Part 1:

- In this homework, the most difficult part of the assignment was simply trying to understand how I wanted to structure events and to comprehend what exactly eventManager does. I reread the slides several times to grasp a better understanding of what I should use handlers for and how they are related to events. Another difficult part of this assignment was that I was facing severe burnout from previous exams and projects, making completing tasks I had not fully understood more difficult.
- The first part I knew that I needed was an Event manager class with the recommended methods, raise(), registerHandler(), and deregisterHandler(). I also realized, as a manager, that this eventManager() would also be a singleton, similar to my gameManager() class, so I also added a getInstance() function. I wanted to use my EventManager() class inside my client but I realized my GameManager class was really going to be raising all the events when it calls my checkInputs() method, so it might be best to create an instance of my eventManager() inside my gameManager() and just raise events there. I also realized my checkInput() function was handling way more than it should be, including my collisions, death, and respawn. Since there needed to be events for collisions, death, spawn, and user input, I created separate functions for all of them and would just raise the respective events in each function.
- Next, I needed to create my Event.cpp what I wanted to attribute to an event. At first, I wanted to use enums to represent my event types since I had not used them since CSC 216 and this seemed like a good excuse to reimplement them. I also created a priority of int64_t type which would be the time of which the event should execute. The priority queue that stores and sorts all of my events would sort by this priority time with the lowest value being at the top. This was somewhat hard to implement since I did not understand I needed to make my own comparison function and had to relook at some old CSC 316 code to get an idea for implementation. I also created a string for event info for extra information I wanted to store in the string. I did not necessarily have a specific purpose for it yet (though it did come in handy later). Lastly, I needed to have a reference to the Actor() that triggered the event. In this homework, I decided to only have events triggered by other players but able to be scaled to other objects.
- Now that I had my standard Event class, I thought to implement my collision events first, see how it works and perfect it, and then finally add the other forms of events. This is where I finally saw the use of handlers in my EventManager(). For the handlers, I created an interface that just had an OnEvent() function like the slides. My CollisionHandler() would then implement this interface with its own implementation of OnEvent(). This was the first time I have used an interface in C++, so I'm glad I got to do that. The OnEvent() function in my handler would

essentially do whatever is in the current contents of my CheckCollision() function in my GameManager() but now my GameManager() would only check if a collision happened and raise an event if true. I also decided to not use the register handler and deregister handler functions, but rather just create an instance of these handlers and permanently use those in my events since it seemed more time-consuming to constantly instantiate new handlers.

- When I finished implementing my CollisionHandler it did not work the first time I tested it. I was able to move but it was not computing at collisions properly. I messed with this for about 20 minutes before realizing that I compute my collisions and process those events after I check my inputs and perform other calculations. So I thought it may be best to implement other forms of events since I was now doing my calculations out of order.
- I then moved to doing user-input and created a handler very similar to my collision handler, except this time I would need some way to represent which inputs were pressed and how to send that as an event. I decided it would be smart to have an array of boolean values that I would set to true if that input was pressed. That boolean array would be sent as a field to my new event constructor I made specifically for user inputs. My user input handler would then interpret this array and make movement to that specific character based on that array. After implementing my user input events, both my input and collision events worked as intended.
- When creating my handlers for the spawn and death, I thought these would be the easier of the four events, but I was proven wrong as I needed to reset the side scrolling object bounds of my player when my character died so the bounds would encapsulate the character. This was a function call in my GameManager() so I thought maybe I can just get an instance of my GameManager in my spawn handler. This would not compile since both the GameManager and the EventHandler both included each other. I was finally able to resolve this issue when I created a new field of my actor witch was a boolean stating whether it was alive or not. This could then tell the main client loop that a death had taken place and to reset the bounds from there. Inside the spawn handler I would simply reset the player to its original spawn point several seconds after the death had taken place. I added a delay to show that the priority queue works in how it sorts since I will always spawn after I die and not before since both events are raised at the same time.

Part 2:

- Part two of this assignment was fairly simple since would I was allowed to send to the client to get the instance of the event was loosely restrictive. I also used a Client-Centric design to handle my events since most of my events would be triggered by a specific player. All my events will be queued on the client side since all actions from these events take place on the client. If I start doing environmental events, that may be better handled on the server.To show an example of one of my events being sent to a server I can send a string of information to the server that informs it that this specific client has died. From here I am able to reconstruct a death event if I like based on the given client ID at a specific time. When the client dies, the server will output which specific client has died to show that these events are able to be understood between both the server and the client.