**Writeup for Homework 5**

**200333470**

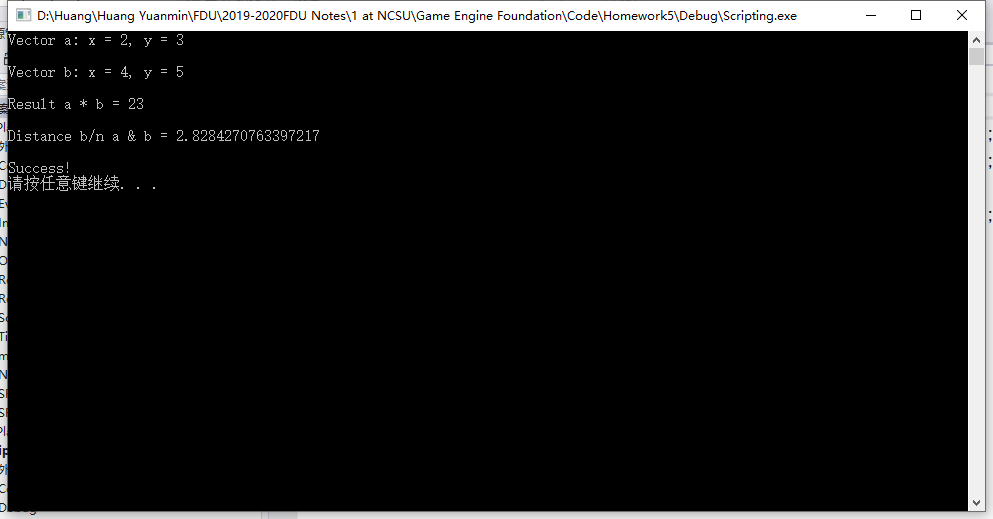
**yhuang64**

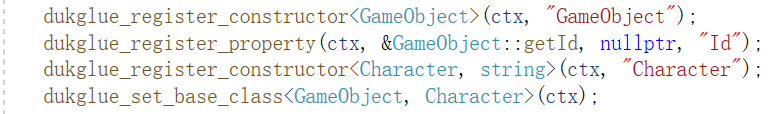
**Yuanmin Huang**

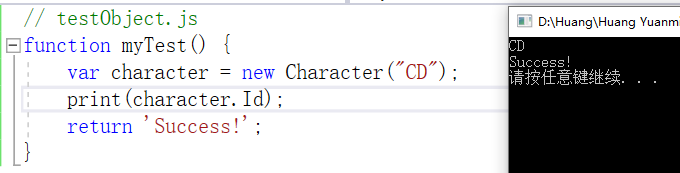
**Section 1**

For section 1, I tried really hard to integrate the scripting module into my project but failed. In the following part of the writeup of this section, I will try to show you the furthest step I have been to and hope that I can at least get some credit for the things I have done.

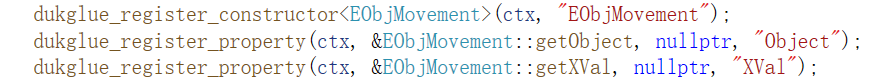
Firstly, I referred to the demo published online by Noah Benveniste on moodle. For test use, I set up a new project “Scripting” to run the code below.

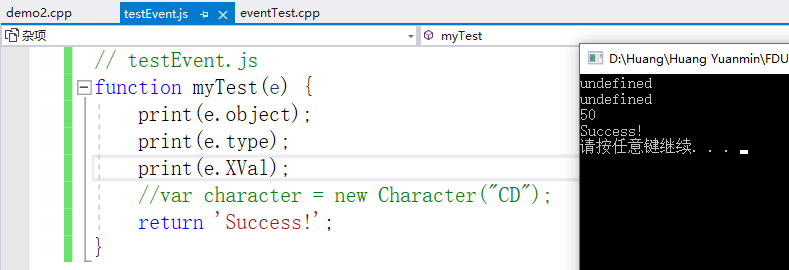
After setting up the environment for duktape and dukglue, I the accomplished effect of the test script as the following screenshot shows. (run demo2.cpp)

Then, I think that to modify game objects, I should try how to gain access to my game object in scripts. Thus, I implemented the following test cpp, which registered GameObject and Character class to the context and specified there inheritance relationship.

At the same time, in the script, I tried to new a character object and referred to its id. The screenshot below showed me the correct effect. (run objectTest.cpp)

Next, for the use of handling events, I wanted to try if dukglue can work with my event objects.

I registered the EObjMovement class to context and pushed an event object of this type to the stack. I want to try if dukglue can return the object pointer (object), double value (XVal) stored in the variant class and enum class value (event type) for me.

After running the script for this test, I achieved the following result. (run eventTest.cpp)

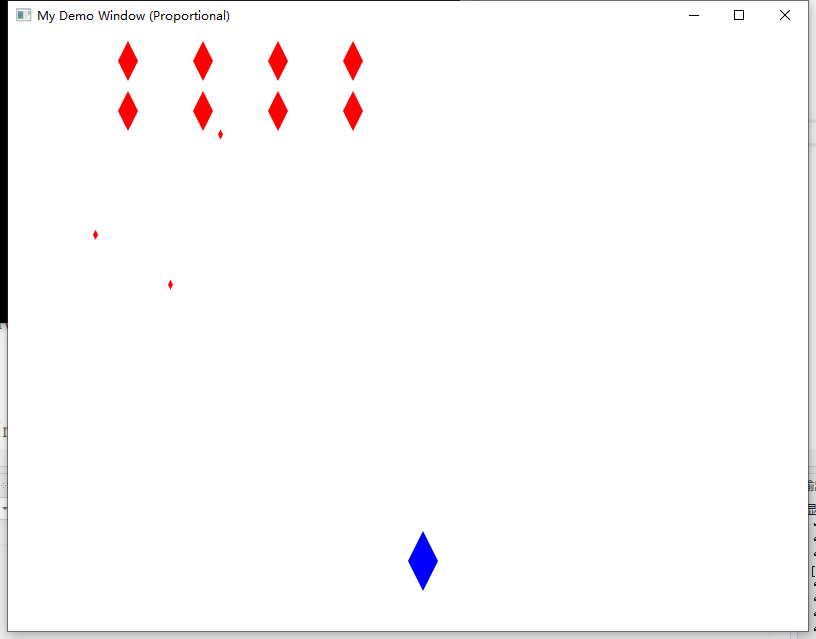
It turned out that I could get the game object pointer and the event type value using dukglue.

