CSC 481 Homework 2
Declan Smith
10/05/2022

Section 1:

For this section I took a pretty simple fork and join approach. I was careful to avoid concurrent threads which might access the same data, so I didn't end up using any mutexes. However, I would have definitely preferred to, since I know they will be necessary in future assignments, but ended up being short on time. But for my part 1 implementation, there really wasn't much data that threads might fight over, so I don't think mutexes would have improved much before adding networking. My game loop stores characters and platforms in two separate vectors, then uses threads to update them in parallel.

The method which updates characters doesn't account for collisions, because I haven't found a super clean way for each character to have access to all the platforms. Instead, after updating all of the platforms according to their movement patterns and characters according to keyboard inputs, I iterate through them and adjust the position of any character which is overlapping a platform. So after using parallel threads to update all characters and platforms, I start a new thread to draw the platforms to the window while the main thread does the collision adjustments.

I tried to approach this section in a way that would scale nicely as more characters were added through multiplayer in section 4. However, It actually ended up requiring a fairly large amount of refactoring in order to make it easier to serialize the state to json.

Section 2:

Creating the timeline class didn't really call for too many big design decisions, as the requirements are pretty specific. If a Timeline is constructed with NULL for the basis, getTime() will refer to std::chrono instead, and returns an unsigned integer number of milliseconds.

Implementing the pause and speed change functionality was a little tricky, as changing the rate in the middle of a loop can cause large spikes in the time delta. However making some simple changes to the last_time whenever you change those settings was an effective solution.

For the changing speed feature, I added a float field scalar, rather than changing the value of the tic speed. That way you can easily get half speed even if the integer tic speed is set to 1.

Section 3:

I made some mistakes in my initial planning for this section. I initially misinterpreted the requirements to be that each client need only receive messages with their own id and iteration rather than including all clients in all messages. Which is a pretty easy problem to solve with request/reply, so that's the method I started with. Subsequently, I used only request/reply for my implementation of part 4, which might have been easier with some publisher/subscriber. After I realized what the actual requirements were for this section, I wasted a lot of effort trying to get it to work with only req-rep before realizing that adding a pub-sub socket would actually make it really easy and straightforward.

My final implementation uses a req/rep and a pub/sub socket. On the server side, the main loop starts with a recv to the reply socket with the dontwait flag to check for incoming requests to add a new client, followed by a loop which increments the iterations of all the clients already connected, publishes a message for each. The client simply sends a request through the rep/req socket to alert the server a new client has joined, then prints out all of the messages published to the subscriber socket.

This implementation is significantly less messy then the networking code I have for part 4. This is in part because after the initialization none of clients need to send anything to the server anymore, but also simply because the publisher/subscriber model is a lot more efficient and sending an identical message to every client.

Section 4:

The first problem to solve was finding a way to send all of the complex state information back and forth between clients and servers. I decided to make a GameState class to store all of the Platforms and Characters which provides a serialize method that converts it into json. Then it can be converted to a string, sent via zmq, and parsed to json and deserialized back to GameState. However, constructing the GameState from scratch each time isn't very optimized. So I also added a json field updateLog to GameState which it would use to keep track of every update thats made to the objects it stores. Then that updateLog can be sent to clients with zmq and each client's GameState can update itself without reconstructing all of the objects every frame.

I only used only one rep req socket for each process, which may have made some things more difficult for me. My initial approach was to have the server do a for loop to do one request and response per client, with a special case for new clients. Each client would update its character movement, send its characters position to the server, and the server would send back the positions for the other characters and platforms. This worked fine with 1 client, but was problematic with more. One tricky problem here is getting all of the characters on the server updated before sending the state update out to the clients. With only one request/response per client not all of the characters' movements can possibly be updated by the time each client is sent their response. To solve this I had two sections for the networking code, each with one request/response per client. For the first section, the client sends its character's position, and the server responds with a confirmation. Then in the second section, each client asks for a state update, and the server sends it to them. I had to add some flags to the json packets being sent back and forth to ensure that each client gets served once per section.

This implementation works great for these requirements, but I expect it will need a lot of refactoring, especially once more multithreading is required. Also, as mentioned in section 3, a publisher/subscriber socket could potentially clean up some of the less pretty sections.