**Writeup for Homework 3**

**200333470**

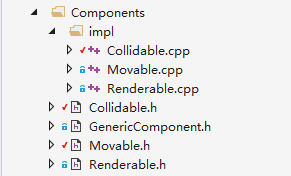
**yhuang64**

**Yuanmin Huang**

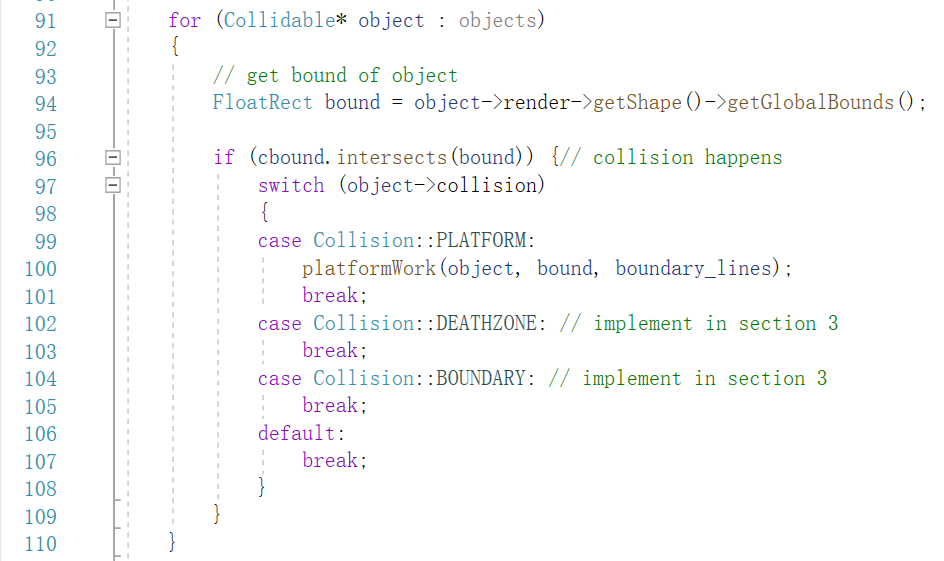
**Section 1**

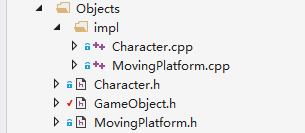
In the first part of this homework, I decided to implement a model which is close to a generic component model. In my design, I have a bunch of components that provide different categories of functionalities and a bunch of game objects composite a map of generic components (component type and pointer) to utilize the functionalities of components. I said my model is close to a generic model because some game objects may need some extra self-defined attributes other than the pre-defined components, which makes the model not purely a generic component model.

For the components, I first designed a GenericComponent class to be inherited by all the other components. The header file contains nearly nothing except for a enumerate class of type of components and a virtual function work() to define different behaviors of each component. The class structure is as shown below.

