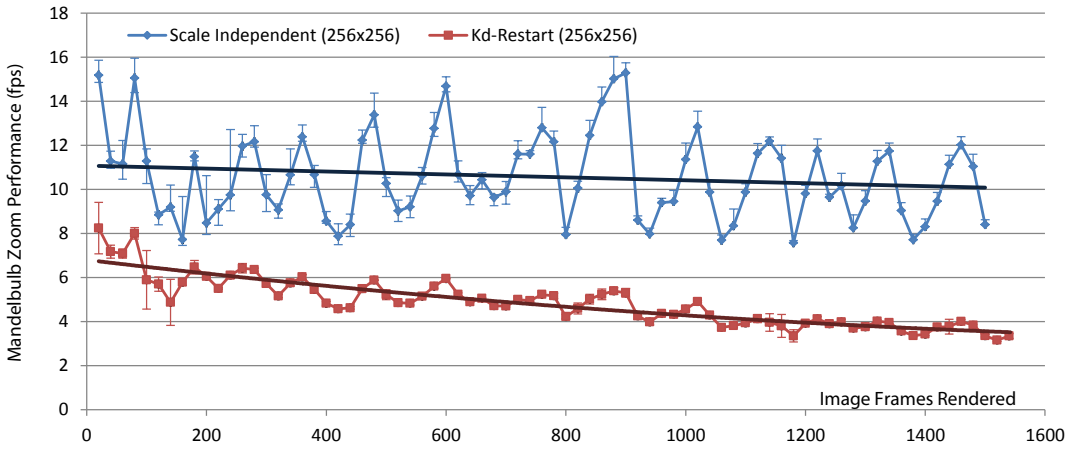


Figure 4.7: Average fps during a Mandelbulb zoom to compare kd-restart with the scale-independent rendering algorithm.

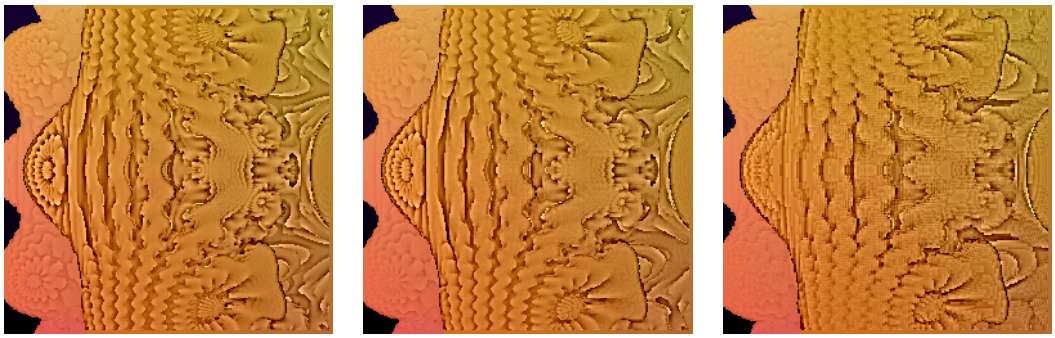
Image Size, Fidelity, and Performance

Hypothesis: Rendering time will be inversely proportional to the square of the image size as the detail in the image is resolution-bound so one ray must be cast per pixel. This performance decrease may be mitigated by allowing the voxels to more roughly approximate the fractal by adjusting the SVO fidelity.

Experiment: Average fps from the beginning of the Mandelbulb zoom and from an external overview of the Menger Sponge were recorded at various images sizes and degrees of voxel approximation fidelity (effects shown in Figure 4.8), so the relationship between image size, fidelity and performance can be measured.

Results & Analysis: Mandelbulb performance was proportional to $size^{1.95}$ with $R^2 = 0.99$ (Figure 4.9). Doubling the width of the image quarters performance because 4 times as many rays will need to be cast and 4 times as many nodes will be subdivided. Halving the fidelity of the octree model increased performance because rays now stop one traversal earlier than before and only a quarter as many subdivisions need be carried out.

The Menger Sponge experiment (Figure 4.10) showed similar trends. However changing the SVO fidelity no longer improved performance to the same extent. This is because the user is further away from the fractal, so fewer nodes in total need to be subdivided. The DE calculations for a Menger Sponge are simpler than the Mandelbulb so less performance is gained through fewer subdivisions. Some rays also miss the root bounding cube so are not affected by voxel fidelity at all.



(a) pixel/voxel ratio: 1 (b) pixel/voxel ratio: 4 (c) pixel/voxel ratio: 16

Figure 4.8: The difference between images rendered with different levels of voxel-approximation fidelity. The difference between rendering at fidelity levels of 1 and 4 are minimal but can lead to a significant increase in performance.

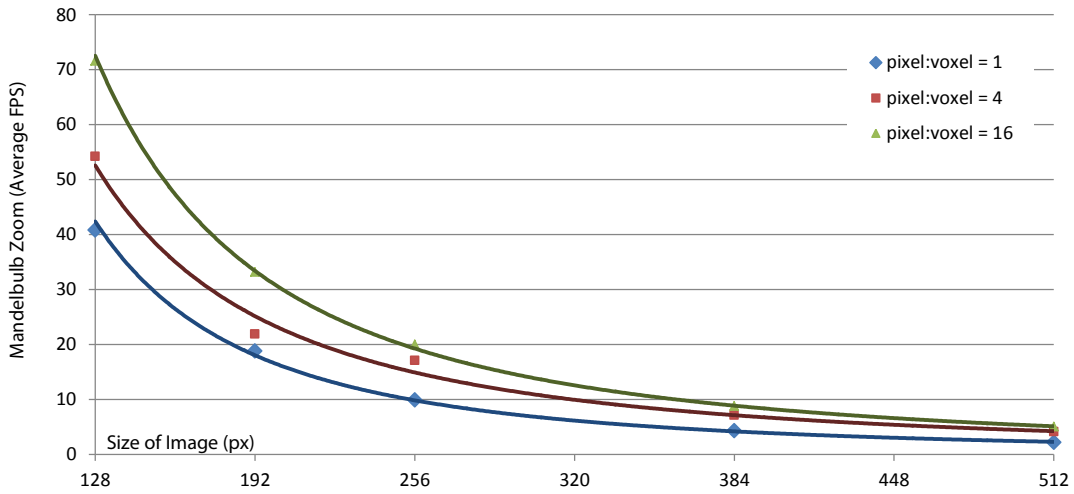


Figure 4.9: The relationship between voxel fidelity, image size, and performance for a Mandelbulb.

