**Writeup for Homework 2**

**200333470**

**yhuang64**

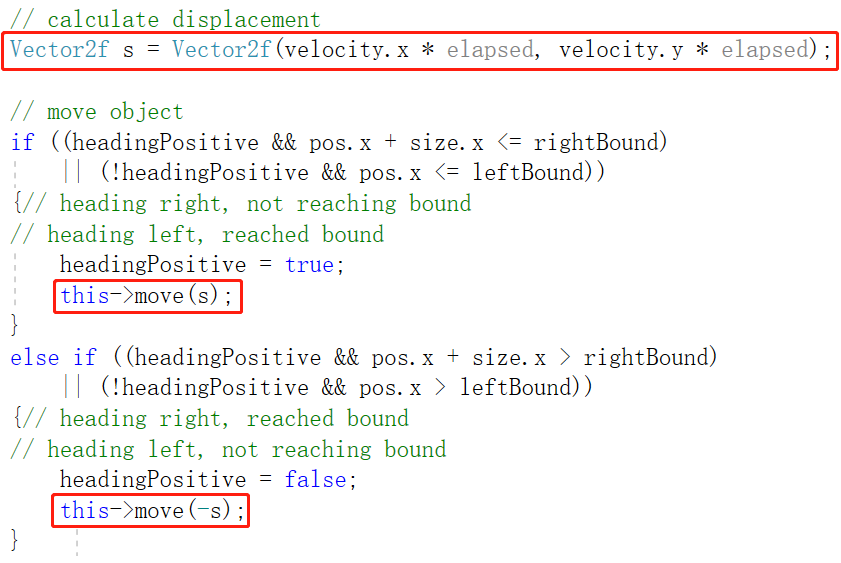
**Yuanmin Huang**

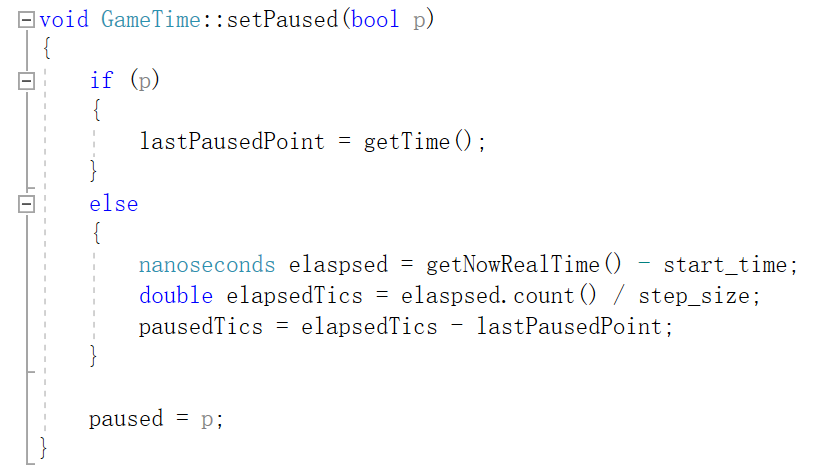
**Section 1**

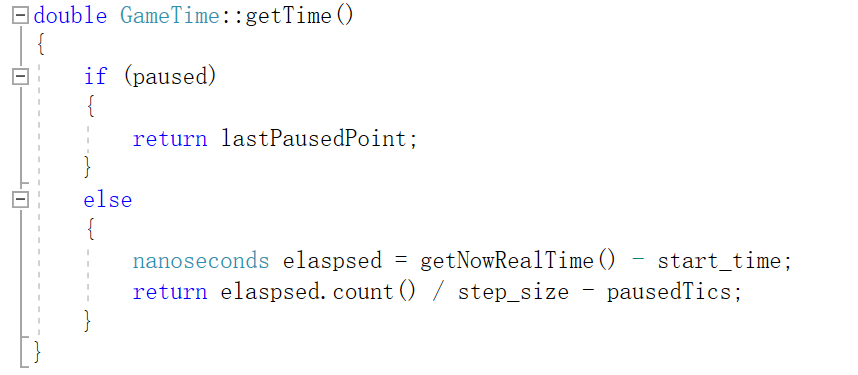
The first section asked us to implement a kind of timeline that was scalable, pausable and on which the movement of a movable object could rely. Thus, I would divide it into three subsections to conquer.

Firstly, it would be refactoring all the movements of movable objects.

