HW2 Writeup

Intro

This project went through a lot of iterations, and I am not particularly proud of some of the code in the final product (especially in part 4). However, since it works and project 3 is around the corner, I figured it would be better just to refactor with that in mind and turn this in. So I will be explaining how it's currently designed, with a few ideas on how I would change it

Multithreaded Loop Architecture

Section 1 involved looking at adding threads to make the game loop more efficient. This was challenging due to all the considerations you have to make when passing data between threads. I started out trying to split up the work like this:

- A thread for handling all movement (platforms, getting player direction)
- A thread for collisions
- The main thread for drawing

I learned real quick that you had to be careful with your mutexes here. I was having issues when drawing in two different threads or moving an entity that was also being drawn in another thread. Using a single rendering mutex to only allow one thread to move or draw at a time. One issue I ran into was that sfml did not like checking user input outside of the main loop, so I decided to put that in with the main drawing thread. So I now had 1 thread for moving and drawing, the other for checking and resolving collisions. My collision algorithm was a decent amount of math, so I felt that it would be a good idea to have it in a separate thread. However, since it resolves the collision in that method, I needed to have a mutex before the resolution algorithm to stop the other thread from moving or drawing when it was going on. I was worried when designing that it would slow down the other thread too much, but it was still working great when I implemented it, even quicker than the singlethreaded design I had. I felt this was the best design due to how intertwined the moving and drawing code needed to be, keeping it all in the same thread felt right. Since the collision detection requires a bit more math, it keeps checking for collisions and quickly pauses the movement while the collision is resolved.

Timeline

The timeline implementation took some time to understand, but was not too bad once I got the logic sorted out. It was hard to understand the anchor at first. I designed my timeline to have 2 overloaded constructors. 1 takes no parameters, and creates an anchor timeline based in realtime. Then, that timeline can be passed to any real time or gametime object. For each frame in the main loop, I calculated the delta time from the last frame and updated my movement methods to take in a delta time.To get pausing to work, whenever the timeline was in a paused state it returned the time the game was paused at to make the frame delta 0 and stop movement. One problem I ran into was the last time recorded in the main loop got messed up when pausing or changing the tic rate, as the delta would suddenly be really big for however long the game was paused or the tic rate changed how getTime() was calculated. The movement of characters and platforms was all thrown off due to the large delta. I had to update lastTime after unpausing or changing tic rate to the current time so the game could get caught back up to where time currently is. For my ticrates, I used 2 as my base rate, with 1 serving as

the ½ speed and 4 as my 2x speed. I had no issues with this other than my lastTime problem explained earlier.

Networking Basics

Learning 0MQ took some time to get used to. My first misunderstanding came form not realizing you can have two different sockets running at the same time. I was struggling to see how you can send out a single message to all clients using PUB-SUB while also detecting a new connection. Once I realized you can use both PUB-SUB and REQ-REP, the assignment just became figuring out the syntax and rules for 0MQ. I decided to use a REP socket on the server, and have the client send a REQ to so the server can know when it connects. It then initializes the map with a new client to track the iterations. The client then subscribe to the publisher socket in the server, sending out the appropriate message for how many are connected. This was just a lot of trial and error.

Putting It All Together

I went through a ton of different iterations for this part. One of the largest challenges was deciding how to have the server distribute information. I realized that currently, I only need the server to distribute character positions. 0MQ only works well with strings, so I created a simple format for sending positions: “id x y”. I gave each entity a global id number that was hard coded into the client and server, which was the first value in the string. The second and third values are the position coordinates for where that entity is. I’d like to come up with a better system for the numbered ids. I'm thinking of making some kind of class or data structure that both the client and server can access so I only have to change the data in one place when adding or changing the platforms/characters on the screen. After deciding on how to send positions, I started to implement moving platforms to be distributed from the server and work on multiple clients. I felt that the moving platforms that all players see needed to be handled by the server, and just publish their new positions every time they move. That way I would not have to sync up clients if they are moving platforms independently. This used a simple PUB-SUB model, and was a little laggier than it probably should be. I did not end up taking the time to optimize, and plan to fix that in the next homework. After getting platforms mostly working, I moved on to characters. Since there was a cap of 3 connected players, I predefined 3 player objects in the client and server, assigning them ids when a new client connected. The client was then sent back this id, and how many other players are connected so it knows what to draw. I don’t like pre defining objects here, and know it will not work for the next homework. I will probably end up creating a new player object and adding it to a data structure for the next one. At first, I thought I was going to have the server save the positions of each player and distribute all of them out every chance it got. However, I ended up changing it to not save any positions, and whenever a client send a new position where their player just moved, it just redistributes that message out to all other clients. This was done because it increases speed and gives the server less work, but meant clients were only getting updated from each other when they moved. This was not an issue most of the time, but added some extra work when connecting, as I had to do some extra work to make sure the positions of all connected players were published when a new client connects so I can get them even if they are not moving. I did not end up incorporating the timeline in this part.