**Writeup for Homework 4**

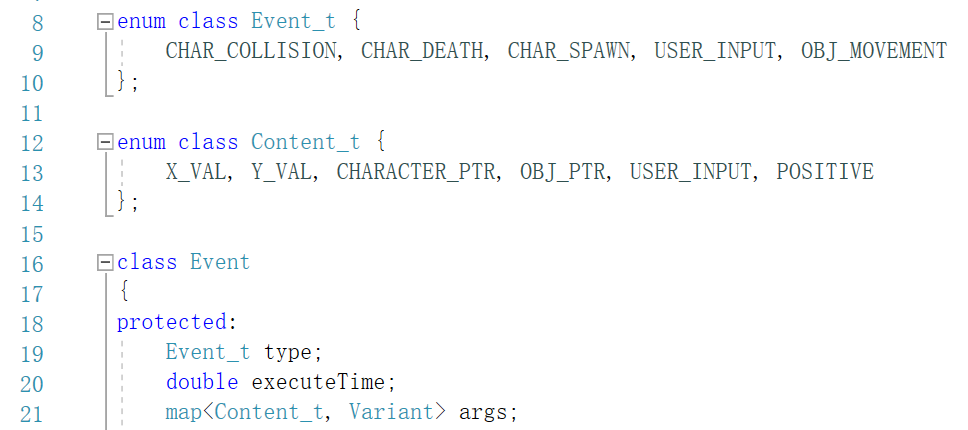
**200333470**

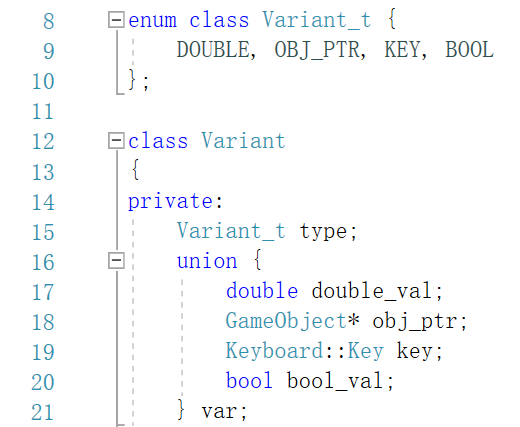
**yhuang64**

**Yuanmin Huang**

**Section 1**

The work for the first section is to implement an EventMgmt System, so let’s make the writeup begin with the event class design (**representation**).

The base class for all events is an Event class. It contains a type, an execution time and a map of arguments, which is content types to variants. The content type here indicates which type of argument (semantically) the corresponding variant contains.

This leads us to the design of Variant class. As is shown on the right, it contains a variant type (syntactically) and a union of values to store the actual value of the variable.

Based on this, I then designed 5 types of events. They are:

1. ECharCollision,

Which has a character pointer and the pointer of the game object character is colliding with;

1. ECharDeath,

Which only has a character pointer;

1. ECharSpawn,

Which has a character pointer and the pointer of the SpawnPoint the character is going to spawn at;

1. EObjMovement,

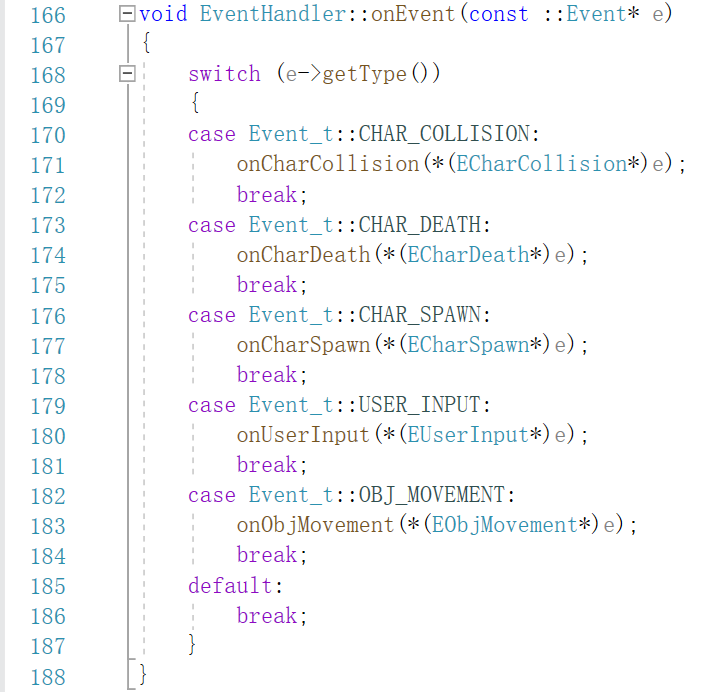
Which has a pointer of game object which is going to move, the position (x\_value, y\_value) it’s going to move to and an optional boolean value of heading positive used for moving platforms;

1. EUserInput,

Which has a character pointer and a Keyboard::Key value to indicate which key has been pressed.

