

Design Decisions

For Homework 2, my first decision was to decide how to incorporate multithreading into my game loop. I first created a new class, `GameThread`, that could handle my separate threads, then I chose to have the first thread handle the moving platform, since it is independent of all other subsystems. I used a wrapper function to have the thread actually perform the platform movement. I modified my existing `FloatingPlatform::move` function to take a mutex as well as a condition variable from the `GameThread`, then I used the unique lock with the mutex to prevent any deadlocks as well as allow the code to be used in the future if the thread crashes.

I chose to have the second thread handle user inputs that move the player character. I first created an `InputSubsystem` class that stores a pointer to the player character, as well as a function to check for inputs. I again used a wrapper class for the thread to access the newly created subsystem. I also used the mutex and condition variable from the `GameThread` in the `checkForInput` function of the `InputSubsystem` class, as well as a unique lock to prevent any deadlocks here also, and allow the code to be used in the future if the thread crashes.

Next, I chose to implement Timelines into the engine by creating a new `Timeline` class and I use it for the movable objects, rather than having them based on the framerate like they were before. To implement the timelines, I first created a main timeline that is used as the anchor timeline. This is also the timeline used to get the frame delta within the main game loop. Instead of a number tied to the frame rate like before, the moving objects now use this frame delta for movement and it allows the game to run at the same game speed no matter what framerate is set. It also allows the game speed to change (speed up or slow down) with the press of a button (In this case two buttons are used). On the same lines as controlling game speed, the `Timeline` class also allows for pausing the game with a button press, since the movement is now tied to the timelines, which are what actually pause. The game loop continues to run, but all movement from in game objects is paused. Having timelines also allows for the game window to run at different FPS. Increasing the FPS allows for the game to run smoother, but does not affect the actual movement of the game itself.

I tried to incorporate creating a separate `Timeline` within the `FloatingPlatform` class, but every way I tried to implement the `Timeline`, it failed and would either not compile or it broke the game. I believe this is due to my lack of knowledge in C++ because I still think it should be possible. I was able to get around this by creating a separate timeline before entering the game loop and then passing a reference to that timeline to the floating platform when constructing it. This seemed to solve my issues and I was able to make that work. I also wanted to incorporate a `Timeline` for the character class as well, but it didn't make sense for the class and it also failed when trying to implement it so I decided to remove it. Instead using the frame delta provided all of the functionality I needed already so I chose to use that for the player movement.

To implement part 3, I chose to use a REP-REQ model. I chose this model because it allowed me to easily track which client was connected to the server and if it is a new client or an existing client. I did this by using an int to track the number of clients that have connected, then incrementing it each time a new client connected, then that incremented int becomes the ID for the newly connected client. I then use an array of ints for tracking the iteration of each client and using the client ID as the position in the array to associate the iteration with the client. For each server reply message, I have the ID as the first character, then each client can use that to determine its ID, which it will assign to its request message. This allows the server to know if the client connected has already connected before, and then use the same ID that was assigned to that client previously. The clients simply send out a request (either "-1" meaning it is a new connection, or the ID the server assigned it), then it receives a response, which is the list of clients and their current iterations, then the client prints that out.

I also wanted to include a simple way to shut down the server without force closing it every time. I chose to run the server as long a bool variable was true (which is the default). Then I have the clients running a specified number of iterations of their loop. The value is currently an int, so it could be any positive integer up to INT_MAX. The loop will then run on each client that connects and the first client to reach the number of iterations set minus one will send an Exit command along with its ID in the request message. The server will then change the loop bool from true to false, and after that response message is sent, the loop will finish and the server will gracefully shutdown, along with the client that sent the Exit command. Since the server will no longer be running, any other clients will not finish their loops and will need to be shutdown manually. Since the server is dependent on reserving a port rather than the clients, I thought it was more useful this way so I don't have to change ports every time I want to run the server again.

