Writeup for Homework 4

200333470

yhuang64

Yuanmin Huang

Section 1

The work for the first section is to implement an EventMgmt System, so let's make the writeup begin with the event class design (**representation**).

The base class for all events is an Event class. It contains a type, an execution time and a map of arguments, which is content types to variants. The content type here indicates which type of argument (semantically) the corresponding variant contains.

```
8   enum class Event_t {
9       CHAR_COLLISION, CHAR_DEATH, CHAR_SPAWN, USER_INPUT, OBJ_MOVEMENT
10  };
11
12  enum class Content_t {
13      X_VAL, Y_VAL, CHARACTER_PTR, OBJ_PTR, USER_INPUT, POSITIVE
14  };
15
16  class Event
17  {
18  protected:
19      Event_t type;
20      double executeTime;
21      map<Content_t, Variant> args;
```

This leads us to the design of Variant class. As is shown on the right, it contains a variant type (syntactically) and a union of values to store the actual value of the variable.

```
8   enum class Variant_t {
9       DOUBLE, OBJ_PTR, KEY, BOOL
10  };
11
12  class Variant
13  {
14  private:
15      Variant_t type;
16      union {
17          double double_val;
18          GameObject* obj_ptr;
19          Keyboard::Key key;
20          bool bool_val;
21      } var;
```

Based on this, I then designed 5 types of events. They are:

1. ECharCollision,
   Which has a character pointer and the pointer of the game object character is colliding with;
2. ECharDeath,
   Which only has a character pointer;
3. ECharSpawn,
   Which has a character pointer and the pointer of the SpawnPoint the character is going to spawn at;

4. EObjMovement,
   Which has a pointer of game object which is going to move, the position (x_value, y_value) it's going to move to and an optional boolean value of heading positive used for moving platforms;
5. EUserInput,
   Which has a character pointer and a Keyboard::Key value to indicate which key has been pressed.

If we want to add new class of events to our engine, we simply need to add new event types to the enum class and define which kind of arguments they would need to composite, which is quite easy to implement given the pre-defined Variant and Event class. We can surely add new Variant types and content types to enum class if needed.

Then, let's take a look at how we handle the events, i.e. the event handler (**handling**).

I have designed a giant event handler that has a general onEvent() function which forwards events to specific onType() functions based on the event type. I chose to implement it this way because I wanted to utilize the enum class of event types and simplify the design for registration in event manager. In this way, the event manager would only need to call the onEvent() function of event handler when handling events instead of notifying a list of separate handlers.

```
void EventHandler::onEvent(const ::Event* e)
{
    switch (e->getType())
    {
    case Event_t::CHAR_COLLISION:
        onCharCollision(*(ECharCollision*)e);
        break;
    case Event_t::CHAR_DEATH:
        onCharDeath(*(ECharDeath*)e);
        break;
    case Event_t::CHAR_SPAWN:
        onCharSpawn(*(ECharSpawn*)e);
        break;
    case Event_t::USER_INPUT:
        onUserInput(*(EUserInput*)e);
        break;
    case Event_t::OBJ_MOVEMENT:
        onObjMovement(*(EObjMovement*)e);
        break;
    default:
        break;
    }
}
```

As for each onType() function, the work we do here is mainly copying the code from previous assignments because the functionality we want to implement hasn't been changed.

Things new here are that when handling some events, they may generate new events. We have several situations here:

1. When handling the collision of a character with a death zone, a new ECharDeath would be generated;
2. When handling the character death, a new ECharSpawn would be generated;
3. When handling the character spawn, a new EObjMovement would be generated.

After discussing how we handle events, let's move on to when we handle the events, and that would be event manager's work.

