Zhongxiao Mei

1. Runtime game object model

The runtime game object model that I choose to use over the course of the semester is the Object-Centric Model. In the design of my engine system, the GameObject is composed of different components, such as position, size, and color components. Each component is an independent object class and each component class contains its own attributes. For example, sizeComponent contains length and width values. PositionComponent contains x-coordinate and y-coordinate values. ColorComponent contains R, G and B values. StaticPlatforms and movingPlatforms are instances of the GameObject. Characters are instances of movingPlatforms. Spawn points, dead zones, and side boundaries are instances of staticPlatforms but they are not drawn or rendered to the screen.

I use the same approach to create GameObjects both in my first 2D platform game and my second game "Space Invaders". For example, firstly, I create a size component, a position component and a color component using "sizeComponents size(10.f, 10.f); positionComponents position(200.f, 300.f); colorComponents color(255, 0 ,0);". Secondly, I create a movable object using "GameObject movableObject;". And then, I set these size, position and color to my movable object using "movableObject.setSizeComponent(size); movableObject.setPositionComponent(position); movableObject.setColorComponent(color);". In addition, these components (size, position, color) can be prepared at the beginning and can be used to create different game objects.

2. Event management system

When doing my second game "Space Invaders", I reused the event management system that I used in my first 2D platform game. My event management system consists of the following classes: Event, collisionEvent, deathEvent, spawnEvent, userInputEvent, scriptEvent, eventManager, eventHandler, charEventHandler, platfEventHandler. These collisionEvent, deathEvent, spawnEvent, and userInputEvent, scriptEvent are subclasses of Event and represent different types of event objects. The eventHandler is the interface of the charEventHandler and the platfEventHandler. It dispatches raised events. The charEventHandler handles all characters' events, including collision, death, spawn, and userInput. The platfEventHandler handles all platforms' events, such as collisions. The eventManager keeps track of which game objects are registered to receive events of arbitrary types. For example, if a collision is detected (registration phase), a new collision event object will be created and added to the event queue (raising phase). Then call eventHandler.onEvent(Event*) for each event that has been raised (handling phase).

In addition, the event management system has an enum type called "eventType" to represent collisionEvent_type, deathEvent_type, spawnEvent_type, userInputEvent_type and scriptEvent_type. The constructor of each event object passes two GameObjects as parameters that representing the event is occurring between these two GameObjects. For example, using "Event* char_coli_platform = new collisionEvent(character, platform);" to create and raise an event that represents the collision between the character and the platform. Each event object has a function called "getType()" which returns its eventType (collisionEvent_type, deathEvent_type, spawnEvent_type, userInputEvent_type, or scriptEvent_type). For example, to know the particular eventType, using “char_coli_platform.getType()” and the return result will be an eventType: collisionEvent_type.

After an event is raised, a function called “onEvent()” in the eventHandler will be called to handle this event. The “onEvent()” function passes an Event as the parameter. For example, when a new event is raised “Event* char_coli_platform = new collisionEvent(character, platform)”, I use “eventHandler* eh_character = new charEventHandler()” to create an event handler for the character. And then, use “eh_character->onEvent(char_coli_platform)” to handle the raised collision event that happened between the character and the platform. When the “onEvent(Event* e)” function is called, I use “switch (e->getType());” to check the type of the raised event. For example, if the Event* e is case: collisionEvent_type, the event handler will handle the collision event. If it is case: deathEvent_type, the event handler will handle the death event. If it is case: userInputEvent_type, the event handler will handle the user input event.

3. Client-server architecture

I also used the same Client-server architecture in my first 2D platform game and my second game “Space Invaders”. The server should be opened first and listen to clients, and then open the client to connect with the server. The server handles user inputs and movement information sent from the client and re-sent to each client and draw on their windows. I use the REQ-REP socket pair and PUB-SUB socket. I use ZMQ_REQ and ZMQ_REP to set up two-way communication between the client and server. To make a connection, the client will send a request to the server using socket_send(client) function. And then the server will receive the request from the client using socket_recv(server) function and send back a reply to the client. If the client receives the reply from the server, the connection is successful. And then using socket_sendmore() and socket_send() function from ZMQ_PUB and ZMQ_SUB to broadcast from server(publisher) to each client(substriber). To handle keyboard inputs that control the movement of a unique object and display the movements on each client, each client send the keypressed message and the character’s position, size and color information to the server. And then the server broadcasts these messages back to each client. Each client will receive keypressed information and characters or moving platforms’ position (x-coordinate and y-coordinate), size (length and width) and color (R,G,B) information to display objects.

4. Multithreaded

I reused the code from my first 2D platform game’s server to implement a multithreaded server for the second game “Space Invaders”. My approach is to create a thread using a function pointer called “void server_thread(string c)”. This function is used to handle key-input information and characters and moving platforms’ position, size, and color information received from the client. Also, there is a list of threads in main(), such as “thread t [numberOfThreads]”. The number of threads is counted when a new client is connected to the server. Using the for-loop to create each thread using “thread(server_thread, c)” and call join() function for each thread. Due to the multi-thread has limitations since I did in my first game, the multithreaded server functionality in my second game “Space Invaders” may not work as expected. The game will work well if just opening one client.

5. Conclusion

I think my game engine is successful at designing for reuse. I reused most of the codes from my first 2D platform game to implement my second game “Space Invaders”, such as runtime game object model, event management system, multi-thread, and client-server architecture. If I am going to start over again to design my game engine again, I will change my client-server architecture to better utilize my runtime object model on both the client and the server. I will keep doing the Object-Centric Model as the runtime game object model and using a similar approach to implement the event management system because the Object-Centric Model is easy to understand and implement.