Jeremiah Knizley

CSC 481

Dr. Alexander Card

November 12, 2023

Part One:

For Part One, I mostly followed the design we discussed in class, but made a few alterations to suit my needs. After homework three, my game loop was in a fairly standardized ordering, which seemed to make the game perform much more consistently, so I went into this homework with the goal of not messing that up.

First, I made the representation for the Events. Events in my design are a single class called Event, which inherits from GameObject. The main benefit of them inheriting from GameObject are so that they can be transferred over the network for part two, so I will forgo explaining that for now. Events have several different preset fields. First, they have a struct called variant, which we talked about in lectures. It proved to be very useful, though a bit tedious to use for events that have a lot of fields (A2). The types of variants I have right now include primitive types like integers, booleans, and floats, as well as GameObject pointers. Events also have a string field called type, which tells the manager what type of event it is, so it can be dispatched appropriately. Where my design differs from the design we talked about in lecture is that it also contains two more predefined fields, namely the integers order and time. In my design, all events need these, though they can simply remain at zero and things won't break. Time is the tic of the global timeline that the event should be handled. Order is the ordering of events should they take place at the same tic, more on this later.

In order to accomplish ordering the events by time, I had to make a few upgrades to my Timeline class. I added a convertGlobal() function which converts a tic from the current timeline and converts it into a global tic rate. Basically if a normal timeline has a tic rate of 8, and the global timeline has a tic rate of 1, it will multiply the supplied value by 8 recursively (A3). This is designed to work with any number of recursive anchoring, so if you have three timelines in a link for example.

Next is the EventManager class. EventManager has two private fields, a GameWindow and a Timeline. These fields are optional, as not all Events may need them, but enough of them do need them that I decided to add it in. It also has two more fields; an unordered map with key-value pairs of strings and a list of EventHandlers. This is how the manager stores its handlers. Each type of event is assigned one or more EventHandler on startup. This allows you to call multiple event handlers for a single event, if you want. The second field is a map of key-value pairs int and a multimap. The multimap contains key-value pairs int and Events. This is the way that priority is handled for events. The first map contains the time priority. For any given Event, at time t, they will be stored at key t, and contain a multimap. The multimap is determined by the order parameter of the Event. Basically, each Event is registered by time and ordered within each time value, should they take place at the same time from different threads (A4). This allows multiple threads to raise events asynchronously and still have an established order, which is very useful. Deregistering works similarly, and simply removes the value from the multimap (Or the outer map, if there are no more events at that time) (A5).

Lastly are the handlers. To make things simple, I stored all of mine in the same header and cpp file. All of them extend from a base EventHandler class, which only contains an onEvent method. This makes it easy to implement new handlers and store them together. It's simple but

effective. Other than that, each event handles events in their own way. My collision event is largely used for when an object collides with my character, such as a moving platform hitting the character. It moves them along with the platform if needed. My gravity event is similar to the collision event, but handles the collision differently. It handles moving the character along horizontal platforms when the character is standing above it, as well as handling vertical-only collision types such as DeathZones. It also, obviously, moves the character down if that won't collide with anything. My MovementHandler handles movement. It's fairly simple. It moves the character right or left based on whatever parameter was sent with the event. It then checks for collisions and handles them on its own. Again, these collisions are handled differently than my collision handler, because the circumstances surrounding the collisions are different. Basically, my "collision" event occurs when something collides with the player, whereas gravity collisions and movement collisions are caused by the player. I could have made three if statements in CollisionHandler to handle each type, but they need to be handled immediately anyway, so there wasn't much point in doing that. Lastly are my DeathHandler and SpawnHandler. In my implementation, these two work together. Whenever a death event is raised, and then subsequently handled, the DeathHandler will raise a spawn event to be handled. This is also the place I decided to demonstrate my EventManager being able to queue up events in the future. In my current implementation, the SpawnEvent is raised with a time value of three seconds (or 3000 tics of the global timeline) in the future. This period would probably be where a "YOU DIED" screen would come up before respawning. If you test it out now, you will see the character fall into the void, and then three seconds later they will appear back at the start. The only problem I ran into when implementing this after I set up my event manager was with queueing multiple events. I ended up having to create a field in character "isDead" or something

like that, so that the Death event wouldn't keep queuing as I collided with the death zone multiple times, and then setting the field back to false when I respawned.