If we want to add new class of events to our engine, we simply need to add new event types to the enum class and define which kind of arguments they would need to composite, which is quite easy to implement given the pre-defined Variant and Event class. We can surely add new Variant types and content types to enum class if needed.

Then, let’s take a look at how we handle the events, i.e. the event handler (**handling**).

I have designed a giant event handler that has a general onEvent() function which forwards events to specific onType() functions based on the event type. I chose to implement it this way because I wanted to utilize the enum class of event types and simplify the design for registration in event manager. In this way, the event manager would only need to call the onEvent() function of event handler when handling events instead of notifying a list of separate handlers.

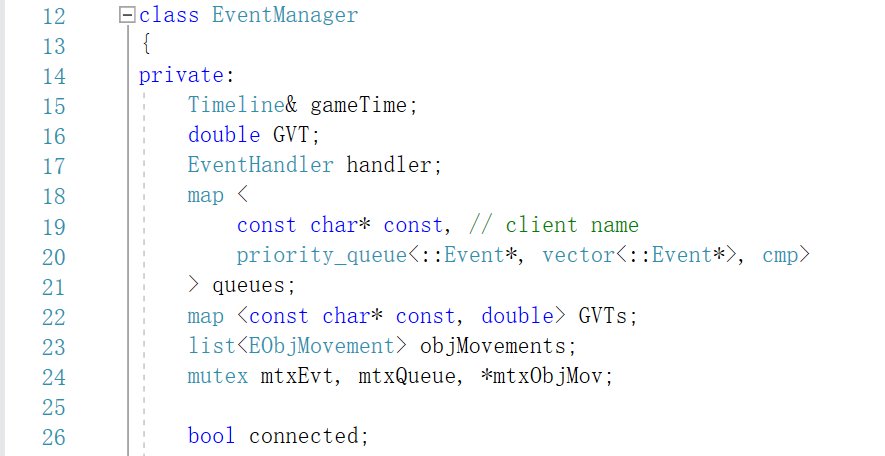
As for each onType() function, the work we do here is mainly copying the code from previous assignments because the functionality we want to implement hasn’t been changed.

Things new here are that when handling some events, they may generate new events. We have several situations here:

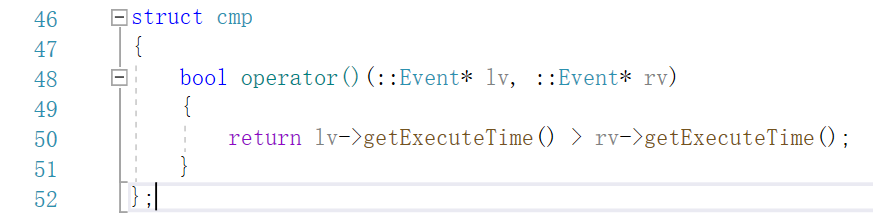
1. When handling the collision of a character with a death zone, a new ECharDeath would be generated;
2. When handling the character death, a new ECharSpawn would be generated;
3. When handling the character spawn, a new EObjMovement would be generated.

After discussing how we handle events, let’s move on to when we handle the events, and that would be event manager’s work.

My design of EventManager mainly has a reference to game time object, a double value of GVT, an event handler (**registration**), a map of client names to their event priority queues, a map of client names to their GVT and a list of EObjMovement events used for network communication.

The key idea here is that we:

1. Maintain a priority event queue sorted by execution time descend;

This is implemented by pass a customized compare function on Events.

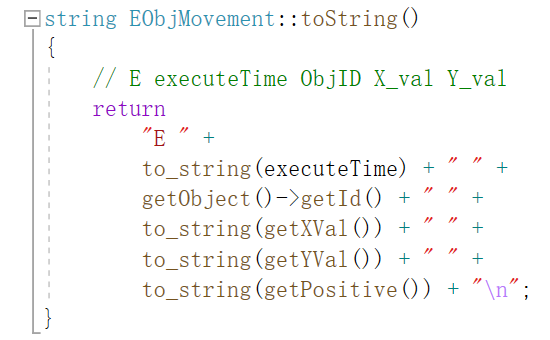
1. Update GVT every time before we handle events;

The work we do here is simply comparing the GVTs stored in the map and taking the least one as our GVT.

