Developing a 3D Graphics Engine to Optimise Collision Detection using Fractal Geometry.

Dissertation

Name: Oliver Green, ID: 20427972, Supervisor: Dr Michael Pound

Abstract

The project is to design a 3D graphics engine combining efficient fractal generation, rendering and storage with a fractal-based collision detection and response algorithm. The engine must accept varying meshes of different mesh counts and involves utilising object file formats, producing renders from different projections. Using the recursive nature of fractals, it aims to exponentially reduce collision computation cost without diminishing accuracy. Fractal generation is optimised with GPU instancing, enabling instant creation of intricate, scalable geometries useful for gaming, simulations and prototyping.

The project provides huge application for efficient terrain systems which could be constructed using fractals, rendered in a fraction of the time, require minimal storage and come instantly pre-packaged with an exponentially efficient but still fully accurate collider, all whilst ensuring there is control over the modelling process.

Contents

1. Background and Motivation

Collision detection is computationally expensive, especially as assets grow in complexity and their geometry increases in mesh count. Real-time collisions are demanding of CPU and GPU resources, which can lower frame rates and responsiveness [1]. However, current applications of 3D graphics, including computer games, physics-based animations, engineering simulations and virtual prototyping, require real-time collision handling.

Intense competitive games need high frame rates to ensure fairness amongst players and it is these games that tend to dominate the market in terms of success. Fortnite, with an average of 221 million actively monthly users and a revenue of $26 billion generated in the span of 5 years [2], still needs to maintain high resolution, large worlds (likely of high mesh count, costly to render and collide with geometry) to attract and maintain users in a competing market space. Although, they cannot expect users to harness the most powerful hardware. Their games must be accessible to all. This is why many game publishers focus on supporting cross-platform releases, enabling broader accessibility and maximizing market reach.

Collision detection is costly but other factors cannot simply be lowered due to their necessity, i.e. real-time responsivity, frame rate, detailed geometry. The core solution is instead finding a balance of speed and accuracy and many methods have been researched to achieve this, driven by the increasing revenue of the industries. Some include: bounding volumes, that is wrapping complex models in simpler shapes, such as bounding boxes [3][4]; using multiple levels of simplified collision meshes based on the camera's distance to reduce computation for distant or partially obscured objects and dividing a scene into hierarchical sections to reduce the number of collision checks needed by focusing only on nearby objects [1]. I believe it would be an interesting dissertation to compare these methods to an approach I would design, making use of fractal geometry.

Fractals can be a simple mesh, which after a number of transformations and recursive iterations become complex structures holding high mesh-counts. Though, the information that these geometries are recursive could allow for a backtracking algorithm that takes advantage of this property to decrease the number of costly calculations required, exponentially without diminishing accuracy. 'Fractal geometry made it possible for the first time to measure cloud formations, the contours of coastlines, the clustering of galaxies, even the folds of mammalian brains, in a rigorously quantitative fashion' [5] and should it be implemented as part of a 3D graphics engine and succeed in reducing the collision cost of objects exponentially it could also be transformative in removing a significant issue in the industry.

The project will aim to generate fractals instantly, rendered for optimal performance using GPU instancing. This provides developers with a powerful tool to create, prototype and refine intricately detailed models without the need for labour-intensive manual work while maintaining control over the generation process. Unlike many procedural generation algorithms, which rely on randomness [8] or evolutionary methods [9] and often limit fine-tuning of meshes, fractals offer precision allowing developers to design the base mesh and sets of transformations. Their structural similarity to natural terrains ensures realistic and visually compelling results.

Fractal collisions are heavily underexplored (see description of work for reasoning why) but the ability to procedurally generate environments instantly and their complimenting low-cost physics colliders, still whilst retaining high mesh counts and detail without negatively impacting frame rate etc, could be a huge step forward for 3D graphics applications.

The project uses OpenGL, chosen due to its support for high-precision floating-point operations in its shader language, GLSL, making it ideal for generating detailed fractal geometries and performing accurate collision detection. Its integration with compute frameworks like OpenCL enables efficient parallel processing, essential for managing the complexity of recursive fractals and ensuring real-time performance.

2. Related Work

2.1. Fractal Generation

Fractal geometries are defined recursively allowing for instant generation of complex environments based on a set of rules. Developers can create intricately detailed models without the need for labour-intensive manual work, significantly reducing production time and costs. Their inherent scalability is ideal for crafting massive open-world games, enabling the creation of vast, detailed landscapes with minimal effort where environments remain detailed at any resolution. This is already being seen in games like No Man's Sky which have been praised for their procedurally generated worlds (where they utilise fractal generation). The game generated $700 million in revenue [7], driven by its standout feature being procedural generation on an unheard-of scale, creating an entire universe with billions of planets to explore!

Much procedural generation techniques have been explored. [8] introduces Perlin noise, a method for generating coherent, continuous random patterns, commonly used for terrain generation. It's a real strong approach, producing smooth gradients that resemble natural phenomena like clouds, waves, or landscapes. Generation can also be achieved through evolution. [9] proposes a model where content is evolved over time through a fitness function, refining meshes for specific gameplay outcomes, such as fun, difficulty, or narrative engagement through genetic algorithms.

Though, advancements in GPU instancing make fractal geometry an optimal performance solution. Modern GPUs and graphics APIs offer greater flexibility, enabling the instancing of not just static geometry, but also dynamic objects with varying attributes, such as texture, colour and shaders and improvements in GPU memory and bandwidth have made it possible to efficiently store and render large numbers of instances. We can leverage this for fractal generation whose geometry is at its core, instances of a base mesh that simply undergo transformations and folds.

Fractals add control. Unlike random noise or an evolution model which requires data to refine meshes, we take a hybrid approach, designing simple base meshes of our choosing and have control when building instructions for how these are transformed. We can use fragment shaders to our advantage to implement recursive fractal algorithms to generate patterns and structures efficiently. Advances in shader programming languages, such as GLSL and HLSL, have further optimized fractal rendering for real-time applications [10]

2.2. Fractal Collision Detection and Response

Determining collision and providing response often as a vector normal to a surface of collision is computationally costly so physics engines often make a lot of approximations balancing accuracy with speed. Accuracy can be prioritised by implementing bounding volume hierarchies: encapsulating objects into simpler meshes [3][4]. Collision detection is first performed on the simple geometries to quickly disregard much of the object's mesh before computing precise collision with the intricate object. Though the design doesn't eliminate costly collisions all together, just reduces them and the computational cost to update the BVH structure in dynamic environments, i.e. to build the simplified geometries is high.

[11] proposed reducing the problem to one-dimensional range tests via a sweep and prune algorithm: sort objects along one or more axes and check for potential collisions based on sorted order. Here, we explore an approach suited to dynamic environments but almost hopes objects lie along specific axes as provides limited benefit in the likely scenario they are scattered across 3D space.

Spatial hashing brings the focus to speed in real time execution, mapping objects into a hash table based on their spatial location [12]. By dividing space, interactions are only checked for objects within the same or neighbouring cells. The method is effective for sparse objects separated in space but when objects are densely packed, it delivers no reward. Also hashing, reduces accuracy significantly.

Tending to the prioritisation of speed and lack of accuracy fails to produce meaningful collision results – it isn't really producing collision on the original objects. Taking this to the extreme with sphere-sphere approximation, we've lost all detail of the original mesh and need to assess our approach, asking what value are we truly providing here? By comparison, prioritising accuracy over speed is where bugs can develop in game engines. This is real time physics we're aiming for where fast-moving objects will slip through one other if collisions aren't detected in speed, breaking immersion or gameplay. Ideally, we want both.

The sole benefit of using fractal geometry is that its recursive nature allows for backtracking of collisions through its instantiated mesh, providing an approach of exponential speed (mathematically, in terms of time complexity, as is recursive, not hyperbole), while maintaining full accuracy as collisions with a fractal's base mesh are computationally inexpensive.

3. Description of Work

The project is to design a 3D graphics engine combining efficient fractal generation, rendering and storage with a fractal-based collision detection and response algorithm. See abstract for elaboration.

A challenge to consider is that most real-world objects, especially that of man-made items, lack the recursive, self-similar properties that define fractals, i.e. a car cannot be made up of cars, etc. This is what I believe to be the greatest barrier to fractal collisions and reason for it being under-researched. It shouldn't be expected that the collision detection algorithm I produce will be able to be widely applied to all meshes as requires using fractal geometry. Instead, I will explore the geometry it can work with and ensure to build a graphics engine that holds multiple objects simultaneously so that this geometry can be integrated with regular meshes, i.e. a car can sit on fractal terrain.

Although there are wide uses for this geometry. Benoit Mandelbrot, recognized for his contribution to the field of fractal geometry, developed a theory of "roughness and self-similarity" in nature as well as labelling his interest as "the art of roughness" [6]. Fractal geometry is effective for representing complex structures that appear in nature such as clouds, mountains, cliffs and trees, with branching algorithms existing online to produce trees. Most examples of fractals have a sci-fi geometry which could be perfect for producing alien planets or galaxies at an instant although they can also produce detailed and varied terrain. Just think of the detail in a snowflake or seashell – naturally occurring fractals.

3.1. Aims and Objectives

The project aims to (1) develop a 3D graphics engine of substantial complexity using the GPU and (2) leverage it to instantly generate fractal geometry and (3) solve a frequent challenge faced in industry: optimising collision detection by making use of the recursive property of fractals.

1) The graphics engine should:

1. Support loading models from the widely used obj file format, including displaying textures so that my engine has full integration with the models designed on others and vice versa. This will become necessary for when I test and compare various meshes against the collision detection algorithms.

2. Produce 3D renders that can be viewed from any position and rotation under both orthographic and perspective projection.

3. Render models correctly using rasterization so that each pixel accurately displays the closest visible face.

4. Include a lighting system where a light source changes the colour values of each vertex.

5. Render a full scene with a camera and multiple objects that can be loaded in and rendered simultaneously. Information on the scene such as camera position, rotation, fov, projection, aspect ratio and all the obj file paths would be stored in a save file my program will read / write to.

6. Provide efficient collision detection for regular 3D models and a simple player to be implemented for testing. I will heavily research existing methods to do this.

2, 3) From there, I will:

7. Procedurally generate fractal geometry, taking in a base case low-mesh 3D model and the fractal pattern / instructions which will inductively build up the structure.

8. Design and implement an optimised collision detection algorithm using fractal geometry. This must be mathematically significantly more efficient (i.e. exponentially quicker).

9. Validate my code, spending a substantial amount of time running tests on my implementation with varying 3D models of different mesh counts (hence the need to use the obj file format or any standard file format), testing how my solution compares to methods on regular geometry, I implement in objective 6.

10. Provide a good UI.

4. Methodology, Design and Implementation

Sections 4.1 and 4.2 break down the class diagrams shown in Appendix C and D. Please refer to these whilst reading.

4.1. Graphics Engine

In this section, we discuss the initial implementation of a custom graphics engine, where we program the maths involved for rendering fractals.

4.1.1. obj files

Aside from simple meshes such as cubes or pyramids, realistically any sort of complexity in models requires the loading of object files. We require storage of mass vertices and face indexes and can set ourselves up for success early on by supporting the widely used obj file format. Already we're providing full integration with models designed in other graphics engines and looking ahead to the scope of the project, it will become necessary for when we test and compare various meshes of a wide range of mesh counts against our collision detection algorithm. It would be great if we could compare a high mesh count, detailed, object to an equally detailed fractal with high mesh but derived from a low mesh base object and loading a file into the GPU is a first step in doing so.

```
# Monitor
# Author: Oliver Green
# Mesh count: 9670

v 24.812761 49.976017 19.514240
v 24.812761 49.976017 19.100040
…

f 134 135 132
f 95 134 132
…
```

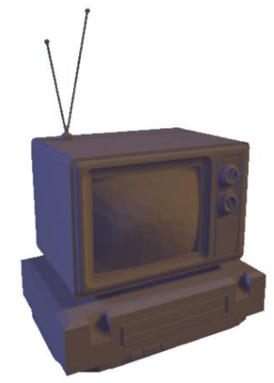


Figure 1. A sample 3D model created in Blender, stored as an obj file, consisting of comments, vertices, and faces, with vertices referenced by hidden indexes starting from 1 and incrementing sequentially which faces refer to. Let's denote our file's mesh count at the top of our obj file as standard practice by constructing a simple script to do so. Now we can effectively determine object's mesh counts when it comes to later testing the efficiency of our collision detection algorithms.

We can extract the data from figure 1 as an array of vertices and an array of face indexes by constructing a simple parser for the context-free grammar we define as:

```
<obj> := <decl>\n<obj> | <decl>

<decl> := <vertex_decl>
       | <face_decl>
       | # <comment> | \n

<vertex_decl> := v <float> <float> <float>

<face_decl> := f <int> <int> <int>
```

We build a scene object containing a camera and a list of objects (who's attributes are these arrays of vertices, faces). Objects are initialised with a file name where there is a dependency within its constructor with the object parser.

Our first obstacle is that the data within these need to be bound to some buffers in our GPU, namely a vertex buffer object (VBO) and an entity buffer object (EBO) where our VBO is a single 1d array of floats – we cannot simply store some vertex type. So, the data we extract must be a float array which lists the components of all vertices one after the other. E.g. "v 1 2 3 \n v 4 5 6" is stored as {1, 2, 3, 4, 5, 6}.

Where do vertices start? At index 3 * i for each i in the number of vertices. Why is this relevant? Vertices are processed in parallel across the GPU's cores so need some way of switching between different sets of vertex data. Therefore, we need to specify how the vertex data is laid out in memory and how it should be interpreted for rendering using a vertex attribute pointer.

4.1.2. Methods of rotation

In preparation for rendering, we design some maths libraries to be used for methods of rotation. These aim to give us more flexibility and control than OpenGL libraries and portability too, i.e. we're not tied to OpenGL's specific implementations.

In a project where efficiency is the end goal (optimising collisions) we want no unnecessary function calls. OpenGL's built-in functions may have overhead from redundant state changes or generalised operations that we don't need for our specific purpose. By creating our own libraries, we can skip unnecessary processing, streamline calculations and improve performance by tailoring operations to our needs.

And that is exactly what we want! Libraries that can fit our bespoke purpose of generating fractals and their colliders. We can't extend OpenGL's matrix class but we can extend one we make, to allow for custom methods and the rotational matrices. A key to rendering objects and performing rotational transformations on a fractal's instances of its base object is converting an Euler rotation to a matrix of local axes, something that we can code into our libraries and then use freely throughout the project, all the while gaining knowledge of the underlying maths which goes into a graphics engine. Really, this understanding is essential as when it comes to calculating fractal collisions, we can't rely on libraries as our purpose becomes too specific.