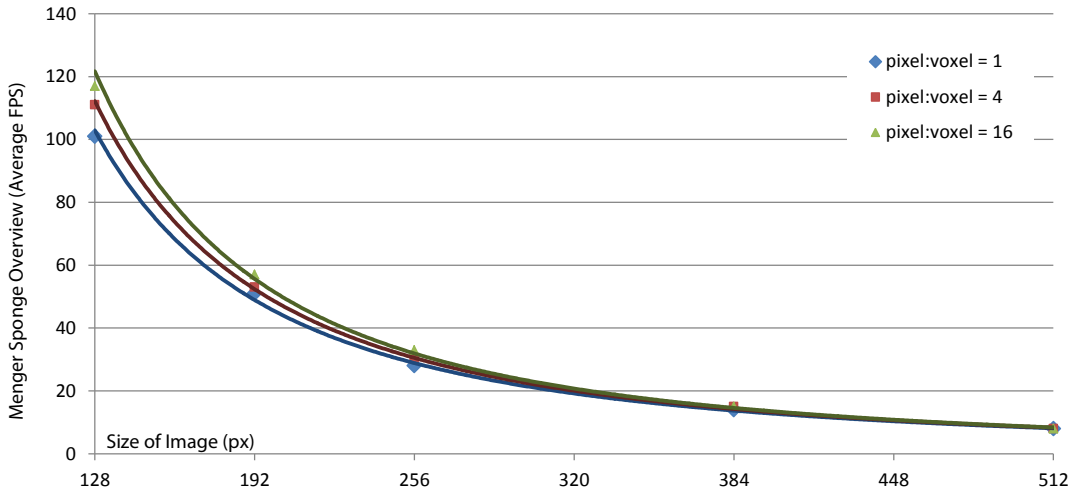


Figure 4.10: Improvements gained in the Mandelbulb by lowering voxel fidelity are not so effective for the Menger Sponge overview.

Exploring the outside of the Menger Sponge could be performed at 256x256 with maximum fidelity at 30fps, but the Mandelbulb zoom ran at 10fps for the same settings, showing the variation in performance between exploring different parts of different fractals. This is the result from some DE functions being more complex than others, so their subdivision threads affect renderer CPU time to a greater extent. Also, In some scenes rays may have to travel long distances over many neighbour traversals while in others they stop relatively short, affecting each ray cast time. By varying on the size of the image and the fidelity the user explores the fractal at, there will always be configurations where the user can experience interactive frame rates during their exploration.

Multi-Threading and Performance

Hypothesis: Rendering an entire image using ray casting is embarassingly parallel, so performance can be improved by allowing the system to use more independent rendering threads. These performance improvements will be limited however by the number of concurrent rendering threads that can be processed on the CPU simultaneously.

Experiment: Average fps from the beginning of the Menger Sponge zoom was measured using 1, 4, and 10 rendering threads in a thread pool.

Results & Analysis: Average performance increased by 120% from 11fps at 1 thread to 24fps at 4 threads, but no futher improvements could be offered by

using more concurrency with 10 threads only producing 21fps. This is because at most 4 rendering threads can simultaneously run effectively on the 4 separate i5 CPU processors (no hyperthreading support), and extra threads only add context switching overhead to rendering time.

4.5 Image Artifacts

During animation, small horizontal line artifacts were noticed around the edges of fractals being rendered. These are caused by the subdivision process and rendering process being run simultaneously. One scanline of the image will render a leaf node before it is subdivided while the next scanline will render it following subdivision. These visual effects are small and would require significant thread synchronization to resolve so were ignored.

Chapter 5

Conclusion

The project fulfilled its core success criteria. A system was implemented that allows a user to navigate a 3D fractal, exploring it in real-time while new fractal detail is simultaneously generated and displayed. A scale-adaptive SVO with dynamic memory management was implemented capable of indefinite fractal data generation. This was evaluated by determining that the JVM memory and working set size are bound. A scale-independent octree traversal algorithm was implemented that allows users to experience exploration of small-scale details with the same interactivity as exploration at larger scales. To show this was a success, an evaluation of system performance over different fractals and rendering conditions was carried out. To achieve high performance, these components can run in parallel, making use of chip multi-processors.

The scale-adaptive SVO and scale-independent rendering algorithm provide a starting point for systems capable of procedurally generating a wide range of fractal-based volumetric models such as foliage, mountain ranges, or even entire planets; all of which may adapt to ensure detail is visible at every scale.

5.1 Future Improvements

This project could be ported to a general purpose graphics processing unit language such as OpenCL[1] to improve performance. In this way the project's parallel nature could be exploited further by using the multitude of GPU cores which

[1] http://www.khronos.org/opencl/

are designed for single instruction multiple data operations; ideal for processing batches of rays through an SVO.

Additional computer graphics effects could be implemented for more realistic images. Improved lighting can be achieved by including surface normals in the SVO and casting additional rays for shadow calculations. Blurry depth of field effects can be efficiently implemented using ray-casting as less detail is needed for out-of-focus areas so rays can terminate early. More post-processing effects such as FXAA for anti-aliasing could be implemented.

Following further study of Distance Estimation, a higher degree of control over fractal maths could be implemented, allowing fractal colouring or modifications to the generation process to distort the fractal shape. This could allow rendering of scale-adaptive organic scenery.

Bibliography

[1] Benoit Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Company, New York, 1977.

[2] Cyril Crassin. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, UNIVERSITE DE GRENOBLE, July 2011. English and web-optimized version.

[3] Dennis Bautembach. Animated sparse voxel octrees, February 2011.

[4] Thomas W. Crockett. Parallel rendering. Technical report, NASA Langley Research Center, April 1995.

[5] Kenneth Falconer. *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley & Sons Ltd., Chichester, 1990.

[6] Mikael Christensen Hvidtfeldt. Distance estimated 3d fractals, syntopia blog. http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i/. May, 2012.

[7] John C. Hart, Daniel J. Sandin, and Louis H. Kauffman. Ray tracing deterministic 3-d fractals. *Computer Graphics*, 23(3), July 1989.

[8] Kristof Rmisch. Sparse voxel octree ray tracing on the gpu. Master's thesis, Aarhus University, July 2009.

[9] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17:1048–1059, 2011.

[10] Enrico Gobbetti, Fabio Marton, and Jose Antonio Iglesias Guitian. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *Vis. Comput.*, 24(7):797–806, July 2008.

Appendix A

Distance Estimation Examples

Listing A.1: Sample Distance Estimation code for the Menger Sponge ported from Fragmentarium.

```
public boolean isInFractal(double x, double y, double z, double d) {

  double r = x * x + y * y + z * z;
  double scale = 3d;
  double MI = 10000; //maximum number of iterations

  int i;
  for (i = 0; i < MI && r < 9d; i++) {

    //folding operations carried out on point
    x = Math.abs(x);
    y = Math.abs(y);
    z = Math.abs(z);
    if (x - y < 0d) {
      double x1 = y;
      y = x;
      x = x1;
    }
    if (x - z < 0d) {
      double x1 = z;
      z = x;
      x = x1;
    }
    if (y - z < 0d) {
      double y1 = z;
      z = y;
      y = y1;
    }

    // scale the point around a fractal vertex
    x = scale * x - 1d * (scale - 1d);
    y = scale * y - 1d * (scale - 1d);
    z = scale * z;

    // cut out the middle section of the Menger Sponge
```

```
36      if (z > 0.5d * 1d * (scale - 1d)){
37        z -= 1d * (scale - 1d);
38      }
39      r = x * x + y * y + z * z;
40    }
41    return ((Math.sqrt(r)) * Math.pow(scale, (-i)) < d);
42  }
```

