Software Design Document

Preface

This document will detail the following in relation to Legend Terrain 0.3: Framework Overview, Detailed Design, GameObjects, Graphics and Rendering, Input Handling, Events, and Examples. Lastly, the document will also detail the key changes from 0.1 to 0.3.

Created by Conor McDonagh Rollo

Computer Games Development (C00260445)

Table of Contents

Abstract

LegendTerrain is a C++ framework designed for procedural terrain and environment generation using OpenGL. It aims to provide developers with the tools necessary to create dynamic game worlds. This framework integrates landscape features through tweaked perlin noise, enabling the generation of procedural scenes.

Introduction

Legend Terrain is a project that has been worked on for the better part of 9 months, and has been through countless iterations and rewrites to perfect the vision that will be explained later on. Currently it is in a semi-usable state that would be comparable to that of an "Alpha" build.

Purpose

This document is for academic evaluation and also seeks to maintain the vision of the projects direction, should it ever again be worked upon. If somone wishes to use the resources in this project, examples will be provided as well as some essential tutorials.

Scope

The scope of Legend Terrain extends to some basic graphical rendering, input handling, advanced memory management, scene management, and automated gameplay looping. It is possible to extend the graphics rendering to utilise other shapes and complex meshes, input may also be easily extended to provide tools such as joystick control on 2 axis, etc. Memory management doesn't need to be touched but may be altered for your specific needs, and scene management can be easily extended to use spatial partitioning[1] for stabilising framerate. The GameObject class provides some simple events that are triggered in the scene manager, these can be extended and built upon, and can even include physics events should you include a physics library in the fork[2].

Usage

This document should be used as both a reference guide and as a tutorial. The Technical Design Doc can also be used in conjunction with this if you need to better understand the class relationship in the project. This document for the most part will be a reference guide, but if you skip to the examples and tutorials section then you will find what you are looking for faster. If you do not understand a certain word or phrase, you will likely find it in the Glossary, please note the subtext on those particular words as they will contain a number leading to the term in the Glossary.

Acknowledgements

Before starting this document, I would like to thank some particular individuals who have helped me throughout this project. *Joey de Vries,* for writing the LearnOpenGL website which helped me at the beginning of my project. *Milo Silvers,* for keeping my head held high during tough and stressful times. *Martin Harrigan,* for being an incredibly supportive supervisor on the project. And finally, my classmates for being a source of joy throughout the year. Thank you.

Framework Overview

The framework is still in its early stages of development and does not yet have a library that could be linked to, so all of the headers need to be included. This can be done by linking and downloading the source files of 0.3.

Architecture

The architectural style that is used in this framework relies on C++17[3] features that use inheritence[4] and polymorphism[5]. The engine does not use the likes of an MVC[6] or and ECS[7] as it is too early in development to use these, although an ECS[7] could be added very easily as the GameObjects are quite configurable. Instead, the engine uses specialised inheritence that makes use of the factory pattern. Lets imagine a Player class, the player inherits the Mesh class, which is something that you created to render a 3d mesh. The mesh class will then inherit the GameObject class, this way the Player can utilize both parents, but when the main function creates the Player, it will create a memory safe GameObject without access to the child classes, automatically creating a shared pointer[8] to itself and adding it to the current scene.

The abstraction of the instantiation process is used to allow for flexible programming while also keeping objects memory safe. This approach decouples the object creation from the actual class systems, making it easier to manage, extend, and maintain your codebase as it grows in complexity.

Core Features

The engine encapsulates all of the core features, and access to the core features should only be made available on the highest level of the application, unless you dedicate a specific manager to perform these tasks for you. The core features are as follows: GameObject, Camera, Input, Scene Manager, Shader, Terrain, and Textured Object. This section will explain the high level aspects of the classes.

- **GameObject** handles object management in a factory pattern[9]. It allows you to utilise certain virtual functions that trigger on specific events like update, start, or on object add. Each object has a default name of "GameObject", but this can be edited and set to something else if you have a need for finding an object by name in the scene.
- **Camera** has a first person look behaviour by default but this can be edited in the class itself. It contains real life camera controls such as YAW[10] ,PITCH[11] , ZOOM[12] , but you may also adjust its speed and sensitivity.
- **Input** is a fully static[13] class that handles the inputs that you map, the axis may also be mapped. The system updates itself in the game loop allowing you to check for button down, up, and held in any update function, and in any class you want. Mapping uses a string and a GLFW[14] key enum.
- **Scene Management** automatically adds a default scene, but you may set and add new scenes whenever you wish, it will simply switch to rendering and updating the objects within that scene. At this current point in time, it doesn't unload and load objects when you switch scenes, it simply keeps them in memory. However, there is a remove object function which can be used for high amounts of instantiated objects.