As a side note, if you want an event to be handled immediately, you can raise an event with the same time value you are currently on, but with an ordering of the current order plus one. Assuming you are iterating through the events correctly, this will allow you to handle new events as they are created by other events being handled. For example, if I wanted my spawn event to be handled immediately, and the Death Event's time value was tic 36, then I would raise a spawn event with time 36, and an order of one, assuming the Death Event's order was zero.

On the topic of iterating through events, with my implementation this needs to be done in a fairly specific way, so as to make it work with both future events as well as events being raised immediately. First, because the outer map of the raised_events parameter is an ordered map, events with the soonest time are listed first. That means that once you get to an event that isn't supposed to be handled yet, you can disregard all events after that, because they will be either equal to or more in the future than that event. This can save time by quite a bit if your implementation likes queueing lots of events in the future. With that in mind, the way that I go through the map is similar to a queue, except I remove the element only after I've processed it. This was originally because I wanted to be able to insert a new element from inside the event being handled (Like how Death Events can create Spawn Events), and still keep iterating through the list while handling the new events. Now though, it's not entirely necessary as long as you insert the new element at order + 1. But I kept it the same nonetheless because it doesn't hurt anything, and you might forget to increase the order for the new event (A6). Also, if you don't want your event to be handled at that time, so as to prevent a chain or something, then you can just put the order at zero, and it will be skipped over.

Overall this part of the assignment wasn't too complicated. The main rationale behind the design was to make it as efficient as possible while also making it easy to be used from multiple threads, as well as at any time and in any order that you want. At that, I think it succeeds fairly well.

Part Two:

For Part Two, I struggled with understanding when or why you would want to send an event to the server. After all, at the time, the only things my clients were sending were their character objects. It seemed rather pointless to send anything else. Also, there was the matter of creating a way to send the objects across the network. Previously, for GameObject, each subclass would create its own toString method which would effectively serialize the object and send it across, and would then use that string to be reconstructed later. Here I faced some problems, as Events were a GameObject, but were also very dynamic. Also, some things wouldn't need to be sent, such as pointers to locally stored objects. After all, there's no point in sending a pointer to a Character object that is held on the client's machine. There isn't even much point in reconstructing the character on the server, the only use for it would be to distribute it to other characters. But then, I'm already doing that without the use of events, and frankly it's much simpler. The other problem is how to synchronize timelines between the client and the server. After all, the client may start at a time of 0 when the server is on tic 80,000. To address this, I did the following.

To address these problems, I did the following: First, I decided that the only things that would be sent across the network using Event's toString would be presets and primitive types. Usually, this is all you should need. There is zero reason to send a GameWindow across the

network for example. Now, you could argue that there is a good reason to send a GameObject over the network but also inside an event. To do this wouldn't be much extra work, I'd simply use the toString() method for that game object. However, I'd also need to include templates in order to construct the GameObject just like I did in my last homework for GameWindow (A1). Doable, but not really necessary if I'm being honest. You could just as easily send the object over normally and then just add them to the event. For the problem of time, what I did is I would send over an event in which its time value was the remaining time from the server's perspective (A7). Then, on the client, I simply added it to the client's current time, and it synced up quite nicely.

Now then, to explain my actual design, I ended up doing a distributed design. Both the client and the server have EventManagers and can register, raise, and handle events all on their own. For example, my Client utilizes the Movement, Collision, Gravity, Death, Spawn, and Closed event. My server utilizes only one event, the Closed event. The closed event is how I decided to handle closing the server, since there wasn't actually a way to do it before. Essentially, at the start of the server, it creates an event that takes place a certain period of time in the future. I believe for the final version I was using 120,000 tics or two minutes(A8). And, after part one, this worked great. After registering the event, coding the handler, and raising it, as well as copy pasting my handling code over from the client, everything worked smoothly.