Before that, I created three timeline classes according to the notes taken in class. They are Timeline, GameTime and LocalTime. There was one thing to make clear that there was a little point Dr. Roberts had made wrong: the duration\_cast function takes a duration as parameter instead of a time\_point like system\_clock::now(). Thus, I used now().time\_since\_epoch() to solve the problem. Or, it could be solved by take the difference of now() and a time\_point of value 0.

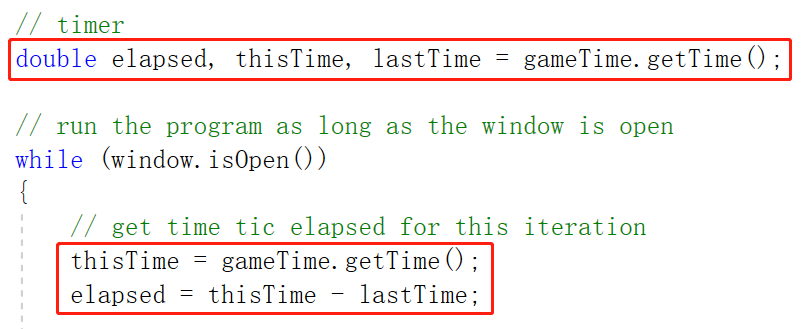
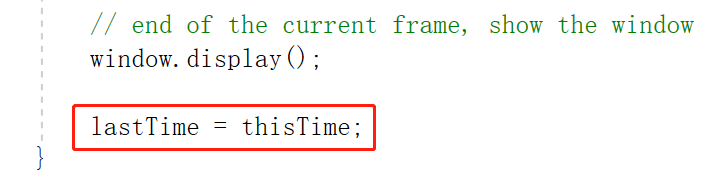
Concerning the refactoring of the movement, I need to refactor the update() function of all subclasses of Movable. I did have a velocity for or movable objects, but my movement was based on iterations of the main loop. So, this time, I must change its foundation to the time elapsed. At first, I added a new attribute *lastTime* to class Movable, which was maintained to help get the time elapsed between each iteration, so that the displacement of objects could be calculated. During the completion of scalable timeline, I moved the attribute to main() as a variable to realize the desired functionality, which I would explain later. Whichever way of implementation is used, the result is the same: getting the displacement of objects and move them accordingly. I also made some changes to the value of the velocity of objects to reach a decent effect of moving based on elapsed time.

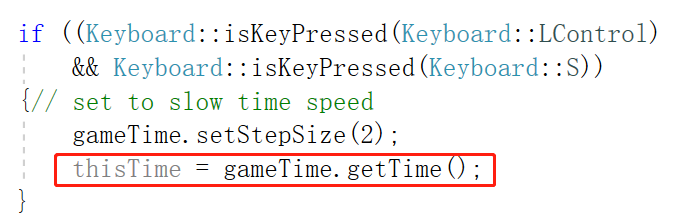
Secondly, I implemented the functionality of pause operation. To achieve this, I added bool paused, double pausedTics and double lastPausedPoint to class Timeline to indicate the status of timeline, the paused time tics and the last time point a pause command was triggered. Moreover, I added a setPaused() function and modified the implementation of getTime().

Take class GameTime as an example, when a pause is triggered, the function setPaused() calls getTime() to record the paused time point and sets *paused* to true. Then, getTime() would always return the paused time point when the timeline was paused, so that all moving objects wouldn’t move because the time elapsed would be 0. Moreover, the character won’t move fast suddenly after resume the game because its velocity won’t be accumulated during the pause. This is implemented by adding acceleration \* elapsed time instead of acceleration directly to the velocity of a character when the detectCollision() function is called.

When an un-pause is triggered, function setPaused() would update the paused tics and set *paused* to false, after which calling getTime() would get virtual game time as real time elapsed minus the paused tics.

There is one more thing to point out for pausing is that I made the program to sleep 0.2 second to prevent multi-triggered pause commands. This is a tradeoff because I must suffer from either disordered paused commands or slightly stuck game experience.

Thirdly, it’s the scalable part, whose goal is that the bigger the *step\_size* is, the slower the time flies in game. To implement this, the design in Timeline class is trivial, which is adding a setStepSize() function which resets *step\_size*. However, I met with a problem that the moving objects would fly away right after resetting the *step\_size*, which I found out was caused by the sudden change of the return value of getTime(), which caused the sudden change of elapsed time. Therefore, I pulled the attribute *lastTime* out to main() and maintain *thisTime (got from calling getTime() at the beginning of each iteration)* and *lastTime* every iteration of the main loop (shown in the picture above). Then, I just need to pass the elapsed time to update() to help calculate the displacement. Moreover, at the end of each iteration, I should assign *thisTime* to *lastTime*.