- **Shader** will set and load shaders that you add to the project. Once you have a new shader that you would like to add, you may create a shader object with it and add it to the Engine's shader vector.
- **Terrain** is the inbuilt terrain generator that utilises the noise class to generate its topography. The object can be instantiated in code and you may specify the amount of detail in the constructor (this is how many subdivisions the base plane has). The generation that you get may be edited in the Noise class, there is currently no way to tweak the variables in the noise class outside as it must be done internally, but that is a feature that is on the radar.
- **Terrain Object** is a class that inherits from GameObject that can be used to create other types of primitive[15] objects. This class is what Terrain inherits from and is incredibly versatile as it also handles transformations such as translation, rotation, and scaling while updating the buffers when doing so. The texture and shader of the mesh are also stored here.

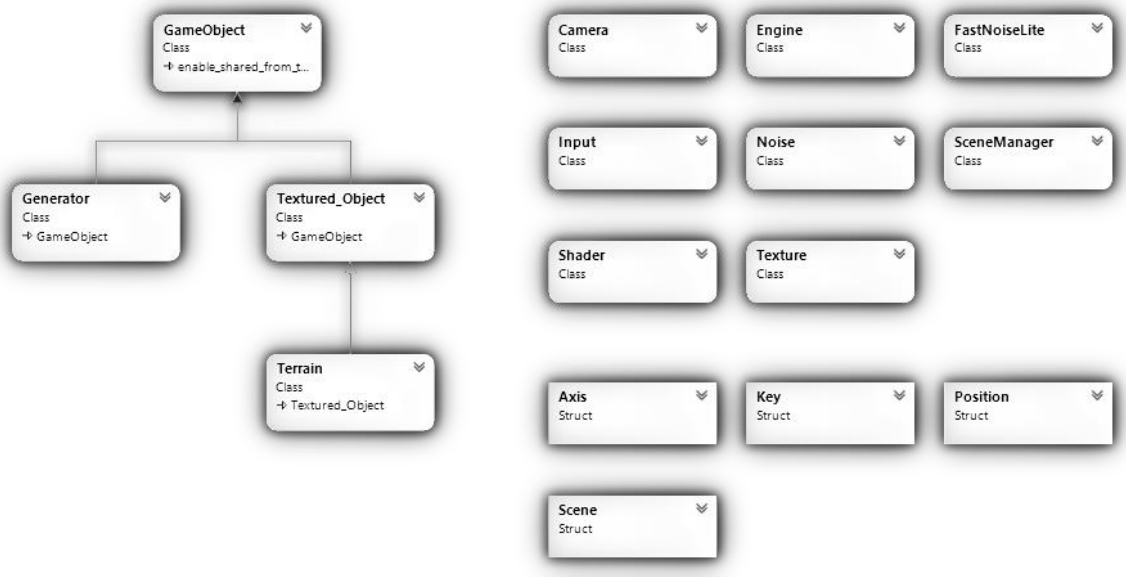
Target Audience

The target audience for this project is mainly C++ developers that have used the likes of Raylib[16] or SFML[17] and they feel like they want to have an easier time developing/prototyping for a project. People who are familiar with Unity style input, classes, and scenes are also apart of the target audience as they come from where the framework's syntax was inspired.

The target audience does not include people who are overly familiar with 3d graphical rendering and engine design, as they may view this project's current state as young in development. However, if they wish to fork the project and make adjustments, that is entirely welcomed.

Detailed Design

As the last section was more of an overview of each of the key features, this section will take a dive into the technical design of them as well as group them into relevant modules. Below is the classes layed out as seen in the Technical Design Doc, with the example usage on the left.



The example that was created is the Generator class which makes use of the Terrain, and Input, as well as instantiation of countless Terrain objects.

Modular Breakdown

Firstly we should group the classes into modules in a way that feels concise and clear. I propose the modules should be as follows:

1. Object Module
    a. GameObject
    b. Textured Object
    c. Terrain
2. Engine Module
    a. Engine
    b. Scene Manager
    c. Input
    d. Camera
3. Loading Module
    a. Texture
    b. Shader
    c. Noise
4. Type Module
    a. Axis
    b. Key
    c. Position
    d. Scene

The way it is broken down helps to draw attention to the subject matter, for example, we want to look specifically at the Scene Manager itself, the way it manages the scenes, not the scene type, that should be seperated into another module.