Introduce vectors with three components: x, y, z. To effectively use them we'll want to add operator overlays. E.g.

$$\underline{v1} + \underline{v2} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \end{pmatrix}$$

When applying vector transformations, we now don't need to concern ourselves with the intricate components of the vectors but trust our previously defined operations to do so, hence we build an abstract interface. It is the prioritisation of abstraction which is fundamental for these libraries. They should ideally be pure (able to be translated to a functional programming language; coded 1:1 in line with handwritten maths), logically defined using lambda expressions and lambda calculus to improve readability and ensure full atomicity and reusability to the extent that nowhere is functionality repeated. Also, to better interface them, we declare frequently used vectors as static variables. Zero is common when referring to an origin and one is used as a default scaling transformation, common in my fractal generator. By referring to these we produce more descriptive, shorter code that is readable.

$$\underline{zero} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \quad \underline{one} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad \underline{i} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \underline{j} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \underline{k} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Critical to matrices are the dot and cross product. For ease, they can be defined here along with many other properties of vectors.

$$dot\left(\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix}\right) = (x_1 \cdot x_2) + (y_1 \cdot y_2) + (z_1 \cdot z_2)$$

$$cross\left(\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix}\right) = \begin{pmatrix} (y_1 \cdot z_2) - (z_1 \cdot y_2) \\ (z_1 \cdot x_2) - (x_1 \cdot z_2) \\ (x_1 \cdot y_2) - (y_1 \cdot x_2) \end{pmatrix}$$

Our techniques above for abstraction really come into play when we introduce matrices. These have three vector components v1, v2, v3 and the multiplication property:

$$(\underline{v1} \quad \underline{v2} \quad \underline{v3}) \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \alpha\underline{v1} + \beta\underline{v2} + \gamma\underline{v3}.$$

We define the frequently used identity matrix as $I = (\underline{i} \quad \underline{j} \quad \underline{k})$ and the operator overloads for matrix multiplication:

$$apply\ matrix\ to\ vector: m \cdot \underline{v} = (\underline{m^T{}_{v1} \cdot \underline{v}} \quad \underline{m^T{}_{v2} \cdot \underline{v}} \quad \underline{m^T{}_{v3} \cdot \underline{v}})$$

$$apply\ matrix\ to\ matrix: m1 \cdot m2 = (m1^T \cdot \underline{m2_{v1}} \quad m1^T \cdot \underline{m2_{v2}} \quad m1^T \cdot \underline{m2_{v3}})$$

$$where\ transpose: m^T = \begin{pmatrix} v1_x & v2_x & v3_x \\ v1_y & v2_y & v3_y \\ v1_z & v2_z & v3_z \end{pmatrix}^T = \begin{pmatrix} v1_x & v1_y & v1_z \\ v2_x & v2_y & v2_z \\ v3_x & v3_y & v3_z \end{pmatrix}$$

And that's a large portion of maths taken care of in just these few lines as we use our vector overloads, we reuse transpose throughout and we use the first operator overload in the second. Even more goes into a matrix's inverse function but by using lambda calculus, conditional expression and reusing the cross product, we result in a simplified expression:

$$inverse: m^{-1} = \left(\lambda det \to \left(det == 0\ ?\ throw\ error : \begin{pmatrix} cross(\underline{v2}, \underline{v3}) \\ cross(\underline{v3}, \underline{v1}) \\ cross(\underline{v1}, \underline{v2}) \end{pmatrix} / det\right)\right) |m|$$

$$where\ determinant: |m| = \left|\begin{pmatrix} \underline{v1} \\ \underline{v2} \\ \underline{v3} \end{pmatrix}\right| = dot\left(\underline{v1}, cross(\underline{v2}, \underline{v3})\right)$$

The code for this is just as short. Note how there is a 1:1 mapping between the code below and the line of maths above due to the lambda calculus. The project is large and complex but by coding imperatively and breaking it down into classes makes it manageable.

```
public Matrix inverse() =>
        ((Func<float, Matrix>)
        (det => det == 0 ?
                throw new InvalidOperationException("Matrix has no inverse.") :
                new Matrix(Vector.cross(v2, v3),
                        Vector.cross(v3, v1),
                        Vector.cross(v1, v2)) / det))
        (determinant());
```

After implementing many useful properties, we extend our matrix to implement the three rotational matrices: roll, pitch and yaw (something OpenGL does not provide in their Matrix3 class) using a lambda calculus like above.

$$pitch(\theta) = \left(\lambda s, c \rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & c & s \\ 0 & -s & c \end{pmatrix}\right)(\sin\theta, \cos\theta)$$

$$yaw(\theta) = \left(\lambda s, c \rightarrow \begin{pmatrix} c & 0 & -s \\ 0 & 1 & 0 \\ s & 0 & c \end{pmatrix}\right)(\sin\theta, \cos\theta)$$

$$roll(\theta) = \left(\lambda s, c \rightarrow \begin{pmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{pmatrix}\right)(\sin\theta, \cos\theta)$$

We extend our vector class to add a local axes method to convert an Euler rotation vector into a matrix of its 3 local axes. Euler rotations are easy to use, needing only 3 inputs versus the 9 local axes use, i.e. no redundancy, hence typically used to store an object's rotation or as an easy way for a user to input a rotation but we require rotation in local axes from for 3D to 2D mapping.

Each of the three Euler angles gives the magnitude of rotation about their corresponding axis, so by applying the rotational matrices in sequence where

- roll is a rotation around the z-axis z degrees anticlockwise
- yaw is a rotation around the y-axis y degrees anticlockwise
- pitch is a rotation around the x-axis x degrees anticlockwise, we construct that:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}.localAxes() = pitch(x) \cdot yaw(y) \cdot roll(z)$$

This proof is simply a geometric interpretation of the linear transformation. The order doesn't matter (despite matrices not being commutative) as long as the order is consistent throughout the program, hence the need for abstraction. By making this the single only line in our program which decides axes orientation, we ensure no errors occur. Though the order will determine the direction of the axis in relation to one another. Figure 2 illustrates the different coordinate systems produced depending on this order.

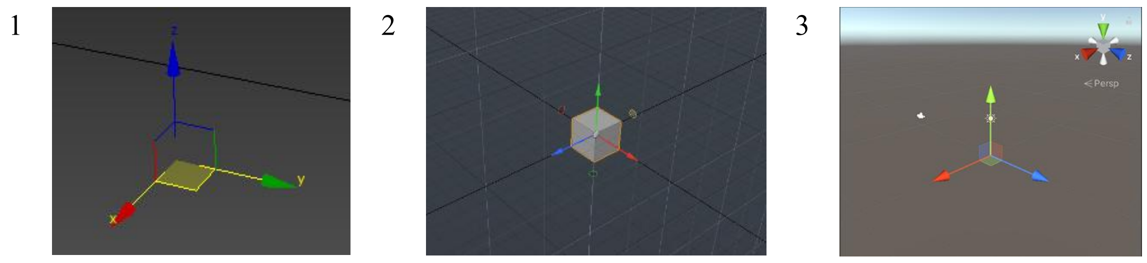


Figure 2. Comparison of coordinate systems in 3D graphics engines. Images 1 and 2 demonstrate Blender's use of a right-hand coordinate system, with differing orientations of axes but identical spatial relationships when normalized, i.e. aligning the z-axis upwards in both images reveals the systems are identical. In contrast, image 3 shows Unity's left-hand coordinate system, where the orientation of the x and y-axes differs from Blender even when aligning the z-axis upwards, eliminating a variable. The distinction between these systems lies in their spatial relationships rather than the individual directions of the axes. For the project, we opt for a left-hand coordinate system.

4.1.3. 3D to 2D mapping

Introduce a camera to be stored in our scene with a position vector $\underline{C_P}$ and an Euler rotation represented as a vector $\underline{C_R}$. As stated, Euler rotations are easy to use, needing only 3 inputs versus the 9 local axes use, i.e. no redundancy, hence typically used to store a camera's rotation.

$$Span(\underline{v1} \quad \underline{v2} \quad \underline{v3}) = \forall \alpha, \beta, \gamma, \alpha\underline{v1} + \beta\underline{v2} + \gamma\underline{v3}$$

From span, we derive that any three non-parallel directional vectors can traverse 3D space. The camera's position (as a position vector) + some amount of the camera's right local axis + some amount of the camera's forward local axis + some amount of the camera's up local axis (as three non-parallel directional vectors) can therefore be some point v (a vertex of an object).

$$\underline{C_P} + \alpha\underline{C_{Right}} + \beta\underline{C_{Up}} + \gamma\underline{C_{Forward}} = \underline{v}$$

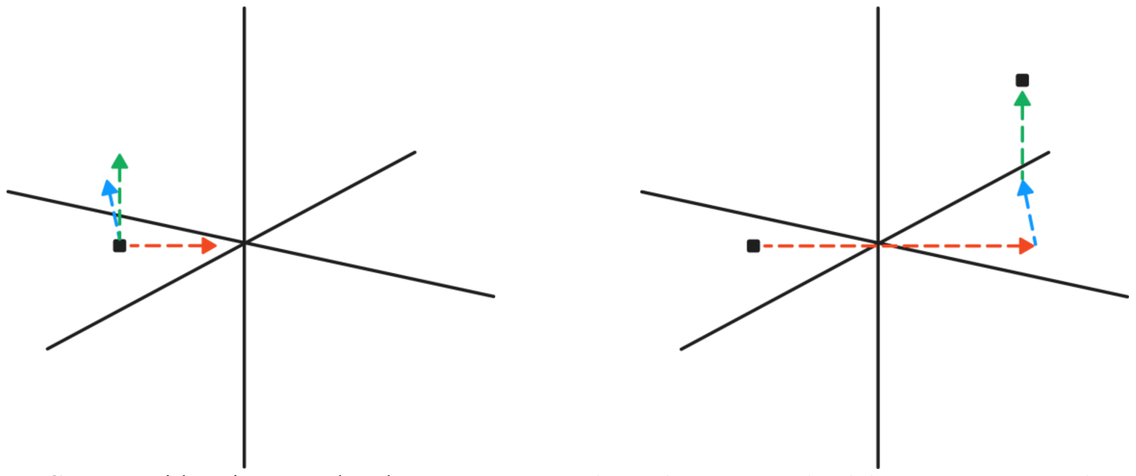

Camera with unit vector local axes.
Shows how a stretch of these axes can reach a point.

Figure 3. Diagram to illustrate how some stretch of a camera's local axes (three non-parallel directional vectors) from the camera's position can reach any point in 3D space. The left pixel represents the camera whilst the right pixel represents a point to map.

We take interest in the values of α, β, γ. Should this be an orthographic projection, α and β are a scale of the 2D screen coordinates of our mapped vertex, with the γ being useful for perspective projection. In solving for α, β, γ, we rewrite $\underline{C_P} + \alpha\underline{C_{Right}} + \beta\underline{C_{Up}} + \gamma\underline{C_{Forward}} = \underline{v}$ as:

$$\rightarrow \underline{C_P} + \left(\underline{C_{Right}} \quad \underline{C_{Up}} \quad \underline{C_{Forward}}\right)\begin{pmatrix}\alpha\\ \beta\\ \gamma\end{pmatrix} = \underline{v} \qquad \text{using } (\underline{v1} \quad \underline{v2} \quad \underline{v3})\begin{pmatrix}\alpha\\ \beta\\ \gamma\end{pmatrix} = \alpha\underline{v1} + \beta\underline{v2} + \gamma\underline{v3}$$

let $C_{LA} = \left(\underline{C_{Right}} \quad \underline{C_{Up}} \quad \underline{C_{Forward}}\right)$

$$\rightarrow \underline{C_P} + C_{LA}\begin{pmatrix}\alpha\\ \beta\\ \gamma\end{pmatrix} = \underline{v}$$

$$\rightarrow C_{LA}\begin{pmatrix}\alpha\\ \beta\\ \gamma\end{pmatrix} = \underline{v} - \underline{C_p}$$

$$\rightarrow {C_{LA}}^{-1} \cdot C_{LA}\begin{pmatrix}\alpha\\ \beta\\ \gamma\end{pmatrix} = {C_{LA}}^{-1}\left(\underline{v} - \underline{C_p}\right)$$

$$\rightarrow \begin{pmatrix}\alpha\\ \beta\\ \gamma\end{pmatrix} = {C_{LA}}^{-1}\left(\underline{v} - \underline{C_p}\right)$$

C_R is the camera's Euler rotation as a vector. C_{LA} is a matrix of the camera's 3 local axes, hence

$C_{LA} = C_R.localaxes()$

$\rightarrow \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = C_R.localaxes()^{-1} \left(\underline{v} - \underline{C_p} \right)$

From this we derive:

$$map(\underline{v}) = \underline{rotation}.localAxes()^{-1} \cdot \left(\underline{v} - \underline{position} \right)$$

Note, we depend heavily on our work for methods of rotation and now that can abstractly be referred to. Also note, each vertex of an object is mapped using the same inverse matrix and position vector, hence we can utilise the GPU's parallel processing capabilities.

We create a shader class consisting of a vertex shader program, a fragment shader program (for colours) and a program to load these programs into the GPU. Our VBO's, EBO's are already loaded into the GPU but we need some way of getting the camera attributes which change based on user input in CPU space into the shader too.

Let localAxesInverse $= \underline{rotation}.localAxes()^{-1}$ be a private attribute in the camera. This is independent of vertices so can be calculated in CPU space, rather than it being processed millions of times in the GPU. It is costly, involving sin and cos calculations in the rotation matrices, matrix multiplication and an inverse function so our program does benefit from running it beforehand. Of course, when we rotate our camera, we need to add a recalibrate method to amend the local axes. Also let us declare a render method in the camera.

Within OpenGL's render loop, we call our camera's render method (this can access its private attributes) where we establish communication between the CPU and GPU, via uniforms, loading localAxesInverse and position into our shader which performs map(v).

4.1.4. Perspective, orthographic projection, frustrum culling and z-buffering

We retrieve $\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}$ from our mapping, per vertex within our shader where α, β are a scale of the 2D screen coordinates of our mapped vertex and γ is its depth along the camera's normal. γ is irrelevant for orthographic projection; just display coordinate $(k\alpha, k\beta)$ for each vertex, choosing k based on the size of the model so it fits in frame. However, perspective must simulate how the human eye perceives depth: object's apparent size decreases proportionally to the inverse of their distance, hence for that $(k\alpha, k\beta)$ must be divided by γ to give an appropriate result.

Much mustn't be rendered – that is drawn to screen. Note, a key distinction between a vertex being rendered and processed. All must be processed in GPU space, mapped and projected but it is actually our camera's render method which renders the results, converting the final data into pixels onscreen with additional steps like shading and rasterization. This distinction allows us to prevent the rendering of a vertex by returning null, should we choose to, during processing in our GPU shader program.

Our eyes cannot see behind us, nor can they see so far ahead to perceive objects that are beyond our focus. Likewise, some vertices are far enough away that they are not visible in the current view frustum of the camera and of course vertices behind us mustn't be mapped into our view so removing these (frustrum culling) reduces unnecessary computational cost and gives clarity to the scene. Let us add to our camera near and far clipping planes, set in CPU space as attributes and passed into the GPU through uniforms. Our shader then returns null, and hence doesn't render, if the γ value lies out of this range. To gain the depth of a vector in some meaningful way, we

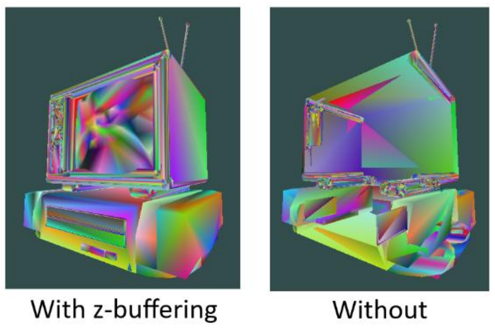


Figure 4. Comparison of rendering with and without z-buffering. Z-buffering ensures accurate rendering by tracking the depth of each pixel in a depth buffer, allowing objects closer to the camera to appear in front of those further away, simply by rendering only the pixel with the closest depth. It simplifies rendering by handling occlusion, eliminating the need for complex sorting algorithms and supports advanced techniques like shadows and transparency.

interpolate γ between the near and far planes to give a depth value between 0 and 1, which can be referred to in z-buffering. Figure 4 shows results.

4.1.5. Context-free grammar for scene data

We need to describe the state of the scene outside of the running execution, namely the attributes of our camera: position, rotation, field of view and any object obj files to be loaded into the GPU with their respective transformations. Constructing a file type with a CFG builds an interface for the user with a precise syntax specification and clear structure. They are easy to extract data from and accept or reject by building a parser that can determine if a file belongs to the language described by the grammar. Here we design the CFG which we can always extend to add more to our scene data:

```
<sdo> := <decl>\n<sdo> | <decl>

<decl> := <camera_decl>
       | <object_decl>
       | <fractal_decl>
       | # <comment> | \n

<camera_decl> := c <position> <rotation> <fov>
       <fov> := <float>

<object_decl> := o "<name>.obj" <transformation>
       <transformation> := (<position>, <rotation>, <scale>) | #

<fractal_decl> := f "<name>.fdo" <transformation>

<position> := <rotation> := <scale> := (<float>, <float>, <float>) | #

<name> := <comments> := <string>
```

For an example file, see Appendix A.

