Hyunsirk Choi

CSC 481

Dr. David Roberts

07 Oct 2022

Homework 2 Writeup

Section 1: Multithreaded Loop Architecture

Based on the homework 1 section 4 program, I have decoupled the `MovingPlatform` and `Character` from the main thread to calculate its movement on a separate thread. To achieve this, I have created an abstract class `ThreadObject` which is used to extend `MovingPlatformThread` and `CharacterThread`. The `ThreadObject` itself is extended from the `sf::Shape` class which allows all children classes of the `ThreadObject` to be directly drawn on the `sf::RenderWindow`. The `ThreadObject` also has a pure virtual function `ThreadObject::run()` to be implemented by its child class for running the function in a thread. This function will calculate and update the object's movement based on its current state which will be updated using setters before starting its own thread. I have used a fork and join approach to implement my multithreaded architecture; however, since I had a poor understanding of how it actually has to be implemented, for every loop a new thread will be created for each of those objects and will join before moving on to the next iteration as shown in the code snippet below. This will result in slowing down the program due to numerous thread creation and joining during its game loop which will defeat the purpose of using multithreaded architecture. This issue will be improved on the next homework assignment.

```
std::thread characterThread(handleThreadObjectRun, &character);
std::thread movingPlatformThread(handleThreadObjectRun, &movingPlatform);

characterThread.join();
movingPlatformThread.join();
```

Section 2: Measuring and Representing Time

The `Timeline` object I have implemented is pretty simple in terms of implementation. The Timeline has two functions to get two types of times which `Timeline::getTime()` returns the total time passed from its creation time and the `Timeline::getDuration()` returns the passed time since the last call of this function. The `Timeline` has its `startTimeStamp` which will be initiated and never change after the object's instantiation. While the `previousTimeStamp` will be updated whenever the `Timeline::getDuration()` is called. Each field is used to calculate the `Timeline::getTime()` and `Timeline::getDuration()`. During these time calculations, it will return the difference between `std::chrono::steady_clock::now()` and one of the time stamps which will then be divided by the `tick` for any potential time speed modification. This tick can

be modified by `Timeline::increaseTick()` or `Timeline::decreaseTick()` to make the time faster or slower. The pausing of time is also possible with the field `bool isPaused` and `Timeline::pause()` and `Timeline::resume()` functions. The `Timeline::getDuration()` will always return 0 when `isPaused` is true but `Timeline::getTime()` will not be affected by the paused time and just returns the elapsed time since its creation. Also, while `Timeline::pause()` simply sets the `isPaused` to true, `Timeline::resume()` will not only set `isPaused` as false but also updates the `previousTimeStamp` to `std::chrono::steady_clock::now()` for proper duration calculation. The `Timeline::getDuration()` function is mainly used to calculate the movement with the concept of "velocity * duration" as shown below. The `TIME_RATIO` constant is used to tweak the calculated movement value by adjusting the unit of time. The `Timeline` object can anchor to another `Timeline` object to return relative to its anchored time, but I have not used this feature throughout this project since I could not attempt section 5 due to lack of time. The main difficulty of this section was figuring out how to cast different `std::chrono` datatypes to calculate the difference in time and return it as a double type.

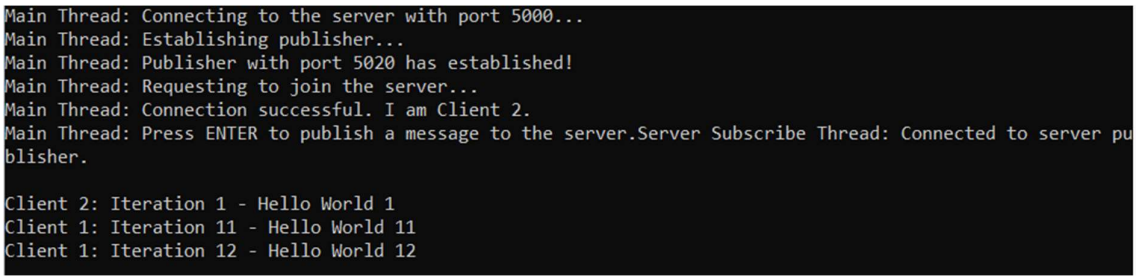
```
object.rotate(ROTATION_SPEED * TIME_RATIO * duration);
```

Section 3: Networking Basics

The server has 2 bind sockets and 1 connect socket while the client has 1 bind socket and 2 connect sockets. The server uses `zmq::socket_type::rep` for responding to any incoming client connection requests which bind to the "well-known" port number of 5000, meaning the client will know the port number when sending requests. When the server receives a request, the server receives the client's port number which is used to connect using `zmq::socket_type::sub` socket to listen to any published client message. Whenever the server receives a published message from a client, then the server publishes that message to all connected clients using `zmq::socket_type::pub` socket which is also bound to a "well-known" port number of 5001.

