COS30031 Games Programming
Custom Project Report

Daniel Coady 102084174

04/11/2021

Contents

1 Introduction

1.1 Background

Computers are incredible and complicated these days, with many components that allow for the more efficient and effective computation of various tasks. For me personally, by far the most interesting modern day development (which many will actually take for granted) is the humble GPU. Short for graphics processing unit, it is a core aspect to any computer whether it is pushing hyper-realistic game graphics or displaying the words on your social media feeds. These GPUs differ greatly from the CPU which many view as the beating heart of any computer. CPUs have been designed to be more general purpose, allowing for operations such as logical branching, maths, and bitwise magic. GPUs on the other hand are entirely focused on floating point maths operations, and have been architected to be able to do those rapidly and in parallel. This means modern day graphics requirements, such as being able to independently address one of the literal millions of pixels on your screen (a standard 1080p screen contains a little over 2 million pixels!), is now a trivial task that can be completed with great speed.

Computers haven't always been like this however, and many moons ago we would have required the CPU to perform all of the logical and graphical work of a computer. This complicated matters greatly as soon as you wanted to display complex graphics, and it only got worse if you wanted to display complex graphics *and* provide complex logic–such as in a game. It's for this reason that many games of yore have had to come up with some very clever techniques to "cheat" graphics. Most games for the longest time were purely 2D since that was about as much as we could reasonably handle with the hardware at the time. Before 3D accelerated technology hit the general consumer market, 3D games were but a dream... Kind of.

Enter the raycaster. There will be more later on about how it works and why it's so fast, but for now just know that raycasters were some of the earliest attempts at creating 3D graphics in games. Not to be confused with ray marching and ray tracing (two very cool 3D graphics techniques as well!), raycasters have a very distinct look to them that many will remember from the first Wolfenstein 3D game. They tend to look very blocky, with billboarded sprites representing the entities within a scene. This technology forms the basis of my custom project.

1.2 The Project

At a high level, this project will take form of a first person shooter using 3D raycaster graphics. The front end will use SFML to display the internal frame buffer as well as receive/process the window events. The back end will be far more complex, consisting of the actual raycasting engine, an ECS implementation for managing enemy entities, multi-threading to assist with performance, and some simple collision detection. The entire project has been written in C++

within a Linux environment using the gcc toolchain, and borrows greatly both from Austin Morlan's writeup on ECS[1] and Dmitry V. Sokolov's tinyraycaster series of tutorials[2].

2 Implementation

2.1 Front End

The bulk of the front end in this project is handled by the Simple and Fast Multimedia Library, or SFML for short. SFML is similar to SDL in a variety of ways–both have many useful abstractions of low level graphics, image handling, audio processing, etc. There are some key differences between the two however:

- SDL has a very C style design and API, while SFML's design and API are much more like C++
- SFML's abstractions tend to be easier to work with due to it's object oriented nature
- SDL's graphics abstractions are fairly limited, only allowing for simple quads and textures while SFML has more primitives and a simple shader pipeline
- SFML just generally has less boilerplate code than SDL

All of the above ultimately influenced my decision to choose SFML over SDL as we had been using for all of our previous tasks throughout the unit. Add to this that it has a very sensible system for event polling that is similar (though in my opinion, slightly better) to SDL's own event system, and it just made sense for the purposes of my project.

There is another option however, one that I think might have made more sense for this project actually: raw OpenGL. It's a scary prospect, for sure, but there's a big benefit in the form of performance that we would have gained. There is of course some added complexity that would come from this since we're essentially working with little to no abstractions. However, considering that there isn't much we need from SFML outside of the events system and drawing textures to a quad, it would have been decently trivial to make the change over to raw OpenGL.

[1] https://austinmorlan.com/posts/entity_component_system/
[2] https://github.com/ssloy/tinyraycaster/wiki/Part-0:-getting-started