Cameras are stored as c followed by position, rotation, fov; objects are their file path followed by a matrix of 3 vectors: position, rotation and scale. Our design also features # as a shorthand, easy way of declaring no transformation, i.e. (0, 0, 0) if position, rotation and (1, 1, 1) if scale, leveraging the benefit of a CFG.

We build a generalised parser, to parse whitespace, characters, floats etc, and extend the class to our scene parser through inheritance, ensuring abstraction and reusability. Functions return a boolean (accept / reject) and output any data to extract plus the remainder of our string

```
public static bool ParseChar(string str, char ch, out string rem) => ⋯
```

Hence, when we parse strings of greater complexity, we can exit upon receiving false input rather than requiring to read the full string, speeding up execution. E.g. The object definition “x “name.txt”” will fail upon not receiving an o as input irrelevant that txt is an incorrect extension also.

```
public static bool ParseObject(string str, out string file, out Matrix? m) =>

    ParseChar(str, 'o', out string rem) &&

    More parsers to return true or false ? ⋯ : ⋯
```

Our scene can extend the scene parser and extract multiple obj files from our scene file, adding them to our object list, but how can we load multiple objects at once into our GPU? Create a RenderObject class with attributes to store our VBO, EBO buffers for each object. By adding a vertex array object (VAO) as an additional attribute we can bind our buffers, along with vertex attribute configurations to each VAO, reserve memory on the GPU to hold these configurations and switch between our VAO’s when rendering, i.e. change the GPU’s active state. This is going to be necessary when performing GPU instancing on the base case of fractals.

4.2. Fractal Generator

This section proposes a novel context-free grammar representation for fractal geometries and generation using GPU instancing and a parser for a custom file format.

4.2.1. The need for a parser

The Sierpiński triangle is a classic fractal starting with a base triangle, or pyramid in its 3D representation. We apply a single set of position transformations where each element in the set is an instantiation of an object:

$$T = [(0, 0.5), (0.5, -0.5), (-0.5, -0.5)]$$

Suppose function f applies a set of transformations to an object:

$$f\ x = \text{apply}(T, x) \text{ then}$$

f.f.f.f.f.f x builds up a Sierpiński triangle of depth 6 recursively. The logic is acceptable for simple fractals like image 1 in figure 5 where the base mesh is clear, but fractals are usually more complex in nature. For image 2, a set of transformations has been applied to a base object to build a tree structure. Already we require rotation and scaling, so each transformation in the set must be a matrix of position, rotation and scale vectors.

$$T_1 = [(p_1, r_1, s_1), (p_2, r_2, s_2), (p_3, r_3, s_3),], \text{ where literals are vectors.}$$

However, a further set of transformations T_2 has also been applied to construct a diamond structure that overlaps the base object. Taking a close look at the fractal, we can see trees converging in to form this diamond, hence:

$$f\ x = \text{apply}(T_2, \text{apply}(T_1, x))$$

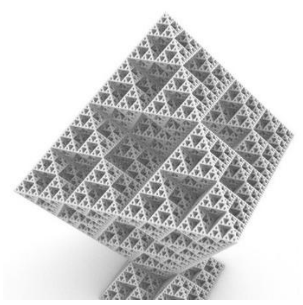
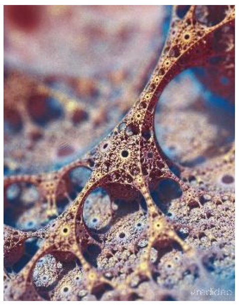
1

2


Figure 5. Image 1 [16] is the Sierpiński tetrahedron, a classic fractal geometry generated recursively using a set of position transformations on a base tetrahedral shape. Each recursive step subdivides the tetrahedron into smaller tetrahedra arranged in a self-similar 3D pattern. Image 2 [17] is a complex 3D fractal structure, combining tree-like branching and a central diamond-shaped convergence. The fractal is generated by applying two sets of transformations: the first set creates a 3D tree structure, while the second set overlaps and integrates these trees into a diamond-shaped formation.

A fractal generator must therefore take in a set of sets of matrix transformations and a base obj file, as shown in figure 6. Figure 6 and 7 shows how the complexity above demands and justifies the implementation of a parser.

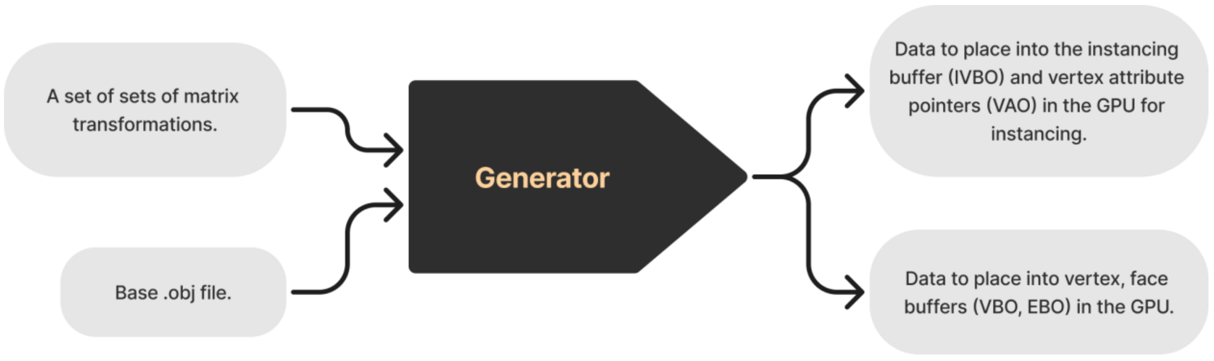

A set of sets of matrix transformations.
Base .obj file.
Generator
Data to place into the instancing buffer (IVBO) and vertex attribute pointers (VAO) in the GPU for instancing.
Data to place into vertex, face buffers (VBO, EBO) in the GPU.

Figure 6. Function diagram of the fractal generator showing its inputs and outputs. The inputs provide no platform for a user to interface – a pretty terrible design! Fractals are complex, especially those with large or numerous sets of matrices and as we look ahead to leveraging their potential to instantly produce cliffs, mountains, terrain systems or galaxies, we demand higher-level control and flexibility for users to experiment with various effects and patterns without being constrained by hardcoding intricate sets of matrices. We want readability in higher-level instructions, not low-level transformational data, justifying a strong need for a custom file format. With this comes a layer of validation as we implement a parser to accept / reject files and extract their data.

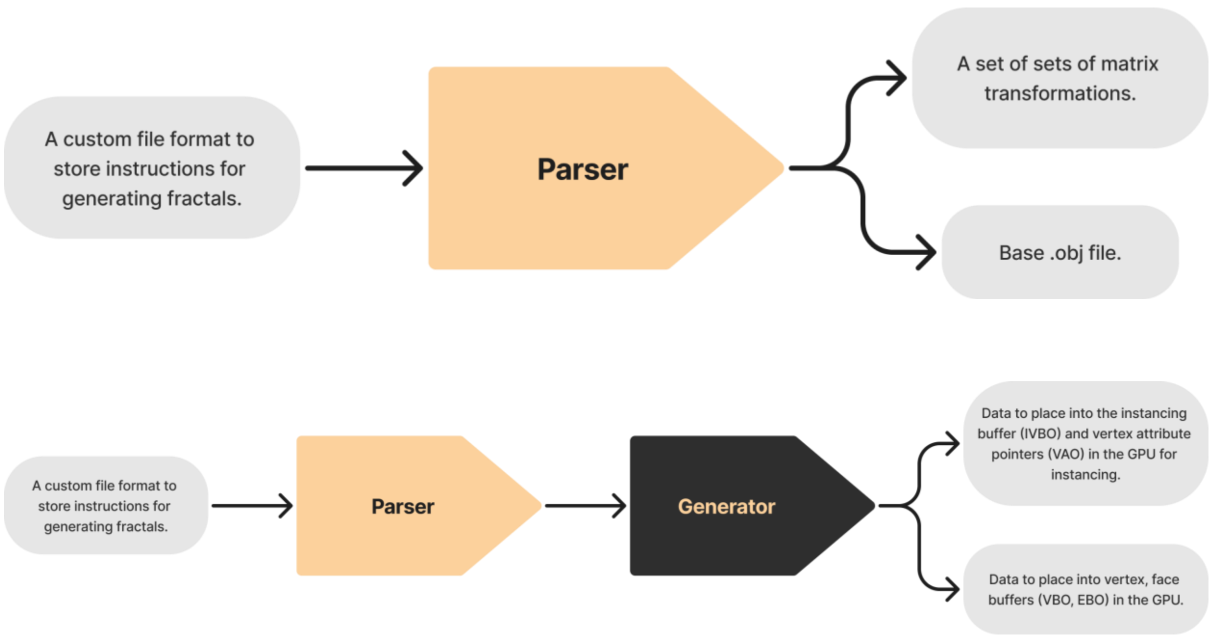

A custom file format to store instructions for generating fractals.
Parser
A set of sets of matrix transformations.
Base .obj file.
A custom file format to store instructions for generating fractals.
Parser
Generator
Data to place into the instancing buffer (IVBO) and vertex attribute pointers (VAO) in the GPU for instancing.
Data to place into vertex, face buffers (VBO, EBO) in the GPU.

Figure 7. A proposed solution featuring a parser and a custom file format for storing instructions to generate fractals – a design which provides the higher-level control, readability and flexibility figure 6 was lacking.

4.2.2. Designing a file type

We design a language where objects can be defined as:

- The base object file in the form "name.obj".
- A set of transformations applied to an object.
- Simply another object.
- A recursive (an object which is not finite) and an integer (to flatten the recursive into an object which is finite).

Recursives are similarly defined but include a base object marked by < >. They represent all possible structures that can be produced from their base object by iterating through some transformations. For instance, in the code snippet:

```
a. "base.obj"
b. X a
r<a>. Y b
where X, Y are some sets of transformations.
```

We conclude:

$$\rightarrow \forall n.\ r = (YX)^n a$$

The language supports transformation functions which take input vectors and return (position, rotation, scale) – a handy tool for readability and concise code. Further readability methods include the addition of vectors so transformations can refer to the vector index instead of repeated code and using #'s to refer to no transformation, i.e. (0, 0, 0) for position, rotation and (1, 1, 1) for scale. We use our defined Vector.zero, Vector.one for this. Our full CFG design is:

```
<fdo> := <decl>\n<fdo> | <decl>

<decl> := <vector_decl>
       | <transformation_decl>
       | <object_decl>
       | <recursive_decl>
       | # <comment> | \n

<vector_decl> := <name> <float> <float> <float>

<transformation_decl> := <name>.<inp><inp><inp> (<out>, <out>, <out>)
      <inp> = <name>. | <null>
      <out> = <name> | <int> | #

<object_decl> := <name>. <object>

<recursive_decl> := <name><<name>>. <object>

<object> := “<name>.obj”
       | [<transformations>] <object>
       | <name>
       | <name> <int>

<transformations> := <transformation>, <transformations> | <transformation>

<transformation> := <name>.<id><id><id> | #
      <id> := <int>. | <null>

<name> := <comments> := <string>
```

For an example file, see Appendix B.

Our parser reads line by line building lists of vectors, transformations, objects and recursives where parsing a declaration appends these. Objects store their name, base file path and a list of lists of matrices. As we backtrack through an object, we apply sets of matrices to a base file, i.e. x = [a] ([b] ([c] ("base.obj"))) which is stored as: x = {name = “x”, file = "base.obj", transformations = [[c], [b], [a]]}, by declaring many constructors to allow for different declarations:

declaration	constructor description	name	file	transformations
x. “base.obj”	base case	x	“base.obj”	new List<List<Matrix>>()
y. […] x	apply set of transformations	y	x.file	x.transformations ++ […]
z. y	assignment	z	y.file	y.transformations
w. r 6	flatten recursives	w	r.file	((r.transformations.removeTail (r.base.transformations)) * 6) ++ r.base.transformations

Here, we produce code to flatten recursives, allow for assignment and inductively build up object’s transformations. Recursives extend the object class as are declared <recursive_decl> := <name><<name>>. <object> in the CFG, storing an additional object attribute, base, used in flattening recursives above. Names can be used to check objects exist in the current context, i.e. for y. […] x, x must exist.

We structure our parser so that each function returns a boolean indicating whether the syntax is accepted or rejected and uses output parameters to extract the relevant data. This approach ensures that when parsing declarations, the parser will first attempt to parse the first acceptable declaration. If that fails, it will then try parsing the next. Hence, we adhere to modularity in incremental parsing and promote reusability, clear error handling and control flow:

```
public static bool ParseObjectDeclaration(string name,string str
        lists of objects,transformations,recursives already declared, out ObjectData? obj) =>
```

```
    ParseFile(...,out string file)
            ? (obj = new ObjectData(name, file),true).Item2
            : ParseTransformationsObject(...,out List<Matrix?>? ms,out ObjectData? _obj)
                  ? (obj = new ObjectData(name,ms, _obj),true).Item2:...
```

The challenge of transformation functions is that they take in 0-3 vector indexes who's values are undecided. For the declaration t.a.b. (a, 3, b), a matrix can only be produced later in code when we set the values of its parameters e.g. t.1.2. Therefore, our constructor must take in a name, some inputs that may be null (t.a.b. has no third parameter) and a string array of outputs which could be vector indexes or variables (undecided values). From this we construct a function Func<int?, int?, int?, Matrix> f to be an attribute of the transformation class. E.g. given the declaration:

```
v 2 4 6
t.a. (1, #, a)
```

We build t ={name = t, f = (x, y, z) => ((2, 4, 6), (0, 0, 0), ParseVar(x))}

The implementation is difficult but justified because it avoids the redundancy of repeated large matrices throughout our fractal data object. Finally, the file returns the fractal object marked by <$>. All other instructions are just steps to get to this return object.

4.2.3. Object transformations

The parser returns a set of sets of matrix transformations on a base object where each matrix consists of three vectors: position, rotation and scale. To generate fractals, we require functions that can apply these transformations to an object. Transforming an object involves applying a transformation to each of its vertices in the GPU. Let *T* represent a generalised method, where *t* is the specific transformation function to be applied:

$$T(t) = foreach\ \underline{v} \in object, \underline{v}' \leftarrow t(\underline{v})$$

For translations, the process is straightforward, as it simply involves adding a translation vector to each vertex, achieved by passing in a lambda calculus as our value for t.

$$translate(\underline{t}) = T(\lambda \underline{v} \rightarrow \underline{v} + \underline{t})$$

For both rotations and scales, we perform the transformation about some pivot $\underline{p}$. Each vertex can be written as the transformation to the pivot followed by the transformation from the pivot to the vertex: $\underline{v} = \underline{p} + (\underline{v} - \underline{p})$ where it is the transformation from the pivot to the vertex, $(\underline{v} - \underline{p})$, which we wish to rotate or scale.

A rotation takes an Euler rotation vector $\underline{r}$ who's local axes we can retrieve using our localAxes() method produced in 4.1.2. To place $(\underline{v} - \underline{p})$ on r's local axes and thus rotate v, we must scale the axes by the components of the vector:

$$(\underline{v} - \underline{p})_x \cdot \underline{r_x} + (\underline{v} - \underline{p})_y \cdot \underline{r_y} + (\underline{v} - \underline{p})_z \cdot \underline{r_z}$$

$$\rightarrow (\underline{r_x} \quad \underline{r_y} \quad \underline{r_z}) \cdot (\underline{v} - \underline{p}) \qquad \text{using } (\underline{v1} \quad \underline{v2} \quad \underline{v3}) \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \alpha\underline{v1} + \beta\underline{v2} + \gamma\underline{v3}$$

$$\rightarrow \underline{r}.localAxes() \cdot (\underline{v} - \underline{p}), hence$$

$$rotate(\underline{r}, \underline{p}) = T(\lambda \underline{v} \rightarrow \underline{p} + \underline{r}.localAxes() \cdot (\underline{v} - \underline{p}))$$

For scaling, we amend our vector class to add a stretch method and simply stretch each vector by the scale vector $\underline{s}$.

$$scale(\underline{s}, \underline{p}) = T(\lambda \underline{v} \rightarrow \underline{p} + (\underline{v} - \underline{p}).stretch(\underline{s})),$$

$$where\ \underline{v}.stretch(\underline{s}) = \begin{pmatrix} s_x \cdot v_x \\ s_y \cdot v_y \\ s_z \cdot v_z \end{pmatrix}$$

We will use 4D matrices to represent translation, rotation and scaling simultaneously. These can then easily be passed into the GPU for processing via buffers.