Listing A.2: Sample Distance Estimation code for the Mandelbulb ported from [6].

```
1  final double pow = 8d;
2  public boolean isInFractal(double x, double y, double z, double d) {
3    // store starting x, y, z
4    double posX = x;
5    double posY = y;
6    double posZ = z;
7
8    double MI = 20;   //maximum number of iterations
9    double dr = 1.0d; //the spatial derivative
10   double r = 0.0d;
11
12   for (int i = 0; i < MI; i++) {
13     r = FastMath.sqrt(x * x + y * y + z * z);
14     if (r > 10) //break if outside bailout value
15       break;
16
17     // convert to polar coordinates
18     double theta = FastMath.acos(z / r);
19     double phi = FastMath.atan2(y, x);
20     dr = FastMath.pow(r, pow - 1d) * pow * dr + 1.0d;
21
22     // scale and rotate the point
23     double zr = FastMath.pow(r, pow);
24     theta = theta * pow;
25     phi = phi * pow;
26
27     // convert back to cartesian coordinates
28     x = zr * FastMath.sin(theta) * FastMath.cos(phi);
29     y = zr * FastMath.sin(phi) * FastMath.sin(theta);
30     z = zr * FastMath.cos(theta);
31
32     x += posX;
33     y += posY;
34     z += posZ;
35   }
36   return ((0.5d * FastMath.log(r) * r / dr) < d);
37 }
```

Appendix B

Ray Cast Thread Code

Listing B.1: Code used in this project to set depth-buffer and image colour values using ray casting.

```
// initialize variables to skip node (startNode) values
OctreeNode node = startNode;
Vector3d boxMin = new Vector3d(startBoxMin);
double boxDim = startBoxDim;

// initialize ray direction and start position
Vector3d pVec = ray.getDir();
Vector3d cPos = ray.getStart();

// calculate t values for each corner of the skip node
double tx0 = (boxMin.x - cPos.x) / pVec.x;
double tx1 = ((boxMin.x + boxDim) - cPos.x) / pVec.x;
double ty0 = (boxMin.y - cPos.y) / pVec.y;
double ty1 = ((boxMin.y + boxDim) - cPos.y) / pVec.y;
double tz0 = (boxMin.z - cPos.z) / pVec.z;
double tz1 = ((boxMin.z + boxDim) - cPos.z) / pVec.z;

// ensure t0 and t1 are in the correct order
if (tx1 < tx0) {
  double temp = tx0;
  tx0 = tx1;
  tx1 = temp;
}
if (ty1 < ty0) {
  double temp = ty0;
  ty0 = ty1;
  ty1 = temp;
}
if (tz1 < tz0) {
  double temp = tz0;
  tz0 = tz1;
  tz1 = temp;
}

```

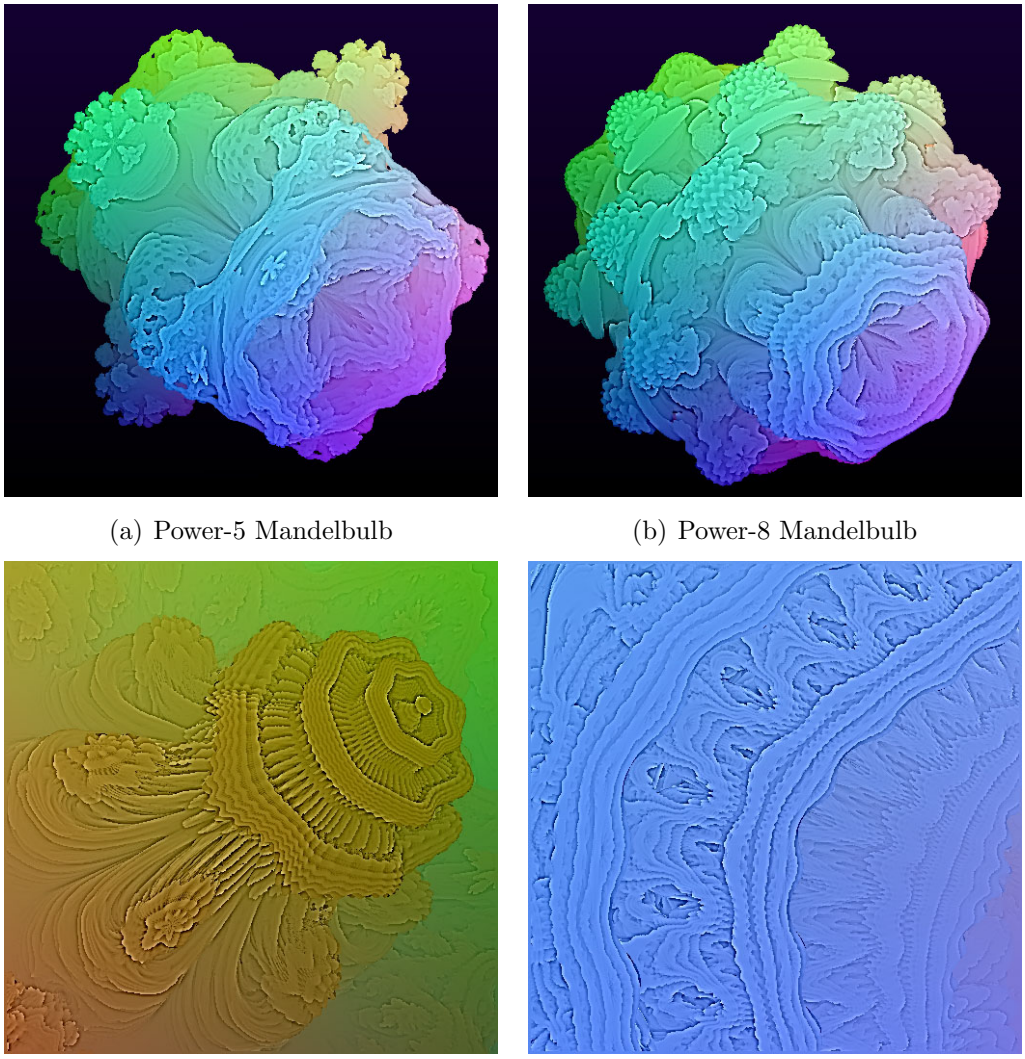
```
35 // tmin = Math.max(tx0, Math.max(ty0, tz0)); tmax = Math.min(tx1, Math.min(ty1,
       tz1));
36 double tmin = tx0 > ty0 ? tx0 > tz0 ? tx0 : tz0 : ty0 > tz0 ? ty0 : tz0;
37 double tmax = tx1 < ty1 ? tx1 < tz1 ? tx1 : tz1 : ty1 < tz1 ? ty1 : tz1;
38
39 // make sure tmin is positive
40 tmin = tmin < 0 ? 0 : tmin;
41
42 // if ray misses bounding cube stop and return black.
43 if (tmin > tmax) {
44   imageColors[index] = ColorUtils.getBackgroundColor((index + 0d) / imageColors.
         length);
45   imageDepth[index] = Double.MAX_VALUE;
46   return;
47 }
48
49 // ------------------------------------------------------------------------
50 // We now know the ray intersects with the bounding cube of the skip node
51 // ------------------------------------------------------------------------
52
53 // create vector for point in space where ray intersects the octree
54 Vector3d P = new Vector3d();
55
56 // will loop until a color is returned
57 while (true) {
58
59 // calculate point where ray hits using current tmin value (adjust slightly to
       take floating point calculations into account)
60 P.scaleAdd(0.0001d * boxDim + tmin, pVec, cPos);
61
62   // while node is not a leaf, descend hierarchy until leaf reached
63   while (!node.isLeaf()) {
64
65     // check if voxel is small enough to terminate hierarcy
66     if (boxDim < (tmin * voxelSizeConstant)) {
67       imageColors[index] = node.getColor();