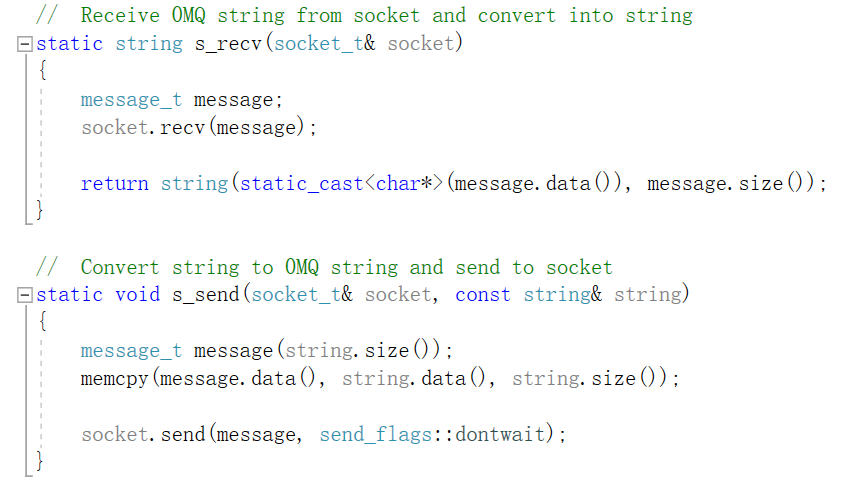
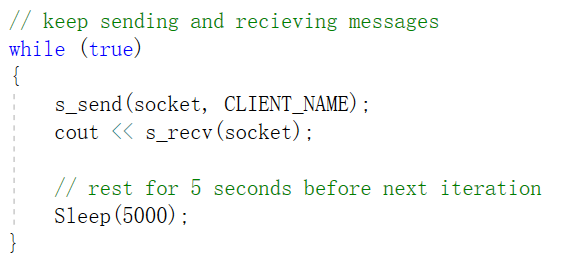
Here comes the non-trivial work. After resetting the *step\_size*, *thisTime* should be reassigned by calling getTime() to narrow the gap between *lastTime (would be assigned thisTime at the end of this iteration)* the new *thisTime* get in the next iteration.

The last work is to set up a mapping of keys and new operations (pause/un-pause, set time speed), which is easy to implement and would be written into README.txt.

**Section 2**

I divided this section into two parts to implement. The first is to implement a single-thread version of a client-server model which has the functionality of interaction between a server and several clients. The second is to implement a multi-thread version in case the server is blocked by one request.

For the first part, I referred to the 0MQ tutorial and took the skeleton of the hello server and hello client program. I also referred to some helper functions which dealt with sending and receiving strings on [GitHub](https://github.com/booksbyus/zguide/blob/master/examples/C%2B%2B/zhelpers.hpp#L76-L95). I tried to run it but failed for some reason. I guessed it was because there was some difference between the interfaces of recv() and send() function of different versions of 0MQ(, which seems weird to me. The interfaces are supposed to remain unchanged, aren’t they?). After I modified a little bit according to the error message, I got codes like this.