4.2.4. Generator design and GPU instancing

The generator concatenates the set of sets of matrices returned by the parser into a single set by applying the matrix transformations to one another. Positions can be summed as can rotations (as they are in Euler vector form). Scales can use the stretch method implemented above, hence two matrices are applied like so:

$$(\underline{p_1} \quad \underline{r_1} \quad \underline{s_1}) \cdot (\underline{p_2} \quad \underline{r_2} \quad \underline{s_2}) = (\underline{p_1} + \underline{p_2} \quad \underline{r_1} + \underline{r_2} \quad \underline{s_1}.stretch(\underline{s_2}))$$

We start with a smaller problem of concatenating a pair of sets of transformations:

$$concatPair(set1, set2) = [m_1 \cdot m_2 \mid \forall m_1 \in set1, \forall m_2 \in set2]$$

Each matrix in the first set is applied to each matrix in the second set to produce a single set. The operation can be recursively expanded to concatenate multiple sets of transformations:

$$concat(set1, set2, set3, \dots, setn) = concat(concatPair(set1, set2), set3, \dots, setn)$$

The generator acts as a factory for creating fractals. Our scene stores a list of fractals, where each fractal inherits from the object class, adding an additional attribute, a set of matrices. When the scene parser encounters an fdo file, it passes the file path to the generator. The generator then uses the fractal parser to extract the data and passes the base file into the fractal constructor to produce a list of vertices and a list of face indexes as inherits the object constructor.

When loading a VAO for each fractal, we store an addition set of matrices into the array buffer, allowing for GPU instancing: a technique that renders multiple instances of the same object with different transformations using a single draw call. Each instance's transformation is provided by its corresponding matrix in the array buffer. The shader then applies these matrices to each fractal instance, improving rendering efficiency by reducing draw calls. We leverage the recursive property of fractals to render them with optimal performance. See 5.2 which expands on this.

It's important to note, fractals store a single iteration of transformations and a depth. For calculations such as collision and dynamic rendering, we only require this single iteration, not the full concatenated mesh. Hence, we concatenate the mesh to its set depth once before loading into the GPU array buffer and use the simpler, much smaller, single iteration of transformations in calculations, requiring less computation. See 5.3 for some images of the generated fractals!

4.3. Fractal Rendering

This section explores rendering of dynamic, infinite fractal structures whose geometry adapts to the camera's position efficiently utilising the GPU.

4.3.1. Reloading the GPU

So far, a scene file is parsed; object / fractal paths are extracted, parsed too, generated and added to lists within a scene. Upon loading the GPU, buffers are created for each object / fractal and bound to their respective VAO's. These store vertices, faces and a concatenated set of matrix transformations, in the case of fractals, each of which could likely be millions in length. For loading once, this presents no significant issue, even with a few million operations, but 'reloading data to the GPU every frame in real-time is highly inefficient and imposes significant performance costs' [13].

By introducing a matrix m, that is applied to all vertices in a mesh, we can move objects over time (see 4.4.1) and dynamically render fractals, simply by updating the value of m. m is stored in the Object class to which fractals inherit, changed when needed and by creating a new buffer, containing this matrix per object / fractal, which is reloaded each frame, we can reduce the reloading process from updating three large buffers to updating a single buffer containing a single manageable matrix.

Still the same work must be done – applying this matrix to every vertex in the mesh – there's no get around to that, but now it is done in GPU space, in parallel, within the shader program rather than by changing every vertex in the CPU.

4.3.2. Dynamic fractals

As the camera travels closer to a fractal, endless detail is displayed. Fractals are infinite, recursive geometries but computers cannot render infinite vertices, so we must use a trick.

A fractal is made up of many transformations of itself (as well as the global transformation m, discussed above). As the camera approaches one of those transformations, i.e. transformation t, we can transform the entire fractal to overlap $m \cdot t$, creating the illusion of infinite detail, see figure 8.

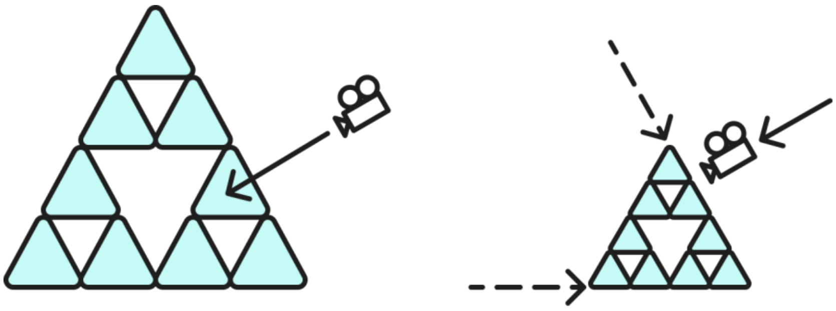


Figure 8. Diagram illustrating the idea that as the camera approaches the fractal, the entire fractal is transformed to overlap which ever sub-transformation is closest to the camera. This is through the assignment $m \leftarrow m \cdot t$.

We ask ourselves, which transformation is closest to the camera? Does the camera collide with that transformation? If so, transform the entire fractal to overlap it and repeat throughout the depth of the fractal.

More technically however, we iterate through each transformation in a single depth of a fractal, find the closest transformation to the camera, t, check the camera is close enough to t and if so, set $m \leftarrow m \cdot t$. We repeat until the camera is no longer close enough to a sub-transformation (i.e. it's only zoomed in to a certain extent), each iteration building up a chain of matrix multiplications as we progress through the depth of the fractal resulting in an assignment of $m \leftarrow m \cdot t_1 \cdot t_2 \cdot t_3 \cdot$ ... We then set the value of the buffer created in 4.3.1 to equal this resulting m when reloading the GPU.

```
Fractal fdo = fractals[index];
Matrix4 m = fdo.m;
while (true) {
        Matrix4 closest = m * fdo.transformations[0];
        for (int i = 1; i < fdo.transformations.Count; i++)
                closest = ClosestMatrix(closest, m * fdo.transformations[i]);

        if ((camera.position - MatrixPos(closest)).magnitude < fdo.radius(closest)) =>
                m = closest;
        else break; }
Reload(m, fdo);
```

The distance between t and the camera can be calculated as $\left|\left(\overline{camera.position} - \overline{t.position}\right)\right|$. Hence, we have a value to compare against when finding the closest transformation to the camera. Here, we construct a function

ClosestMatrix(m1, m2) that takes in two matrices and compares the two magnitudes, returning the matrix with the smallest magnitude from the camera and use this to iterate through the transformations.

Similarly, to check if the camera is close enough to t, we calculate the distance between t and the camera (the magnitude) and determine if it is lower than a certain value. This value needs to decrease with the depth of the fractal: a value of 1 unit, may be too small for low depths but too large for high depths. A good measure would be the radius of the transformed fractal.

It's important to note, we cannot replace our original m (fdo.m in the code above) with the resultant value for m. That is because if we wish to zoom back out of a fractal or stray onto a different path of the fractal, the resultant m has lost that information. Instead, store the original m in the fractal, leave it untouched and compute the above algorithm each frame, parsing the resultant m into the GPU buffer.

This does mean that if a camera is within depth 10 of a fractal and in a single frame travels further to depth 11 then the above would iterate 11 times rather than just once as must start from the original global matrix m. Although, given it's time complexity is $O(\text{depth} \cdot \text{transformations})$ and the number of transformations in a single iteration of a fractal is very low, it means the cost of this algorithm is low. Yes, if we zoom in to a depth 1 million, it may crash.

4.3.3. Radius of an object

We add a radius function to our object class, used above for checking if the camera is close enough to a transformation, t. The furthest vertex in an object defines the maximum distance from its central point and so is a simple approximation for the radius. We must also factor in the global matrix m when finding the furthest vertex as it could comprise of a scale transformation, which would stretch all vertices and hence the radius.

We apply m to each vertex but subtract its position to remove the effects of translation, ensuring distances are measured relative to the object's transformed centre. Then, by iterating through each vertex, we return the greatest magnitude.

$$A.radius(m) = max\left(\left|\left(m \cdot \underline{v} - \underline{m.position}\right)\right| \; for \; v \in A\right).$$

To find the radius of a transformed fractal (a fractal is made up of many transformations of itself), we substitute m in the above calculation as $m \cdot$ the chain of transformations built up as we traverse the depth of the fractal. This works because each transformed fractal is simply a transformation of the same base mesh. Hence, 'fdo.radius(closest)' is used in the code above as 'closest' is $m \cdot$ the chain of transformations we're building in the while loop. Our function therefore takes in a matrix, though for regular objects, we just substitute their global matrix m.

Calculating an object's radius is necessary for producing sphere colliders in 4.4.2, which is why we place the function in our object class to which fractals can inherit.

4.3.4. Fractal trails

Refer back to figure 8. The camera zooms in to see an infinitely detailed fractal before glancing out to the distance. Much of the fractal has disappeared and that is apparent! The illusion is broken.

We add a queue, used to retain a few previous depths of transformation. That way, if we zoom in to say depth 10, maybe the entire fractal is only transformed to depth 8 and we retain 2 levels of transformation beyond what we have zoomed into. Hence, when we look out to distance, we still see some geometry exists there. The length of the queue is the number of depths we retain (the fractal trail), see figure 9.

Zooming into a depth of 4, previously would return $m \cdot t_1 \cdot t_2 \cdot t_3 \cdot t_4$. With a trail of 2 (setting the queue size to 2), we return $m \cdot t_1 \cdot t_2$, ensuring we retain 2 levels of transformation.

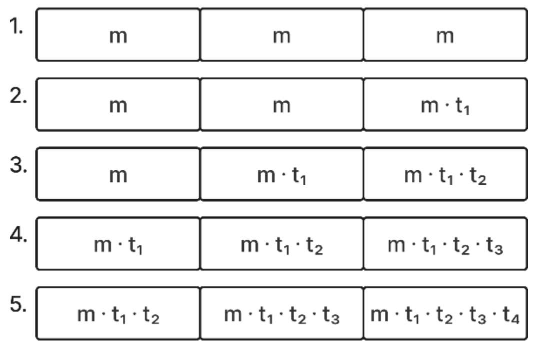


Figure 9. Demonstration of the queue at use, keeping track of previous transformations. In this example, we use a fractal trail of queue size 3, meaning we retain 3 levels of transformation. We set the trail length as a variable and construct a queue of that size during runtime, allowing for us to easily change the number of levels we wish to retain. When the fractal is transformed, it uses the head of the queue.

4.4. Fractal Collisions

In this section, we extend our engine to include accelerations, velocities, colliders and coefficients of restitution and provide an optimised collider for fractals, giving appropriate detection and response.

4.4.1. A simple physics engine

We need a system to allow for objects to move over time, under forces, and respond to collisions. For this, we extend the scene file's context-free grammar to include physics objects. They are declared with their object file path, their global transformation and a coefficient of restitution, e, i.e. bounciness. This parameter allows for various materials to be represented in the software and plays into how we respond to collision in 4.4.4. When choosing a value between 0 and 1, 0 represents a perfectly inelastic collision, i.e. a steel ball, and 1 represents a perfectly elastic collision, i.e. a rubber ball (which bounces without losing energy).

```
<decl> := <camera_decl>
       | <object_decl>
       | <fractal_decl>
       | <physics_decl>
       | # <comment> | \n

<physics_decl> := p "<name>.obj" <transformation> <e>
       <e> := <float>
```

Physics objects extend objects but are packaged with a collider, velocity, a physicsMove function and this coefficient of restitution. When the scene file is parsed, physics objects are created; added to a list within the scene and each loaded into the GPU.

To move an object, for each force applied to it, we calculate its acceleration: $\underline{a} = \underline{F}/m$. Though, this may not always be necessary, e.g. for gravitational force, we always use $\underline{a} = \begin{pmatrix} 0 \\ -9.8 \\ 0 \end{pmatrix}$. Its velocity starts at vector zero and increases each frame by $\underline{a} \cdot \Delta t$, where Δt is the time elapsed during that frame. This follows from acceleration being the rate of change of velocity over time.

We then increment the global matrix m's position by $\Delta \underline{v}$, since velocity is the rate of change of displacement over time. Therefore, our physicsMove function must take in a deltaTime (the time elapsed during that frame) as a parameter. We must also extend this function further for responding to collisions, see 4.4.4.

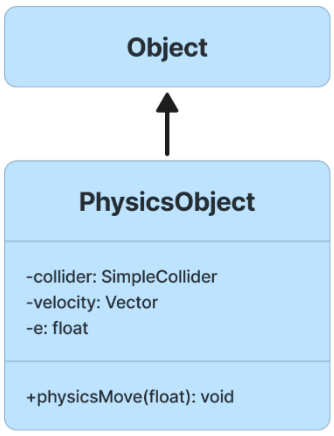


Figure 10. Class diagram showing the inheritance relationship between physics objects and objects. It displays the collider, velocity and coefficient of restitution attributes and the physicsMove function.

4.4.2. Object collider

We introduce a simple collider to be attached to a physics object. It will calculate the closest point in a scene full of meshes, check whether that point is close enough to the object to determine a collision has taken place and later provide a response. When instantiating the collider, we pass into it a position and a radius, calculated from our function in 4.3.3.

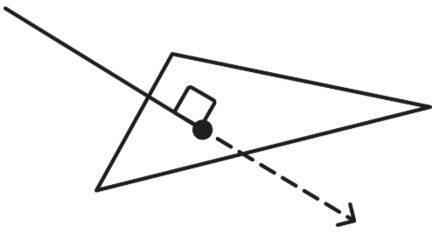


Figure 11. Sketch showing the path from the collider's position to the closest point on the plane. Notice how the line is perpendicular to its surface. This is information we can work off.

