Matthew Warren

CSC 481 HW4 Writeup

When deciding how to design my event manager, I started with what events I thought would be beneficial for my system. Starting with the 4 required ones, I knew I would be doing spawn, death, collision, and input. I knew that I would want more than this, as only doing these 4 would split my engine, as there is a lot more data passed around than just these. I came up with these other ones: Adding a new player, moving a player, Player disconnects, and gravity. This covered just about everything that I originally had sending between my client and server. One major consideration I thought about was if I should make an event about moving platforms changing position. This is something that would be happening every server loop iteration, and I felt like it would clog up the event manager with events. I decided to keep that in its own thread, just sending positions without events to improve performance. Gravity was something I also had this concern with, but slightly altered my implementation to only create a gravity event if the player is actually in the air. That way constant gravity events are not being made if the player is resting on the ground. With this many events, and each one needed to know about different things, I figured it would be best to create child event classes inheriting from a base event class. This way I can take advantage of polymorphism in the event manager, and can easily design events that only know about what I need. The base class has a timestamp and a priority (an enum that is either High, Medium, or Low).

The next thing I considered was where to do the event processing. I first tried a server centric design, where the client only makes events, sending them off to the server where everything will be processed. The more I added to the server, the slower it got sending events, and made the clients laggy. There were probably ways I could have optimized it to make it faster, but I figured the easier solution would be to divide up the work between the clients and server. This has the added benefit of being able to process player movement on the client, making inputs always feel responsive. I went through a few iterations of this, but settled on a design where each client is solely responsible for its own player. I ran into issues where the client and server would have their object positions out of sync, especially when lateral side scrolling got added. As discussed in class, events can be a lot harder to debug, and I felt a simplification was needed. So I made the server responsible for all the world objects, constantly broadcasting their positions. Now all the clients are guaranteed to have the same world.

The event handlers were relatively simple to implement, but the hard part was deciding what handlers to make and what events they should respond to. I decided to make my handlers around broad parts of the game. So in the client, I have a handler for that client's player, and one for events in the world. The player handler was responsible for input, gravity, collision, and spawning events. These all needed the player object to be moved, so it made sense to group them together. This was the main chunk of processing that needed to happen in the client. The world handler handled adding and removing other players, death, and translate. I originally had death with the player handler, but I have a player death completely reset the level, so I wanted it in a broader scope than just the player. All of these events in the world handler are ones that were sent from the server (or from another client that the server broadcasted out), and just required minor changes to the world. Translating was an interesting one to do with my design of clients only handling their own player. When one client triggers a translation, it sends the event to the server where it translates all world objects other than players. It then broadcasts the event

out to all clients, which will move their player if they are not the one who triggered the translation. I only made 1 handler in the server, as most of the processing there just involved moving the world objects. It handles adding, moving, and removing players, death, and translations.The server does not actually raise any events itself, just handles events sent by the client. For events like this that were just going to be send to the server, I didn't even make an event object in the client. I just made a string representation of it and sent it off to the server as a method to save processing power for creating an event I am not going to use.

The event processor design was based on the design in the lectures, but with a few changes. For my event queue, I used the priority queue from the c++ standard library. It allowed me to use a custom comparator to sort an event into the queue based on the timestamp of the event. [1] None of my events needed to be handled in the future, so I really did not take advantage of this in my engine, but it is a good feature to have for the future. It was nice not having to think about the order events are popped off the queue. As I was implementing my manager, I realized that just sorting by timestamp is not enough to handle the priority enum I had in my events. Thinking back I could have expanded my comparator to sort by timestamp and priority in one queue, but that did not come to me at the time. Instead, I made 1 queue for each priority level. The manager's raise function (I named it add to queue) looked at the event's priority and added it to the appropriate queue. [2] This meant in my event processor, I had to go through the top of 3 queues instead of just 1, adding a little more code (even if it is mostly the same for each one). [3] For each queue, I check if there is anything in it and use a while loop to go through and pop each event in the queue less than the current timestamp of the server. This is how the priority queue I am using is taken advantage of, as I just keep popping until I get to an event that has a time set in the future. I then go through each handler mapped to that event and call the onEvent method of that handler. Handling the high queue first, then medium, and lastly low. Since the client and server both have events they need to process, I had to use this code in both places. In my engine I just copied it into both of them. I did briefly wonder if it would be better to put this into another file or the manager class to promote better code practices, as I currently have to edit it in both places if I want to make a change. I did not have to change it much so I did not feel it was worth exploring, but something to consider for the future.

One way I was able to take advantage of the priorities in events was in movement and collisions. I made movement a high priority event, so it always got processed first in the frame. This allowed me to make collisions a medium priority event, so they are always processed after movement. I did not want the collision to be handled, then the player moved again which could just move them back into whatever it was colliding with.

One important thing I had to consider when designing the system was how to manage the lifespan of events in memory. I decided to use shared pointers when creating events. I did not know these existed before, as I am relatively new to C++. They made it really easy to make a persistent event, and I did not have to think about deleting them when I was finished with them. Popping them off the queue should be the last reference to it, and removing that last reference will automatically delete it. Moving forward I will probably using shared pointers for all my objects, as I am still using normal ones for some of the more permanent objects in the scene.

Another important thing to consider was using mutexes to prevent simultaneous data access. This was even harder this project, as the data structures were not being accessed

across different threads, in the client/server and event handlers, etc. There was a lot of different places where data was being changed, and I had to pass mutexes into my handlers to make sure they could block off the data being changed by the event.

APPENDIX

```
bool CompareEvent::operator()(std::shared_ptr<Event> e1, std::shared_ptr<Event> e2){
    return e1->timeStamp > e2->timeStamp;
}
```

[1]: Custom comparator for inserting into the event queue.

```
void EventManager::addToQueue(std::shared_ptr<Event> e){
    {
        std::lock_guard<std::mutex> lock(mutex);

        if(e->priority == HIGH){
            eventQueueHigh.push(e);
        }
        else if(e->priority == MEDIUM){
            eventQueueMedium.push(e);
        }
        else if(e->priority == LOW){
            eventQueueLow.push(e);
        }
    }

}
```

[2] raise (addToQueue) function.

```
{
    std::lock_guard<std::mutex> lock(eventManager->mutex);
    while(!eventManager->eventQueueHigh.empty() && eventManager->eventQueueHigh.top()->timeStamp <= gameTimeline.getTime()){
        std::shared_ptr<Event> ev = eventManager->eventQueueHigh.top();
        for(EventHandler* h : eventManager->handlers[ev->eventType]){
            h->onEvent(ev);
        }
        eventManager->eventQueueHigh.pop();
    }
    while(!eventManager->eventQueueMedium.empty() && eventManager->eventQueueMedium.top()->timeStamp <= gameTimeline.getTime()){
        std::shared_ptr<Event> ev = eventManager->eventQueueMedium.top();
        for(EventHandler* h : eventManager->handlers[ev->eventType]){
            h->onEvent(ev);
        }
        eventManager->eventQueueMedium.pop();
    }
    while(!eventManager->eventQueueLow.empty() && eventManager->eventQueueLow.top()->timeStamp <= gameTimeline.getTime()){
        std::shared_ptr<Event> ev = eventManager->eventQueueLow.top();
        for(EventHandler* h : eventManager->handlers[ev->eventType]){
            h->onEvent(ev);
        }
        eventManager->eventQueueLow.pop();
    }
}
```

[3] event processor