I think that this makes it nearly impossible for me to use this to handle the object movement event because the generic component embedded in my game object, which is the part I must have access to in order to handle events, can only be referred to using enum class value. Not to mention the inaccessible game object through the event object.

I think that the main reason for the failure may be the deep call stack when accessing the actual attribute which I need to modify. Therefore, not to waste time on this, I decided to implement the second part first.

To test the functionality I have tested above, you can build and run the Scripting project in the solution. What you need to do is to switch between the three mentioned test files (demo2.cpp, testObject.cpp, testEvent.cpp), which asks you to comment two of them and uncomment the other one before running the project.

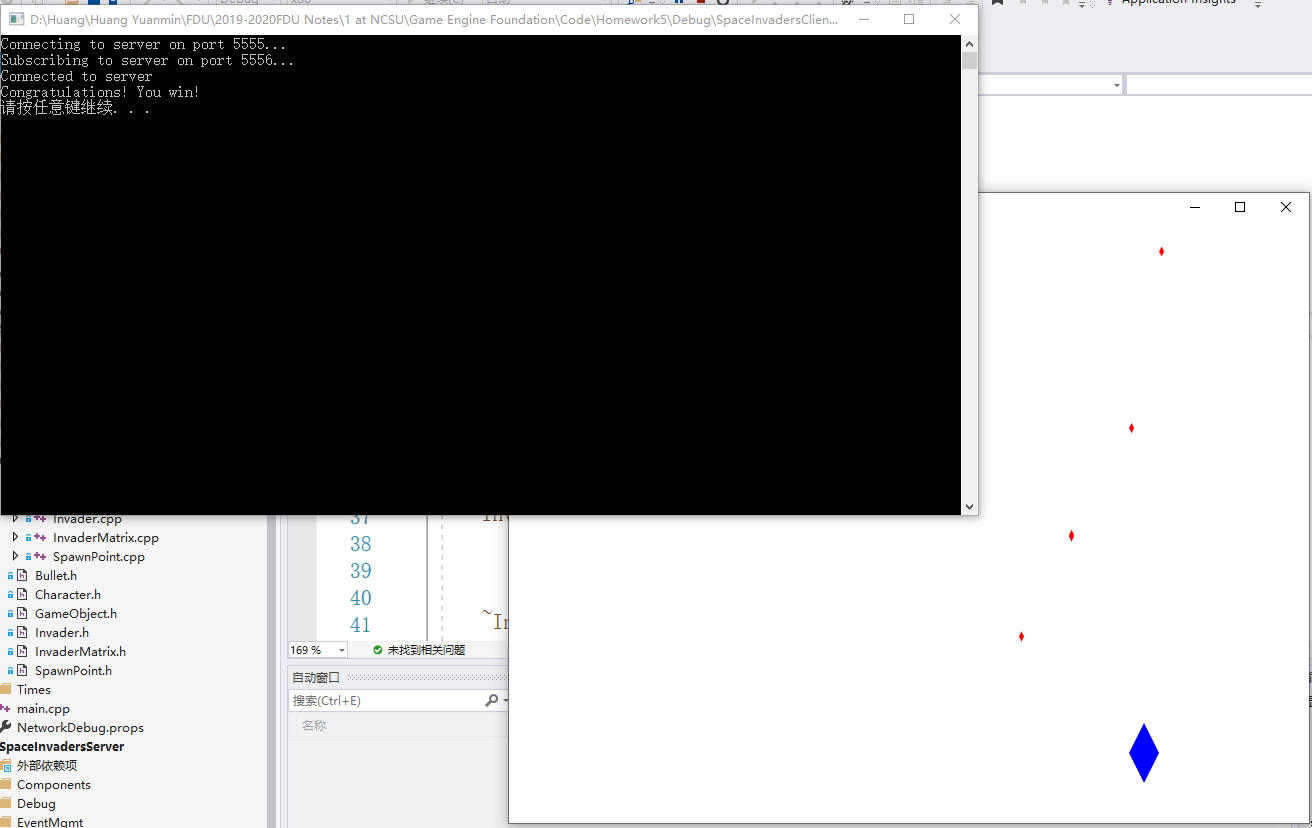
**Section 2**

For section 2, I implemented a space invader using my engine.

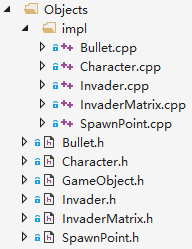
As is shown above, the 2 \* 4 red diamonds are invaders and the blue diamond is the character (defender). The tiny red diamonds flying are the bullets fired by invaders.

The invaders will move from left to right and step down once and then move from right to left, which is the same as the original game. The characters can fire bullets back, but at most two at a time and at most one every one second.