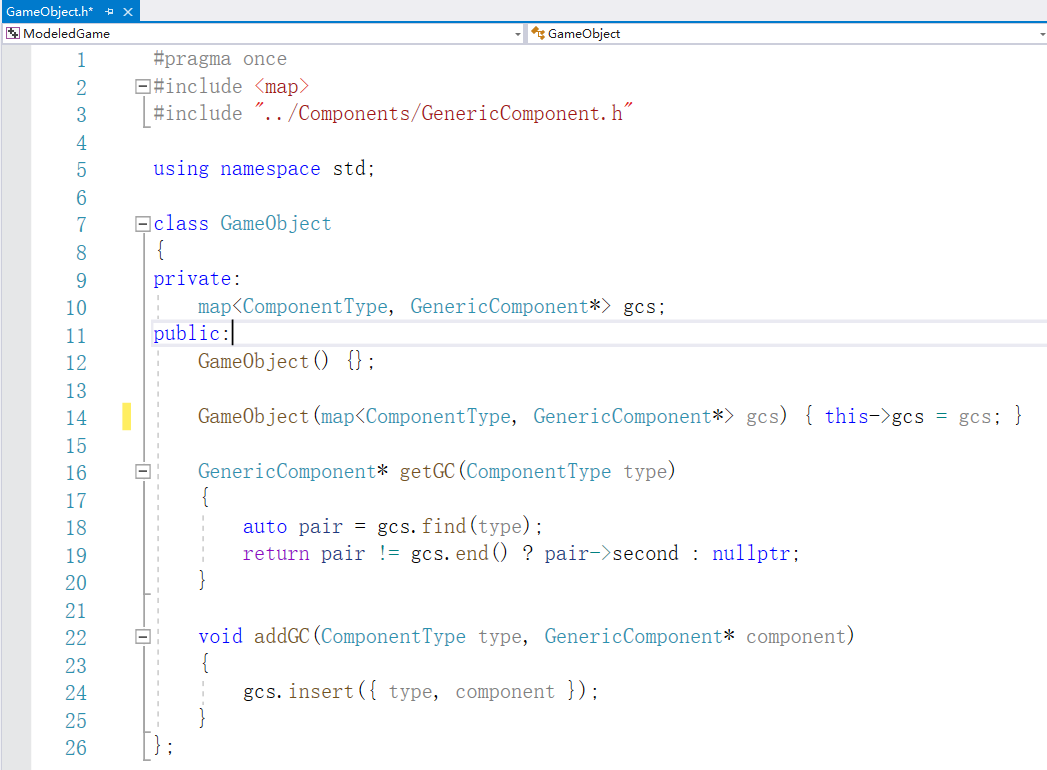
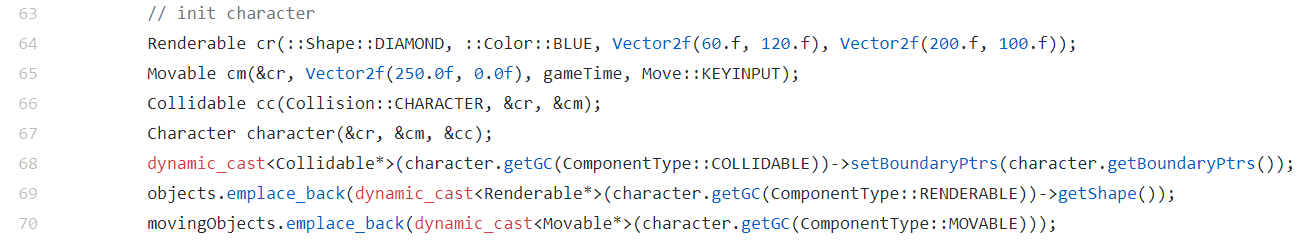
At first, I wanted to design an extensible function work() using … parameter of C++. Later, I found it too complicated and not practical to do so, so I turned to implement overload functions in subclasses instead of override ones so that I could have changeable parameters.

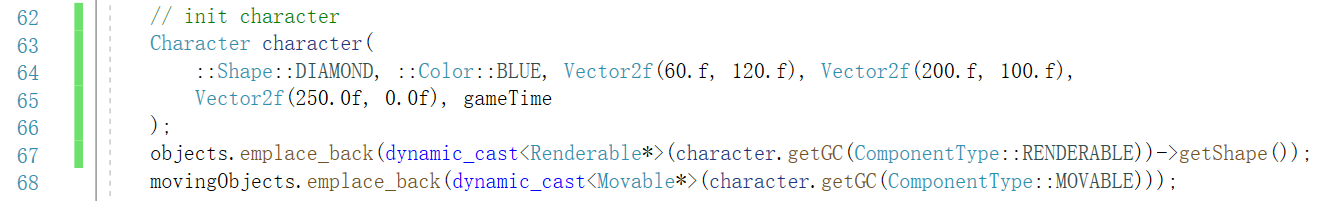
As is shown above, I have three kinds of components:

1. Renderable, which is responsible for defining an object’s shape and color. I provided three shapes: rectangle, diamond and circle and three colors: red, green, and blue. They are pre-defined enumeration elements which can be chosen by users when constructing objects. Within this component, the size of the shape can also be decided by users.
2. Movable, which is responsible for the movement of objects. Thus, the work() function within this class is used for moving. Obviously, to move an object, the class composites a pointer to a Renderable object to gain access to the object to be moved. To indicate which specific type of moving the object wants to implement, I defined HORIZONTAL, VERTICAL and KEYINPUT three ways. The work() uses a switch-case to decide which actual move() function would be called to move the object. As for the three types, KEYINPUT simply deals with character movements using keyboard input, while the other two deals with back and forth movement in either horizontal or vertical direction of platforms. To make it more extensible, the speed, start point and range of the movement can be defined by users easily by passing corresponding arguments to the constructor.
3. Collidable, which is responsible for dealing with the collision of objects. The work() function here works more like detectCollision() in previous homework. Again, a collidable object needs to be renderable and may need to be movable (for platforms and characters). Therefore, there are pointers of Renderable and Movable in this class. An enumeration class with CHARACTER, PLATFORM, BOUNDARY and DEATHZONE is defined to indicate the actual type of the game object compositing this component, which would be used to lead to different behaviors when collision is detected. I only implemented the function for PLATFORM here, and the other two would be implemented in section 3.

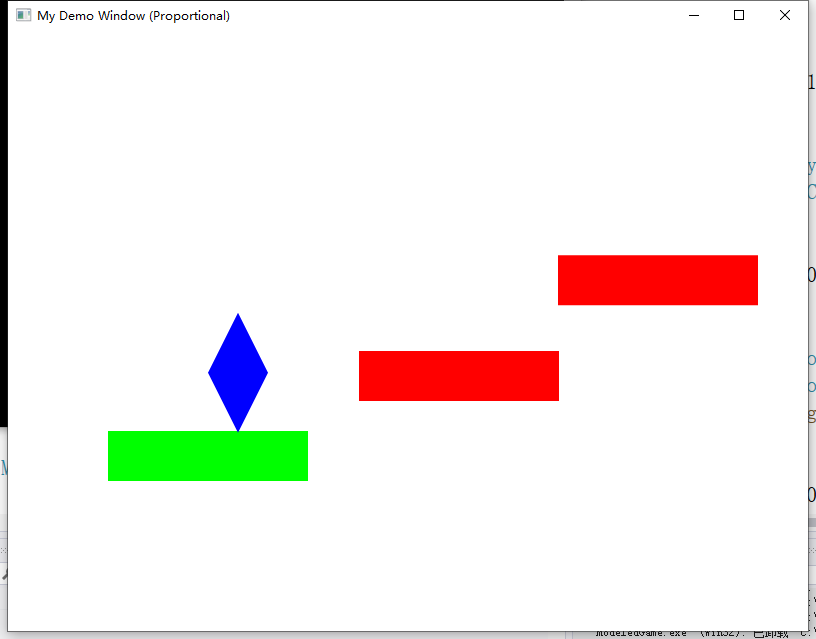
Then, for the game objects, similarly, I designed a GameObject class to composite the common map from ComponentType to GenericComponent\* and functions to add and get GenericComponents.

MovingPlatform and Character are the only two classes of game objects till now (, static platforms can be implemented using moving platform of velocity zero). They all composite the three components designed above to implement the functionality they need. Specifically, Character has a vector of four Collidable pointers to indicate if the character is having collision with any collidable objects in all four ways. The pointer to the vector would be passed to the character’s Collidable component to implement the functionality of work().

Originally, I made the constructor of game objects relatively simple, which only receives pointers of pre-constructed components and stores them by calling addGC() of GameObject. However, this makes the construction of game objects looks quite heavy in main.cpp.

I felt that this part should be encapsulated into game objects as it is part of the construction of the object. So, I refactored the constructors and got the code as follow.