My design of EventManager mainly has a reference to game time object, a double value of GVT, an event handler (**registration**), a map of client names to their event priority queues, a map of client names to their GVT and a list of EObjMovement events used for network communication.

```
class EventManager
{
private:
    Timeline& gameTime;
    double GVT;
    EventHandler handler;
    map <
        const char* const, // client name
        priority_queue<::Event*, vector<::Event*>, cmp>
    > queues;
    map <const char* const, double> GVTs;
    list<EObjMovement> objMovements;
    mutex mtxEvt, mtxQueue, *mtxObjMov;

    bool connected;
```

The key idea here is that we:

1. Maintain a priority event queue sorted by execution time descend;
   This is implemented by pass a customized compare function on Events.

```
struct cmp
{
    bool operator()(::Event* lv, ::Event* rv)
    {
        return lv->getExecuteTime() > rv->getExecuteTime();
    }
};
```

2. Update GVT every time before we handle events;
   The work we do here is simply comparing the GVTs stored in the map and taking the least one as our GVT.
3. Handle events according to GVT;
   For every priority queue in the map, we keep handling the events on top of the queue until the top event's execution time is larger than GVT. After an event is handled, remember to delete the pointer because it's "newed" and pop it out.

Finally, what we need to clarify is when do we generate event (**raising**). The simple answer is that when we occur where we originally deal with directly in our previous assignments. For example, in Movable component, when a platform moves, we need to generate a new EObjMovement event and store it in the event manager, while we simply moved the platform in the past.

Another important design in this section is how my networking part cooperate with the event management part.

My thought is that there is no need for other peers to know all the events occurs on my machine. The only thing important other machines need to know is the object movement happened on my machine. Here comes my purpose of designing the list of

```
string EObjMovement::toString()
{
    // E executeTime ObjID X_val Y_val
    return
        "E " +
        to_string(executeTime) + " " +
        getObject()->getId() + " " +
        to_string(getXVal()) + " " +
        to_string(getYVal()) + " " +
        to_string(getPositive()) + "\n";
}
```

EObjMovement in my event manager. This list is used to store all the object movement events and then be used for communicating with other peers in the network to notify others how the objects of mine have moved during the short amount of time passed.

This calls for a way to transport EObjMovement across the internet. I didn't choose to serialize it, instead, I wrote a toString() function to convert it into a string which contains essential information of the event. Then, in the other end of the network, the peer can use the string to generate a new event equivalent to the original one.

```
// insert new Event anyway
manager->getMtxQueue()->lock();
manager->insertEvent(
    new EObjMovement(
        atof(infos[2].c_str()) - connectedTime, // add time bias
        objects->find(infos[3])->second,// character
        atof(infos[4].c_str()),
        atof(infos[5].c_str()),
        infos[6] == "1"
    ),
    (const char* const)infos[0][0]
);
manager->getMtxQueue()->unlock();
```

The list of EObjMovement events is maintained by storing the object movement events of self's and clear up all the events in the list after sending the corresponding strings out.

```
// store the event for publishing if is object movement event
if (e->getType() == ::Event_t::OBJ_MOVEMENT && client_name == SELF_NAME)
{
    mtxEvt.lock();
    objMovements.push_back(*(EObjMovement*)e);
    mtxEvt.unlock();
}
```

```
// generate events string, SELF_NAME E executeTime ObjID X_val Y_val
list<EObjMovement>* newObjMovements = manager->getObjMovements();
manager->getMtxEvt()->lock();
for (EObjMovement e : *newObjMovements)
{
    // only send this character movement event
    if (e.getObject()->getId() == SELF_NAME)
        message += SELF_NAME + (string)" " + e.toString();
}
newObjMovements->clear();
manager->getMtxEvt()->unlock();
```

The tricky thing here is concurrency control, because we must avoid storing events after we have generated strings but before we clear the list, or we will lose some of the events, which may cause a short slowing down of displaying on other peers.

As you may have noticed in the code of parsing string to generate events, there is a " -connectedTime". This is another important part of networking communication structure: the time control.

As different clients can start at various time points, the timeline of their event managers rely on would differ from each other. So, the current time we get may not reflect the actual current time. My design is to let client first send its time point of connecting to server, then server records the bias between server time and client time by computing the difference between

the time client has sent and the time of server's. Finally, the server sends the bias value back to client so that client also knows the value.