For part 4, I chose to use both a REP-REQ model as well as a PUB-SUB model. I chose the REP-REQ model to be used in a separate thread for both the client and the server as a way for the server to determine connected clients and receive information from clients, then I chose the PUB-SUB model as a way for the server to send out information to all clients. I decided to have the server do all of the environment movement (the static and platforms) and I have the clients each handle their own player movement. This should allow for a smooth experience for the individual players on each client, since their character movement is managed locally, and the server is sending out an update on each iteration of the game loop for any moving platforms. Included with each iteration update the server sends out is the latest known character location from each of the connected clients, which each connected client can then render in their individual game windows.

To accommodate the client information to be sent, I made a ClientStruct that includes an ID, the player character for the client, and a pause Boolean. A ServerStruct was used for the server to send its information to the clients that includes a pause Boolean, a static platform, a floating platform, a vector of ClientStruct type, and a running Boolean. This allows OMQ to send the structs as data in the messages sent to/from the server and clients, then recreate that same struct when it is received. By recreating each ClientStruct received from the clients, the server is able to send all clients the positions of each Character to all connected clients, and by having each client recreate the ServerStruct, each client is able to display the platforms after the server has moved them as well as display each of the other connected clients' player characters.

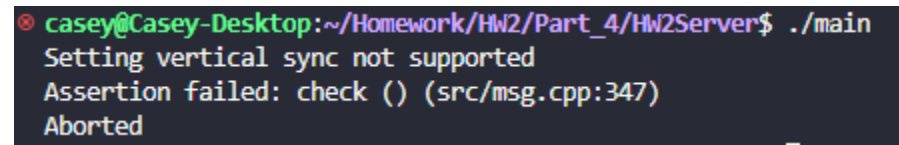
Unfortunately, after over 90 hours on part 4 alone, I was unable to get part 4 to work. I believe that this is due to my lack of knowledge with both C++ as well as OMQ. Although many tutorials exist and there is a lot of documentation for OMQ, I quickly found that most of what I would try would not work. Either the syntax was completely different than the tutorials or documentation would show, or certain functions didn't exist that I was expecting based on the documentation. I was able to get part 3 working, so I know that I am able to send messages between a server and multiple clients, but when I try to run the REP-REQ along side the PUB-SUB I received errors. Currently, both the client and the server for part 4 will compile, but as soon as the server tries to send a message through the PUB_SUB connection, it will throw an "Assertion failed: check ()" error and the client will be stuck waiting to receive from the server (See Figure 1). Again, I believe that this is due to my lack of experience with OMQ and more specifically the PUB-SUB model of OMQ.

I read online that many people were having issues when passing a struct as a ZMQ message, which could be my issue as well. My next step will be to remove all of my structs and try just sending string messages and parsing each item out individually. This seems unnecessary, and I do believe that a struct should work still, but I found many posts on forums about people needing to pass strings instead. Unfortunately, by the time I found those posts, it was too late to implement into Homework 2. I am hopeful that I can implement this into Homework 3, since that builds upon the networking components of Homework part 4.

I did plan ahead with my design for part 4 though which will be useful in Homework 3. I built in functionality to accommodate smooth disconnects when a client quits and disconnects from the server. By using a pointer to a Boolean to determine when the client is running, I will be able to determine when a client exits the game loop, making the last message it sends to the server change the ID to negative. The server will then remove that client from the current connected clients and free up the ID. Because the client information was removed, the server will no longer pass information on the disconnected client to the other connected clients, and they will not draw the character onto their screens.

APPENDIX:

Figure 1:



```
⊗ casey@Casey-Desktop:~/Homework/Hw2/Part_4/Hw2Server$ ./main
Setting vertical sync not supported
Assertion failed: check () (src/msg.cpp:347)
Aborted
```

A terminal window screenshot showing a C++ program execution. The prompt is 'casey@Casey-Desktop:~/Homework/Hw2/Part_4/Hw2Server\$'. The command executed is './main'. The output shows 'Setting vertical sync not supported', followed by an assertion failure message 'Assertion failed: check () (src/msg.cpp:347)', and finally 'Aborted'.