First, we find the closest point on a single face (made of 3 vertices $\underline{a}$, $\underline{b}$ and $\underline{c}$) to the collider at position $\underline{p}$. This point makes a line with $\underline{p}$ perpendicular to the surface of the face – along the face's normal $\underline{n}$. In other words, the closest point $\underline{v}$ plus some amount, k, of the face's normal $\underline{n}$ is position $\underline{p}$.

$$\underline{p} = \underline{v} + k \cdot \underline{n}$$

By finding the value of k, we can substitute it in and rearrange to express $\underline{v}$ in a computational form:

$\rightarrow (\underline{p} - \underline{a}) = (\underline{v} - \underline{a}) + k \cdot \underline{n},$ by subtracting $\underline{a}$ from both sides

$\underline{v} = \underline{a} + \lambda(\underline{b} - \underline{a}) + \mu(\underline{c} - \underline{a}),$ since $\underline{v}$ lies in the plane, it can be represented using the vector plane equation

$\rightarrow (\underline{v} - \underline{a}) = \lambda(\underline{b} - \underline{a}) + \mu(\underline{c} - \underline{a}),$ by subtracting $\underline{a}$ from both sides

$\rightarrow (\underline{p} - \underline{a}) = \lambda(\underline{b} - \underline{a}) + \mu(\underline{c} - \underline{a}) + k \cdot \underline{n},$ by substituting $(\underline{v} - \underline{a}) = \lambda(\underline{b} - \underline{a}) + \mu(\underline{c} - \underline{a})$

$\rightarrow (\underline{p} - \underline{a}) \cdot \underline{n} = (\lambda(\underline{b} - \underline{a}) + \mu(\underline{c} - \underline{a}) + k \cdot \underline{n}) \cdot \underline{n},$ by applying the dot product with $\underline{n}$ to both sides

Since $\underline{n}$ is perpendicular to any vector in the plane, the dot product of $\underline{n}$ with $\underline{b} - \underline{a}$ and $\underline{c} - \underline{a}$ is zero, hence:

$$\begin{aligned}\rightarrow (\underline{p} - \underline{a}) \cdot \underline{n} &= (\lambda \cdot 0 + \mu \cdot 0 + k \cdot \underline{n}) \cdot \underline{n} \\ &= (k \cdot \underline{n}) \cdot \underline{n} \\ &= k(\underline{n} \cdot \underline{n}) = k, \qquad \text{since } \underline{n} \text{ is a unit vector, } \underline{n} \cdot \underline{n} = 1\end{aligned}$$

Hence, $k = (\underline{p} - \underline{a}) \cdot \underline{n}$

Referring back to our original expression, we substitute in k to give:

$$\underline{v} + \left((\underline{p} - \underline{a}) \cdot \underline{n}\right) \cdot \underline{n} = \underline{p}$$

$$\rightarrow \underline{v} = \underline{p} - \left((\underline{p} - \underline{a}) \cdot \underline{n}\right) \cdot \underline{n}$$

Where:

$$\underline{n} = \frac{cross(\underline{b} - \underline{a}, \underline{c} - \underline{a})}{|cross(\underline{b} - \underline{a}, \underline{c} - \underline{a})|} = cross(\underline{b} - \underline{a}, \underline{c} - \underline{a}).unitVector()$$

Now, we can compute the closest point on a single face. Under this model however, this point may lie outside of the face, as shown in figure 11, since it is simply the projected vector on an infinite plane. We need to determine if it does lie outside and respond to it in that case. By using barycentric coordinates, common in 3D graphics although used little in this dissertation, we check if the closest point is within the bounds of the triangle (by expressing it as a weighted combination of the triangle's vertices) and if so, return the computed value above.

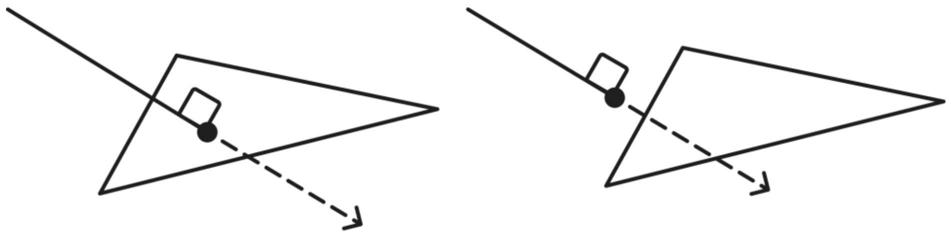


Figure 12. Sketch highlighting that the closest point on a face's plane to the collider's position may lie outside the face as is simply the point which forms a perpendicular line with the collider's position to the surface of the face.

Otherwise, we'll determine the closest points on each of the 3 edges of the triangle and return the closest. For this, we require finding the closest point on an edge (made of 2 vertices $\underline{a}$ and $\underline{b}$) to the collider at position $\underline{p}$. Let $\underline{ab} = (\underline{b} - \underline{a})$ be the vector from a to b (the direction vector of the line). Our closest point, $\underline{v}$, is then position vector $\underline{a}$ to get onto the line plus some amount, k, of $\underline{ab}$.

$$\underline{v} = \underline{a} + k\underline{ab}$$

We also know the vector from $\underline{p}$ to $\underline{v}$ is perpendicular to $\underline{ab}$ as $\underline{v}$ is the closest point. Hence $\underline{v} - \underline{p} = (\underline{a} - \underline{p}) + k\underline{ab}$ is perpendicular to $\underline{ab}$. Again, by finding the value of k, we can substitute it in and rearrange to express $\underline{v}$ in a computational form:

$\rightarrow \left((\underline{a} - \underline{p}) + k\underline{ab}\right) \cdot \underline{ab} = 0,$ since the dot product of two perpendicular vectors is zero

$\rightarrow (\underline{a} - \underline{p}) \cdot \underline{ab} + k(\underline{ab} \cdot \underline{ab}) = 0,$ by expanding the dot product

$\rightarrow k(\underline{ab} \cdot \underline{ab}) = -(\underline{a} - \underline{p}) \cdot \underline{ab} = (\underline{p} - \underline{a}) \cdot \underline{ab},$ by rearranging

$\rightarrow k = \dfrac{(\underline{p} - \underline{a}) \cdot \underline{ab}}{\underline{ab} \cdot \underline{ab}}$

Similar to how the closest point can lie outside of a face, it could also lie outside of the edge, as is the projected vector on an infinite line. We must clamp our k value between 0 and 1, so that if it does lie outside, we return a corner.

$$\rightarrow \underline{v} = \underline{a} + clamp\left(\frac{(\underline{p} - \underline{a}) \cdot \underline{ab}}{\underline{ab} \cdot \underline{ab}}, 0,1\right) \cdot \underline{ab}$$

Using a lambda calculus and combining our two solutions together, we produce the function:

$ClosestPointFace(\underline{a}, \underline{b}, \underline{c}) =$

$$\left(\lambda\underline{v} \rightarrow \underline{v} \text{ lies inside triangle } ?\ \underline{v} : \underline{a} + clamp\left(\frac{(\underline{p} - \underline{a}) \cdot \underline{ab}}{\underline{ab} \cdot \underline{ab}}, 0,1\right) \cdot \underline{ab}\right)\left(\underline{p} - \left((\underline{p} - \underline{a}) \cdot \underline{n}\right) \cdot \underline{n}\right)$$

Now we simply iterate through all the faces of an object to determine the closest point in the entire mesh. For this, we construct a mapping function to get the vector values for $\underline{a}, \underline{b}, \underline{c}$.

Our objects contain a flat float vertices array storing all vertex positions sequentially, i.e. $\{x_1, y_1, z_1, x_2, y_2, z_2, \ldots\}$ and an indices array storing all faces as a flat sequence of 3 vertex indexes. To retrieve a single coordinate of a vertex we multiply obj.indices[x] by 3 to jump to the correct starting position in the vertices array and add a y to select the specific coordinate:

$$g(x, y) = obj.vertices[(3 \cdot obj.indices[x]) + y]$$

We retrieve the full vertex as $\begin{pmatrix} g(x,0) \\ g(x,1) \\ g(x,2) \\ 1 \end{pmatrix}$ and apply the global matrix, m, to give: $f(x) = m \cdot \begin{pmatrix} g(x,0) \\ g(x,1) \\ g(x,2) \\ 1 \end{pmatrix}$. When we loop through the faces, we pass in these mapped values to return the closest point on an object to the collider:

```
    Vector v = ClosestPointFace(f(0), f(1), f(2));
    for (int i = 3; i < obj.indices.Length; i+= 3)
        v = ClosestVector(v, ClosestPointFace(f(i), f(i+1), f(i+2)));
```

We run this code for each mesh (all objects, fractals and physics objects) in the scene, ensuring to ignore itself otherwise we would always detect a collision, and gain the closest point for a whole scene. If the distance between this point and the collider's position (the magnitude) is smaller than the radius, a collision has taken place.

Finding the closest point gives us a perfectly accurate response, even with overlapping meshes and provides a really easy and efficient way to respond to collision, see 4.4.4.

4.4.3. Optimised collider for fractals

The method above is computationally expensive. A mesh with millions of faces would require millions of operations per frame as it computes the closest point on each face. Many existing collision detection methods reduce this cost by diminishing accuracy to achieve real-time performance, however by leveraging fractal's recursive geometry, we can exponentially decrease the number of operations whilst maintaining full accuracy. See 5.2 for a full evaluation.

Our aim is to recursively find the closest point on a fractal. As a reminder, a fractal is made up of many transformations of itself (as well as the global transformation m). For each transformation, we can think of it as a base mesh and find the closest point on it to the collider. As we iterate through, we compare which transformation, t, has the closest point and set $m \leftarrow m * t$. Hence, considering that transformation as the new fractal. We repeat for the depth of the fractal, each iteration building up a chain of matrix multiplications as we progress, resulting in an assignment of $m \leftarrow m * t_1 * t_2 * t_3 * \ldots$ This leaves us with a final closest point, that we return.

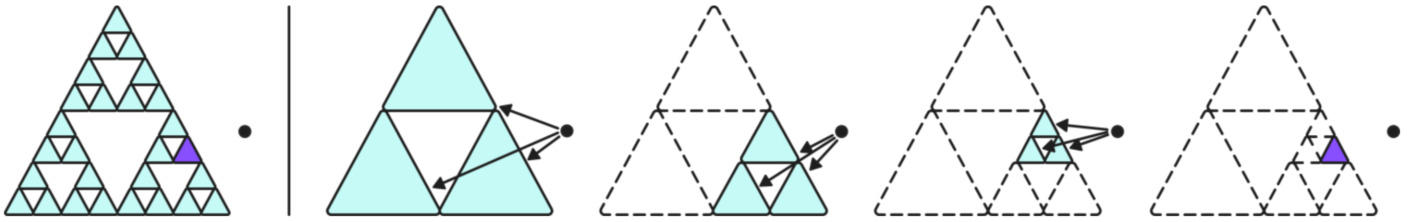


Figure 13. Demonstration of the process to recursively find the closest point on a fractal. At each level, we compare each transformation as if it were a base mesh, then repeat for the transformation that we find is the closest. Notice how much of the fractal is exponentially being ignored, resulting in a collision detection where nearly all of its faces aren't checked for their closest point – a vast reduction in the number of operations.

Our object collider is reasonable for small meshes, so can use it to calculate the closest point on a base mesh (which could be as little as 10 faces) in our algorithm above. Note, fractals should be using inexpensive base meshes which under transformations produce complex geometry.

This only applies to fractals due to their recursive properties, hence for each object and physics object we use our object collider and for each fractal we use this improved method. We create multiple collider classes which inherit a base collider class, allowing us to reuse functions and easily change which collider we wish to use in our physics object.

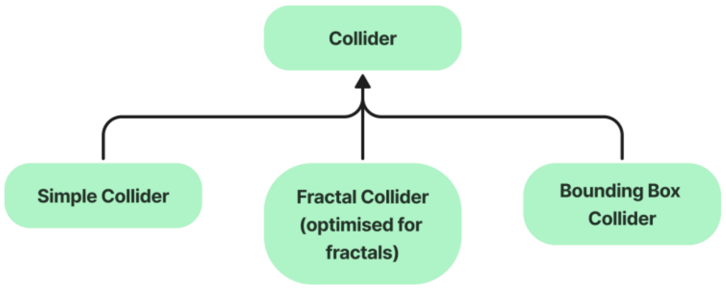


Figure 14. Class diagram showing our Collider base class with three derived types, each representing the different collision strategies. This could further be expanded through polymorphism.

4.4.4. Response

A physics object starts with an initial velocity, $\underline{u}$, and overshoots into another object. Within the same frame, we must provide a response to the collision. We compute the closest point above as it can efficiently determine both the direction and depth of the overlap. This tells us how much the collider has moved inside the object, allowing us to scale this direction accordingly to push it back out and resolve the collision.

We calculate the displacement vector between the position of the collider and the closest point as $\underline{d} = \underline{p} - \underline{v}$. If the magnitude of this displacement is lower than the radius of the collider, $|\underline{d}| < r$, a collision has taken place and we determine the depth (amount its overshot) as:

$$overshoot = r - |\underline{d}|$$

By scaling the unit vector of the displacement by how much the collider has overlapped the mesh, we achieve a response to the collision, as shown in 3 of figure 15:

$$\underline{response} = (r - |\underline{d}|) \cdot \frac{\underline{d}}{|\underline{d}|}$$

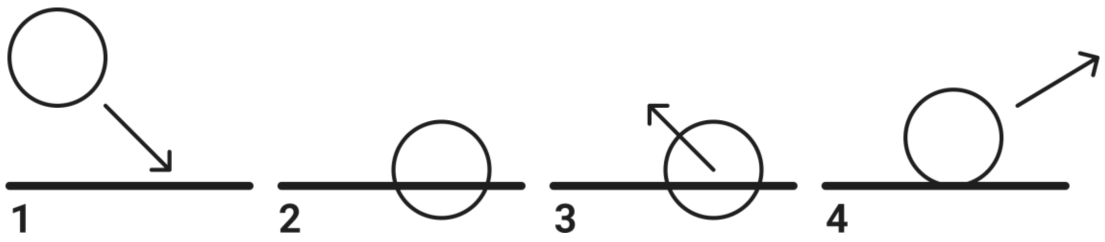


Figure 15. The collision process in a single frame of execution. 1. The physics object starts with initial velocity and is re-positioned based on this and the time passed since last frame. 2. This results in the object overlapping the mesh. A collision has taken place and must be detected. 3. We adjust the ball accordingly by the distance between the closest point on the mesh and the collider's position, in the opposite direction. 4. If the velocity is left unchanged, due to the objects constant acceleration it would soon become a particle accelerator. We must override this velocity to either let it roll across the mesh or bounce based on its coefficient of restitution.

Our physics object now sits atop the mesh it once overlapped, but as its acceleration continues to increase its speed, it soon moves so far per frame that it completely overshoots the mesh entirely. Hence, no collision is detected despite the object still passing the mesh. So, as well as adjusting for the overlap, we must also correctly append the object's speed. For this, we use coefficients of restitution, thinking about how different materials respond.

Perfectly steel balls (e = 0) lose all their speed in the normal direction to the surface of collision. Perfectly rubber balls (e = 1) retain all of their speed in this normal direction but inversed – creating a bounce. Both maintain the same velocity along the surface of collision. Therefore, our collider needs to output both the response calculated above and the normal to the surface of collision.

```
public bool Collision(out Vector response, out Vector normal) {…}
```

By passing the closest point and the normal to the plane's surface through the collider's functions in a struct, we keep track of the normal and output it along with the boolean.

We resolved the physics object's velocity into the vector component along the normal and the component perpendicular to the normal (along the surface):

$$\underline{v_n} = (\underline{u} \cdot \underline{n}) \cdot \underline{n}$$

$$\underline{v_s} = \underline{u} - \underline{v_n}$$

Note, its original velocity is simply the sum of these vectors:

$$\underline{u} = \underline{v_n} + \underline{v_s}$$

But by applying the coefficient of restitution to the normal part and inverting it, we gain the desired bounce response or at least halt its speed in the normal direction, preventing it eventually clipping through the mesh:

$$\underline{v} = -e\underline{v_n} + \underline{v_s}$$

5. Evaluation

5.1. Evaluation Methods

The project sets out with purpose of proposing the idea that 'terrain systems could be constructed using fractals, rendered in a fraction of the time, which require minimal storage and come instantly pre-packaged with an exponentially efficient but still fully accurate collider, whilst ensuring there is control over the modelling process.' This should be broken down into the following components of success to determine how realistic and feasible the idea is, hence reflecting the structure of the methodologies above:

- Rendering
- Storage
- Collision and response
- Control – see 6.1. This marked the need for a language as constructed in 4.2.

We set out to explore four areas of evaluation: theoretical, from the methodologies; performative, based on implementation and observations; visual, reflecting perspective observation that goes beyond the statistics and applies to how we really use and interact with software and finally a discussion, mainly regarding the control component of success. This isn't about mathematics or performance but feasibility by determining whether meshes of ranging variety and complexity can genuinely be easily modelled.

5.2. Theoretical Evaluation

We'll produce theoretical computational complexity analysis and a mathematical evaluation. If a strong indicator of potential benefit is found here, we're likely highlighting an advantage, especially in terms of time complexity.

A fractal is made up of many transformations of a base mesh.

Let $b =$ number of faces in the base mesh.

Let $t =$ number of transformations in a single depth of the fractal.