68       imageDepth[index] = tmin;
69       return;
70     }
71
72     // descend hierarchy
73     boxDim /= 2d;
74
75     // use step function to adjust boxMin ((boxMin.x + boxDim) is boxMid)
76     boolean[] s = { (P.x >= (boxMin.x + boxDim)) ? true : false, (P.y >= (boxMin
           .y + boxDim)) ? true : false,
77         (P.z >= (boxMin.z + boxDim)) ? true : false };
78     boxMin.x += s[0] ? boxDim : 0;
79     boxMin.y += s[1] ? boxDim : 0;
80     boxMin.z += s[2] ? boxDim : 0;
81     node = node.getChild(s[0] ? 1 : 0, s[1] ? 1 : 0, s[2] ? 1 : 0);
82
83   }
84
85   // mark node and its bricks as having been visited by a ray
86   node.visit();
87
88   // a leaf node has now been reached; if node is a non-empty leaf, return its
         color
```

```
89  if (!node.isEmpty()) {
90    if (boxDim > (tmin * voxelSizeConstant) && !node.isQueuedSubdiv())
91      subdivider.queueNode(node, boxMin, boxDim);
92
93    imageColors[index] = node.getColor();
94    imageDepth[index] = tmin;
95    return;
96  }
97
98  // otherwise node is empty so we must set ray tmin to tmax of current node, so
        we need to calculate tmax again calculate t values for each corner
99  tx0 = (boxMin.x - cPos.x) / pVec.x;
100 tx1 = ((boxMin.x + boxDim) - cPos.x) / pVec.x;
101 ty0 = (boxMin.y - cPos.y) / pVec.y;
102 ty1 = ((boxMin.y + boxDim) - cPos.y) / pVec.y;
103 tz0 = (boxMin.z - cPos.z) / pVec.z;
104 tz1 = ((boxMin.z + boxDim) - cPos.z) / pVec.z;
105
106 // ensure t0 and t1 are in the correct order
107 if (tx1 < tx0) {
108   double temp = tx0;
109   tx0 = tx1;
110   tx1 = temp;
111 }
112 if (ty1 < ty0) {
113   double temp = ty0;
114   ty0 = ty1;
115   ty1 = temp;
116 }
117 if (tz1 < tz0) {
118   double temp = tz0;
119   tz0 = tz1;
120   tz1 = temp;
121 }
122
123 // find tmax
124 tmax = tx1 < ty1 ? tx1 < tz1 ? tx1 : tz1 : ty1 < tz1 ? ty1 : tz1;
125
126 // find neighbor node if one exists and adjust boxMin position
127 OctreeNode neighbor = null;
128 int neighborId = 0;
129 if (tmax == tx1) {
130   neighborId = (ray.getDir().x > 0 ? 1 : 0);
131   boxMin.x += ray.getDir().x > 0 ? boxDim : -boxDim;
132 } else if (tmax == ty1) {
133   neighborId = (ray.getDir().y > 0 ? 3 : 2);
134   boxMin.y += ray.getDir().y > 0 ? boxDim : -boxDim;
135 } else if (tmax == tz1) {
136   neighborId = (ray.getDir().z > 0 ? 5 : 4);
137   boxMin.z += ray.getDir().z > 0 ? boxDim : -boxDim;
138 }
139 neighbor = node.getNeighbor(neighborId);
140
141 // if no neighbor node found return black
142 if (neighbor == null) {
143   imageColors[index] = ColorUtils.getBackgroundColor((index + 0d) /
        imageColors.length);
144   imageDepth[index] = Double.MAX_VALUE;
145   return;
```

```
146   }
147
148   // if coarser neighbor node found, snap boxMin onto coarser grid
149   if (neighbor.getDepth() != node.getDepth()) {
150     // save old position of box for setting neighbor once found
151     double oldBoxMinX = boxMin.x;
152     double oldBoxMinY = boxMin.y;
153     double oldBoxMinZ = boxMin.z;
154
155     boxDim = Math.pow(2d, -neighbor.getDepth() + 1d);
156     boxMin.x = Math.floor(boxMin.x / boxDim) * boxDim;
157     boxMin.y = Math.floor(boxMin.y / boxDim) * boxDim;
158     boxMin.z = Math.floor(boxMin.z / boxDim) * boxDim;
159
160     //optimize sub-optimal neighbor pointer
161     if (!neighbor.isLeaf()) {
162       node.setNeighbor(neighborId, neighbor.getChild(oldBoxMinX >= (boxMin.x +
              boxDim / 2) ? 1 : 0,
163           oldBoxMinY >= (boxMin.y + boxDim / 2) ? 1 : 0, oldBoxMinZ >= (boxMin.z
                + boxDim / 2) ? 1 : 0));
164     }
165   }
166
167   // finally set node to neighbor found and tmin to new position along ray
168   node = neighbor;
169   tmin = tmax;
170 }
```

