Jayden Sansom, jksanso2

Homework 4: Event Management

Introduction

This assignment allowed for an event management system to be added to the engine using the SFML and ZeroMQ libraries. The following sections explore the design decisions made in each of the assignment parts.

Event Management System

The design of the event management system relies on event representation, registration, raising, and handling. By utilizing how these factors are implemented and integrated into the existing system, I can ensure that the events will not show a visible difference from past implementations.

The event representation is mainly handled within the Event class, where there are the enumeration classes EventType, ParamType, and VariantType, describing the four types of events as well as the types of parameters they can contain. The Variant class is used to represent the parameter types, accepting any and placing it within a parameter of the Variant object. The Event class will add these Variants to it and can be inherited by the individual Event type classes.

The *EventInput* class will contain the parameters of a player object pointer, KeysPressed struct pointer, and a float representing elapsed time, all required to run the update function of the player. The *EventCollision* will require a player object pointer and a game object pointer, as the event's purpose is to resolve the collision and that requires both objects. The *EventDeath* class was specially designed to contain all of the required Variant parameters of *EventSpawn* as well since the event of death will lead to the creation of an *EventSpawn* Event. In addition to these parameters, a pointer to a vector list of SpawnPoints is required to randomly select a place for the player to spawn and those coordinates are sent as the X and Y positions of the *SpawnEvent*. The SpawnEvent needs more parameters than the others due to the implementation of side-scrolling areas. When the player dies and respawns, the window, camera, and side scroll areas need to reset to center on the player in its new position. By obtaining these parameters, everything can be reset and the player can be moved to the SpawnPoint generated in the *EventDeathHandler* OnEvent function.

Event Handling is the next step and the general *EventHandler* class was created for the individual handlers to build upon, requiring a reference to the *EventManager* and a pointer to the event that will be run, as well as the virtual class onEvent() that will run the function of each event. The *EventInputHandler* utilizes the input to run the update function of the given player, the *EventCollisionHandler* runs the resolveCollision function of the given player, *EventDeathHandler* obtains a random spawn point from the given list and initializes an *EventSpawnHandler* that will set the position of the player to that random spawn point and reset the camera and side-scrolling areas. The handlers require a circular dependency from the *EventManager* but it is needed due to the need of adding an event to the queue from

within an Event handler. The issues from this are minimized via forward declarations and ensuring that the program remains safe when and how the *EventManager* is obtained and used.

Event registration is done differently from the pseudo-code provided in class in order to fit my implementation. The *EventManager* contains a queue of EventHandler pointers and when an event is registered, it is added to the queue. Due to the low amount and usual sequence of events, a priority queue was not implemented but could easily be created if necessary by adding a parameter when creating an event handler and using it as the priority when registering Event raising is done by locking a mutex, going through the queue of the manager, and running the OnEvent function of all event handlers. This raise function is called multiple times in the game loop to ensure that events are played at opportune moments but it could easily be added to run in a separate thread instead to constantly raise events if there exists one in the queue.

Ultimately, with these implementations, the events run seamlessly and create a window that delivers the same result as past implementations, only with the event manager running beneath it. For the next portion of the assignment, a new event and event handler will be added and sent across the network, proving to have a simple implementation due to the work done on the extensibility of the event system.

Networked Event Management

When considering what network event management system to implement, I prioritized what would cause the least disruption to the existing system, instead of adding new features rather than refactoring what was previously done. Therefore, for the purposes of my engine, I chose a design similar to the Distributed management system, where I have each of my clients running their own event management. At the same time, the server also handles its own. To best showcase this, I opted to add a new type of event, *EventClientDisconnect* that would be registered once a client sent to the server that they had disconnected. The event would take in the name of that client and send a message to the other clients in the subscriber thread that it had occurred. Then, in each client, the event would be received, registered with their own event managers, and then raised to remove the client from the scene.

Implementation of the event itself as well as its handler followed the same implementation of previous events, attempting to build off of the last part rather than reformat all others. In addition to the name, the event would also request a pointer to the list of PlayerClients. This list, while not used on the server side of the event, allows the client to obtain the list from which they are to delete the client with the given name from. This list was not passed from server to client and instead was obtained in each respective call to register the event. This was due to the possibility that the list would contain large amounts of data, taking a long time to parse the string data and fully obtain the client list. Therefore, since the client and server both had a list of clients, identical to each other but having the possibility of being out of order, it was determined that this way of passing this certain information would be best.

There is much that could be done to continue to improve the server-client event management system but for the purpose of displaying an example event being passed over the network to a client. In future iterations, this could be improved by changing the way that the server sends information to the clients, as strings have their limitations over .json files in this scenario. Ultimately, however, the connection is demonstrated and there is room for extensibility in event creation and handling across client connections.