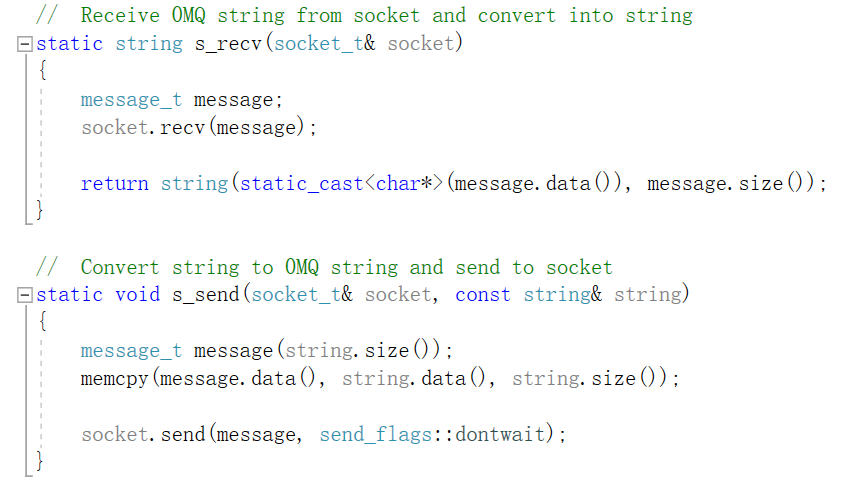
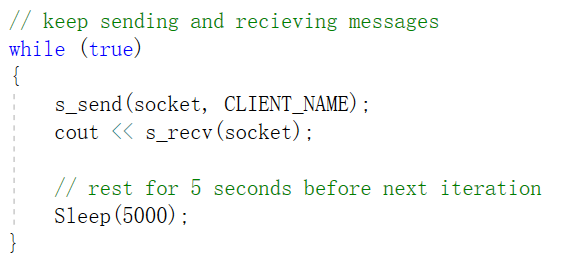
The new constructor looks like this. The similar refactor was done to MovingPlatform as well.

After you run the project, the effect is supposed to be like the following picture.

**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.

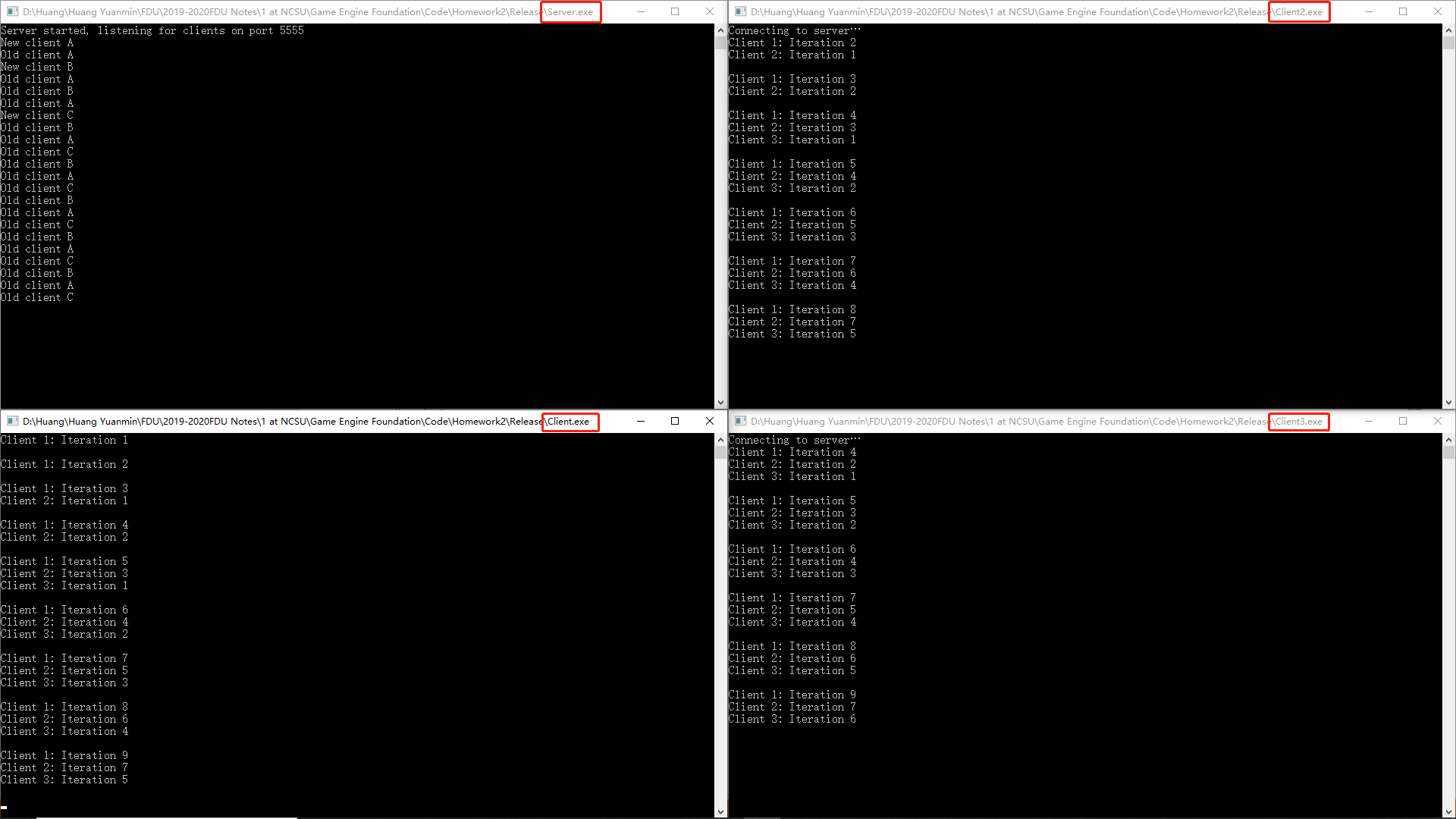
Then, clients are simple in my design. In one iteration of the loop, they send their names to the server and receive response from server. After sleeping for 5 seconds, next iteration begins. I made the name of client constant value so that I could easily make duplicates from one client project.