Later, the value is used when sending and subscribing time-related messages to and from server. My goal here is to keep the server receiving and publishing all the time value according to server timeline. Therefore, when a client sends message out, it needs to add the bias to the

```
void Client::sendHandler()
{
    if (!connected) // don't send if disconnected
        return;

    // first, name GVT double
    string message = SELF_NAME + (string)" GVT " +
        to_string(manager->getRequestGVT() + connectedTime) + "\n";
```

GVT; when the server receives the message about events, it also needs to add the bias to the

```
// insert new Event anyway
manager->getMtxQueue()->lock();
manager->insertEvent(
    new EObjMovement(
        atof(result[2].c_str()) + connectTimes[result[0]], // add bias time
        characters.find(result[3])->second,// character
        atof(result[4].c_str()),
        atof(result[5].c_str())
    ),
    (const char* const)result[0][0]
);
manager->getMtxQueue()->unlock();
```

execution time. When the server publishes message, it directly sends all the event strings out; and when a client subscribes the message, it subtracts the bias on all time values. In this way, we make all the timepoints agree with each other on when events truly happened.

But, we need to understand that this only works when SERVER STARTS FIRST because the client wouldn't send the connect message several times. It would only send once and wait for the server to respond, so the it would only send a time point really close to its startup, which means that the server would not be able to get a big negative value of time point if a client starts really early before the server.

In addition, you may have notice that every time I insert an event to event manager, I add a mutex. This is due to another tricky concurrency control which lies in handling events meeting with inserting events. Inconsistency of GVT computation at one execution may occur when an event is inserted to the top of a queue (this may happen because of network communication latency) after we have set the GVT and begin to handle events.

This may also become a matter when a client disconnects, and we need to erase the corresponding queue and GVT. Imagine that we suddenly remove a queue when the event manager is working in a loop of all queues. Unexpected errors would occur if we don't add any concurrency control to it.

Originally, I tried to implement a sequential event management that I only call executeEvents() once every time the client receives a message. After running it, I found that that was too slow, and events would be clustered. Hence, I wrote another function to keep

executing events and made it a new thread. This works much better than the original design. I think that this echoes what we have talked about multi-threaded design in class.

```
// init event manager
EventManager manager(gameTime, &mtxObjMov);
thread exeEvent(&EventManager::keepExecutingEvents, &manager);
exeEvent.detach();
```

```
void EventManager::keepExecutingEvents()
{
    while (true)
    {
        mtxQueue.lock();

        if (!connected)
        {
            mtxQueue.unlock();
            break;
        }

        executeEvents();

        mtxQueue.unlock();
    }
}
```

However, may be because of the multiple threads server and clients have, when I try to run multiple clients and the server on my computer, whose CPU has only dual-core and is running at 100%, it doesn't work well.

When one client is online, there are only some slight slow downs at some random point, but the character can still move smoothly. When two clients are online, the situation becomes unstable that the movement of one character may have somehow visible latency to reflect on the other client. Also, there is a chance that the character fall through the platform and death zone because of the insufficient running loops of collision detection. When there are three or four clients running on my computer at the same time, the game is unplayable, and the characters will surely fall through the platform.

I hope that my program would work better on a computer with better CPU. I think that it should do because the lack of cores would be removed, but I don't have a chance to test. Also, I think that originally, it is unrealistic to run multiple clients together with the server on one machine. And to test the functionality of event management system and network design, running two clients simultaneously should suffice.

Section 2

The task for this section is to implement a replay system using event management system and timeline system.

For my engine specifically, the replay system would be storing EObjMovement events when recording and then make the event manager execute those stored events when replaying. This will work because all the movement events give us a trace of how all the objects have moved on the screen. Thus, by executing them again, we achieve a replay effect. In addition, during the replay, the timeline can be set to different type of step size so that we can control the

replay speed.

Here is the design of my Replay class.

```
12  class Replay
13  {
14  private:
15      Timeline* gameTime;
16      GameTime replayTime;
17      double startTime;
18      bool isRecording, isPlaying;
19      priority_queue<::Event*, vector<::Event*>, cmp> records;
20      EventManager* manager;
21      Client* client;
```

The records priority queue here is used for queuing recorded events. The manager and client pointer here are used for manipulating stuff in them when handling start/end record/replay events.