However, then comes an important problem; if the server closes, the client must also close, ideally at the exact same time if possible. You don't want to be stuck in the server waiting for an update after it's been closed (which did happen when I wasn't using convertGlobal exactly correctly!). To do this, I created my toString and constructSelf methods for my Event class. Now, originally, I wanted to have an event called a "NetworkEvent" that would be raised whenever you wanted to transfer information over from the server to the client, or from the client to the

server. You would give it a socket, or a port, etc. and things like that. I included a screenshot showing one attempt at doing this (out of three) (A9). I actually still have the SOCKET type in my Event for when I was trying to do this, because I may revisit it later. However, I ran into issues with 0MQ because it seems to very much not like it when you try to use a socket from a different thread than the one that created it, similar to RenderWindows in SFML. Even when being thread-safe, and making sure not to send/receive at the same time, I still ran into issues regarding the ordering of these events that made my program too inefficient with multiple clients. The other reason I abandoned this idea is that most of the time, it's completely unnecessary. The whole idea of the events is for them to be queued and then handled. It's a waste of time to queue the event on the server, then handle it by sending it to the client, and then have the client queue that same event that should have already happened, and then handle it later. So, I took a different approach.

By that, I mean I simply sent the event over to the client the moment that I wanted to, just like I would for any other GameObject. In RepThread (The thread that communicates with individual clients), at the very beginning, I send an Event with type "Client_Closed", and a time value of 120,000 tics - the current time of connection. I also sent a string message (A10). This message should be the reason for disconnecting. It can be anything, such as "Game Over" (which is what I used), "Times up", "We are conducting maintenance…" or "You have been issued a VAC ban, and have been disconnected from the server". When the client receives the event, it uses the GameObject abstract function createSelf to reconstruct the Event and raise it. It had already registered the Client_Closed event previously. Then, presumably several thousand tics later, the event gets handled and the client is shut down. By the way, the way I deal with the "parameters" map in the Event object is that I iterate through them, adding their key, type, and

value to the string so that they can be reconstructed correctly in constructSelf (A11). I tried this out by using several parameters of different types, and it seemed to make the string and construct it fine.

Originally I ran into a bit of a problem with this. My timing was extremely accurate, however my original handling loop didn't happen until about halfway through the tic, and the first thing that happened in the loop was that I waited for more things from the server. That wouldn't do, because then the client would be waiting for a new message before realizing that the server was closed. So, I added another loop that would handle any events that needed to happen at the beginning of the tic. This is very reminiscent of the way it was presented in the lectures, where it would handle High priority tasks at the beginning, then Medium, then Low. Anyway, if you run this version on multiple clients, they will run for two minutes (try and see how many you can open), and then they will all close at the exact same time, along with the server. It's very satisfying.

As a small side note, the reason why I didn't use PUB/SUB for sending the message is twofold. First, it's not a message that needs to be repeated. It really only needs to be sent once, which is kind of the beauty of the Event system. Once it's queued, you don't need to do anything but wait. So I simply needed it to send it to each new client as they connected, which should have improved efficiency (because I send much less often). The other reason is because I'm still conflating messages on the subscriber end. I have an idea of how to get around that using the event system, but unfortunately I wasn't able to get it to a working state by the deadline. Perhaps I'll finish it before the next one. But the main problem was that if I'm conflating messages, there would be a chance that I would miss important events, which could be very bad. Whereas for my Req/Rep, my client receives every message. Also, up until now, there wasn't really any purpose

in sending anything back to the client through req/rep, but now there is so it isn't wasteful anymore.

Overall, part one definitely was the simplest to figure out. Once I had an idea of what data structures I wanted to use, and what I wanted to prioritize in my game engine, it became pretty simple. Part two on the other hand was more challenging, because I had a lot of decisions to make about what would serve the game better overall. Also networking is just more challenging to be honest. I spent a lot of time trying to get that NetworkHandler to work, but just couldn't crack it. That being said, I don't think it was really needed, it just would have simplified my code a lot more.

KNOWN BUGS:

It's become something of a tradition for me to make this section, so here it is.

1. I still have the problem with the character moving very slightly then the platform changes direction. This varies depending on the tic rate, and is pretty much imperceptible if your client is running at a tic size of 8, which I've been able to easily manage since my optimization in homework three.
2. Not really a bug, more like something I want to change in the future. I'd like to change how my GameWindow works. Sometimes it's very nice to have all of those methods in there, but other times it makes it very annoying to have to give everything access to my GameWindow, even though they don't need to draw anything or check for collisions.
3. That's pretty much it. There are things I didn't go super in-depth on during this homework, partly because I didn't want to mess up what I did in homework three, and

there are still things I want to add. I actually think the optional replay feature wouldn't have been that difficult to implement. I already use setPosition over move anyway for my objects, so it should have just been a matter of making all of my game object updates pass through my event manager, and then exporting them to another manager or map to be replayed.