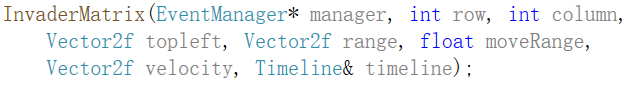
The character will die if it is hit by a bullet, which is also the same for invaders. However, the bullets in my game will not disappear after hitting someone. They will only disappear after they fly out of the window. Also, due to the latency of networking, there maybe some weird death.

Once all the invaders are dead, the character will receive a message of wining in console and the game ends.

After the overview, I will show you how I designed and implemented the game. I will divide them into five main parts.

1. **Objects**

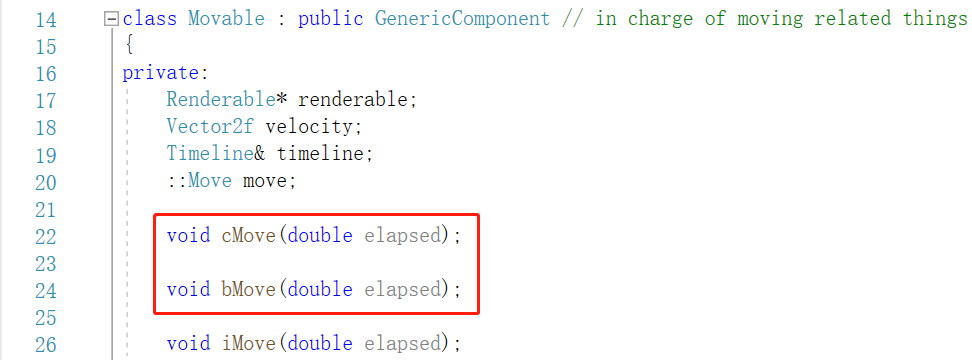
To me, besides the Character and SpawnPoint, Invader and Bullet are the other important objects in this game. I made them be renderable, movable and collidable.

Furthermore, I implemented a InvaderMatrix to manage my invaders. When constructing it, how many rows and columns of invaders are needed can be flexibly chosen. Then, a vector<vector<Invader\*>> would be initialized accordingly.

Also, it takes care of the movement and firing logic of invaders. That is, the whole invader matrix should move together for some distance at a time. And, randomly selected invaders would fire a bullet towards the character.

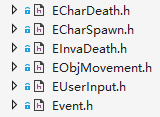
At the same time, the character is restricted to move only horizontally, and can only have two bullets in the window at a time. The gun of the character is limited to fire one bullet every one second. This is implemented by maintaining a list of current bullets and recording the last time of firing.

1. **Components**

To define the movement of invaders and bullets, I implemented cMove and bMove functions in Movable component.

The cMove function moves the cluster of invaders towards the direction computed by the invader matrix, while the bMove function simply moves the bullet towards up or down.

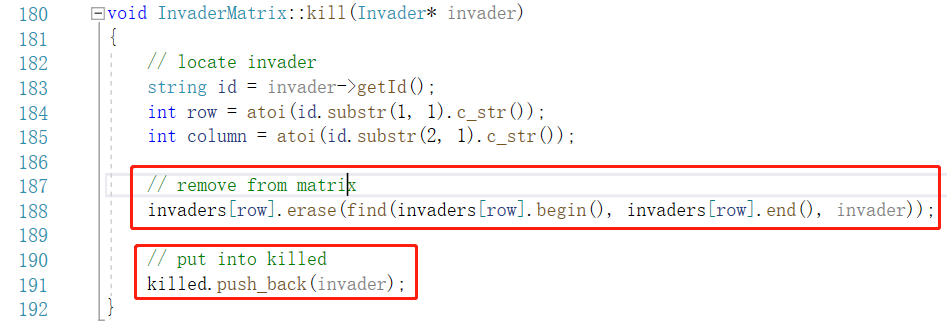
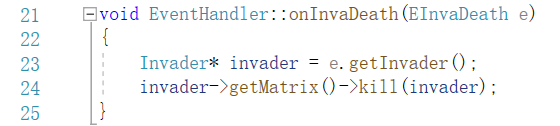
In addition, the work function of Collidable implements the collision detection between invaders and character’s bullets and between character and invaders and their bullets. In either cases, corresponding death events are raised.

1. **Events**

This leads us to the events design. The new event here is the EInvaDeath which represents the death of a invader. The others are basically the same as the 2D platformer game.

1. **Event Handlers**

What has changed are the event handling functions so that they can provide functionalities needed by the space invader game.

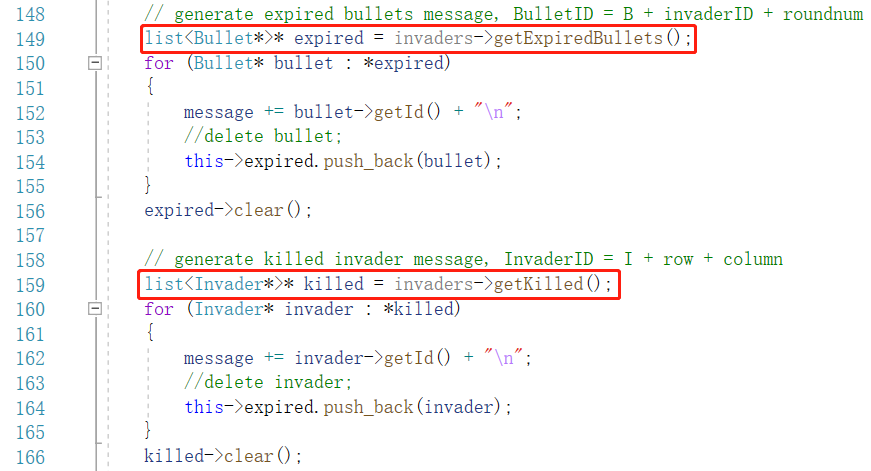
For example, when an invader is killed, the handler calls the kill function of the invader matrix, which will move the invader pointer away from the matrix to an killed invader list. Later, the killed invaders will be deleted and cleared.