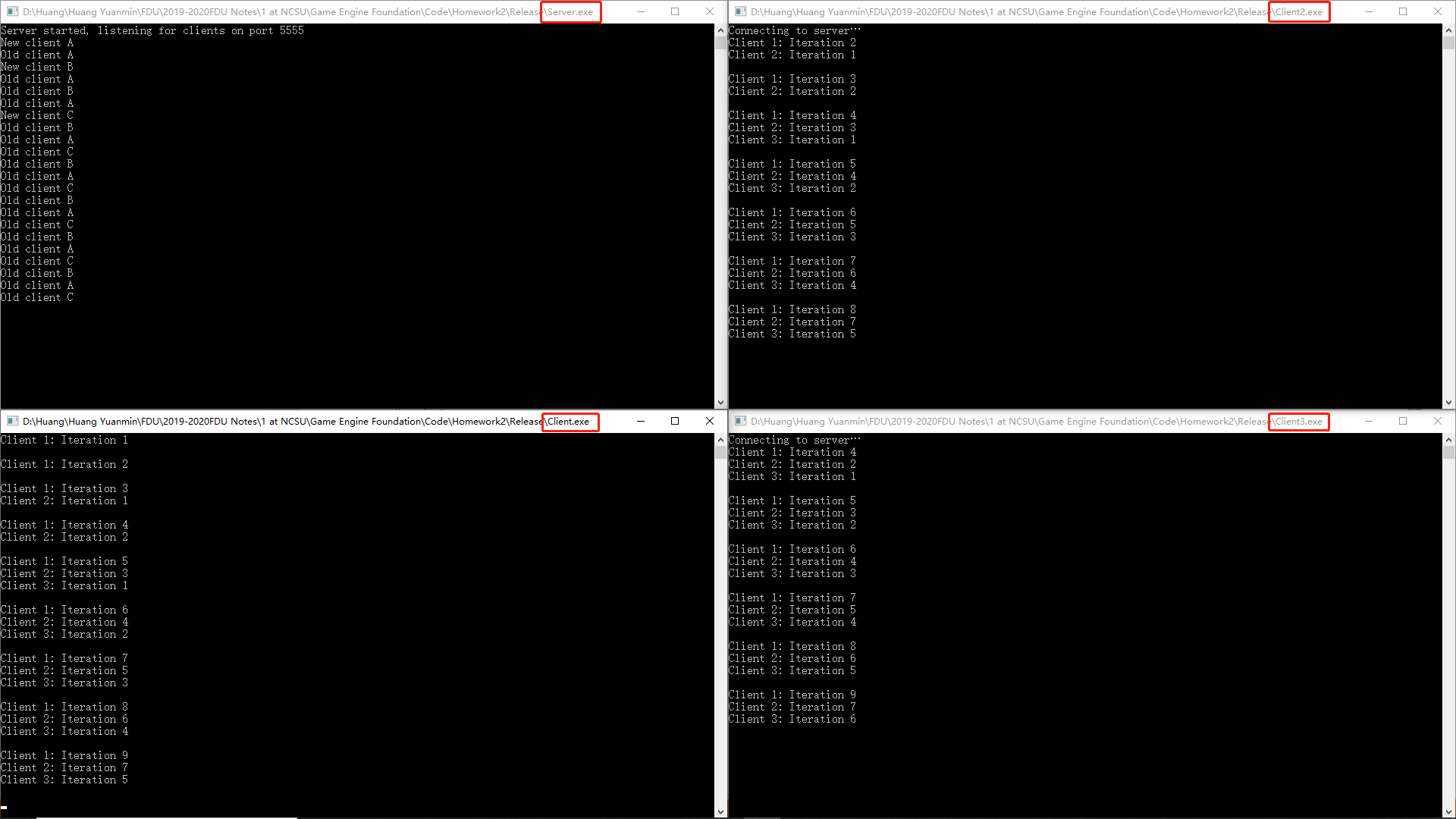
Then, clients are simple in my design. In one iteration of the loop, they send their names to the server and receive response from server. After sleeping for 5 seconds, next iteration begins. I made the name of client constant value so that I could easily make duplicates from one client project.

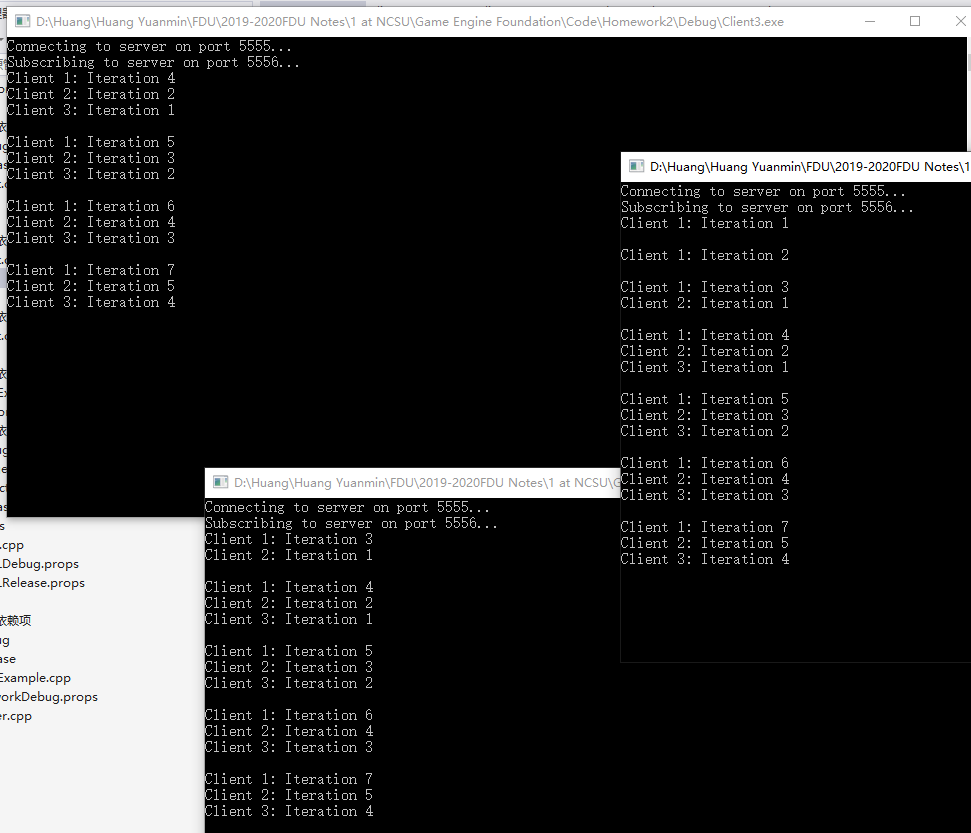
For the server, it maintains a vector of Clients (including information of client name and the count iteration) so that it can tell how many iterations a client has gone through. Every time when there is a request, the server update the vector according to the client name received. Next, a string message is generated from the information given by the vector and sent to client.