Appendix C

3D Fractal Range



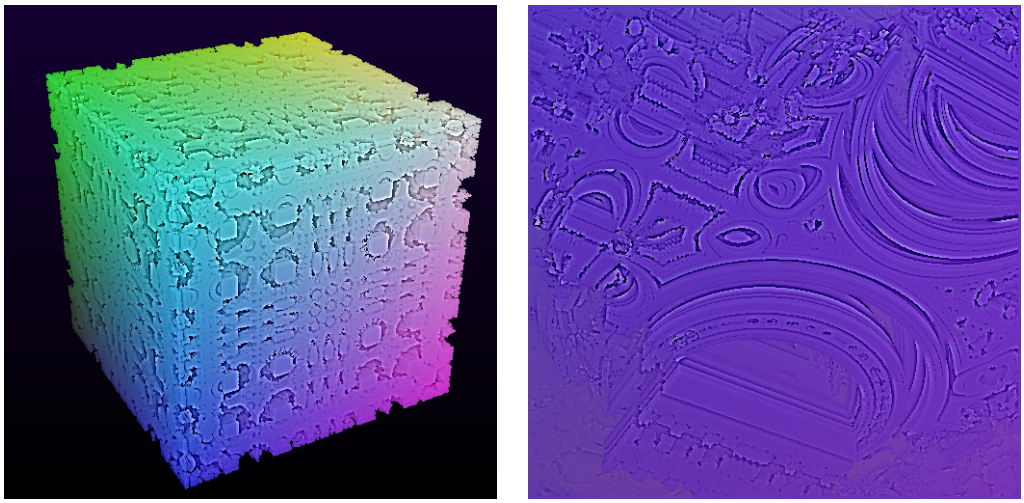
(a) Power-5 Mandelbulb

(b) Power-8 Mandelbulb

(c) Mandelbulb Stem Detail

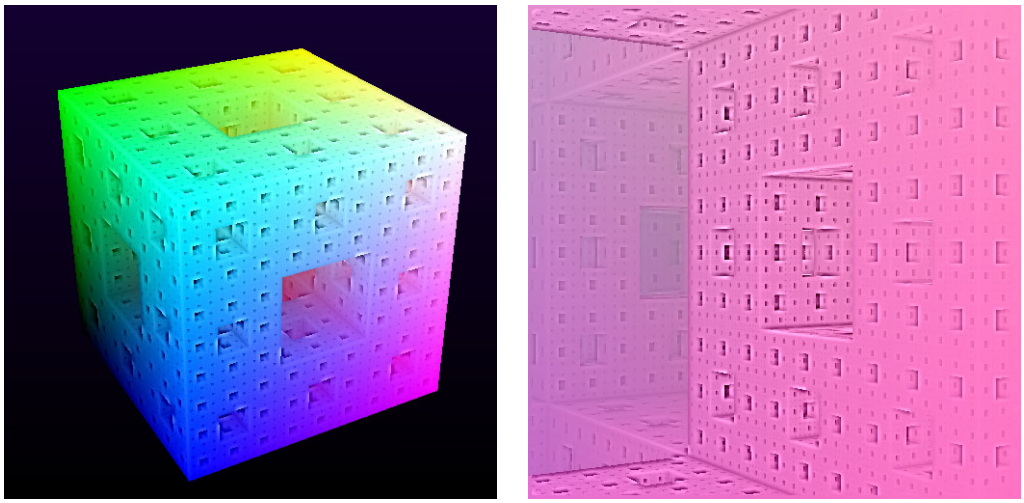
(d) Mandelbulb Rib Detail

Figure C.1: Different versions of the Mandelbrot-esque Mandelbulb Escape-Time Fractal.



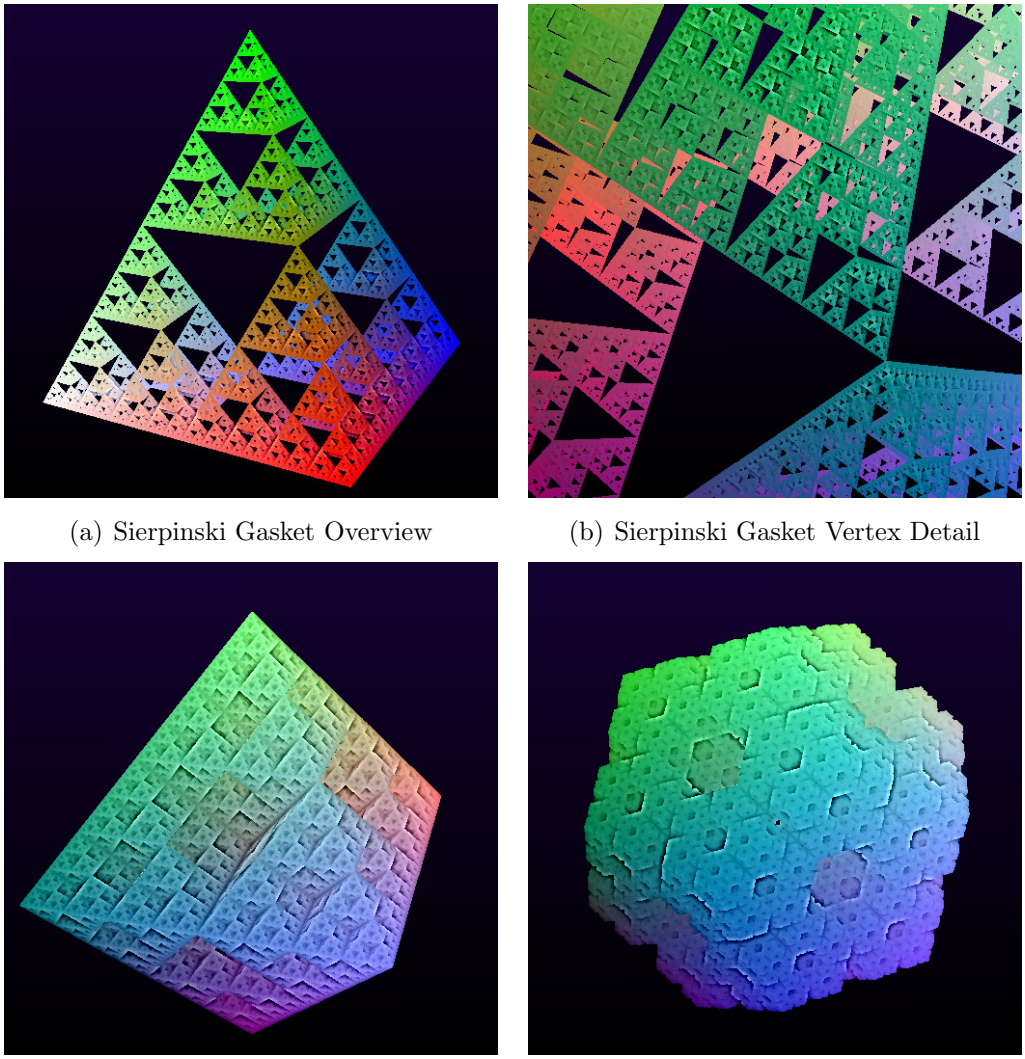
(a) Mandelbox Overview (b) Mandelbox Ring Detail

Figure C.2: Mandelbrot-esque Mandelbox Escape-Time Fractal.



(a) Menger Sponge Overview (b) Menger Sponge Internal Detail

Figure C.3: Kaleidoscope IFS Menger Sponge Fractal.



(a) Sierpinski Gasket Overview

(b) Sierpinski Gasket Vertex Detail

(c) Sierpinski-esque Octagon

(d) Sierpinski-esque Dodecahedron

Figure C.4: Kaleidoscope IFS Sierpinski-esque Polyhedra.