1. **Networking**

My design for networking of this game is having all invaders and their bullets on server and character and its bullets on client.

The client would send the movement information of character and its bullets to the server so that the server can create corresponding objects. Then, those objects are used for collision detection after the update of invaders and their bullets. Finally, the movement of invaders, bullets and character (if killed, respawn will generate a character movement) is sent back to the client so that the client can interact accordingly.

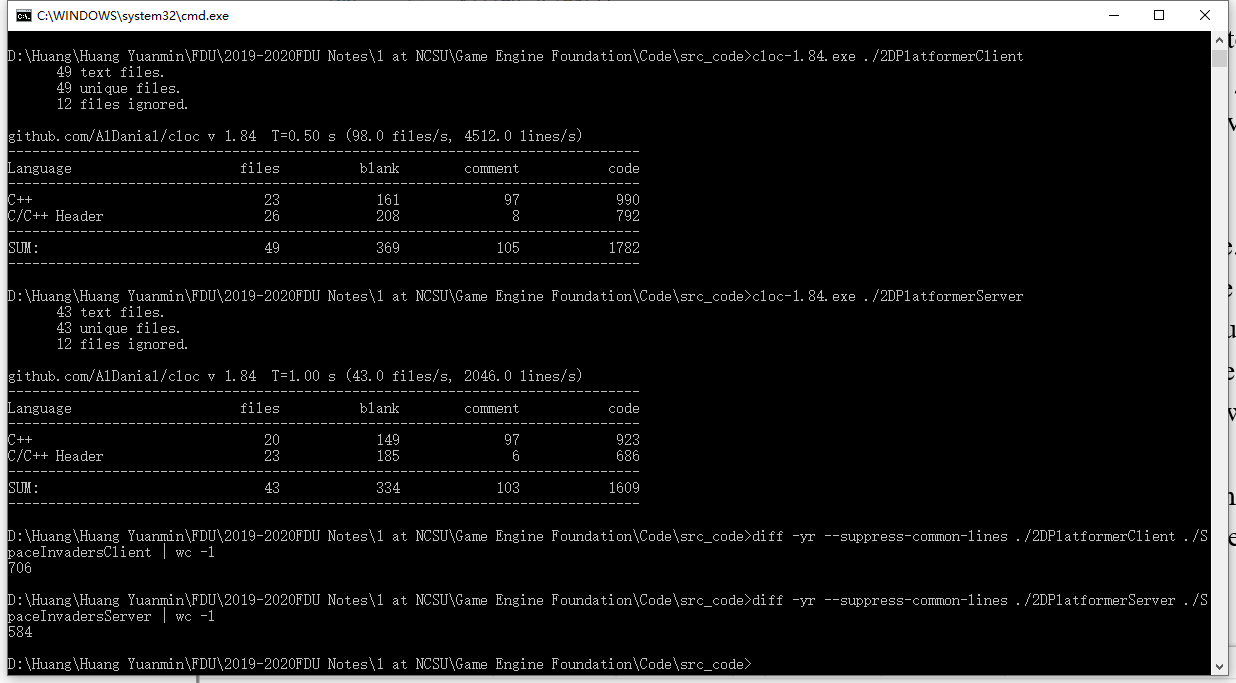
Specifically, when server finds out that invaders are killed, an additional message to indicate this would be sent to client. At the same time, expired bullets of both invaders and character would also cause additional messages to be sent.

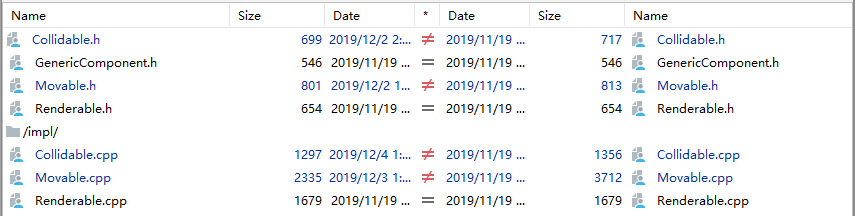
This is implemented by utilizing the expired bullets list and killed invaders list.

Another thing needs to be mentioned in this part is the latency. Maybe due to my architecture, there are some situations in which the latency would affect the game experience. For instance, it seems on the client end that your bullet has hit an invader, but it hasn’t on the server end.

Then, I will show you the difference in code between my new game and the old one.