APPENDIX

A1:

```
//Scan through each object
while (sscanf_s(updates.data() + pos, "%[^,]%n", currentObject,(unsigned int)(updates.size() + 1), &newPos) == 1) {
    //Get the type of the current object.
    int type;
    int matches = sscanf_s(currentObject, "%d", &type);
    if (matches != 1) {
        free(currentObject);
        throw std::invalid_argument("Failed to read string. Type must be the first value.");
    }
    //Push the newly created object into the array.
    nonStaticObjects.push_back((templates.at(type))->constructSelf(currentObject));
    //update position and skip past comma
    pos += newPos + 1;

}
```

A2:

```
Event c;
{
    c.time = line->convertGlobal(currentTic);
    c.type = std::string("collision");
    Event::variant upPressedVariant;
    upPressedVariant.m_Type = Event::variant::TYPE_BOOLP;
    upPressedVariant.m_asBoolP = upPressed;
    Event::variant doGravityVariant;
    doGravityVariant.m_Type = Event::variant::TYPE_BOOLP;
    doGravityVariant.m_asBoolP = &doGravity;
    Event::variant characterVariant;
    characterVariant.m_Type = Event::variant::TYPE_GAMEOBJECT;
    characterVariant.m_asGameObject = character;
    Event::variant ticLengthVariant;
    ticLengthVariant.m_Type = Event::variant::TYPE_FLOAT;
    ticLengthVariant.m_asFloat = ticLength;
    Event::variant differentialVariant;
    differentialVariant.m_Type = Event::variant::TYPE_INT;
    differentialVariant.m_asInt = currentTic - tic;
    c.parameters.insert({ { "upPressed", upPressedVariant }, { "doGravity", doGravityVariant } });
    c.parameters.insert({ "character", characterVariant });
    c.parameters.insert({ "ticLength", ticLengthVariant });
    c.parameters.insert({ "differential", differentialVariant });
}
```

A3:

```
int64_t Timeline::convertGlobal(int64_t time)
{
    if (anchor) {
        return anchor->convertGlobal(time) * tic;
    }
    else {
        return time * tic;
    }
}
```

A4:

```
std::map<int, std::multimap<int, Event>> raised_events;

std::unordered_map<std::string, std::list<EventHandler*>> handlers;
```

A5:

```
void EventManager::registerEvent(std::list<std::string> list, EventHandler* handler)
{
    for (std::string i : list) {
        if (handlers.count(i)) {
            handlers[i].push_back(handler);
        }
        else {
            std::list<EventHandler*> newList;
            newList.push_back(handler);
            handlers.insert({ i, newList });
        }
    }
}

void EventManager::deregister(std::list<std::string> list, EventHandler* handler)
{
    for (std::string i : list) {
        if (handlers.count(i)) {
            handlers[i].remove(handler);
            if (handlers[i].empty()) {
                handlers.erase(i);
            }
        }
    }
}
```

A6:

```
//Handle all events that have come up.
bool erase = false;

for (const auto& [time, orderMap] : em->raised_events) {
    if (time <= line->convertGlobal(currentTic)) {
        for (const auto& [order, e] : orderMap) {
            for (EventHandler* currentHandler : em->handlers.at(e.type)) {
                currentHandler->onEvent(e);
            }
        }
        if (erase) {
            em->raised_events.erase(em->raised_events.begin());
        }
        erase = true;
    }
    else {
        break;
    }
}
if (erase) {
    em->raised_events.erase(em->raised_events.begin());
}
```

A7:

```
try {
    std::shared_ptr<Event> e(new Event);
    //Convert to an event pointer
    e = (std::dynamic_pointer_cast<Event>(e->constructSelf(replyString)));
    e->time = line->convertGlobal(currentTic) + e->time;
    //Raise event
    em->raise(*e);
}
```

A8:

```
//Set up EventManage for server
EventManager manager(&global);
std::string serverClosed("Server_Closed");
std::string clientClosed("Client_Closed");
std::list<std::string> types;
types.push_back(clientClosed);
types.push_back(serverClosed);
manager.registerEvent(types, new ClosedHandler);

//Add server closed event.
Event e;
e.time = GAME_LENGTH; //FAME_LENGTH into the future
e.type = "Server_Closed";
std::string message = "Server Closed";
Event::variant messageVariant;
messageVariant.m_Type = Event::variant::TYPE_STRING;
messageVariant.m_asString = message.data();
e.parameters.insert({ "message", messageVariant });
manager.raise(e);
```