Appendix D

Automated Exploration Screenshots

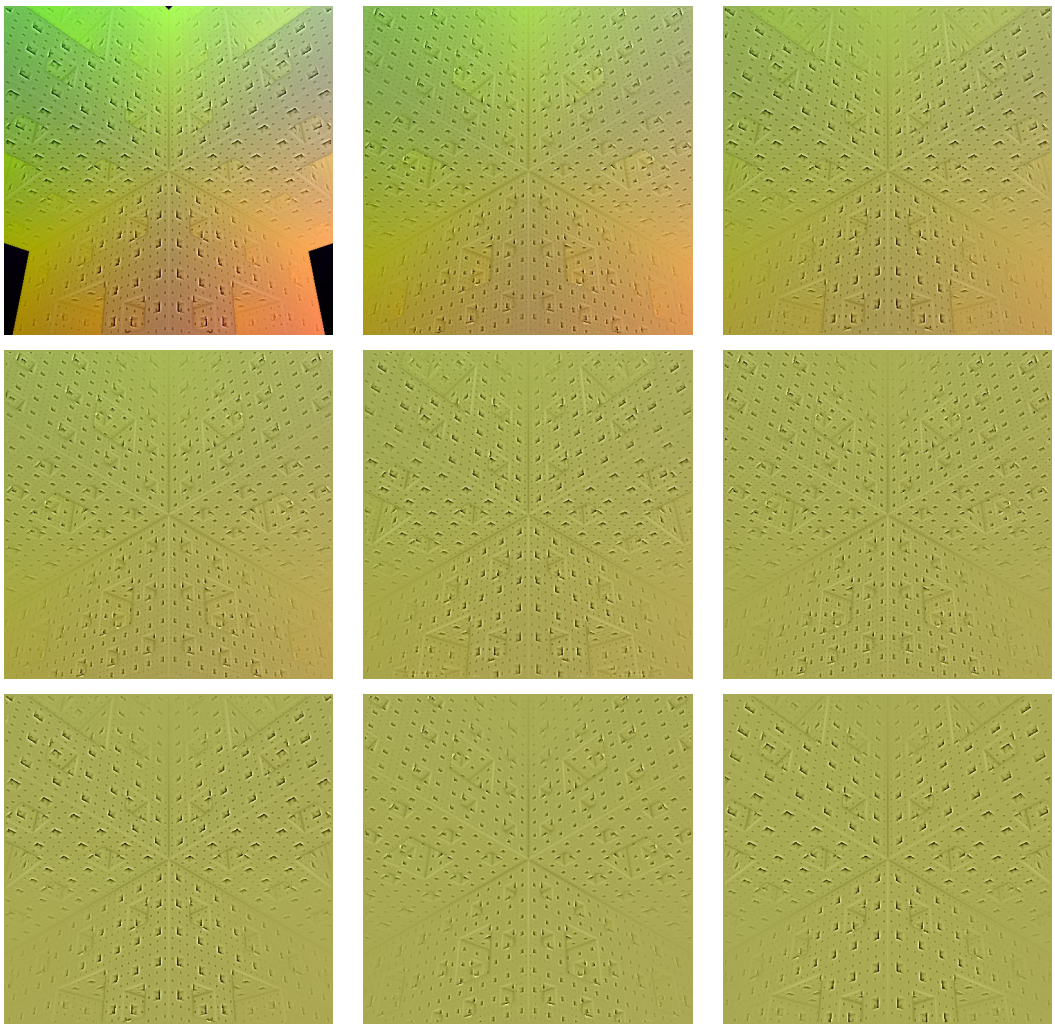


Figure D.1: Screenshots of Menger Sponge point of recursion zoom exploration. Presented fractal detail remains relatively identical as scale varies as a result of strict-self-similarity.

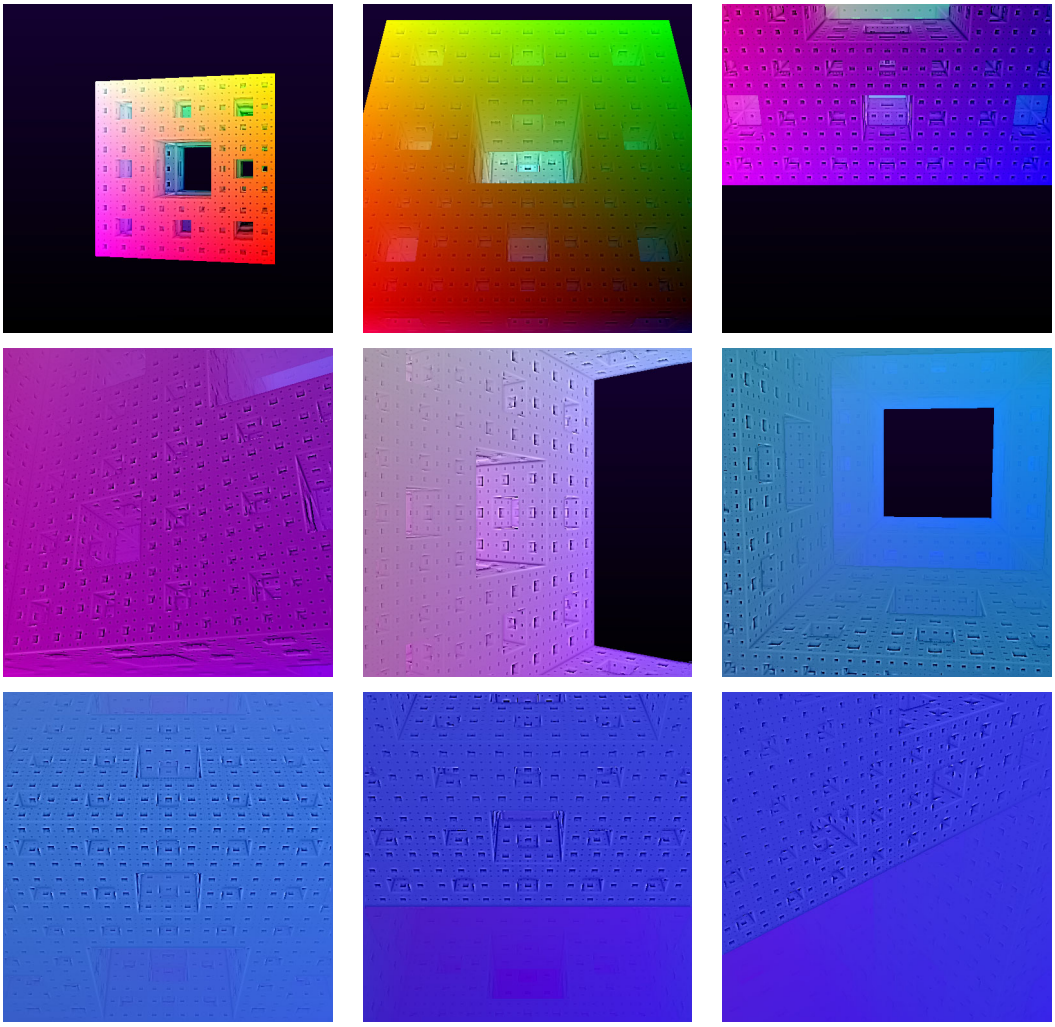


Figure D.2: Screenshots of Menger Sponge tour exploration navigating the camera inside the fractal at various levels of recursion. Video available online: www.youtu.be/BU5UXhPGvU4

