Writeup

Section 1: Multithreaded Loop Architecture

In Section 1, I simplified my game code by moving less important game logic to a separate thread. This helped reduce errors and make the game less complicated. Later on, I extended this idea to moving platform logic, using two threads to share the workload and ease the main thread's burden. Managing dependencies on the main thread required using something called a "mutex."

To make the code structure cleaner and more organized, I switched from a basic single-function design to an object-oriented approach. This made it easier to add new features to the game.

Section 2: Measuring and Representing Time

In Section 2, I set up a real-time clock outside the main game loop and a global time clock (dt_clock). Inside the game loop, I compared the time before and after resetting the global clock and stored the difference as 'dt,' which represents the time for each game step. This 'dt' was vital because it allowed game objects to be independent of the frame rate, making it possible to adjust game speed or pause/unpause the game. I added three preset speeds (0.5x, 1x, and 1.5x) for adjusting 'dt.'

Eventually, I created a timeline class to make it easier to manage time in the game and allow for scalability.

Section 3: Networking Basics

In Section 3, I explored a networking library called zeroMQ. I looked into two networking models, PUB-SUB and REQ-REP, implemented them for practice, and mixed elements of both for the third part.

I used the REQ-REP model to set up connections between clients and the server. In this model, the server gives each client a unique ID. I used the PUB-SUB model to send messages about client iterations. This meant that when a new client joined the game, they could start from the current point without missing anything.

Section 4: Putting it All Together

Section 4 was the most challenging part of the project. I had to figure out how to exchange data between clients and the server, what data to send, and how to send it. To tackle some lag issues related to message queuing, I used a feature called "conflate" on the sub socket.

One significant challenge was sending a specific kind of data structure through a zeroMQ socket. So, I came up with a way to turn that data into a text string, send it, and then convert it back to its original form on the receiving end.

I also ran into an issue with the REQ-REP model, where I had to send a reply. To work around this, I sent dummy messages as replies.

Lastly, there was a minor hiccup when drawing characters on the screen. At first, I was only creating one character object in a loop, but I later realized I needed a whole array of character objects to handle the game correctly.

Section 5: Asynchronicity!

In Section 5, I took the game to the next level by making it asynchronous. This means that different clients could now run at their own individual speeds, and all the game processes could run independently. To achieve this, I assigned threads to each client, allowing them to send their responses and receive replies without being tied to a single, synchronized pace.

This project was a big learning experience, helping me build a multithreaded game loop, set up a solid time management system, and establish a working client-server setup for a multiplayer game.