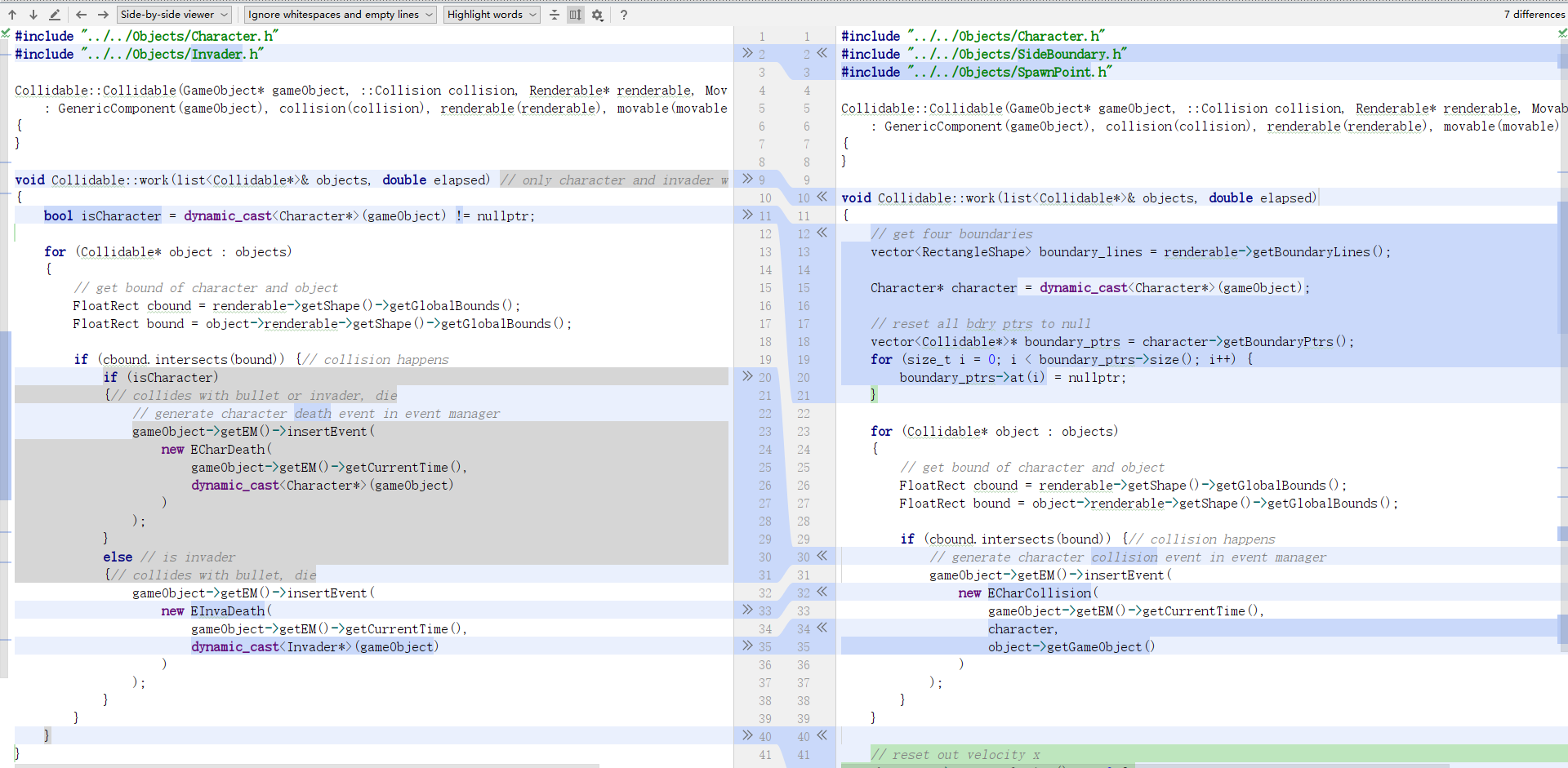
In the screenshots below, the left side would be the code of the new game, and the right side would be the old one. Due to the limitation of the tool I used, I can only observe the number of blocks of different codes between comparable files (files with the same name) instead of line-wise counts of different codes between two directories. I will try my best to explain where the difference lies in my code.

\*Add-on: I used the diff tool after I finished the whole part of writeup and found the percentage of different lines is (706-client-diff + 584-server-diff) / (1782-client + 1609-server) = 38.04%\*