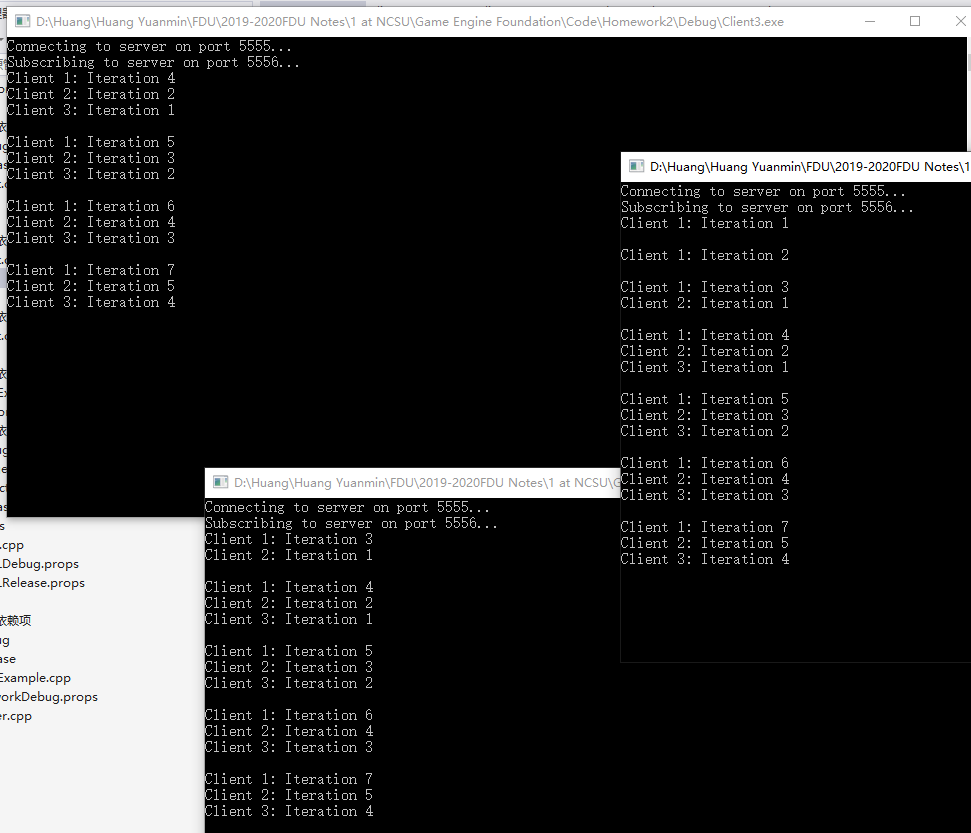
For the server, it maintains a vector of Clients (including information of client name and the count iteration) so that it can tell how many iterations a client has gone through. Every time when there is a request, the server update the vector according to the client name received. Next, a string message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