Object Module

The object module is the key part of the framework, it handles memory, sets the groundwork for events, and prepares the mesh type using OpenGL.

The GameObject's most important component is the static[13] "Create" template function. This function works by forwarding the arguments placed in the base classes constructor. It then asserts that the base class is a GameObject through inheritence, if it isn't, then the code will not compile. It then creates the shared pointer to the object and calls the engine singleton's[18] AddObject function which passes it to the current scene for looping the events. Finally it returns the shared pointer to the object.

The virtuals inside of the class are used as events by the scene manager, awake invokes upon the scene's start, start invokes when the base class starts, render invokes after update , update invokes at the start of the scenes loop, on object add invokes when it is added to the scene, and remove invoke when the object is removed.

Textured Object inherits from the GameObject class, it has a generate function meant to be called by the class that inherits from the Textured Object class. It needs things like the VBOs[19], EBOs[20], VBO Texs, Indices, etc. Render takes over the virtual from Object, but can be tweaked in the inherited class if you override it.

The transformation functions include move and rotate, which adds to the current position and rotation angle variables. There is also a set shader function which can accept a shader object as well as a protected texture pointer which can be loaded from the texture map.

Finally, Terrain is more of an example class that includes procedural generation, the constructor creates the mesh as it is supposed to be as if you implemented your own. Then the displace vertices function is what uses the Noise class to generate the topography of the mesh with your given detail/subdivisions.

Engine Module

The Engine class is the first thing and last thing that should be used in the main function. You need to load the engine at the start, instantiate all of your relevant GameObjects, then start the engine at the end.

When the constructor is called, the engine becomes a singleton[18], and the load function sets up all of the relevant parts for you, like the scene manager, the input manager, the window, and camera. The functions that are used by other modules would be the get window, get current scene, and then add object. There are also static smart pointers[8] to the default shader and main camera for convenience.

Start and Stop are the functions that handle the beginning and end of the engine's lifecycle, as it was said earlier, the start function should only be called once all needed objects are instantiated, then stop can be called whenever you want it to be called. In the example in the source, it is called once the escape key is pressed.

Moving onto the private access specifier of the engine, the shader vector is stored here, as well as a raw pointer[21] to the window for GLFW's[14] convenience, the screen height and screen width are also kept here as constants. The only other relevant parts that need to be mentioned are the scene manager being kept here, not as a pointer but actually as a regular member, then the singleton[18] instance of the Engine.

The Scene manager is an interesting class as it handles all of the incoming GameObjects and stores them in memory. There is a private list of shared pointers to scenes as well as a static shared pointer[8] to the currently active scene. The constructor creates an initial default scene and sets the current scene as that. The main functions relevant to the scenes are as follows: the add scene function which adds a new scene to the list, set scene which sets the current scene that is being displayed, get scene to get a non active scene, get active scene to get the currently active scene. Then there is the add game object to scene and remove game object from scene which acts upon the currently active scene.

Finally there is a loop scene function that iterates over all of the objects in the current scene. This function is public but it is not meant for use other than by the engine's start function.

The Input class is a widely used class that is encouraged to be utilized in the basemost classes of GameObject. It is a static[13] class that can be called from almost anywhere. There isn't much need to go over the private static[13] variables as they are all essentially just unordered maps with strings as keys pointing to keys or axis.

Initialize is a function that must be called in the engine's load function, and update is called in the game loop. The map button and map axis functions may be called during initialisation functions or start events in any of the GameObject subclasses but can even be called in the main function. They require an enum from GLFW[14] to map the keys.

Get button, down, and up, are ways that you can determine what sort of input you are seeing as a boolean. There is also a get axis that will return a number between -1 and 1, depending on which two keys are being held. The input class can also get mouse buttons as booleans as well as the 2d mouse position in the window.

The final part of the Engine module, is the Camera class, the camera acts as a sort of global main camera for the engine, it contains important vectors for the relative front, up, and right directions as well as the world up. Its yaw[10] and pitch[11] can be modified directly in order for you to create a sort of post processing effect if you need alternative camera movement. There is also configurable movement speed, mouse sensitivity, and zoom. The view matrix[22] and projection matrix[23] are passed to the active shader and do not need to be modified. The process keyboard and process mouse movement functions are where the default camera movement can be set up. It is advised to modify these directly at this current point in time, or if you wish to have an external camera controller, comment out the definition.