According to the assignment, I also implemented EStartREC, EEndREC, EEndPlaying events to utilize the event mgmt. system to handle start/end recording instructions.

To be clearer, the complete flow for a replay in my engine would be:

1. When player hit R key during game play, an EStartREC event would be generated, which only carries the pointer of a replay instance;
2. When handling the EStartREC event, the startRecording() function of replay will be

```
19  void Replay::startRecording()
20  {
21      replayTime = GameTime(1);
22      startTime = gameTime->getTime();
23      replayTime.setPaused(true);
24      isRecording = true;// manager will begin to record EObjMovement events now
25  }
```

called;

In the function, a new GameTime object is generated and paused at once, which would be used by the event manager to handle events when replaying; the start time is recorded also for handling events because the new game time object will count time from 0; the isRecording is set to true so that the event manager would begin to store object movement events to the records queue.

3. When the player hit the E key, an EEndREC event would be generated;

```
126         // store the event for publishing if is object movement event
127         if (e->getType() == ::Event_t::OBJ_MOVEMENT)
128         {
129             // only store self movement events for networking
130             if (client_name == SELF_NAME)
131             {
132                 mtxEvt.lock();
133                 objMovements.push_back(*(EObjMovement*)e);
134                 mtxEvt.unlock();
135             }
136
137             // store the event for replaying if is recording
138             if (replay->getIsRecording())
139             {
140                 replay->pushEvent((EObjMovement*)e);
141             }
142         }
```

4. When handling EEndREC event, the endRecording() function of replay would be called firstly;

```
27  void Replay::endRecording()
28  {
29      records.push(new EEndPlaying(gameTime->getTime(), this));
30      isRecording = false;
31  }
```

In the function, one last event EEndPlaying to end the replay is pushed to the bottom of the priority queue records; isRecording is set to false to stop the event manager from storing events to records.

5. After that, immediately, the startPlaying() function would be called;

```
33  void Replay::startPlaying()
34  {
35      isPlaying = true;
36
37      manager->setTimeline(&replayTime);
38      client->disconnect(true);// this will also clear events in manager queues and GVTs
39      manager->addQueue("R", &records);
40
41      replayTime.setPaused(false);
42  }
```

isPlaying is set to true so that the client, manager and main loop would know that a replay is been playing. When replaying, the client will no longer send or subscribe messages; the manager would use different time management to handle events; the main loop will no longer receive key inputs for moving characters. Instead, it receives key inputs for changing replay speed.

We then set manager's time line to the replay time we generated when we start recording; make client temporarily disconnect from server for replay so that server will not waiting for the client to send event messages to it. The replay on one client will not affect other clients game play as well. The old events and GVTs stored in manager will be cleared to prevent us from handling events wrongly.

Finally, we add the queue of records we stored to event manager and start the replay time. The event manager will then automatically execute those events for us and shows a replay to us.

6. Remember that the last event in queue is a EEndPlaying event;

```
44  void Replay::endPlaying()
45  {
46      manager->setTimeline(gameTime);
47      manager->setReplay(false); // this will remove queue for replay
48      manager->removeQueue("R");
49      client->connect();// reconnect to server
50
51      // clear recorded events for this recording
52      while (!records.empty())
53      {
54          delete records.top();
55          records.pop();
56      }
57      isPlaying = false;
58  }
```

In the function, the timeline of manager is returned to normal; the queue for replay is removed; the client reconnect to server so that the game play return to normal.

At this point, other clients can see the client's character appear again.
Moreover, we need to delete and clear all the events in record queue to prepare for next replay.

Specifically, to change the speed of replay, we just need to reset the step size of our replayTime object. This will automatically change how fast the time flies when the replay is on.

Also, to me, letting the replaying client disconnect from server is a quite straightforward way to handle replay without affecting other clients as well as handling replay more convenient (as we no longer have other events or GVTs in the event manager so that we can utilize the manager without modifying the main logic of event handling).