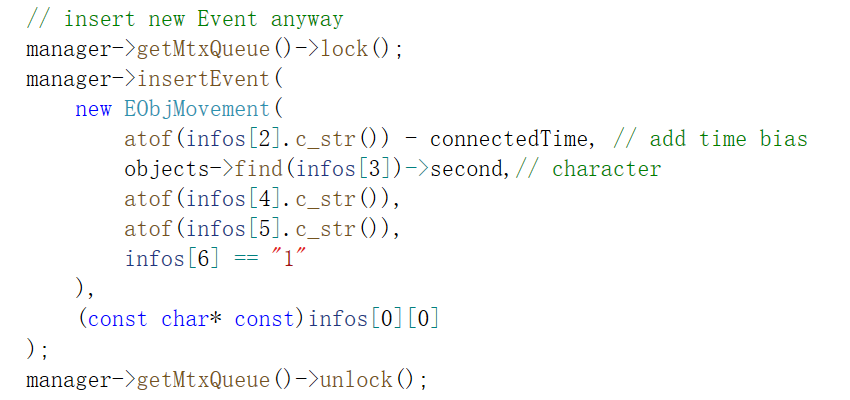
1. Handle events according to GVT;

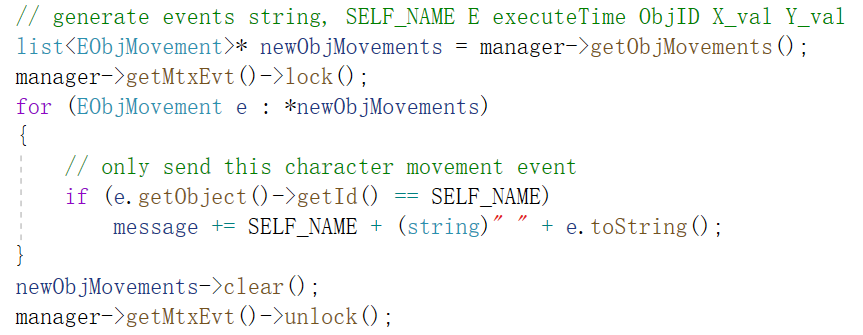
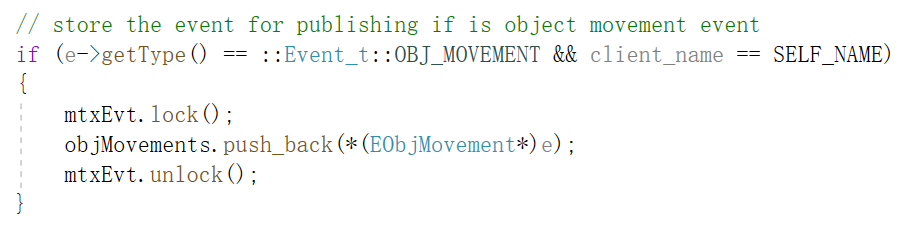
For every priority queue in the map, we keep handling the events on top of the queue until the top event’s execution time is larger than GVT. After an event is handled, remember to delete the pointer because it’s “newed” and pop it out.

Finally, what we need to clarify is when do we generate event (**raising**). The simple answer is that when we occur where we originally deal with directly in our previous assignments. For example, in Movable component, when a platform moves, we need to generate a new EObjMovement event and store it in the event manager, while we simply moved the platform in the past.

Another important design in this section is how my networking part cooperate with the event management part.

My thought is that there is no need for other peers to know all the events occurs on my machine. The only thing important other machines need to know is the object movement happened on my machine. Here comes my purpose of designing the list of EObjMovement in my event manager. This list is used to store all the object movement events and then be used for communicating with other peers in the network to notify others how the objects of mine have moved during the short amount of time passed.