Let $d =$ depth of the fractal.

Hence, the number of faces in a fractal is $b \cdot t^d$. If we wish to compare this to a regular mesh then we ideally need to pick a mesh with $b \cdot t^d$ faces.

5.2.1. Rendering

When loading a model, each vector is processed by the GPU in parallel which allows us to render models of hundreds of thousands of vertices. In an ideal GPU with unlimited cores, the time complexity is $O(1)$ – just give each core a vertex to process, but in a realistic GPU with P cores, the time complexity of a regular mesh that has $b \cdot t^d$ faces is $O(b \cdot t^d / P)$, since the number of faces is proportional to the number of vertices.

By GPU instancing fractals of the same face count (where a fractal is t^d instances of a base mesh), we process the base mesh only once on the GPU in $O(b)$ time, then for each of the t^d instances, we apply a transformation matrix in the GPU's shader program, in parallel, taking time $O(t^d / P)$. The overall time complexity reduces from $O(b \cdot t^d / P)$ to $O(b + t^d / P)$.

The number of faces in the base mesh of a fractal is intended to be low, i.e. under 50 sounds reasonable. We make a simple, low-cost object then build up many transformations of it to create a truly detailed mesh, hence this reduction in processing is somewhat valuable, e.g. divide rendering time by 50, but the true benefit comes from efficient rendering and memory use.

For regular meshes, the entire geometry is stored in memory: $b \cdot t^d$, since every face could be unique and there's no opportunity for reuse. But by instancing, only one copy of the small <50 face base mesh is stored in memory: b which the GPU then reuses. This is a dramatic save in GPU memory bandwidth (loading b faces vs $b \cdot t^d$ faces). For a mesh of hundreds of thousands of vertices, that's tons of data that will saturate bandwidth, even with cache, but <50 faces is insignificant. Instancing also has less vertex shader invocations and improved rasterisation.

5.2.2. Storage

Refer to the context-free grammar for .obj files in 4.1.1. For each vertex, we store 'v' followed by 3 floats and a new line. For each face, we store 'f' followed by 3 integers and a newline. Including spaces, that totals to $(5 + 3 \cdot sizeof(float))$ characters per vertex and $(5 + 3 \cdot sizeof(int))$ characters per face.

Faces are composed of vertices and many vertices are shared between adjacent faces, hence the total number of vertices in a mesh is always at least as many as the number of faces – likely more. Therefore a 1 million face mesh, has a file size of at least $(5 + 3 \cdot sizeof(float)) \cdot 1{,}000{,}000 + (5 + 3 \cdot sizeof(int)) \cdot 1{,}000{,}000$ characters. Even if we take $sizeof(float)$ and $sizeof(int)$ to each be 1 character of text, that is a file size of 16 million characters or 16Mb – an extreme underestimate. (In 5.3, we see a mesh of 237,664 faces has a file size of 36.9Mb)

A fractal is made up of a base mesh .obj file and a .fdo file to store the sets of transformations in a readable format (the language we wrote in 4.2), see appendix B for reference. A base mesh will have <50 faces, so up to 800 bytes using the approximation above and even a .fdo file with many sets of transformations is about 1Kb in storage. It's minimal lines of text and irrelevant of a fractal's depth. Hence, increasing the depth of a fractal in the .fdo file and scaling it to a 1 billion face mesh will not change the file size, since only a single integer in the file is changed and the base mesh remains the same.

This means fractals can reduce files tens of megabytes in size to a couple of kilobytes at most. Also, by storing them in this format, we can make large changes to them easily. By making a simple edit to the base mesh, we can modify the entire fractal. Likewise, we can add in an extra set of transformations, remove a set of transformations or increase / decrease the fractals depth, changing the entire model. This just is not possible when storing objects in a long list of vertices and faces as we would have to manually change every one.

5.2.3. Collision and response

A simple collider contains a function that calculates the closest point on a face to the collider's position. It simply iterates through all the faces of an object to determine the closest point in the entire mesh, see 4.4.2. A mesh with millions of faces would require millions of operations per frame, as it computes the closest point on each face – far too computationally expensive.

Given f is the number of primitive operations to calculate the closest point on a face to the collider's position, the collision cost (number of primitive operations each frame to determine a collision and response) of a mesh that has $b \cdot t^d$ faces, using a simple collider, is $b \cdot t^d \cdot f$. However, our implementation in 4.4.3, reduces this exponentially when used with fractals that also have $b \cdot t^d$ faces.

In 4.4.3, we recursively find the closest point on a fractal. At each depth, we iterate through all t transformations and select the one with the closest point to the collider's position. To do so, we consider each transformation as a base mesh and compute the closest point, incurring the cost of $b \cdot f$ primitive operations per transformation. That is $b \cdot f \cdot t$ work to evaluate all transformations within a single depth and select the one with the closest point. We consider this as the new fractal, exponentially ignoring all other branches and repeat for the depth of the fractal, d, totalling to $b \cdot f \cdot t \cdot d$ primitive operations.

This reduces the time complexity of collision and response from $O\left(b \cdot t^d \cdot f\right)$ for the simple collider to $O(b \cdot t \cdot d \cdot f)$ for the fractal collider. However, since b and f are small (the base mesh of a fractal should be composed of <50 faces and the cost to calculate the closest point on a face is simply the vector cross product, possibly with some barycentric coordinate conversion and a few more vector operations), we can treat them as constants and simplify the time complexities to $O\left(t^d\right)$, $O(t \cdot d)$ respectively (exponentially faster).

Worked example:

A fractal with
- a base mesh composed of 10 faces, $(b = 10)$
- 50 transformations in a single depth $(t = 50)$
- and a depth of 4 $(d = 4)$

has $10 \cdot 50^4$ faces = 62.5M faces.
- Simple collider cost $= b \cdot t^d \cdot f = 10 \cdot 50^4 \cdot f = 62{,}500{,}000f$
- Fractal collider cost $= b \cdot t \cdot d \cdot f = 10 \cdot 50 \cdot 4 \cdot f = 2{,}000f$

This is x31,250 faster!

This makes sense – fractals are recursive geometries so it sounds reasonable to find an exponential time improvement. However, we must remember our simple collider is extremely expensive and existing collision detection algorithms exist which make improvements on this, e.g. bounding boxes. The difference is these diminish accuracy to gain greater efficiency. They are a compromise of accuracy and efficiency and hence where many collision bugs in video games arise. Our fractal collider achieves an exponentially faster speed, whilst maintaining full accuracy. The drawback: we must use fractal geometries, see 6.1.

5.3. Performative Evaluation

We extend our theoretical evaluation to test on real technology. Most notably, we're actually using the graphics card (crucial to optimisation through parallel processing) and testing the effects of GPU-instancing. This way, we can account for hardware limits and real-world costs like memory use, draw calls and GPU load which steps away from our idealised model.

We'll select a range of models for testing, choosing a mix of fractals and non-fractals with varying mesh counts, from low to high detail, evaluating performance at different levels of complexity, see figure 16.

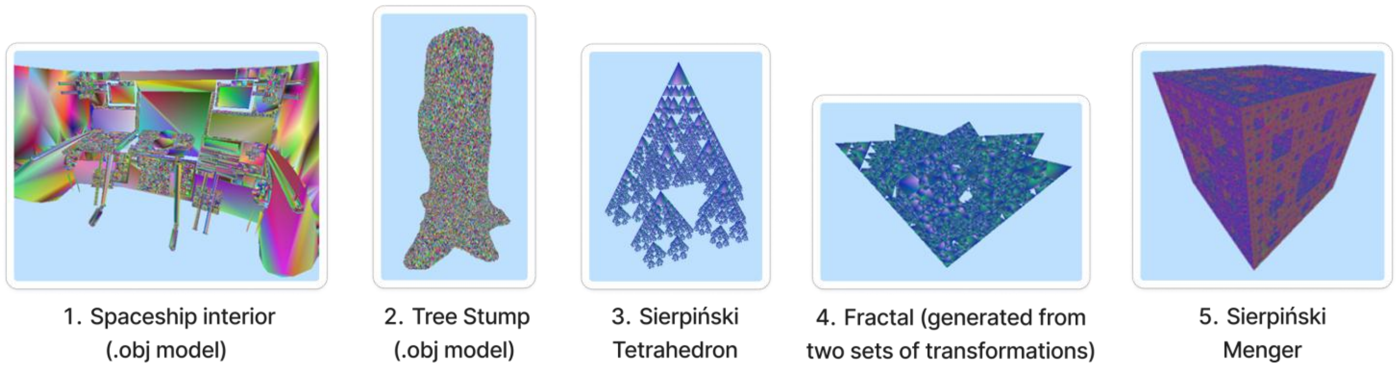

1. Spaceship interior (.obj model)
2. Tree Stump (.obj model)
3. Sierpiński Tetrahedron
4. Fractal (generated from two sets of transformations)
5. Sierpiński Menger

Figure 16. Models used for evaluation. The first two are regular objects. Mesh 2 was found online [18] and specifically chosen due to its natural properties (it being a tree stump), meaning it could likely be constructed using fractals and so should prove an interesting comparison to mesh 3, of similar face count. The last three are fractals generated in the project. Mesh 3 and 5 are generated from a single set of transformations (it is clear mesh 3 is a simple tree structure) whereas mesh 4 is generated from two (that are a tree structure and a diamond). These are less obvious but the mesh is essentially a tree that is caved in on itself. When generated from multiple sets of transformations, it may become less clear what this mesh is (which allows for much more visual complexity, hence the need for the language produced in 4.2). Note, mesh 5 has an exceedingly high mesh count (38.4M faces!) but surprisingly this prooves fine.

Then, evaluate our pre-described components of success by implementing the following strategies:

Rendering – By adding a stopwatch to our implementation, we can record the load time of meshes and output to the console. We ensure to load only a single mesh into the scene at once and take an average of 10 attempts to gain accurate results.

Storage – We'll peek at the file sizes of the models. For meshes 1 and 2, they are simply a .obj file and can directly note down their storage size. For meshes 3, 4 and 5, these are fractals broken into their base .obj file and the .fdo file which stores the sets of transformations in a readable format (the language we wrote in 4.2). We'll note down the sum of their sizes.

Collision and response – Each time the base collider's ClosestPointFace function, to which all colliders inherit, is called, we'll increment an operation counter. This will then record the number of closest point on face calculations made each frame and will output to the console in through the physics object to which the collider is attached. Given f is the number of primitive operations to calculate the closest point on a face to the collider's position, the collision cost (number of primitive operations each frame to determine a collision and response) will be our output multiplied by f.

Model ID	1	2	3	4	5
Fractal? (Y/N)	N	N	Y	Y	Y
Face count (complexity of mesh)	277,654	237,664	168,070 (10*7^5)	1,176,490 (10*49^3)	38,400,000 (12*20^5)
Rendering cost (average load time for 10 attempts in seconds)	85.577	74.467	0.231	0.075	0.741
Storage (bytes)	10,799,578	36,871,183	749 (388+361)	1,176 (388+788)	1,049 (384+665)
Collision cost (number of closest point on face calculations)	277,654	237,664	420	1,470	1,200

Figure 17. Table comparing results for the models in figure 16. As face count increases, we expect rendering cost, storage and collision cost to also. The fact fractal models are demonstrating significantly lower costs despite greater complexity suggests that their recursive nature can provide far more efficient computation across the board. Note, this is tested using a laptop's RTX4070 GPU; we use our fractal collider for fractals and our (inefficient but fully accurate) simple collider for regular meshes.

Figure 17 shows the results. Fractals perform wildly more efficient. Even a fractal mesh of 38.4 million faces renders in a fraction of a second, takes up minuscule storage and has barely a collision cost! Section 5.2, really breaks down the reasoning as to why this is happening and may not be too intuitive, for instance it isn't predominantly the reduction in time complexity from GPU instancing which is causing this dramatic reduction in rendering cost, but more related to the memory bandwidth which is being saved, so I strongly suggest referring to this for an explanation.

There are some things to note, however. It appears strange that the 1.17M face fractal is rendering faster than the 116k face fractal. This is likely just tiny inaccuracies caused by granularities (backround tasks, CPU scheduling, etc). Every time the solution was run, it gave values a couple 0.1s apart. We're talking fractions of seconds and really insignificant when comparing with meshes taking over a minute to load. Also, the storage size of mesh 2 seems a little high, especially when mesh 1, whose face count is similar, is stored in a third of the size. This was a model found online whose .obj file did seem to be storing a large amount of unnecessary data. Just running through the file, it appears that a cleaner save could half its storage size, though still is largely off the size of the fractals.

Interestingly, the collision operation counter is exactly equal to the maths described in 5.2.3. For instance, mesh 3 is a fractal with $b = 10$, $t = 7$, $d = 5$ and its collision cost $= 10 \cdot 7 \cdot 5 = 420$.

5.4. Alternative Collision Approach: Bounding Boxes

We build a bounding box octree class to subdivide a 3D space of an array of vertices into small bounding boxes. These can be represented by simply a minimum and maximum vector, where the span of these is the span of the bounding box. It follows that the centre of a bounding box is the average of its minimum and maximum vectors. Likewise, all other vertices on the box can too be derived from just a combination of these two vectors. Our octree is then a root node, which itself stores a bounding box and 8 children nodes and we set it to a sensible depth.

We construct a bounding box collider which instantiates an octree for each object in a scene by passing in their vertices. Then, we recursively traverse through the tree finding the closest point on each bounding box in a single depth; note down the node with the closest bounding box; exponentially ignore all other nodes and repeat considering that node as the new tree, for the depth of the tree. This extends our collider class, so we reuse functionality and allow for easy selection of colliders in each physics object. It is a very similar approach to fractal collision, described in 4.4.3, so can expect a similar time complexity. The large difference, however, is bounding box octrees must be constructed for each object, which takes some processing also. Figure 18 shows the results.

Model ID	1	2	3	4	5
Fractal? (Y/N)	N	N	Y	Y	Y
Face count (complexity of mesh)	277,654	237,664	168,070	1,176,490	38,400,000
Simple collider collision cost	277,654	237,664	168,070	1,176,490	38,400,000
Fractal collider collision cost	N/A	N/A	420	1,470	1,200
Bounding box collider collision cost	1,736	1,714	1,666	1,936	2,419

Figure 18. Table comparing the alternate collision approaches. Both simple and fractal colliders are fully accurate as don't approximate meshes, whereas bounding boxes do, diminishing accuracy. The drawback of fractal colliders is they must be used on fractals as exploits their recursive geometry. Still, fractal colliders prove more effective than bounding boxes, though only slightly, in terms of collision cost but largely excel in accuracy. Note, depth of bounding box octree is taken as $3 \cdot log_8\, FaceCount$.

5.5. Visual Evaluation

Some aspects – especially the accuracy of a collision – are best measured by the human eye that can easily perceive quality and spot clear errors (e.g. colliders clipping through one another, rendering lag). We'll implement shrinking for physics objects to get a good view of this and assess whether they can collide appropriately with the intricate grooves of fractals, whilst colliding appropriately, as well as comment on some intuitive visual benefits / drawbacks of the solution.

Each frame, we reduce the physics object's global matrix's scale and its collider's radius by a factor of the change in time during that frame. We wrap this in a shrinking method that can be enabled / disabled when required.