Loading Module

The loading module is imperative for the framework's rendering and visual functionality. It encapsulates the Texture, Shader, and Noise classes, all of which have to be loaded in some regard.

Firstly the Texture class uses STB_IMAGE[24] as a dependency, which is an external image loading library. In the set function, it takes the name of the texture you wish to map, as well as the path. It gets the data from STB_IMAGE[24] and then generates the texture within OpenGL. After it successfully fills the texture id variable, it passes a pointer to itself to the texture map so that it may be loaded on as many objects as you wish, reducing overhead for loading images more than once.

Textures should be loaded before objects in the main function, the textures themselves are then bound on individual object draw calls.

The shader class isn't as well integrated as the texture class yet. These work by being added to the Engine's shader vector, and then they can be used by referening their index position in the vector on your added objects.

The shader constructor takes a vertex shader path and then a fragment shader path, if the Shader you are using is just one, you can add an empty vertex or fragment shader to the path. There are also bools, ints, floats, vec4s[25], and mat4s[26] that you can set within you shader if you need to set uniforms inside your objects update function.

The loading part of the shader only uses the standard library's input filestream rather than a fancy external library.

The noise class is actually just a wrapper for the Fast Noise Lite[27] library which implements the perlin noise in its most basic form. The class uses the perlin noisemap that it seeds and calculates the heightmap for the terrain through manipulation of the likes of amplitude, fequency, height, and octaves of samples.

During the loading phase of the noise, which is the first thing to do once a singleton[18] instance of it is made, it sets the noise type to perlin, the frequency, and then seeds the noise using a random engine. Once the noise is properly seeded, there is a base function that carries out the first type of generation, this is called the get height function. It samples noise on 12 octaves, multiplying the amplitude by the given persistence, and the frequency by the lacunarity[28].

The get masked height function is what gives the terrain its unique generation. This is the function that is called by the displace vertices function within the terrain class. The masked height is retreived by first getting the base height from the other function. The mask is then retreived from a scaled version of the noise map that is currently being used. The mask uses a low threshold to create rare features within the terrain, be it high mountains, or deep ravines. Finally it checks if the base height is less than 0, and if it is, it multiplies by 0.1 so that the water level stays constant by the end.

Type Module

The type module is specifically for the structs that exist within the other modules. They will be listed below and given respective details as to what they do/are used for.

- Axis simply contains a positive key and a negative key, it is used solely for the Input class, for things like getting a horizontal axis between -1 and 1, on the keyboard keys left and right.
- Key is also for the Input class, and it takes a code which points to a physical key on a keyboard, controller, etc. It saves booleans for if it is pressed currenty, was pressed over one frame ago, and was released one frame ago.
- Position was a depracated struct for the shader class, from an earlier point of the framework, it is no longer used as it was replaced by the set vec 4 function.
- Scenes are a cornerstone of the scene manager, they are the data structure that holds the name of the scene as well as a list to all of the shared pointers to GameObjects within that scene. The lifetime of the shared pointers relies on the scene, if the scene is removed, all of the memory for the objects within the scene will be freed.

For a look into the class design please see the technical design document.

Creating your own GameObject Class

By the end of this tutorial we will have created a class that generates terrain objects for us with a sort of "Render Distance", it will also control the camera and allow us to move around the scene. Lets start by creating a class that inherits from GameObject.

```
#include "GameObject.h"

class Generator : public GameObject
{
public:
    Generator();
};
```

We wont put anything in the constructor body just yet. We can test if this works by adding it to our main function. So lets head over to our main function and do that.

```
#include "../include/Engine.h"
#include "../include/Generator.h"

int main()
{
    Engine* engine = new Engine{ "Tutorial" };

    auto generator = GameObject::create<Generator>();

    engine->start();
    return 0;
}
```

Now what we did in the main function is simply create an instance of the engine class, and named our window, "Tutorial". This will load the engine class as well as create a singleton[18] instance of it. The "auto generator" part is simply the name of our local variable that will store the instance of our class. We set it equal to "GameObject::create<Generator>()" because the static[13] create function templates our class and returns a shared pointer to the instance of the generator, adding it to the scene, and preparing it for the gameloop.

It is safe to run it now, and you should have a screen with nothing on it.

