CSC 481 Homework 4 Write-Up

Sarah Willis

10/29/20

For this project, I was tasked with creating programs in C++ using Microsoft Windows 10, Visual Studio 2019, SFML 2.5.1 and ZeroMQ release 4.3.2 that satisfied the requirements of three different sections (the last of these being optional). I did not complete the third section. The first section involved adding an event management system to my game engine utilizing a priority queue. Due to my need to add a replay system later on as well as my need to sort out lingering issues in miscommunication and synchronization between my client and server, I decided to go with a server-centric design for my event management. With this design, my server handles all of the game logic and event management, with the client only handling rendering, updating the positions of its local objects based on updates from the server, and sending updates on user input to the server to allow for the raising of user input events. This design works well because having all of the game logic and event management in one place makes it far easier to synchronize the various activities and debug any issues. It also provided a nice foundation for the replay system by allowing replays to focus on simply resending positional updates to clients through the event management system.

After deciding on a server-centric design, I considered whether to integrate the event handling into my game object model. As my game object model is property-centric, I thought it might be possible to have the properties themselves serve as the event handlers. After some consideration, I decided that this would not be a good idea as it would limit the reuse of the properties for various types of objects due to the specificity of the handling. Having the Collision property be the handler for collisions, for example, would make it difficult to generate collision events specific to different types of objects, such as a character collision event versus a moving platform collision event. Thus, in the end I decided that the event handlers would be separate objects from the properties.

For the events themselves, I decided to generally follow the recommended model, creating a generic Event class with a timestamp, a String type, and a list of variant arguments. Having the type as a String allows for greater extensibility than more primitive types would allow while also avoiding the complexity that having different event types as different objects would bring. The list of variant arguments also allows for greater extensibility by allowing each event object to have a variable number of arguments with variable types. The variant currently only allows for int and float arguments as those are the only ones I found myself in need of for my implementation, but other types could be supported in the future if necessary.

Having created an Event class and decided on having my event handlers be separate from my properties, I decided to generally go for the recommended model once again in creating an abstract EventHandler class that can be inherited from in concrete subclasses to form specific handlers. In order to allow for communication with the game object model, the EventHandler class contains a pointer to the map of all properties, allowing its subclasses to interact with properties whose ids can either be stored in the handlers themselves or obtained through event arguments. This design allows for the event manager to store event handlers without needing to know their specific type, while the handlers themselves can have a specific type so that specific behavior can be defined for particular events. New handlers can easily be created in the future by extending the EventHandler class, allowing for extensibility.

For managing the events, I created a singleton EventManager class that contains a map pairing event types with lists of registered handlers, a priority queue that prioritizes events based on their timestamp (earlier times having higher priority), and a timeline used to time events and determine which events need to be handled. Having the class be singleton makes sense as there is no need for multiple

managers and it allows for the event handlers to be able to reference it for event raising while still being contained by it. The mapping of event types to their registered handlers makes it easy to go through the list of registered handlers for each event being handled. Having the priority queue be based on event timestamps (which indicate when they should be handled) allows for event ordering to be coordinated effectively, and having the manager itself contain a timeline makes it easy for other classes to reference its current time when raising their events.

After creating my different classes surrounding event management, I set about refactoring my game to utilize the event management. The first step was to refactor my client-server setup to support the server-centric model, shifting the properties surrounding characters to the server and restricting the client to only properties related to positions and rendering. I also set it up so that a connecting client would receive the information for all objects and their related properties it needed to create, allowing me to avoid defining objects and properties twice across client and server. As I wasn't able to get my game fully functioning in the last homework, I ended up mostly starting fresh with my implementation of the game logic using the event system, using my previous implementation to help guide me while creating new event types and corresponding handlers to satisfy the various necessary functions. I also added multithreading to the server so that each client's updates to the server on user input could be performed in a separate thread. Due to the need for separate character collision, death, and spawn events, I created a different handler for each of those. I also created handlers for user input, platforms moving around characters, and positional updates to the client. Overall, my event management system is working well, though there are a few improvements that could be made, such as eliminating some redundant event raising to help performance and improving the way that the jump is handled (as it currently doesn't work as well as before, as there is less time for the user to give input and move left or right).

For the second section, I was tasked with using my event management system to implement a replay system. As I had already designed my system with replays in mind by ensuring that all the client handled was positional updates from the server, adding in the replays was fairly simple. I created a ReplayHandler event handler class to use in handling the recording of replays, as well as the changing of speeds during replays. I then made a few additions to the EventManager class to support replays, such as adding in a separate timeline and queue specifically for replays as well as a way to unregister from events (allowing the ReplayHandler to unregister from events when not recording) and a way to set that the manager is in the state of replaying. Once a client selects to record a replay, the ReplayHandler registers for all of the events it is set to record, adding each event to a list of recorded events. Once the client selects to stop the recording, the ReplayHandler is notified so that it can unregister from events it records, register for user input events (to allow handling changes in speed), set that the manager is replaying, and raise all of the recorded events so that they can be stored in the manager's replay-specific event queue. I also added a Boolean value to the EventHandler indicating whether a handler should be notified of events when a replay is occurring.

While a replay is playing, the EventManager will handle events in its replay queue as needed, only sending the event to handlers who are set to be notified while replaying. This allows for only the ReplayHandler itself as well as the PositionalUpdateHandler to be notified while a replay is occurring. The ReplayHandler thus only has to register for events involving a position update, and each of these events is then sent to the PositionalUpdateHandler with the proper order/timing when a replay is playing. The replay thus involves the client being resent the same positional updates as before, with the final update bringing the objects back to their state before the replay started. As none of the objects on the server are touched during the replay, there is no need to save any of their state, and the game logic should not be tampered with by the replays.

Though implementing the replays was fairly simple, I did have some issues with the timing and the changing speeds. To resolve these issues, I had to make sure that the timestamps of the events to be replayed were not relative to a particular speed and that they were related more to the time they were

actually handled than the time they were simply set to be handled (as some events are set with lower times to ensure moving them up ahead of others in the ordering, and some events may be handled a bit late). I also had to make sure that checking whether events were to be handled took into consideration the current speed when comparing times in order to allow for the speed being changed in the midst of a replay. There are still a few remaining hiccups in the system, such as a slight delay in the replay sometimes when changing the speed, but it works fairly well overall at allowing for the various positional updates to be captured and replayed. Also, due to the server-centric design, only one replay can be recorded/played at a time, and the replay will be reflected across all clients (preventing all clients from moving during this time).

To conclude, my design for the event management system and the replay system focuses on centralizing the game logic on the server, simplifying the addition of new logic as well as allowing game changes to be reflected in replays. The use of generic Event objects, specific EventHandler implementations, and a singleton EventManager allows for reuse and extensibility by making it easy to create new event types, implement new event handlers with unique game logic, and manage a variety of event handlers and event types. The focus of replays on merely resending positional updates to the client makes it easy to add to or change the game logic without having to make drastic changes to the replay system or to the event system itself.