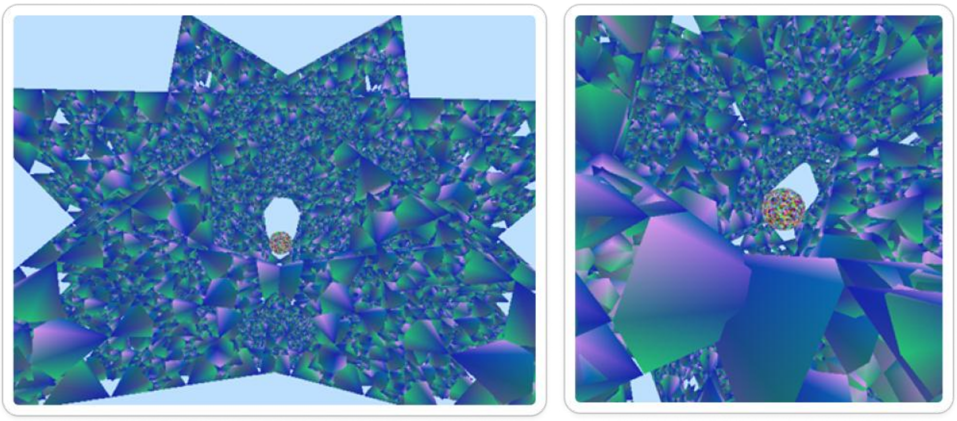


Figure 19. View of a collision between a physics object and a generated fractal. Here the object has collided with the mesh, bounced around a bit and fell into one of its intricate grooves. Collision is accurate – the object collides precisely, not clipping into the fractal. We visually see it respond to collision by bouncing from face to face, eventually settling inside the mesh. As it shrinks, it soon passes through the gaps in the fractal, falling to collide once more with its interior and roll of the mesh entirely. It's exciting to see real-time collision with a 1.18 million face mesh, especially as it enters the intricate details of the model!

The results in figure 19 do not follow for bounding boxes. Although not largely noticeable with small geometry, the physics object failed to make proper contact during collisions, often responding too early, as shown in figure 20. This would be a stronger concern for large terrains with rough landscape and hills, whose surfaces are complex, as would present visibly unrealistic physics.

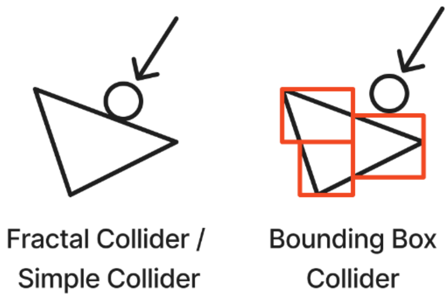


Figure 20. Diagram comparing the accuracy of colliders. The bounding box acts as an approximation so the physics objects respond too early in the collision, whereas when using fractals or the fully accurate but inefficient simple collider, response is provided correctly.

6. Discussion

6.1. Reflection and Application of Fractals in Graphics

Fractal geometry demonstrates significantly greater efficiency to regular meshes. The concern then lies in how far their application can realistically and feasibly be pushed. [14] showcases examples of complex fractals in nature, see figure 21. Such grand detail can be produced easily and can be taken further to generate entire mountain ranges, islands, caves, cloud systems or snow with complex ice crystals. Video games can have larger, more detailed, open, explorable worlds without killing performance that seem organic and endless. By combining geometries, we could produce forest covered cliffs or rock formations on coastline, adding greater visual depth to landscapes. Plus, with the infinite detail our fractals provide from 4.3 comes unbelievable visual detail, immersion and potential for massive world building.

Figure 21. Some exciting appearances of fractals in nature from [14]. Romanesco broccoli (image 1) looks almost alien-like with a clear set of transformations (if this was to be constructed digitally). Pinecones and tree branches bring more realism. We can imagine constructing terrains with these and their sets of transformations / base objects are less apparent.

This relies on control of modelling – meshes of ranging variety and complexity can be easily modelled. Our language in 4.2 allows this. It adds higher-level control and flexibility for users to experiment with various effects and patterns without being constrained by hardcoding intricate sets of matrices and provides readability in higher-level instructions rather than low-level transformational data.

Most importantly, the language allows for n sets of transformations, as described in in figure 22. This dramatically improves visual complexity as many sets of transformations, then iterated to a set depth, can create a resultant mesh far from its original base model. Notice how for each fractal in figure 21, it's unclear what their bash mesh could be – their shape is simply too complex to tell – unlike the simple Sierpiński tetrahedron where its base mesh is abundantly clear to be a tetrahedron. This is what our language provides and mesh 4 in figure 16 begins to demonstrate this. 4.2.1 elaborates on this with discussion of the transformations involved.

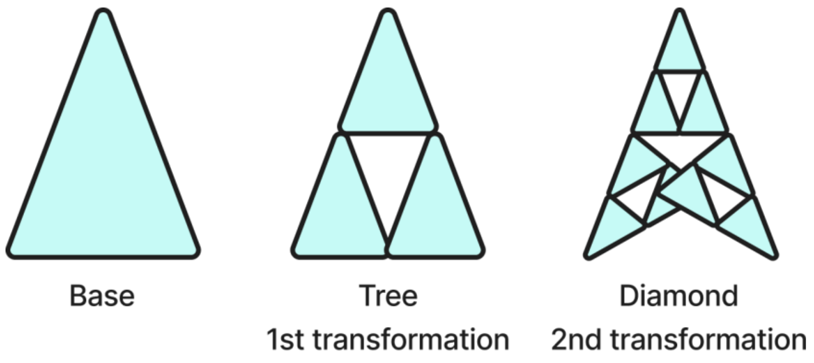


Figure 22. Illustration of the process of constructing a fractal made of two sets of transformations. First, we duplicate and translate the base object to form a tree structure, then duplicate, translate and further rotate this tree structure to produce a diamond. The process is repeated for the depth of the fractal, each iteration considering our built up diamond as a base mesh.

6.2. Prospects for Further Research

The project could be extended for fractal-optimised lighting. It is likely possible to approximate global illumination and ambient occlusion, capturing subtle lighting details that methods for regular meshes may miss or be computationally costly to replicate. [15] demonstrated fractal-based procedural techniques offer powerful tools for simulating natural phenomena making them valuable for advancing lighting systems in real-time rendering engines.

Since we have found an optimal way of calculating the closest point on a factal to a position in space in 4.4.3, I strongly believe this could be extended beyond collision, to finding the exact point where a ray of light hits a fractal mesh, in exponentially efficient time. Paired with collision, storage and rendering reduction in costs, this could potentially prove as a powerful improvement in 3D graphics. Less computation means that power can be spent on grander detail and larger environments.

The advantages gained are only achievable for fractal geometry. An interesting continuation could be taking any regular mesh (.obj file) and approximating it as a fractal (.fdo file with a small base .obj) via implementation of a neural network. This would be challenging as would require rewriting 3D models through AI, but if implemented would result in a huge payout as described by the results of the evaluation.

Note, it is impossible to turn most regular meshes into fractals: not all meshes are recursive geometries, but all recursive geometries are by intuition meshes, meaning that it is required that the neural network uses an approximation. This does diminish the accuracy of collision, which was the motivation for using fractals and the large contributor to its visual success, however it would remove the need for further processing to generate bounding boxes and comes with all the efficiency of rendering cost and storage outlined in 5.1, 5.2.

6.3. Legal, Social, Ethical and Professional Issues

The project complies with copyright laws and respects intellectual property rights. Legal consideration was taken when finding obj files used for testing, ensuring all were open-source, royalty-free, or created by myself to avoid copyright infringement. I properly cited all sources, including any code and research papers to avoid plagiarism.

Usability is a primary social consideration. The engine requires an intuitive interface to ensure accessibility for users with varying technical backgrounds. This extends to fractal generation where we demand a platform for users to interface. Fractals are generated from a set of sets of matrix transformations and a base obj file though the project prioritised readability by allowing users to work with higher-level instructions, rather than directly with low-level transformational data. For social consideration, the code files are packaged with clear intended-use documentation and instructions in a user manual file outlining its functionality. Also, the project is intended to be open source.

Ethical considerations are central throughout the project. Academic integrity was upheld by rigorously citing sources, ensuring original code, and maintaining transparency about results and limitations when it came to testing the efficiency of the fractal collision and response algorithm. These were documented in detail and produced via alternate methods: incrementing a counter in code to count the number of processes executed (hence mitigates fabrication or misinterpretation of results as takes out the human factor) and judged by the feel of the software when interacting with a fractal mesh as a player character (to still provide some human factor in results – much collision accuracy can only be interpreted by the smoothness of the interaction). To mitigate bias, all tests were run against an efficient collision detection algorithm (bounding boxes) and results were based on an average of many tests.

Professionalism was demonstrated through rigorous testing of the engine to validate its performance and reliability, throughout. By maintaining transparent documentation and ensuring all sources were properly cited, the project adhered to high professional standards. Regular reviews of progress through an agile methodology helped mitigate the risk of time management to complete sprints and data loss was be prevented using version control. Code followed design principles (e.g. fractal generator is a factory for fractals) and was largely imperative using lambda calculus's for readability and efficient debugging.

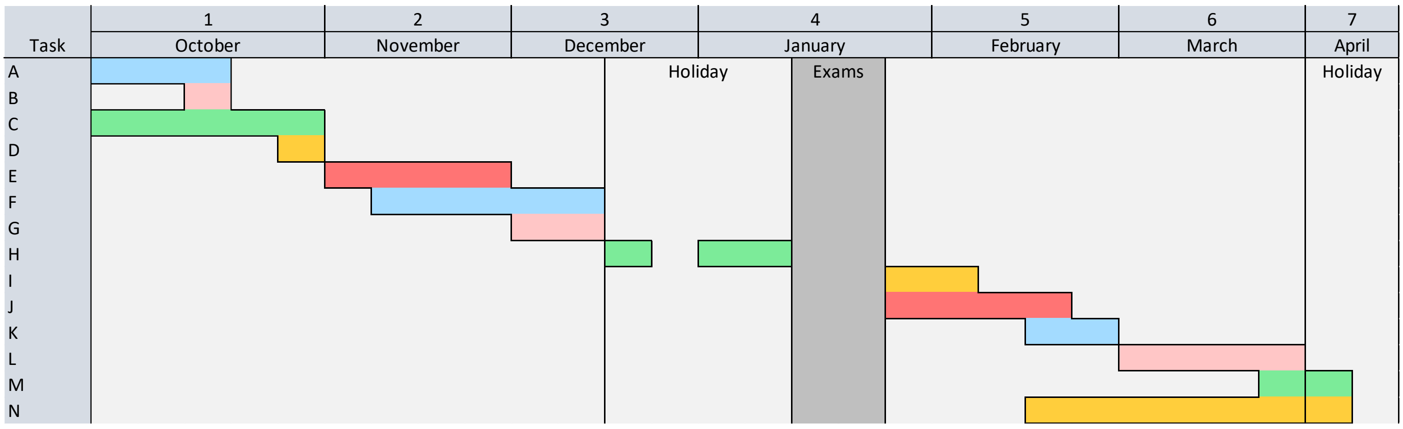
6.4. Project Plan and Progress

I have fully implemented and evaluated all methodologies following an agile methodology including sprints laid out below, meetings with my project supervisor and reviews of progress. I knew the project would be large given I was attempting to produce a graphics engine to not only load and render object files but generate fractal geometry and their colliders in real time. Additionally, learning OpenGL and how to interface with the graphics card through shaders along with creating my own math libraries to handle 3D-to-2D mapping, projection, and object transformations from scratch, I anticipated, would be time-consuming. However, I planned accordingly and dedicated much time to the project each week. Though, from doing so, I've gained invaluable knowledge, which has significantly enhanced my understanding of graphics engines, techniques to improve abstraction and readability and the potential of fractal geometries and how they are constructed.

I expanded my project to include infinite dynamic fractal rendering and a physics engine used for collisions involving coefficients of restitution. I successfully divided the workload in half to split between the two terms and promptly began research and implementation as I expected the maths behind fractal collisions and comparing it to alternate collision detection and response algorithms on various meshes would consume much time – which it very much did.

My initial project plan didn't feature fractal generation in first semester as I greatly underestimated the complexity of this task (discussed in 4.2.1, the need for a parser) upon research. Realising this, I worked solidly on finishing the graphics engine early and began the task of generating fractals to mitigate the risk of the project falling to failure, in the case I struggled to generate them. I also swapped lighting and textures with fractal generation, infinite dynamic fractal rendering and the physics engine as believed it to be more valuable, producing new innovative design rather than re-implementing existing graphics algorithms. In a fractal-based project, I felt it was imperative to produce a render of a fractal in the first semester rather than leave it too late, so prioritised this. This is the sort of reviews of progress I conducted as part of following an agile methodology.

My revised Gantt chart:



First Semester

A Decide on a project idea and a graphics API (OpenGL) and research how to use it alongside the strengths and weaknesses of existing graphics engines (Blender, Unity etc).

B Load a simple obj file storing a 3D model and shader programs into the graphics card.

C Create classes to handle the maths required to map from the global axes to a camera's local axes and create a camera object which handles rendering using uniforms as communication between CPU and GPU. Respond to user input, to produce my first render, providing both orthographic and perspective projection.

D Complete project proposal.

E Load multiple obj files at once into the same scene. Incorporate a z-buffer depth algorithm to determine closest face per pixel.

F Research into whether fractal collisions have been studied much before. Generate 3D fractal geometry using code, taking in a base case low-mesh 3D model and the fractal pattern which will inductively build up the structure.

G Write interim report.

H Research and implement efficient methods for collision detection of regular 3D structures.

Second Semester

I Produce infinite dynamic fractals, reloading GPU buffer data based on the camera's position.

J Design the algorithm to efficiently calculate collisions with fractal geometry, backed up by mathematical proof.

K Implement this approach.

L Validate my code by creating physics objects to interact with the geometry and ensure my algorithm uses the recursive property of fractals to exponentially decrease time to calculate collisions, i.e. by running tests with various different 3D object files of different mesh counts. How does the collision cost of a 1 million vertex fractal model using my algorithm compare with that of a regular 1 million vertex model using a regular collision method?

M Improve the UI.

N Write final dissertation and work on demonstration video.

7. References

[1] Ericson, C., 2004. *Real-time collision detection*. Crc Press.

[2] Online: DemandSage. "Fortnite Statistics 2024 (Active Players & Revenue)." *DemandSage*, https://www.demandsage.com/fortnite-statistics/. (Accessed: 8 December 2024).

[3] Ernst, F., Heidrich, W., and Seidel, H. P. (1999). A Survey of Bounding Volume Hierarchies and Their Applications. Computer Graphics Forum, 18(4), pp. 123-134.

[4] Gottschalk, S., Lin, M.C. and Manocha, D., 1996, August. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (pp. 171-180).

[5] Online: IBM. "Benoit B. Mandelbrot." *IBM History*, https://www.ibm.com/history/benoit-mandelbrot. (Accessed: 8 December 2024).

[6] Online: Encyclopaedia Britannica. "Benoit Mandelbrot." *Britannica*, https://www.britannica.com/biography/Benoit-Mandelbrot. (Accessed: 8 December 2024).

[7] Steam Revenue Calculator (n.d.) *No Man's Sky*. Available at: https://steam-revenue-calculator.com/app/275850/no-man's-sky (Accessed: 8 December 2024).

[8] Perlin, K. (1985) 'An image synthesizer', Computer Graphics (ACM), 19(3), pp. 287–296.

[9] Togelius, J. and Yannakakis, G.N. (2007) 'Towards optimized procedural content generation', Proceedings of the 4th AIIDE Conference, pp. 161–166.

[10] Green, S. (2010) 'Fractal rendering using GPU shaders', NVIDIA Developer Blog. Available at: https://developer.nvidia.com (Accessed: 8 December 2024).

[11] Glencross, M., Pattanaik, S., and Hubbold, R. (2002). Efficient Collision Detection for Interactive 3D Graphics Applications. ACM Transactions on Graphics, 21(3), pp. 333-352.

[12] Klosowski, J. T., Held, M., Mitchell, J. S. B., Sowizral, H., and Zikan, K. (1998). Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. IEEE Transactions on Visualization and Computer Graphics, 4(1), pp. 21-36.

[13] Smith, J., 2020. Introduction to Computer Graphics and GPU Programming. Cambridge University Press.

[14] Pearson, C. (2021) 9 amazing fractals found in nature. Treehugger. Available at: https://www.treehugger.com/amazing-fractals-found-in-nature-4868776 (Accessed: 11 April 2025).