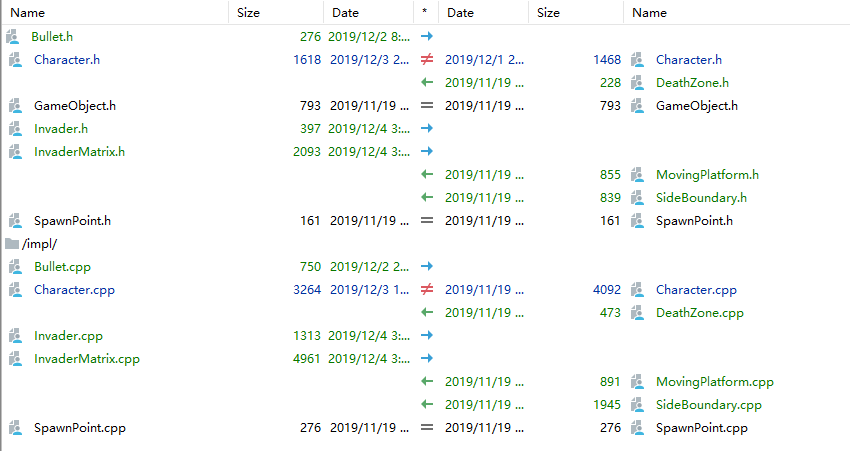
1. **Components**, total difference: 28

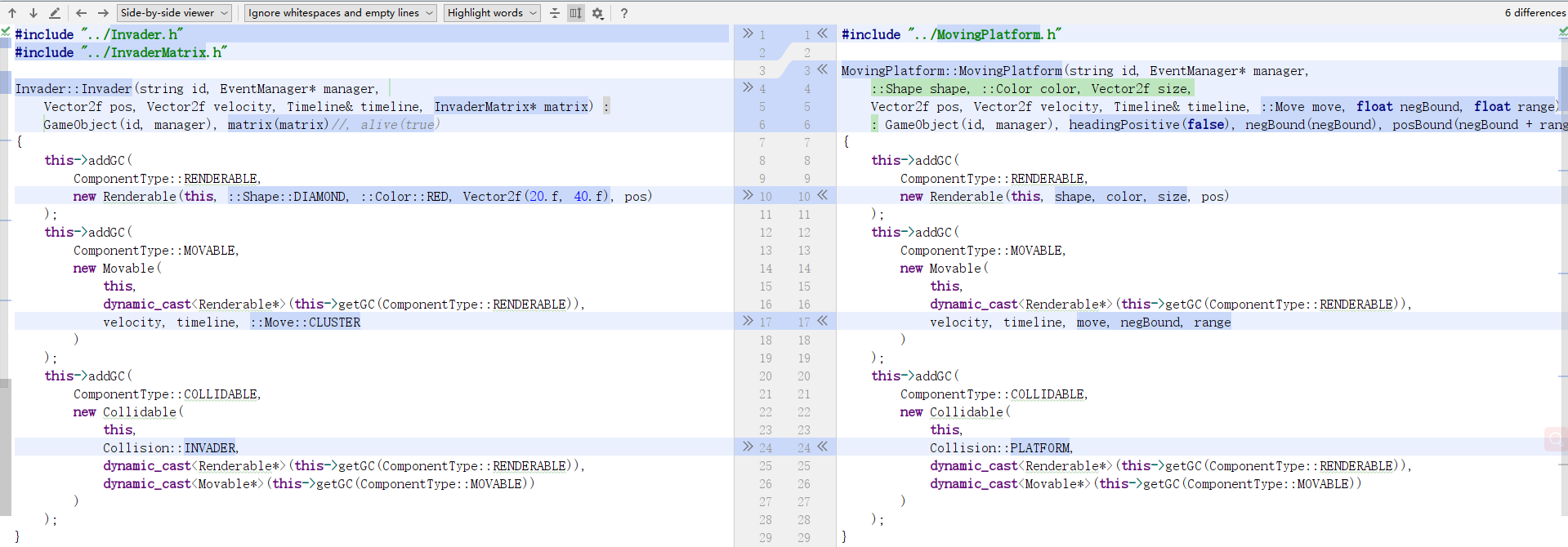
As you can see, the original structure of component model hasn’t been changed. There is no new component, and the GenericComponent abstract class and the Renderable component remain the same.

What’s new here mainly occurs in Movable and Collidable, which is because of the change of behaviors that the game objects would have in the new game. Therefore, I mainly modified the move and collision detection function implementations and kept the original structure.

I will show the comparison between two Collidable.cpp files below for instance.

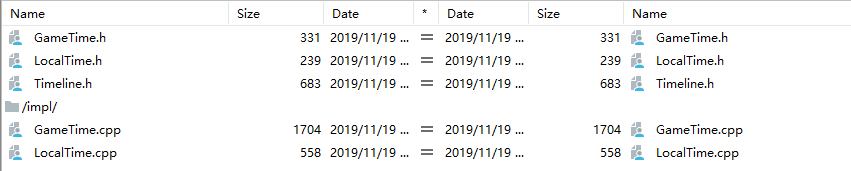
1. **Objects**, total difference: 40

****The change in this part is severe, though the game object structure hasn’t been changed.

As you can see, except GameObject, only SpawnPoint remains unchanged. Character has been changed a lot. The rest of them are totally different files (thus are not counted into the difference).

However, this is because these two games are totally different games which need different functionalities. It’s fair that they have different set of game objects and design of them.

The main purpose for the game engine is that the mode we design our objects should be reusable, which is preserved in my design. As you can see above, different objects in both games still have similar ways of construction. (I’m showing the difference between Invader and MovingPlatform).