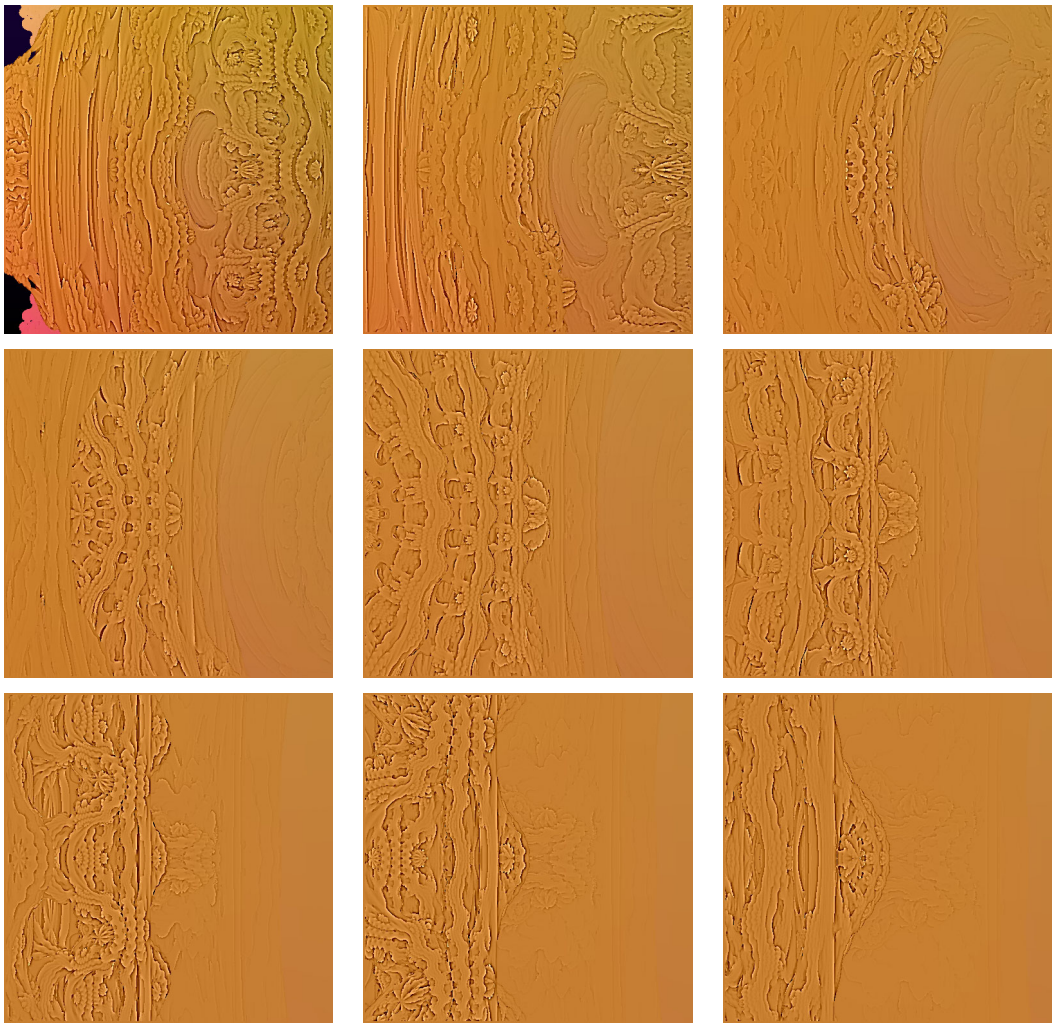


Figure D.3: Screenshots of Mandelbulb zoom exploration. Detail varies at all scales as a result of quasi-self-similarity. During exploration the fractal section in the left side of the image gets relatively further away from the foreground fractal section, requiring more neighbour traversals to reach.

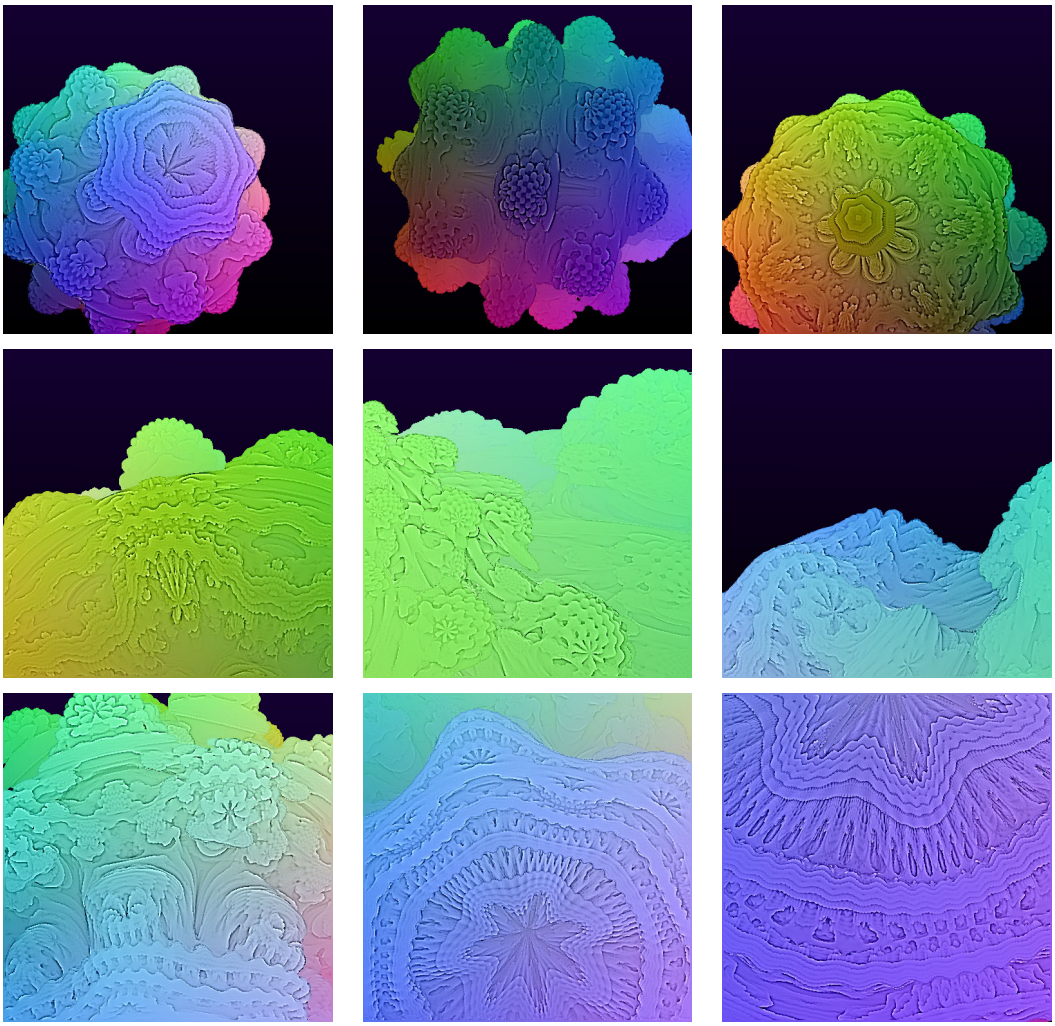


Figure D.4: Screenshots of Mandelbulb tour exploration showcasing external surface bulb detail. Video available online: www.youtu.be/oFAiiiJTo50

Appendix E

Original Project Proposal

Introduction and Description of the Work

Fractals are geometric shapes that exhibit the property of self-similarity, meaning that it is possible to repeatedly deconstruct them into smaller shapes "each of which is (at least approximately) a reduced-size copy of the whole."[1] This means that by zooming into them, they can be explored indefinitely to reveal their intricate and complex details at all levels of magnification.

Two common techniques for generating fractals are:

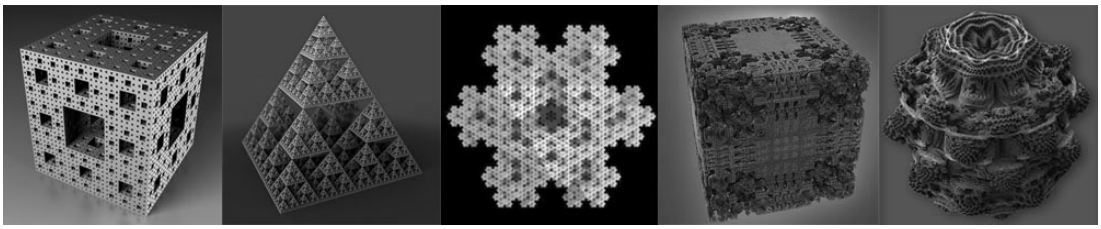
- *Iterated function systems*: These produce a fractal that is the result of the union of several copies of itself, with each copy transformed by a function. The Sierpinsky carpet is a well known 2D example; its analogue in three dimensions is the Menger Sponge.
- *Escape-time systems*: These produce a fractal based on a formula or recursive relation iterated over every point in space. The most famous example of this is the Mandelbrot set which has the Mandelbulb as its current closest 3D analogue.

The formulae and functions involved in the generation of these types of fractals depend on parameters which, when changed, can lead to different looking shapes. An example of this is altering the power m in the Mandelbrot generating function of zn+1 = znm + c leading to fractals with m-1 degrees of rotational symmetry.