We will now update our Generator class to make use of the update event from the GameObject, this will be looped over 60 times per second in the scenes loop. We will also make a pointer to the main camera which we will take from the engine.

```
#include "GameObject.h"
#include "Camera.h"

class Generator : public GameObject
{
public:
    Generator();

private:
    void update(float dt) override;

    Camera* mainCamera = nullptr;
};
```

Now for the implementation.

```
#include "../include/Generator.h"
#include "../include/Engine.h"

Generator::Generator()
{
    Input::MapAxis("Horizontal", GLFW_KEY_D, GLFW_KEY_A);
    Input::MapAxis("Vertical", GLFW_KEY_W, GLFW_KEY_S);
    Input::MapAxis("UpDown", GLFW_KEY_SPACE, GLFW_KEY_LEFT_CONTROL);
    Input::MapButton("Close", GLFW_KEY_ESCAPE);

    mainCamera = Engine::mainCamera.get();
}
```

In the constructor, we map the Horizontal, Vertical, and "UpDown" axis to their respective keys. This will allow us to take the keyboard input so that way may translate the camera. We map the escape key for the close button which will help us close the window. The main camera is kept as a static[13] unique pointer in the engine so once we include the engine, we have access to that.

```
void Generator::update(float dt)
{

    if (Input::GetAxis("Horizontal") > 0.f)
    {
        mainCamera->ProcessKeyboard(RIGHT, dt);
    }
    if (Input::GetAxis("Horizontal") < 0.f)
    {
        mainCamera->ProcessKeyboard(LEFT, dt);
    }
    if (Input::GetAxis("Vertical") > 0.f)
    {
        mainCamera->ProcessKeyboard(FORWARD, dt);
    }
    if (Input::GetAxis("Vertical") < 0.f)
    {
```

```
        mainCamera->ProcessKeyboard(BACKWARD, dt);
    }
    if (Input::GetAxis("UpDown") > 0.f)
    {
        mainCamera->ProcessKeyboard(UP, dt);
    }
    if (Input::GetAxis("UpDown") < 0.f)
    {
        mainCamera->ProcessKeyboard(DOWN, dt);
    }

    if (Input::GetButtonDown("Close"))
    {
        Engine::getInstance().stop();
    }

}
```

In the update function we simply check the value of each axis which will if -1, 0, or 1, and then call the main camera's process keyboard function with the direction we want to move, and the delta time[29]. The get button down part simply closes the GLFW[14] window from the engine's instance.

We will now add the terrain generation part of the class.

```
#include "GameObject.h"
#include "Terrain.h"
#include "Camera.h"
#include <map>

class Generator : public GameObject
{
public:
    Generator();

private:
    void update(float dt) override;
    void generateTerrain(const glm::vec3& position);

    std::vector<std::shared_ptr<Terrain>> terrain;
    std::map<std::pair<int, int>, bool> currentTerrain;

    Camera* mainCamera = nullptr;
    const int GEN_DIST = 10;
};
```

We need to include Terrain in the header as well as map. We will utilise a generate terrain function to delegate the generation code from the movement in update. We will store a vector of shared pointers to the terrain, as well as a map to make sure we don't generate terrain more than once. We will also have a generation distance to control how far ahead we will generate terrain.

```
void Generator::generateTerrain(const glm::vec3& position)
{
    int posX = static_cast<int>(position.x);
    int posZ = static_cast<int>(position.z);

    auto checkAndCreate = [&](int x, int z) {
        if (currentTerrain[{x, z}] == false) {
            currentTerrain[{x, z}] = true;
            terrain.push_back(GameObject::create<Terrain>(glm::vec3{ x, 0,
z }, 5));
        }
    };

    checkAndCreate(posX, posZ);

    for (int i = 1; i <= GEN_DIST; ++i) {
        for (int dx = -i; dx <= i; ++dx) {
            for (int dz = -i; dz <= i; ++dz) {
                checkAndCreate(posX + dx, posZ + dz);
            }
        }
    }
}
```

The generate terrain function is pretty straightforward. We take the position and put it into an X and Z integer, we write a quick lambda[30] that takes the X and Z which will check the terrain map and see if there is terrain in the current space, and if there isn't, we tick it off and create the terrain gameobject in the terrain vector, passing in its position and the level of detail we want in the terrain. We then call the lambda[30] for the current position and then do a nested for loop for a square around the camera, as far as the generation distance, calling check and create each time.