1. **Times**, total difference: 0

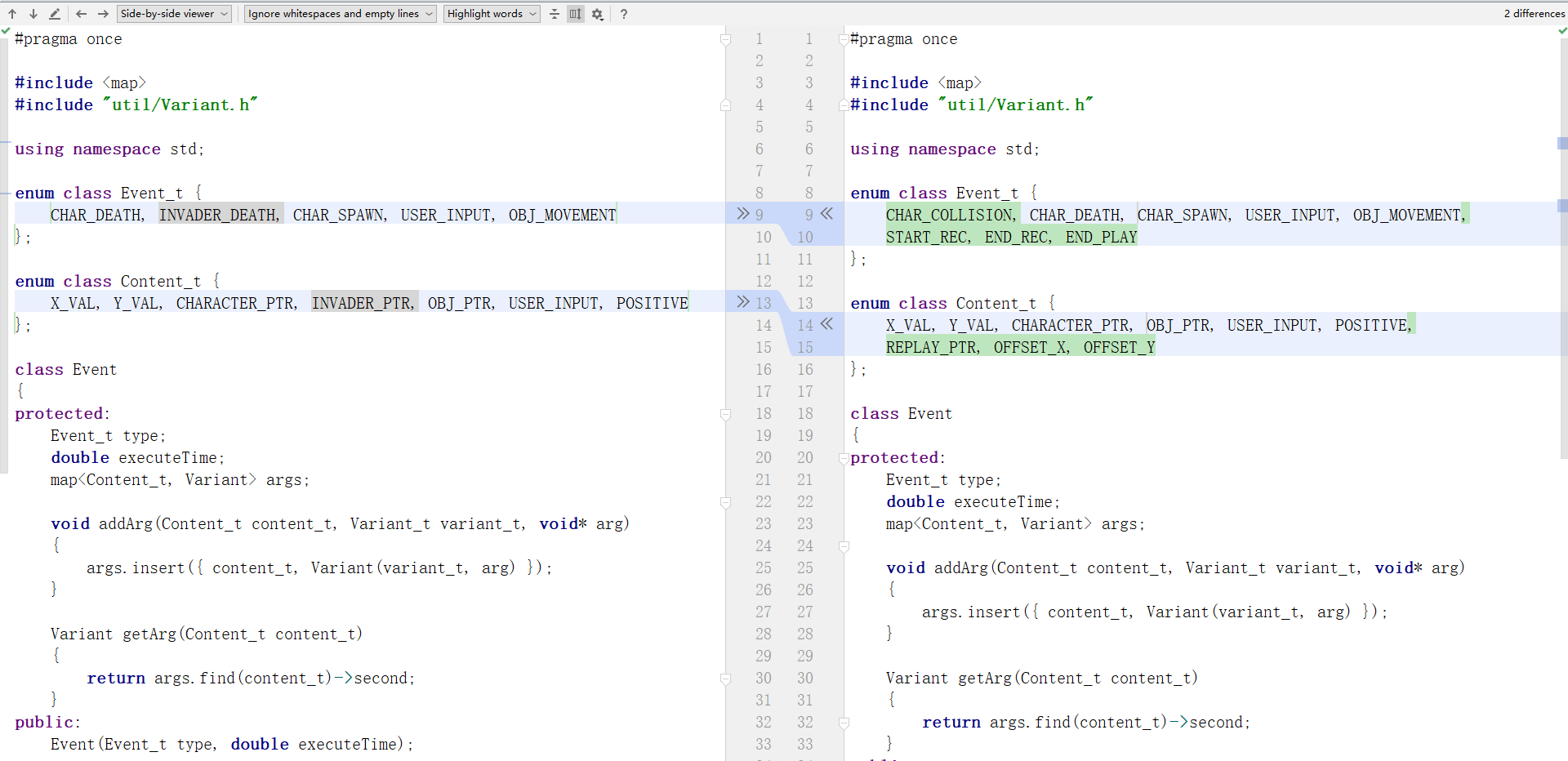
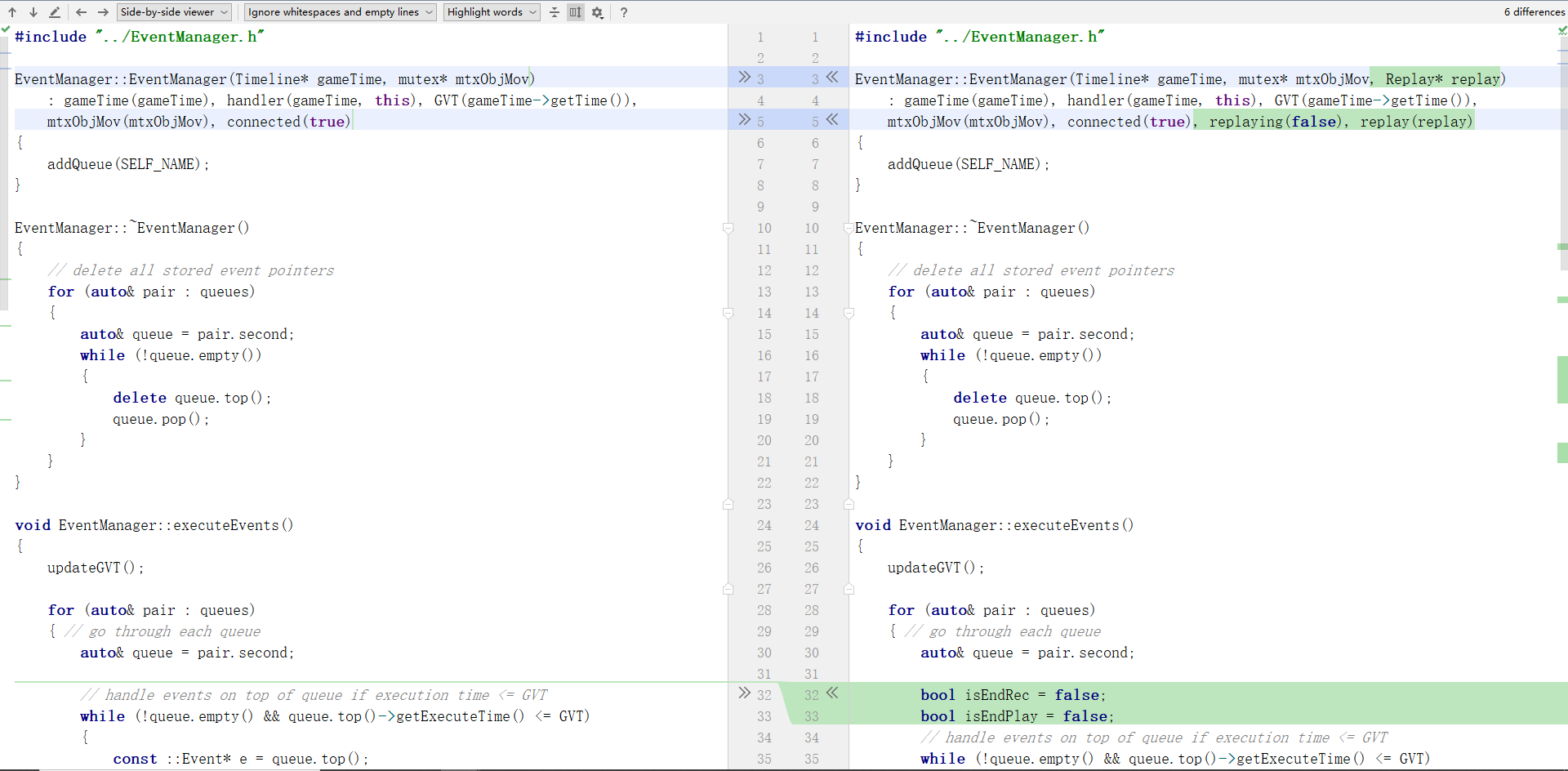
The time module between two games are completely identical. The time part is portable enough for my engine to utilize in different games.

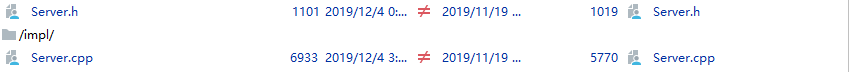
1. **Event Management**, total difference: 40

Like objects, the event types and handling have been changed a lot during the shifting between two games. In addition, I removed the replay system and its related files, which may also cause some difference.