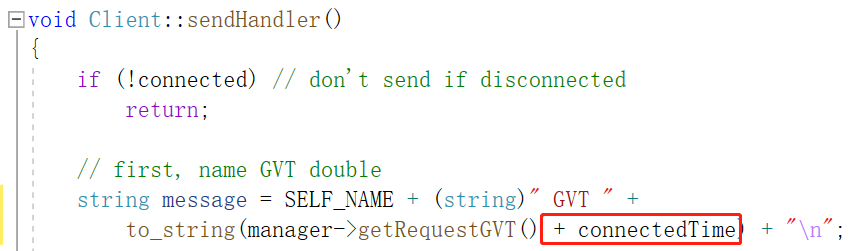
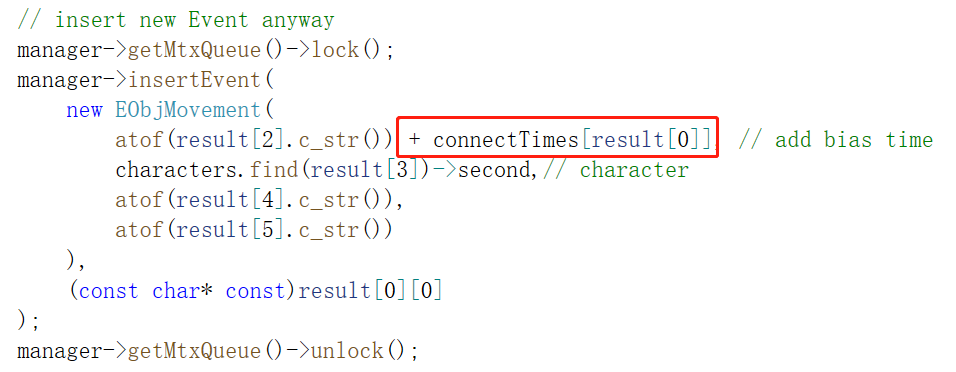
This calls for a way to transport EObjMovement across the internet. I didn’t choose to serialize it, instead, I wrote a toString() function to convert it into a string which contains essential information of the event. Then, in the other end of the network, the peer can use the string to generate a new event equivalent to the original one.

The list of EObjMovement events is maintained by storing the object movement events of self’s and clear up all the events in the list after sending the corresponding strings out.

The tricky thing here is concurrency control, because we must avoid storing events after we have generated strings but before we clear the list, or we will lose some of the events, which may cause a short slowing down of displaying on other peers.

As you may have noticed in the code of parsing string to generate events, there is a “ - connectedTime”. This is another important part of networking communication structure: the time control.

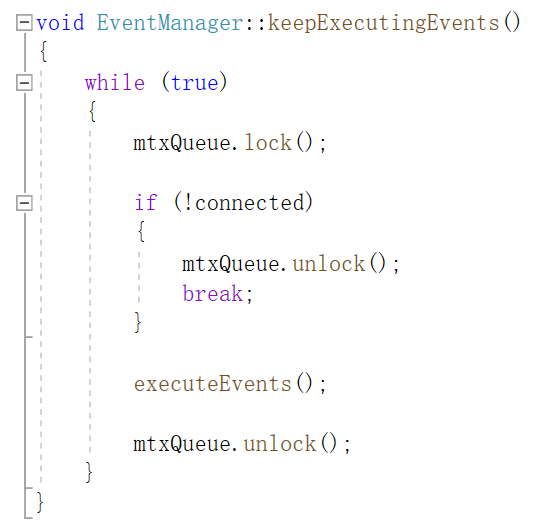
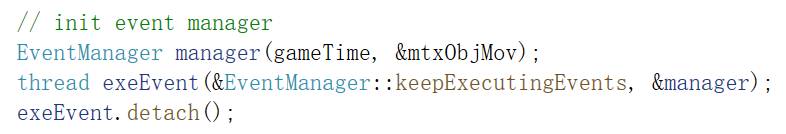
As different clients can start at various time points, the timeline of their event managers rely on would differ from each other. So, the current time we get may not reflect the actual current time. My design is to let client first send its time point of connecting to server, then server records the bias between server time and client time by computing the difference between the time client has sent and the time of server’s. Finally, the server sends the bias value back to client so that client also knows the value.

Later, the value is used when sending and subscribing time-related messages to and from server. My goal here is to keep the server receiving and publishing all the time value according to server timeline. Therefore, when a client sends message out, it needs to add the bias to the GVT; when the server receives the message about events, it also needs to add the bias to the execution time. When the server publishes message, it directly sends all the event strings out; and when a client subscribes the message, it subtracts the bias on all time values. In this way, we make all the timepoints agree with each other on when events truly happened.

But, we need to understand that this only works when SERVER STARTS FIRST because the client wouldn’t send the connect message several times. It would only send once and wait for the server to respond, so the it would only send a time point really close to its startup, which means that the server would not be able to get a big negative value of time point if a client starts really early before the server.