A9:

```
//NetworkHandler::NetworkHandler(EventManager* em)
//{
//    this->em = em;
//}
//
//void NetworkHandler::onEvent(Event e)
//{
//    zmq::context_t context(2);
//    zmq::socket_t socket(context, zmq::socket_type::rep);
//    socket.connect("tcp://localhost:5556");
//    //try {
//    //    socket = e.parameters.at("socket").m_asSocket;
//    //}
//    //catch (std::out_of_range) {
//    //    std::cout << "Invalid socket parameter in NetworkHandler" << std::endl;
//    //    exit(3);
//    //}
//    if (e.type == "Client Closed") {
//        e.time = GAME_LENGTH - em->getTimeline()->getGlobalTime();
//        std::string rtnString = e.toString();
//        zmq::message_t reply(rtnString.size() + 1);
//        memcpy(reply.data(), rtnString.data(), rtnString.size() + 1);
//        socket.send(reply, zmq::send_flags::none);
//        zmq::message_t update;
//        zmq::recv_result_t received(socket.recv(update, zmq::recv_flags::none));
//    }
//}
```

A10:

```
zmq::context_t context(2);
zmq::socket_t repSocket(context, zmq::socket_type::rep);
repSocket.connect(portString);

Event init;
init.type = "Client_Closed";
std::string message = "Game Over";
Event::variant messageVariant;
messageVariant.m_Type = Event::variant::TYPE_STRING;
messageVariant.m_asString = message.data();
init.parameters.insert({ "message", messageVariant });

Event::variant socketVariant;
socketVariant.m_Type = Event::variant::TYPE_SOCKET;
socketVariant.m_asSocket = &repSocket;
init.parameters.insert({ "socket", socketVariant });

zmq::message_t update;
zmq::recv_result_t received(repSocket.recv(update, zmq::recv_flags::none));

init.time = GAME_LENGTH - manager->getTimeline()->getGlobalTime(); //Time differential
std::string rtnString = init.toString();
```

A11:

```
while (sscanf_s(self.data() + pos, "%[^.]%n", current, (unsigned int)(self.size() + 1), &newPos) == 1) {

    char* key = (char*)malloc(strlen(current) + 1);
    Event::variant::Type type;
    int innerPos = 0;

    //Get the key and the type of value
    int matches = sscanf_s(current, "%s %d%n", key,(unsigned int)strlen(current) + 1, & type, &innerPos);
    //Throw if key or type is invalid
    if (matches != 2) {
        free(key);
        throw std::invalid_argument("Failed to read string. Type must be the first value.");
    }
    //If it's an int
    if (key && type == Event::variant::TYPE_INT) {
        int value;
        sscanf_s(current + innerPos, "%d", &value);
        Event::variant intVariant;
        intVariant.m_Type = Event::variant::TYPE_INT;
        intVariant.m_asInt = value;
        e->parameters.insert({ std::string(key), intVariant });
    }
    //If it's a float
    else if (key && type == Event::variant::TYPE_FLOAT) {
        float value;
        sscanf_s(current + innerPos, "%f", &value);
        Event::variant floatVariant;
        floatVariant.m_Type = Event::variant::TYPE_FLOAT;
        floatVariant.m_asFloat = value;
        e->parameters.insert({ std::string(key), floatVariant });
    }
    //If it's a boolean
    else if (key && type == Event::variant::TYPE_BOOLP) {
        int value;
        sscanf_s(current + innerPos, "%d", &value);
        bool *boolValue = new bool(value);
        Event::variant boolVariant;
        boolVariant.m_Type = Event::variant::TYPE_BOOLP;
        boolVariant.m_asBoolP = boolValue;
        e->parameters.insert({ std::string(key), boolVariant });
    }
    //If it's a string
    else if (key && type == Event::variant::TYPE_STRING) {
        char* value = (char *)malloc(strlen(current) + 1);
        sscanf_s(current + innerPos, " %[^.]", value, (unsigned int)strlen(current) + 1);
        Event::variant stringVariant;
        stringVariant.m_Type = Event::variant::TYPE_STRING;
        stringVariant.m_asString = value;
        e->parameters.insert({ std::string(key), stringVariant });
    }
```