Whereas, the main mechanism of event representation, raising, queuing and handling remain unchanged. What has been changed are the event types enumerations, new events and there handling functions.

The Event class is shown below to illustrate that only the enumerations are different between the new one and the old one.

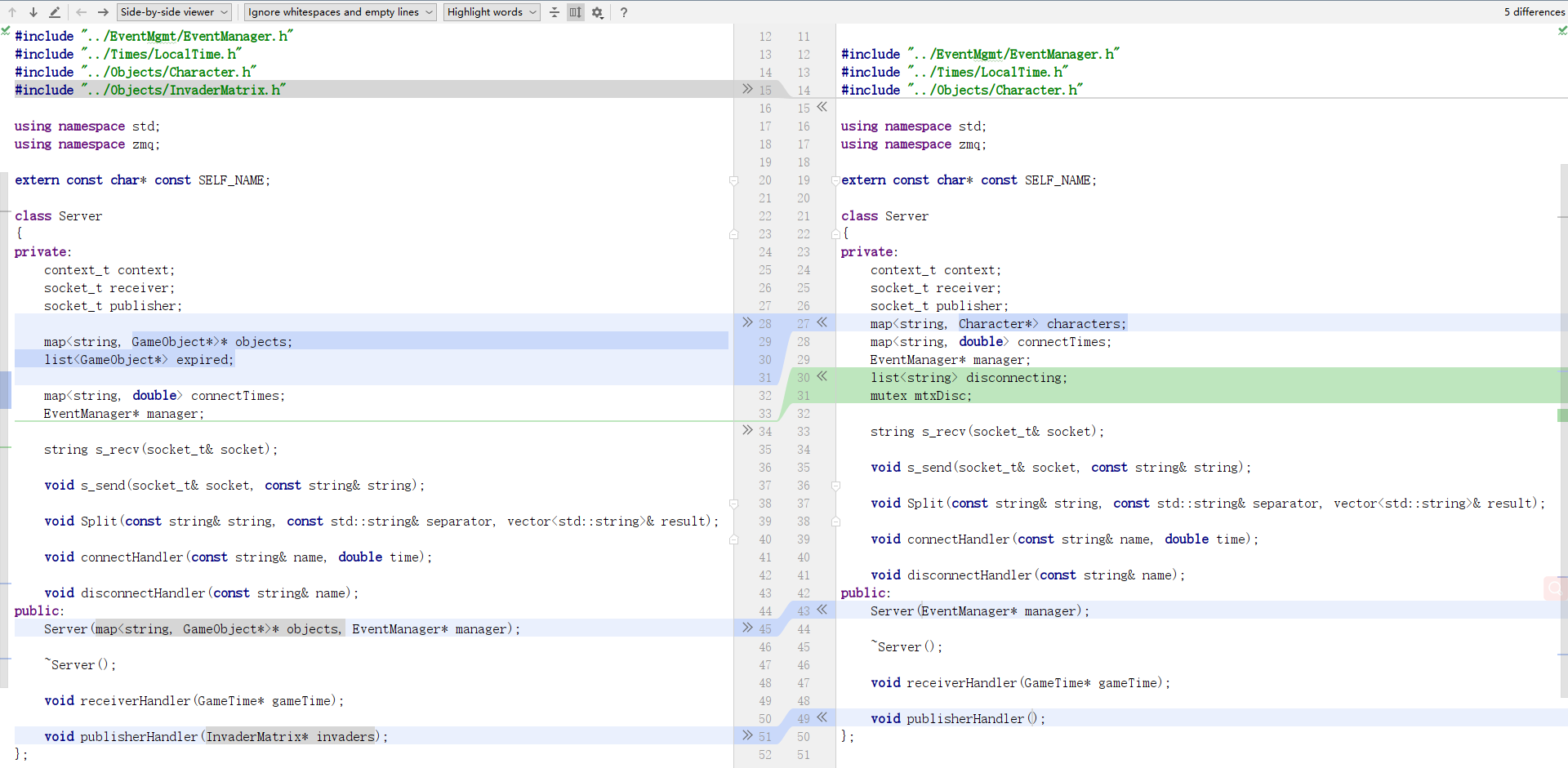
The two event manager implementations are also identical except for the removed replay related design.

1. **Networking**, total difference: 30 (client) + 25 (server)

To be honest, the modification of this part is the most painful part among all. The connection code that can remain unchanged are really few. Because of the difference between two games, the content of message that are teleported between client and server must be redesigned and thus the way of processing and interpreting messages.

The only good news is that the design for most interfaces remain unchanged or only has been changed a little, while the implementations beneath have huge differences.

The picture below is the difference between two Server header files. As you can see, most function signatures are similar to each other.

1. **Main**, total difference: 34 (client) + 12 (server)

Again, different games, different main loops. That’s fair.

We are surly going to have different object initializations and game-related implementation in the main loop.

Despite those differences, the order of everything happens in the main loop should remain the same. For example, we shall answer key input, move objects, do collision detections and render the objects on the window. Beyond that, not much code is reusable.

1. **Conclusion**

All in all, the most design part of the game engine of the two games are identical to each other. But some additions and deletions in code still have to occur because of the difference between the games we want to implement.

I believe that the design of my engine isn’t too bad, so that I can utilize the generic design pattern I defined to create new classes to provide new functionalities, like components, game objects and events. In my opinion, this is the key point of reusing a game engine.

Also, I think that there will be better practice of the design of the engine such that we can achieve new functionalities without too much painful work in modifying the code. Instead, we may only need to modify some configuration files, which will be the reward of a more flexible and better-designed game engine.