Project Report

Legend Terrain

Abstract

LegendTerrain is a procedural terrain generation/game framework developed using C++ and OpenGL, aimed at facilitating the creation of dynamic and expansive virtual environments in games. This report outlines the architectural decisions, roadmap, and analyses of the framework.

Conor McDonagh Rollo

C00260445

Computer Games Development

Table of Contents

Introduction

Legend Terrain is a project where I wanted to create a framework that would help newer developers get into the low level graphical programming space by providing an easy-to-use design that reminisces of the likes of Unity in a way doesn't hold their hand too much while also allowing them to get into the depths of the framework, essentially programming it themselves.

The vision was to have procedural terrain be the talking point that would generate interest and get developers excited about prototyping something they enjoy, to go on and program it from the lowest level themselves.

I had always wanted to experiment with OpenGL and that was my reason for undertaking this project. I looked at other options such as Vulkan and Direct3d and decided that with the scope of the project, I should choose the one with infinitely more documentation and tutorials as I was also new to it.

My original idea was to essentially make a game engine, but as I got more into the project and deepended my understanding, I found that the idea of creating something of that scale was futile and I should settle for a framework that could do a chunk of the things that a game engine could do.

Project Overview

The core functionalities of the project were to be firstly centred around Objects in a scene, these objects would be GameObjects and would be able to be inhereted and built upon. Creating a controller that could handle the gameloop for the developer as well as some other helpers such as a scene manager and an input manager were also needed. I wanted to abstract from the low level graphics programming to a certain degree, and make the focus more on building upon what is presented while not having to worry about the likes of memory management.

Some of the inspirations for the project were various youtubers that I watch who set out on journeys to make their own engines. A big help in the design of the framework came from one called TheCherno, who creates amazing C++ tutorials, and who had actually released a fully fledged game engine with only a few developers. Not to mention the excitement of the growing scene of 3rd party engines such as Godot 4. I knew that I couldn't make an engine in the allotted time, but I could try and make something that would resemble my vision.

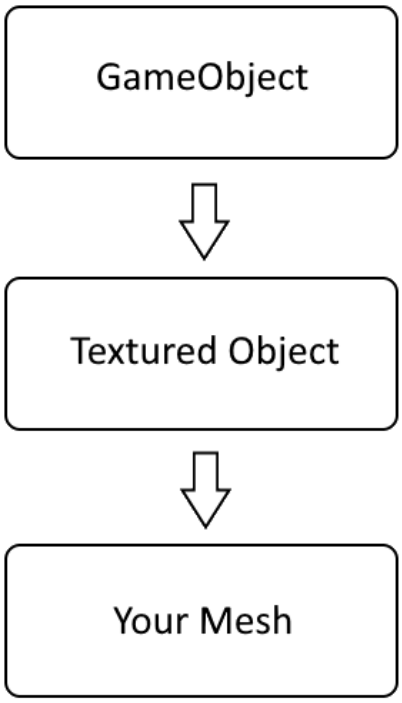
System Architecture

The idea of the architecture was that the GameObject class would implement a factory pattern that handles memory mangement for the lifetime of the scene. The GameObject would also have virtuals that could be invoked at specific points in time, and would be easily extendable for the likes of a physics engine.

The likes of Textured Object would inherit from the GameObject class so that it would not longer have to worry about memory management and then that could handle the OpenGL related tasks, as well as shaders and such.

Then whatever would inherit Textured Object, only really needed to detail the vertices and position of where the object was going. This simple inheritence and polymorphic structure would be the cornerstone of the framework.

The engine class would be completely decoupled from the GameObjects and the scene manager would handle their lifetime. Then the input manager can be called from anywhere within the project.



Roadmap

The project wasn't immediately started. I spent the first few weeks actually just learning core concepts of OpenGL from the LearnOpenGL website, I made a few side projects to practice during that time. The one thing I didn't think about then was framework architecture, it was quite a big mistake because I just got straight into the bulk of the project, getting things working left, right, and centre. By Christmas, I had realised that the project was no longer going in the direction I had wished for, this was due to poor design choices.

At Christmas time, I began rewriting from the ground up, detailing the architecture and properly defining the GameObjects and how they interact with the rest of the system. I started to look at it more like a framework that other people are going to use, and that really helped me get the core components working together in the correct way.

When I was finally done rewriting the framework, I had realised that I had made errors somewhere in the low level graphical rendering parts. I wasn't able to get it working no matter how hard I had tried, so I began working on it less and less.

Near the end of the allotted time for the project, I picked it back up and took the first and second rendition of the framework, and began merging them by hand, keeping the working parts of the first and the better architecture of the second. When I was done, I had the third rendition of the project, which finally worked exactly how I had envisioned it.

Performance Analysis

The performance analysis was something I left until the very end. There were some interesting performance related errors but most of them were able to be fixed by changing one or two lines of code.

One of the more interesting performance errors was actually in the process of loading images for textures. The entire time that I was doing the project, I hadn't realised that I was loading images in real time as objects were being created. So when I instantiated an object in real time, it would load the texture exactly when it was instantiated. I fixed this by creating a static texture map that can be picked from and loaded at the start. This reduced the overhead of the program by over 50% in that test.

One of the other big errors was one that I didn't have enough time to fix. I realized when I was creating objects that were very far away from the player, they were just sitting there still using memory. I had realised that I needed to apply spatial partitioning to the scene manager.

Conclusion

The project was an interesting dive into the world of graphics programming, I would certainly like to learn more about it. I think that I would like to continue working on it in the future, and some things that I would consider for it would be another full framework rewrite. I would have liked to use decoupled patterns in order to make it viable to create a UI for the framework, and then even a scripting pipeline to make it closer to an engine.

Literature Review

Procedural Generation in Game Development

Procedural generation employs algorithms to create vast game content with minimal user input, enhancing environments and gameplay mechanics. Togelius et al. (2011) examine its application in expansive games like "Minecraft" and "No Man's Sky," where it crafts diverse and extensive landscapes.

Frameworks in Game Development

Game development frameworks streamline the creation process by providing reusable components and workflow management. They integrate critical functionalities like graphics rendering and physics. Well-known examples include Unity and Unreal Engine, which support both 2D and 3D development (Gregory, 2014).

Combining Procedural Generation with Frameworks

Merging procedural generation with game frameworks optimizes the development of dynamic environments, allowing developers to focus on gameplay mechanics. Adams (2017) notes that this combination enhances game performance and developer productivity by managing the complexities of environment generation.

References

Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. IEEE Transactions on Computational Intelligence and AI in Games.

Gregory, J. (2014). Game Engine Architecture. CRC Press.

Adams, E. (2017). Fundamentals of Game Design. Pearson Education.