For my requests of clients are not so intensive and the time consumed for each response of server is not so long, my design already can work quite well under 3 clients connected to the server simultaneously. (Shown in the screenshot below)

Next, I tried hard to implement a multithreading server which can respond to various requests from clients.

According to my previous experience on multithreading socket programming, the server would have one socket listening for requests, and then it distributes tasks for each request out as new threads so that the listening socket wouldn’t be blocked. Therefore, I tried to implement my server like this, but got an uncaught exception after I created a new thread to handle request got and listened to next request. I found the errno to be 156384763, which means error EFSM: the socket not being in the appropriate state. I recognized that in ZMQ, a request received must be followed by a reply sent out, or an error would occur, which made my implementation of multithreading impossible to work.

Thus, I went to the office hour and showed Dr. Roberts my work. It turned out that my implementation had some difference with the expected behavior. The clients should be receiving broadcasted messages at the same time instead of requesting for replies themselves, which might cause inconsistency of state of clients.

Finally, I found a way which combines REQ/REP model with PUB/SUB model to implement the functionality expected. The REQ/REP model is used once to build connections between clients and server, while the PUB/SUB model is used to publishing current state of connections to all subscribers in a separate thread. In this way, a server is either a replier or a publisher, and a client is a requester and a subscriber, which receives messages and prints them out. The implementation can be seen in source code, and here’s a screenshot when running.