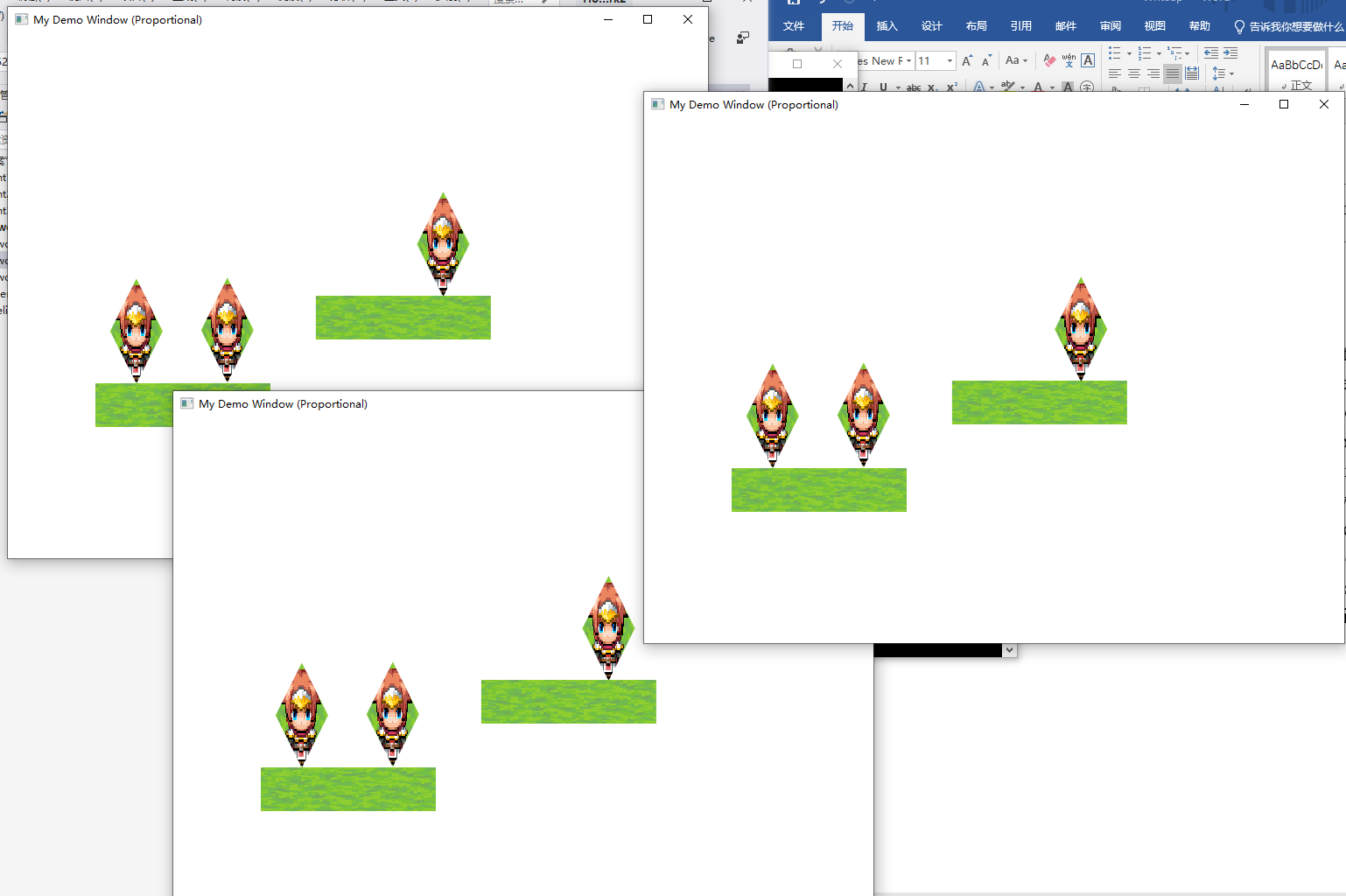
My main strategy for this part is letting the server keep receiving messages from clients which indicates current position of each client’s character and publishing messages which contains current positions of platforms and all the characters of all clients as well as different current time for each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