We then call generate terrain with our camera's position at the end of the update function.

```
generateTerrain(mainCamera->Position);
```

We also add an initial call at the start of our constructor.

```
glm::vec3 pos{ 0.f,0.f,0.f };
generateTerrain(pos);
```

Now we will add all of the other relavent parts to the main function.

```
#include "../include/Engine.h"
#include "../include/Generator.h"

int main()
{
    Engine* engine = new Engine{ "Tutorial" };
```

```
    // SET TEXTURE MAP
    Texture* snow = new Texture();
    snow->set("snow", "assets/defaults/snow.jpg");
    Texture* stone = new Texture();
    stone->set("stone", "assets/defaults/stone.jpg");
    Texture* water = new Texture();
    water->set("water", "assets/defaults/water.jpg");
    Texture* ground = new Texture();
    ground->set("ground", "assets/defaults/ground.jpg");

    Noise noise;
    Noise::instance->Load();

    auto generator = GameObject::create<Generator>();

    engine->start();
    return 0;
}
```

We start the engine as normal. Then we set the texture map, by default, the displace vertices function within the terrain class looks for the "snow", "stone", "water", and "ground" entries in the texture map. To use them, all you need to do is create a pointer to a new texture, and then call its set function with the name of the texture and the path to the texture.

We also need an instance of the noise class for the terrain, and we must call load or else we wont get the dynamic terrain generation.

And that's it.

Glossary

1. **Spatial Partitioning:** An optimization technique that divides the GameObjects into regions to efficiently manage.
2. **Fork:** Branching off of the main branch of the frameworks code to customize or enhance its functionalities.
3. **C++17:** The version of the C++ programming language that this project uses.
4. **Inheritance:** An Object oriented programming feature that allows new classes to derive properties and behaviour from existing classes, such as the GameObject class.
5. **Polymorphism:** A concept in C++ allowing functions to do different things based on the object it is acting upon.
6. **MVC:** Model-View-Controller, a design pattern that separates the data model, user interface, and control logic in programming, this would be used if UI was integrated into this project.
7. **ECS:** Entity Component System, a design pattern that uses composition instead of inheritence, it usually helps with speed.

8. **Shared Pointer:** A type of smart pointer in C++ that manages shared ownership of a dynamically allocated object, helps with memory leaks.
9. **Factory Pattern:** A design pattern that provides an interface for creating objects in a super class, but allows subclasses to alter the type of objects that will be created.
10. **YAW:** Rotation around the vertical axis, used in the camera .
11. **Pitch:** Rotation around the lateral or transverse axis, controls the up and down in the camera system.
12. **Zoom:** A camera operation that enhances or reduces the field of view.
13. **Static:** Static refers to variables or functions within a class that are shared among all instances, or properties that do not change during runtime.
14. **GLFW:** A library for OpenGL that provides a simple API for creating windows, contexts and handling user input.
15. **Primitive Object:** Basic geometric shapes like cubes, spheres, and planes.
16. **Raylib:** A simple and easy-to-use library to learn video game programming, similar to SFML.
17. **SFML:** Simple and Fast Multimedia Library, a C++ library that provides low and high-level access to graphics, sound, and input devices for developing games.
18. **Singleton:** A design pattern that restricts the instantiation of a class to one "single".
19. **VBO:** Vertex Buffer Object, used in OpenGL to store vertex data.
20. **EBO:** Element Buffer Object, used in OpenGL to store indices that define which vertices to draw.
21. **Raw Pointer:** A type of pointer in C++ that directly manages the memory address of an object.
22. **View Matrix:** Transforms world space into camera (view) space.
23. **Projection Matrix:** Transforms view coordinates into screen coordinates.
24. **STB_IMAGE:** A single-header-file public domain library for loading images.
25. **Vec4:** A four-dimensional vector.
26. **Mat4:** A 4x4 matrix.
27. **Fast Noise Lite:** A lightweight noise generation library.
28. **Lacunarity:** A measure in procedural generation that describes the texture's gap, influencing the appearance of detail at various scales.
29. **Delta Time:** The time elapsed between the current and the last frame.
30. **Lambda:** A feature in C++ allowing the creation of anonymous functions.

Afterword

This has been my most interesting project to date, and the intricacies captivated me the whole way through. I'm very thankful for everyone that helped me along the way and I will always look back on this project with fond memories. When I have time to come back to graphics programming, I will certainly be expanding upon this idea. Thank you for reading and I hope that you found something interesting here.

Kind regards,

Conor McDonagh Rollo