In addition, you may have notice that every time I insert an event to event manager, I add a mutex. This is due to another tricky concurrency control which lies in handling events meeting with inserting events. Inconsistency of GVT computation at one execution may occur when an event is inserted to the top of a queue (this may happen because of network communication latency) after we have set the GVT and begin to handle events.

This may also become a matter when a client disconnects, and we need to erase the corresponding queue and GVT. Imagine that we suddenly remove a queue when the event manager is working in a loop of all queues. Unexpected errors would occur if we don’t add any concurrency control to it.

Originally, I tried to implement a sequential event management that I only call executeEvents() once every time the client receives a message. After running it, I found that that was too slow, and events would be clustered. Hence, I wrote another function to keep executing events and made it a new thread. This works much better than the original design. I think that this echoes what we have talked about multi-threaded design in class.

However, may be because of the multiple threads server and clients have, when I try to run multiple clients and the server on my computer, whose CPU has only dual-core and is running at 100%, it doesn’t work well.

When one client is online, there are only some slight slow downs at some random point, but the character can still move smoothly. When two clients are online, the situation becomes unstable that the movement of one character may have somehow visible latency to reflect on the other client. Also, there is a chance that the character fall through the platform and death zone because of the insufficient running loops of collision detection. When there are three or four clients running on my computer at the same time, the game is unplayable, and the characters will surely fall through the platform.

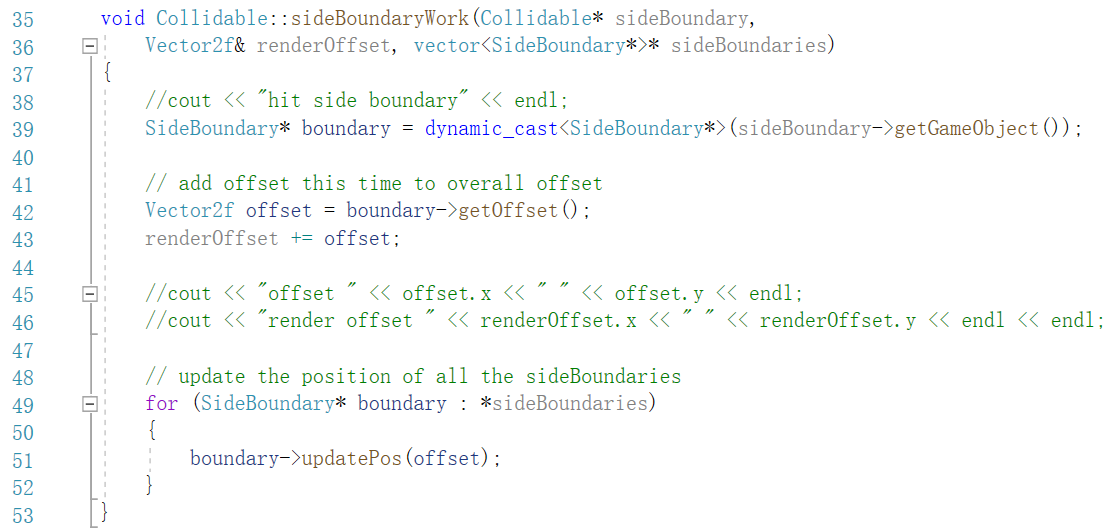
I hope that my program would work better on a computer with better CPU. I think that it should do because the lack of cores would be removed, but I don’t have a chance to test. Also, I think that originally, it is unrealistic to run multiple clients together with the server on one machine. And to test the functionality of event management system and network design, running two clients simultaneously should suffice.

**Section 2**

**Section 3**

What I have done for this section is mainly some add-ons to the object model I have designed for section 2.

I implemented SideBoundary, SpawnPoint and DeathZone for this section.

SideBoundary is a class designed to define a side boundary in the game. My goal is to set two boundaries at the left and right side of the window. They are both 100 units far away from the side (this value can be set to whatever reasonable value by game designers). When a character collides with a side boundary, it will then add a value of offset to the variable render offset in main.cpp. The value added is computed based on the given window size, how far the side boundary is from the side of the window and which side is the collided boundary on. The render offset is later used to add to the position of objects other than the character of the client before being drawn. In this way, I made my client move to “next” view relatively. Of course, when a collision happens, the side boundaries should also “move” with the character, which means I would add the negative offset to all the side boundaries. Actually, what I have described above is mainly implemented in the work() function of Collidable component.