On the other hand, using the `zmq::socket_type::req` socket the client sends a request to the server's "well-known" port 5000 to establish a connection with the server. The client includes its `zmq::socket_type::pub` socket port number so that the server can listen to the client for any messages. This port number is dynamically decided based on the number of the running local clients where the port will increment by 10 starting from port number 5010. For example, Client 1 will have a port number of 5010 and Client 2 will have aa port number of 5020. The client will also use the `zmq::socket_type::sub` socket to subscribe to the server's 5001 port to receive any messages that were received by the server including the client's echoed message.

The below screenshot shows client 2 who joined in the middle while client 1 was sending messages to the server.



Section 4: Putting it all Together

Given that most of the features were implemented in sections 1, 2, and 3, the main part I must consider in section 4 is to decide where to calculate and update the game state and how to share information between the server and client through the network. I have decided to have the game server calculate and update the entire objects in the game state. This means, the client only has the objects ready to update their positions based on the received message from the server. When the client first connects to the server through a request, the client will receive a response with the client number of the game. Then, the client will listen to any published message that looks something like this:

```
"MovingPlatform-Position:407.000000/300.000000,
Player0-Color:4278190335-Position:173.500000/475.500000,
Player1-Color:65535-Position:559.500000/475.500000"
```

This message contains all of the information about moving objects in the game and their current positions. While the `MovingPlatform` only contains its position information that is formatted in "x-position/y-position", `PlayerX` which the value of X is the client number, and the `Color` is the color of the client character that the value is from `sf::Color::getInteger()`. As the client already knows all the shapes of all objects in the game, the client simply checks for the name of the object and updates its position. For `PlayerX`, the client keeps track of an array of players with their own colors to update their position and draw. The client also checks if the number of clients has changed and adds the new client to the list. One thing to note is that the list of players includes the running client itself as well. This means the client does not really know which `PlayerX` object in the game represents the running client. Then how does the running client take input and update its position? As all of the updates are done from the server, the client just has to pass the client number that was received at the beginning of the connection to the server. When the client receives keyboard input, it publishes a message to the server that looks something like this:

```
"1, Left"
```

The first value is the client number that is sending the message and the second value is simply the keyboard input that the client has received. As the server receives this message, the client updates its game state and publishes the game state to all clients including the client who just sent the keyboard input message. Since all of the messages were in one long string, it was important to parse the string correctly, especially on the client side. Fortunately, C++ `<iostream>` library supports a sscanf()

function that I can format the string making the parsing much simpler as shown in the screenshot below.

```
int playerIndex;
int color;
float positionX;
float positionY;
sscanf(tokenized[tokenIndex].c_str(), "Player%d-Color:%d-Position:%f/%f", &playerIndex, &color, &positionX, &positionY);
```

There are some tricks that I have used during the implementation of the game due to some unexpected issues. The server runs an infinite while loop to call `Timeline::getDuration()` from the game time to calculate the movements of each component; however, this while loop does not render anything and the program is simple enough to iterate each loop in less than a millisecond. This causes a problem with getting the duration since the `Timeline` uses a millisecond as its time unit which `Timeline::getDuration()` will always yield 0 making the no movement change during its `velocity * duration` calculation. To resolve this, I had to forcefully make the thread sleep for 1 millisecond using `std::this_thread::sleep_for()`. This resolved the problem of the objects having no movement changes, but this may be a potential problem in an environment with a slow CPU and lack of computational resources.

```
bool isRunning = true;
while (isRunning)
{
    std::this_thread::sleep_for(1ms); // Optimized thread execution time
    double duration = timeline.getDuration();
```

Another trick that I have used is related to the unsmooth animation of the input-based character movement. As the moving platform or the character's jump movement (referring to the physics and not the input-based jump trigger) does not depend on the character's input through the network, both movements are smooth and clear; however, the character's left or right movement solely relies on client's input through the network. While it does not seem like the latency is the issue, it seems like the client manually setting the position using `sf::Shape::setPosition()` is causing the animation cutoffs as the character seems to "teleport" to its next updated position rather than smoothly moving. To resolve this problem, I had to add more frames between each movement by tweaking the constants to speed up the time but decrease the moving distance per time (decrease velocity); however, this approach affects the non-input-based movements such as the moving platform and the physics which was extremely hard to get the optimized settings. Eventually, I decided to send a "burst" of input messages from the client side which means the client will publish several copies of the same input message for a single input. This allows the server to move the character object several times at a small distance for a single client input without affecting the non-input-based objects. This trick is not the best as this will be highly affected by congested networks and lack of computational resources causing the client to experience latency.

```
for (int i = 0; i < BURST_PUBLISH_COUNT; i++)
    publisherSocket.send(zmq::buffer(inputMsg), zmq::send_flags::none);
```