[15] Musgrave, F. K., Kolb, C. E., & Mace, R. S. (1989). The synthesis and rendering of eroded fractal terrains. ACM SIGGRAPH Computer Graphics, 23(3), pp. 41–50.

Images of fractals:

[16] Reddit, u/proceduralgeneration (2021) 3D fractal art created in Mandelbulber. Available at: https://www.reddit.com/r/proceduralgeneration/comments/rsnq4q/3d_fractal_art_created_in_mandelbulber/ (Accessed: 11 December 2024).

[17] Discourse McNeel, McNeel Community (2023) 3D Printing Vase Mode for Fractals. Available at: https://discourse.mcneel.com/t/3d-printing-vase-mode-for-fractals/159007 (Accessed: 11 December 2024).

Model of tree stump:

[18] Available at: https://www.thingiverse.com/thing:2824918 (Accessed: 11 April 2025)

8. Appendix

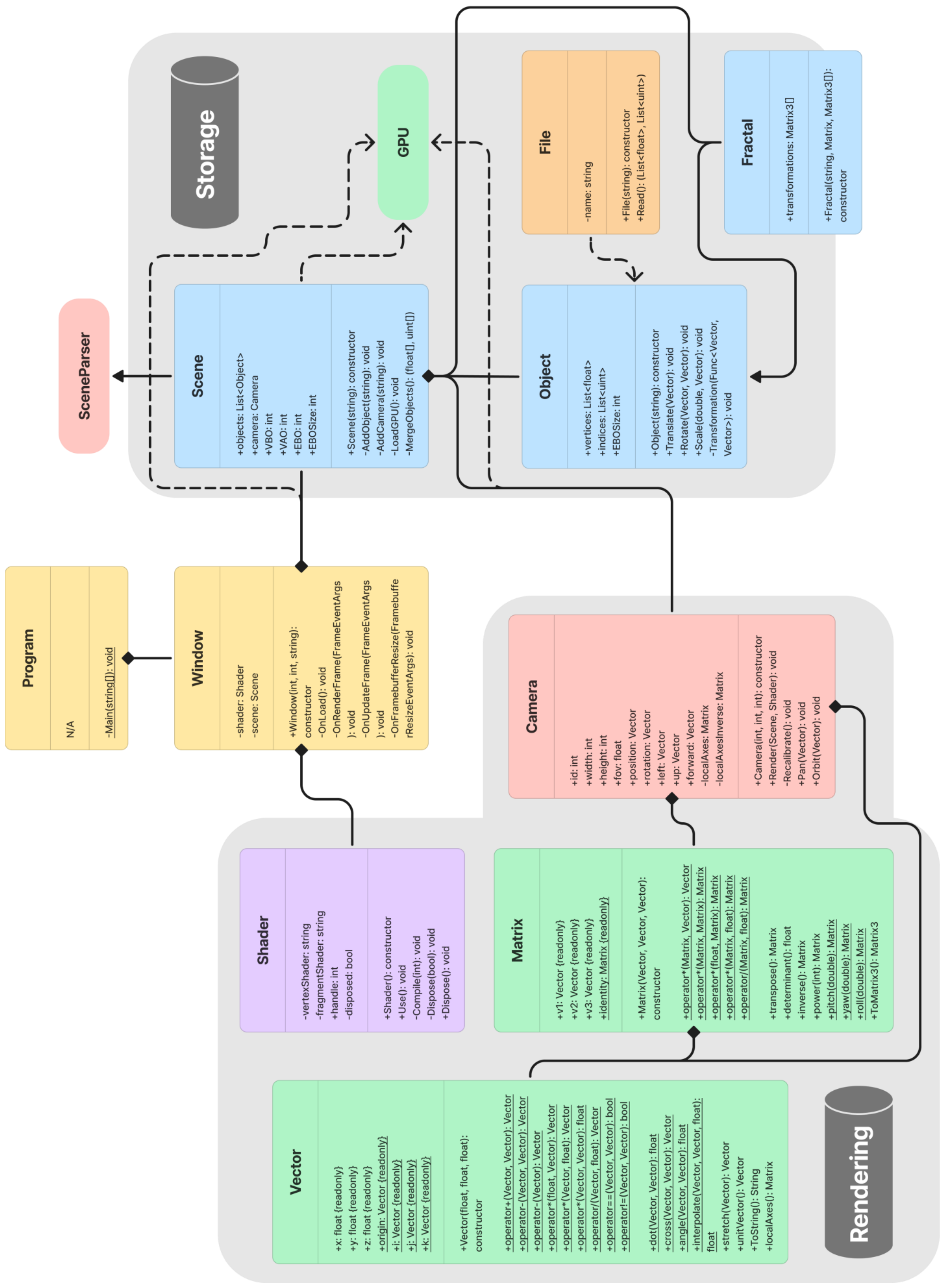
Appendix A. Example sdo (scene data object) file

```
# Scene Data

# Camera attributes
c (-48.761894, 25.567242, 64.429054) (-0.071402885, -2.4999454, 0) 60

# Object file paths and transformations.
o "SampleAsset1.obj" ((1, 1, 1), (90, 0, 30), #)

o "SampleAsset2.obj" #
```

Appendix B. Example fdo (fractal data object) file

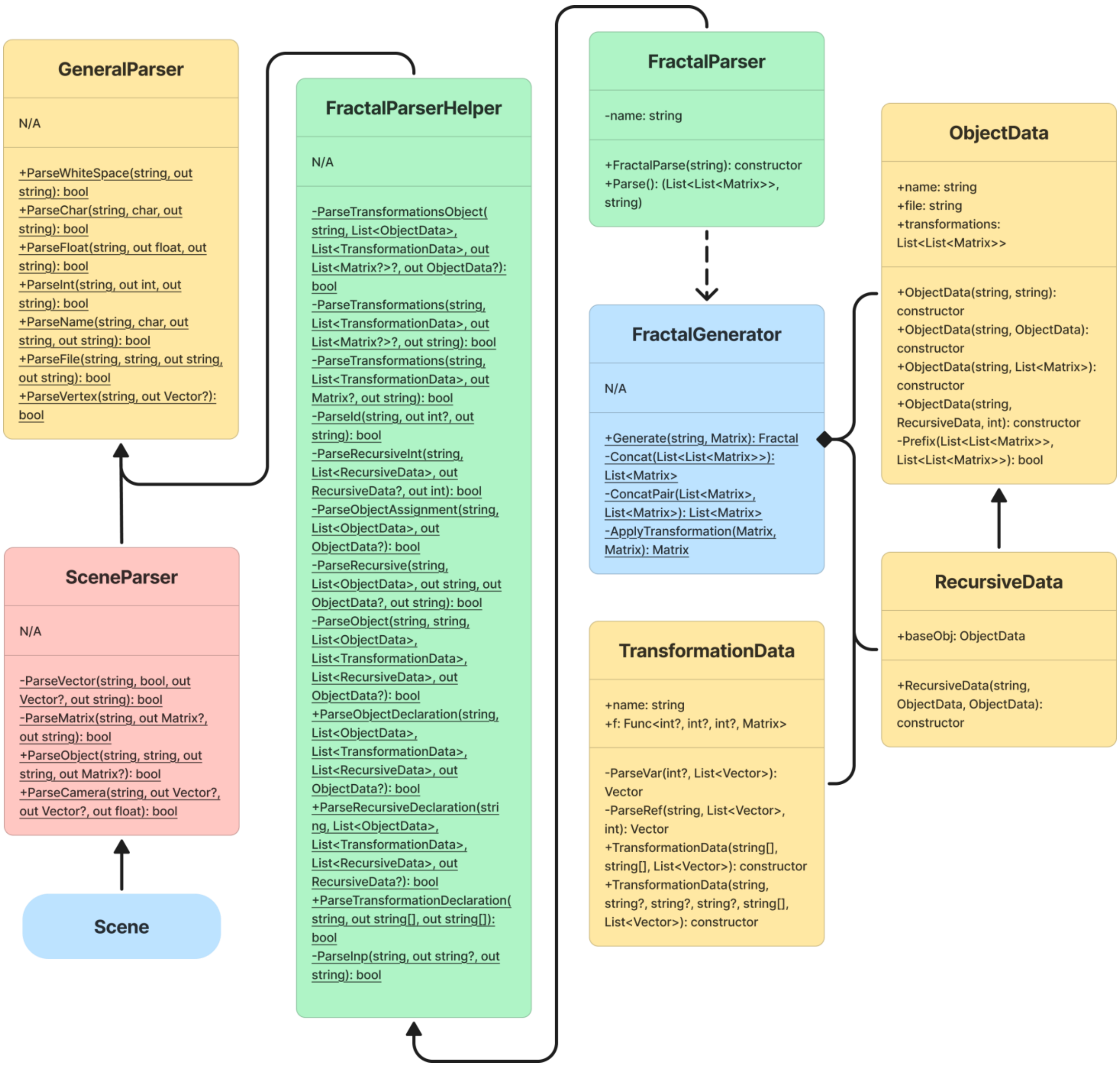
```
# Fractal Data Object

# Vector transformations
v 1 1 -2
v 1 -1 -2
v -1 1 -2
v -1 -1 -2
v 0.5 0.5 0.5

# Base Object
x. "FractalBase.obj"

# Transformation function indexing vertices above
t.a. (a, #, 5)

y. [#, t.1, t.2, t.3, t.4] x

v 10 10 -10
v 10 -10 -10
v -10 10 -10
v -10 -10 -10

v 135 45 0
v 135 -45 0
v -135 45 0
v -135 -45 0
v 0.1, 0.1, 0.1

u.a.b. (a, b, 14)

# Recursive object
z<x>. [#, u.6.10, u.7.11, u.8.12, u.9.13] y

# Returns z as a finite fractal of depth 6.
<$>. z 6
```

Appendix C. Class diagram for the graphics engine based on the design discussed in 4.1 and 4.2.



Storage
GPU
SceneParser
Scene
+objects: List<Object>
+camera: Camera
+VBO: int
+VAO: int
+EBO: int
+EBOSize: int
+Scene(string): constructor
-AddObject(string): void
-AddCamera(string): void
-LoadGPU(): void
-MergeObjects(): (float[], uint[])
File
-name: string
+File(string): constructor
+Read(): (List<float>, List<uint>)
Fractal
+transformations: Matrix3[]
+Fractal(string, Matrix, Matrix3[]): constructor
Object
+vertices: List<float>
+indices: List<uint>
+EBOSize: int
+Object(string): constructor
+Translate(Vector): void
+Rotate(Vector, Vector): void
+Scale(double, Vector): void
-Transformation(Func<Vector, Vector>): void
Program
N/A
-Main(string[]): void
Window
-shader: Shader
-scene: Scene
+Window(int, int, string): constructor
-OnLoad(): void
-OnRenderFrame(FrameEventArgs): void
-OnUpdateFrame(FrameEventArgs): void
-OnFramebufferResize(FramebufferResizeEventArgs): void
Camera
+id: int
+width: int
+height: int
+fov: float
+position: Vector
+rotation: Vector
+left: Vector
+up: Vector
+forward: Vector
-localAxes: Matrix
-localAxesInverse: Matrix
+Camera(int, int, int): constructor
+Render(Scene, Shader): void
-Recalibrate(): void
+Pan(Vector): void
+Orbit(Vector): void
Shader
-vertexShader: string
-fragmentShader: string
+handle: int
-disposed: bool
+Shader(): constructor
+Use(): void
-Compile(int): void
-Dispose(bool): void
+Dispose(): void
Matrix
+v1: Vector {readonly}
+v2: Vector {readonly}
+v3: Vector {readonly}
+identity: Matrix {readonly}
+Matrix(Vector, Vector, Vector): constructor
+operator*(Matrix, Vector): Vector
+operator*(Matrix, Matrix): Matrix
+operator*(float, Matrix): Matrix
+operator*(Matrix, float): Matrix
+operator/(Matrix, float): Matrix
+transpose(): Matrix
+determinant(): float
+inverse(): Matrix
+power(int): Matrix
+pitch(double): Matrix
+yaw(double): Matrix
+roll(double): Matrix
+ToMatrix3(): Matrix3
Vector
+x: float {readonly}
+y: float {readonly}
+z: float {readonly}
+origin: Vector {readonly}
+i: Vector {readonly}
+j: Vector {readonly}
+k: Vector {readonly}
+Vector(float, float, float): constructor
+operator+(Vector, Vector): Vector
+operator-(Vector, Vector): Vector
+operator-(Vector): Vector
+operator*(float, Vector): Vector
+operator*(Vector, float): Vector
+operator*(Vector, Vector): float
+operator/(Vector, float): Vector
+operator==(Vector, Vector): bool
+operator!=(Vector, Vector): bool
+dot(Vector, Vector): float
+cross(Vector, Vector): Vector
+angle(Vector, Vector): float
+interpolate(Vector, Vector, float): float
+stretch(Vector): Vector
+unitVector(): Vector
+ToString(): String
+localAxes(): Matrix
Rendering

Appendix D. Class diagram for the fractal generator and parsers based on the design discussed in 4.1 and 4.2.


GeneralParser
N/A
+ParseWhiteSpace(string, out string): bool
+ParseChar(string, char, out string): bool
+ParseFloat(string, out float, out string): bool
+ParseInt(string, out int, out string): bool
+ParseName(string, char, out string, out string): bool
+ParseFile(string, string, out string, out string): bool
+ParseVertex(string, out Vector?): bool
FractalParserHelper
N/A
-ParseTransformationsObject(string, List<ObjectData>, List<TransformationData>, out List<Matrix?>?, out ObjectData?): bool
-ParseTransformations(string, List<TransformationData>, out List<Matrix?>?, out string): bool
-ParseTransformations(string, List<TransformationData>, out Matrix?, out string): bool
-ParseId(string, out int?, out string): bool
-ParseRecursiveInt(string, List<RecursiveData>, out RecursiveData?, out int): bool
-ParseObjectAssignment(string, List<ObjectData>, out ObjectData?): bool
-ParseRecursive(string, List<ObjectData>, out string, out ObjectData?, out string): bool
-ParseObject(string, string, List<ObjectData>, List<TransformationData>, List<RecursiveData>, out ObjectData?): bool
+ParseObjectDeclaration(string, List<ObjectData>, List<TransformationData>, List<RecursiveData>, out ObjectData?): bool
+ParseRecursiveDeclaration(string, List<ObjectData>, List<TransformationData>, List<RecursiveData>, out RecursiveData?): bool
+ParseTransformationDeclaration(string, out string[], out string[]): bool
-ParseInp(string, out string?, out string): bool
FractalParser
-name: string
+FractalParse(string): constructor
+Parse(): (List<List<Matrix>>, string)
FractalGenerator
N/A
+Generate(string, Matrix): Fractal
-Concat(List<List<Matrix>>): List<Matrix>
-ConcatPair(List<Matrix>, List<Matrix>): List<Matrix>
-ApplyTransformation(Matrix, Matrix): Matrix
ObjectData
+name: string
+file: string
+transformations: List<List<Matrix>>
+ObjectData(string, string): constructor
+ObjectData(string, ObjectData): constructor
+ObjectData(string, List<Matrix>): constructor
+ObjectData(string, RecursiveData, int): constructor
-Prefix(List<List<Matrix>>, List<List<Matrix>>): bool
RecursiveData
+baseObj: ObjectData
+RecursiveData(string, ObjectData, ObjectData): constructor
TransformationData
+name: string
+f: Func<int?, int?, int?, Matrix>
-ParseVar(int?, List<Vector>): Vector
-ParseRef(string, List<Vector>, int): Vector
+TransformationData(string[], string[], List<Vector>): constructor
+TransformationData(string, string?, string?, string?, string[], List<Vector>): constructor
SceneParser
N/A
-ParseVector(string, bool, out Vector?, out string): bool
-ParseMatrix(string, out Matrix?, out string): bool
+ParseObject(string, string, out string, out Matrix?): bool
+ParseCamera(string, out Vector?, out Vector?, out float): bool
Scene