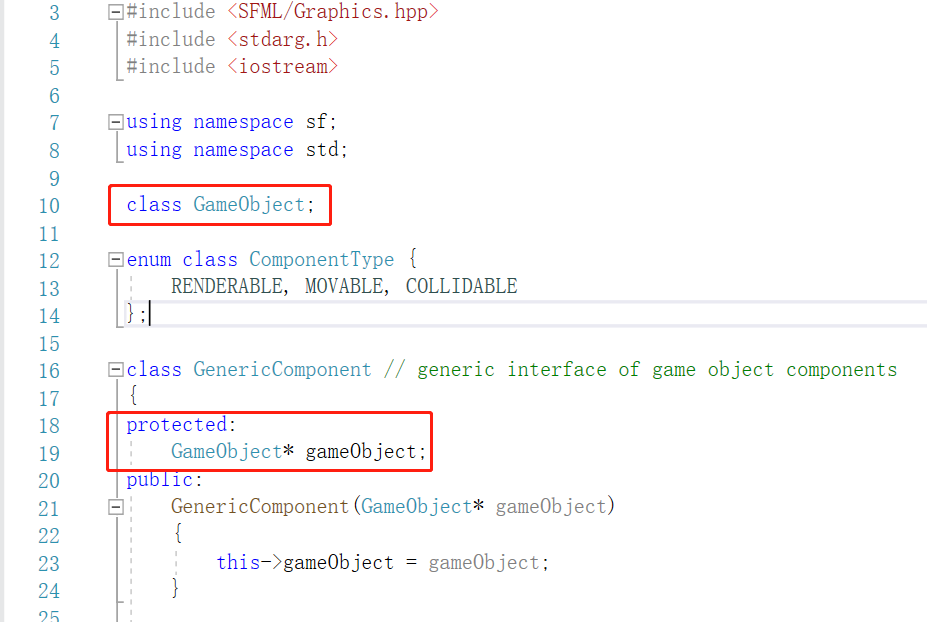
The design of SpawnPoint is quite easy because it needs nothing, but a position defined in Renderable component. (I still make it have a Renderable component even though it needn’t to be rendered.)

Finally, the DeathZone is an object with Renderable, Collidable. When a character collides with it, the character should be transferred back to a random spawn point in the list in my design, which is quite plain.

However, when I was implementing my design on the original generic component model, I encountered big problems which pushed me to refactor the model. The problem happens when I want to cast a pointer to a SideBoundary type in Collidable.cpp when implementing the work() function. I found that I needed to include the header file of SideBoundary if I wanted to do so. Whereas, SideBoundary had included Collidable.h, which made it become a recursive include. It made me impossible to link the cpp files together when compiling.

This made me to reconsider my design. I realized that I should composite a pointer of the game object in my game component so that I could gain access to the data I need when implementing some of the behaviors. My original design asked me to keep several attributes of all the game objects may be possible to composite the component in the corresponding component, which I found is inextensible and redundant.

Although the new approach couldn’t prevent me from recursive including, I could at least remove all the redundant attributes in my components and no longer need to worry about the inconsistency of attributes values in components and corresponding game objects.

As for the recursive including, I googled the problem and found out that it can be solved by declare a incomplete definition of game object class in component’s header file and then include the header file of objects in cpp files of components.

As a result, I designed a game with size of 1600 \* 600. Four death zones stand around the area, with which character collides would cause a transfer back above the static platform in the first view. In the second and third view, there is a horizontally moving platform and a vertically moving platform respectively. The window size is 800 \* 600, and the padding of side boundaries from the side of window is 100. Thus, in my design, every time a character collides with a boundary will cause a 600 change in render offset.

Moreover, after I started to run the whole program, I found that there were some instants that when the character was in the second view, the static platform in the first view emerged. I tried to print out the value of the render offset because I thought that it would be caused by render offset being modified unexpectedly, but I found nothing wrong. Until I realized that this was caused by read/write conflicts of multithreading. There were surely be moments when the main thread was rendering the window, the position of the platforms were modified by subscribeHandler() of the client. Therefore, I added a mutex to prevent these conflicts and protect the platforms from being modified when the main thread was drawing them. After that, everything moves as expected.

**Section 4 (Optional)**

Though I may not have the time to implement this section, I thought something about it.

For the networking model, one way to implement it would be only transferring the position of objects and generate the actual object accordingly on the other end. The other way would be use json or other libraries to transfer a whole serialized object and only needs decoding on the other end.

Then what I should do would be creating objects and run the program to measure the time used under different number of objects and under different transfer model.