[1] Mandelbrot, B.B. (1982). The Fractal Geometry of Nature

This project aims to develop a system to allow users to explore 3D fractals and their infinitely complex detail. They will be able to describe a fractal by choosing a fractal archetype from a library and specifying relevant parameters involved in its construction. The system will then generate geometric data on the fly as the user navigates around and into it in real time.

The fractal archetypes available to the user will be: Menger Sponge, Sierpinski Pyramid, 3D Koch Curve, Mandelbox, Mandelbulb (as seen below).



Resources Required

This project will rely on software libraries to allow the use of GPU processors and facilitate the large amounts of vector arithmetic necessary for ray casting. A machine with a graphics card and multiple processors will be necessary for development; my own desktop PC should be sufficient however.

Starting Point

As well as the ability to program in Java, this project's development will depend on themes in the following courses:

1. **Computer Graphics and Image Processing** introduced the fundamentals of rendering 3D objects in 2D, including ray-casting.
2. **Concurrent and Distributed Systems** will be necessary to achieve the highest possible real-time performance by using threads and writing thread-safe code
3. **Floating-Point Computation** will be used by formulas and functions in fractal data generation and tracking of rays during rendering

Substance and Structure of the Project

The key concept of the project is to be able to navigate and indefinitely zoom into a 3D fractal in real time, so a data-structure must be chosen to facilitate the dynamic geometry generation and real-time rendering.

Voxels will be used to represent the geometric data as they are ideal for representing rough or fragmented objects such as fractals with their often high frequency displaced detail resulting from recursive generation (which also makes rendering them with polygons unsuitable). A data structure called a sparse voxel octree will be used to store the voxels which minimizes its memory footprint and functions as an acceleration structure supporting efficient ray-cast rendering.

In order to ensure that the user is experiencing resolution-bound detail of the 3D fractal (rather than seeing individual voxel cubes), this data structure will have to be modified into a scale-adaptive sparse voxel octree. The system will have to keep track of the current level of detail stored in the sparse voxel octree and procedurally adding more complex detail in the area of the fractal the user is currently viewing by further subdividing octree leaves and intelligently caching geometric data. In order to keep performance high we can employ the dynamic computation of fractal depth subject to frame-rate; by which low powered machines can still view fractal detail at arbitrary scale, but with higher latency between zooming in and detail being calculated.

This volumetric data will then be rendered as efficiently as possible using ray-casting through the sparse voxel octree. Since each ray is independent from the others all rays can be calculated in parallel. Because of this highly parallel nature of the rendering algorithm, implementations involving multi core systems will be explored to improve rendering speed significantly. Ambient occlusion should be present as an optional shading method dependent on performance to allow the fractal to display a better approximation of a real life 3D object.

The viewing camera will be navigated around the fractal in one of two modes: either *Interactive* or *Cinematic*. During Interactive mode the user can rotate the camera either freely or around the fractal, move this camera in three dimensions, move it towards or away from the fractal and increase or decrease its field of view (zooming in and out). This control system will be designed to be as intuitive as possible using either mouse or keyboard or both. During Cinematic mode the user will choose a preset flight-path from a list and the camera will be navigated through and into their chosen fractal to highlight regions of interest for the casual observer.

A simple graphical user interface will be developed to allow the user to choose a fractal archetype from a list and modify parameters specifying its generation.

An evaluation of the system will be performed by measuring performance and accuracy over a range of different fractals and viewing conditions. Rendering frame-rates at different magnification levels and resolutions will be recorded over a number of machines to determine the system's efficiency and assess to what extent rendering time is independent of scale. An error metric will be defined to measure how approximate a given level of detail is at a certain distance from the camera to determine computational accuracy of different levels of fractal detail. It will also be possible to measure the caching efficiency of the scale-adaptive sparse voxel octree by keeping track of the number of times the octree has to re-calculate a value or discard a value; giving an indication of success or failure of the smart caching algorithm.

The machines tested on should have a range of different CPUs and GPUs to examine what difference the number of cores makes on performance. These can then be compared to current 3D fractal rendering solutions in terms of frame-rate, resolution and magnification.

Success Criteria

The following should be achieved for the project to be deemed a success:

1. It must be possible to generate volumetric data described by 3D fractals and store this information in a scale-adaptive sparse voxel octree.
2. Further generation of fractal detail must be computed procedurally and added to the data structure to allow indefinite magnification.
3. The user must be able to specify the fractal they wish to view and then navigate around and into it as the octree is rendered in real-time: with at least 20 frames / second and at at least VGA resolution - 640x480.
4. A detailed examination of system performance under different environments and with different data-sets must be carried out.

Timetable and Milestones

Preliminary Work

Determine which software libraries to use through research and experimentation. Study sparse octree data-structure and its related ray-casting rendering algorithm.

Milestones: Be prepared to start coding.

Weeks 1 & 2 — 04/11/11 (date of deliverables)

Finalize the modular structure of the system to be developed. Study mathematics involved in the construction of fractals. Design unit tests for and construct a standard vector maths engine. Begin work on a renderer using static sparse-octree data.

Milestones: Standard vector maths engine should be complete. By this stage all preparatory work should be complete. Render a basic cube to test the partially developed renderer.

Weeks 3 & 4 — 18/11/11

Continue work on the renderer and implement a simple keyboard event handler and camera system. Rigorously unit test vector maths engine and camera control system. Continue studying construction of fractals.

Milestones: Be able to simply navigate around an object model using preliminary renderer and camera control system.

Weeks 5 & 6 — 02/12/11

Finish work on a preliminary ray-casting renderer and test it. Design unit tests for and begin implementing the fractal maths engine. Measurements will be taken for the evaluation of the renderer without any procedurally generated detail.

Milestones: The renderer should be complete. Be able to pre-compute a simple iterated function system fractal - the Menger Sponge, store it in a sparse voxel octree, and then render it.

Weeks 7 & 8 — 16/12/11

Finish developing and test the fractal maths engine. Begin work designing the scale-adaptive level of detail algorithm to populate the sparse octree procedurally as more detail is required.

Milestones: The fractal maths engine should be complete. Be able to pre-compute an escape-time system fractal - the Mandelbulb, store it in a sparse voxel octree, and then render it.

Weeks 9 & 10 — 30/12/11

Complete developing and testing the scale-adaptive level of detail algorithm. Measurements will be taken for the evaluation.

Milestones: A completed scale-adaptive sparse voxel octree. The user should be able to simply navigate around and into a 3D fractal at arbitrary magnification with basic camera controls of their choosing in close to real-time.

Weeks 11 & 12 — 13/01/12

Optimization of renderer, fractal maths engine and scale-adaptive algorithm to achieve better performance. Begin work designing the graphical user interface and improving the camera control system by further developing the mouse and keyboard event handler.

Weeks 13 & 14 — 27/01/12

Finish implementing the more comprehensive graphical user interface and improved interactive camera controls. Implement and test the cinematic camera system.

Milestones: The user can now choose a fractal archetype, specify parameter values and then navigate around that chosen fractal in either interactive or cinematic mode. This should be the final version of the 3D fractal explorer.

Weeks 15 to 20 — 02/03/12

Evaluation and comparison of the system's performance and complete the dissertation. If time allows, extensions will be made to the system.

Milestones: A finished dissertation