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.

2.2 Back End

2.2.1 Enemy AI

Enemies are all treated as entities in an Entity Component System, or ECS for short. This is quite a simplified version of ECS, as it only need serve one specific

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

purpose with little to no generics involved. This has allowed me to keep it as simple as it is, while also ensuring maximum performance from both algorithmic and cache optimisations.

Each entity will contain a component for movement, the enemy type, and the distance it is from the player. They have been separated like this since there are situations where not all of the data related to an entity would be necessary, and so by lowering the amount of data we need to retrieve in each of the related systems we can help ensure higher amounts of data are being fetched. In terms of systems present, there are a few present. The main ones are the render system, and the enemy behaviour system. The render system is fairly uninteresting as all it does is read the components for each entity to render them accordingly. The enemy behaviour system on the other hand is far, far more interesting. This system, as the name would imply, is where all of the actual AI for each enemy lives.

The enemy AI is incredibly simple, as it exhibits only two behaviours:

- Chasing the player
- Waiting to see the player

Chasing the player is fairly simple: calculate the unit vector pointing towards the player, then add that vector to the current position scaling by speed and current delta time. This only happens when the enemy can see the player however, and this was a fair bit more complex. You see, physics raycasting with my current solution is rather complicated due to there being no real collision detection to speak of. If I had implemented some simple collision detection with axis-aligned bounding boxes then I may have been able to do some simple maths to check for ray intersections with edges of the AABB, but there is no such thing implemented. So there's only really one option left for me: a sort of ray march. To be clear, this isn't the graphics technique for volumetric raycasting, I just don't have a better name for it at the time of writing this. Simply put, the way that the physics raycasting works is by continually "marching" in a given direction until it hits something. If we hit something, then we register what that is and make a decision based on that. In our case, if we hit a wall then we don't want to move the entity. If it hits the player, then we start chasing.

This gets expensive to perform every frame however, so I wanted a way to make it more performant. This posed an interesting issue since more performance seemed to require me to give up some amount of accuracy. This lead me to perform a series of tests on how performant and accurate different levels of coarseness were when performing a physics raycast, but perhaps most importantly I wanted to find out at what point did the believability of the AI suffer. Much to my surprise, I could go incredibly coarse before it started to show any sort of behaviour that was odd. And so, I made the raycast treat each map tile as a single unit, and it will draw a line from enemy to player checking for a hit. This hit a good balance, where the entities never ended up showing strange behaviour, but performance is hardly impacted due to how coarse the physics raycast calculations are.

2.2.2 Collision Detection

2.2.3 Multi-threading

2.2.4 Raycaster

3 Challenges

This has been an absolutely enormous project, one that has spanned months at the time of writing this. That of course means that there have been many different and interesting challenges that have arisen during development. I have not mentioned these challenges above for the sake of brevity, but for the interested reader I have detailed what they were, why they were problems, and how I did or didn't manage to resolve them.

3.1 Raycaster Depth Testing

3.2 Mouse Looking

With the way that the raycaster had been written, changing the player's view angle would be trivial–and indeed it is. There is a simple method on the Engine class that allows you to adjust the current player's view angle, which is stored in radians, by a provided delta. At first this was achieved with a simple button press which was easy to create: while the key is held down, increment the view angle by a set delta. This worked excellently, but it would be *way* cooler if you could control it with your mouse instead.

This is where the complications begin, but not because mouse movements are hard to track–quite the contrary actually! Really, all that needed to be done was store the previous position of the mouse, store the next position of the mouse, and then calculate the magnitude of the two vectors. In fact, this is made even easier since the only axis we care about is the x axis (raycasters by design only allow for looking left and right) so we need only store the previous and current x axis and calculate the difference between the two to get our delta.

In SFML we have, much like SDL does, an event loop. These are pretty simple, allowing us to listen for all window events. It works incredibly well, and indeed is perfect for most of what is needed in this project. So when handling a mouse event, it would seem most reasonable to also add it to this event loop. In fact, checking the SFML documentation we can see that there is actually an event type called MouseMoveEvent which is perfect for us. So in theory, we should just be able to add this event to the loop and calculate our deltas like this:

```
sf::Event event;
int last_x;
while ( window->pollEvent( event ) ) {
    // handle other event types here
```

```
    switch ( event.type ) {
        case sf::Event::MouseMoveEvent:
            const int delta = event.mouseMove.x - last_x;
            last_x = event.mouseMove.x;
            move_view( delta );
            break;
    }
}
```

Here is where we encounter our first problem, since if we just calculate the deltas between the last position and the current position then we may eventually (and probably will) run out of screen space to travel. To fix this is actually fairly simple, since we need only do the following:

```
// on program init
window->setMouseCursorGrabbed( true );

// during event loop
sf::Mouse::setPosition( window_center, *window );
```

The general idea is that we are ensuring that the cursor is locked to the window so we can always capture events, and we're resetting the position to the center of the window every trip through the loop. This works well, but introduces yet another issue. You see, when we set the mouse position it actually adds another MouseMoveEvent to the event stack. This is a problem, because we're in the middle of polling that very same stack and if we add another event to that stack while we're in the event loop... Well you're just going to spin endlessly in a loop–which is exactly what ended up happening.

This problem was the cause for much pain over the course of a few days as I couldn't figure out how to stop the event stack from registering a new event when I move the mouse cursor through code. Eventually I happened across a post on the SFML forums (which unfortunately, I have since lost) which keyed me in to the solution which I have ultimately landed on. That solution is a simple one: don't use the event loop. It might be a hack, it might actually be intended, really I'm not actually sure. What I do know, however, is that calculating the delta myself outside of the event loop ended up being the solution I needed to side step the issues with endless looping. However, I do fully admit that there's a potential issue with performance from this approach (perhaps a small one, but and issue all the same) since we're still adding these events to the stack and we're now doing delta calculations on every single frame. These are sacrifices I'm willing to make however since early optimisation is the enemy of progress, and there is only so much time to complete this project. Since it didn't pose any immediate problem with the performance of the project, I've left it like this in the final product.

4 Conclusion