**Section 3**

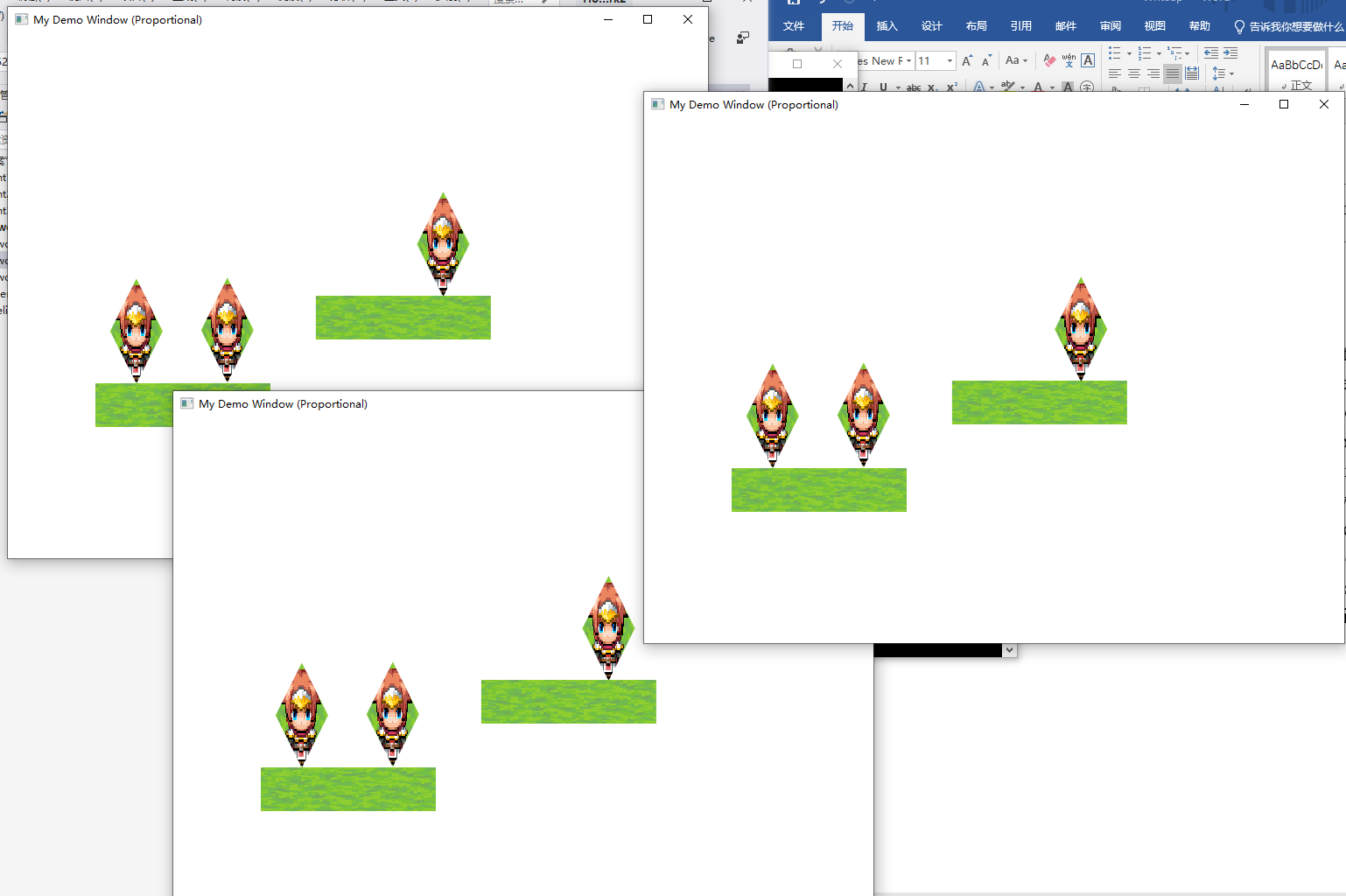
My main strategy for this part is letting the server keep receiving messages from clients which indicates current position of each client’s character and publishing messages which contains current positions of platforms and all the characters of all clients as well as different current time for each client (would be useful for dealing with asynchronicity).

At the meantime, clients are responsible for calculating positions of their own characters and keeping sending the positions to the server as well as subscribing and analyzing the message received from server, then updating positions of all the local objects accordingly.

To avoid combining codes for networking with those for SFML, I created Server and Client classes to deal with networking issues and instantiated a server/client in the main function of server/client project.

To achieve a reasonable frequency of update of information, I made both two main functions multithreading. On the server end, the receiving function is assigned to a new thread to keep track of information of newly connected clients as soon as they connect to the server, while on the client end, the subscribing function is assigned to a new thread to update the positions of platforms and characters.

One of the tricky things I encountered is that the main loop of server end is running far faster than the one of client end because it didn’t need to deal with SFML windows, which makes the publishing function called at the end of each loop of server being executed too frequently, causing the client end being too stressed. Therefore, I made the publishing function sleep for 16 milliseconds so that it wouldn’t be executed more than 60 times in a second, which I thought would be a reasonable fresh rate.

Another thing worth to mention is how to deal with temporary variables created when analyzing subscribed messages to create new characters for newly connected clients. I used to instantiate characters directly in client functions and use list of pointers to store all the characters in the main function. However, it turned out to be unacceptable to store the address of a temporary variable. Finally, I came up with a way which only stored the map of names and positions of new characters and instantiated them before drawing in the main loop.

The other things I think are just the normal coding that needed for implementing the correct functionality so I would not list them out one by one. And I gave out a screenshot of my three clients running together above.

**Section 4 (Optional)**

If I have enough time to dig deeper into this assignment, I would be quite willing to implement this part.

Moreover, I think that I have done many preparing works for implementing this part, such as maintaining separate local time timelines for different characters of clients on the server, having flexibility for adding time step information to passed messages and the corresponding way on server to handling changing in step size.

However, I have already spent too much time on this assignment, and I also have many other works to do. Thus, I think I shall stop here and leave more time for next assignment.