Hyunsirk Choi

CSC 481

Dr. Robert

14 November 2022

Homework 4 Write-Up

I have created 2 additional classes to implement the Event Management System on my currently existing project. These classes are `EventManager` and `EventObject` classes which the `EventManager` takes care of registering and polling events and `EventObject` is the representation of any event. There is only a single `EventManager` in the entire program where different `GameObject` will have access to this `EventManager` for registering new events and listening to existing events to handle them. There are 4 different event types which are character input, character and platform collision, and character death and spawn. These different types are declared as `static const` variables in the `EventObject` which is easy to scale by declaring a new event type.

Event Representation

`EventObject` is consist of 4 fields which are event type, priority, event object header, and event content. The event type is one of those static constant values that were declared. The priority, for now, is set to a hard-coded value to enforce execution orders of different event types such as polling character death event first before character input; however, this priority can be dynamically assigned using a Timeline object to execute all events in chronological order. The event object header is the unique header ID of the source object which will be affected by this event. It is used to identify which object will be able to listen to and handle the event action. The last field is the event content which is often set to empty as the `EventObject` itself has all of the necessary information to trigger an event, but there are some events such as character input which require additional information which can be stored in the event content field.

```
const std::string EventObject::CHARACTER_INPUT = "CharacterInput";
const std::string EventObject::CHARACTER_DEATH = "CharacterDeath";
const std::string EventObject::CHARACTER_SPAWN = "CharacterSpawn";
const std::string EventObject::CHARACTER_PLATFORM_COLLIDE = "CharacterPlatformCollide";
```

Event Registration

`EventManager` has a vector array of these `EventObject` to represent a queue of events. New `EventObject` can be registered using `EventManager::registerEvent(EventObject *newEvent)` function which will push the new event to its appropriate index based on its priority. This function call can be called anywhere in the program including the main thread and in any `GameObject`. For example, the character input is registered in the main thread after detecting any keyboard inputs and character-platform collision and character death/

spawn are registered in `CharacterObject` and `DeathZoneObject` while updating their states. There is no limit on where an `EventObject` can be registered as long as there is access to the `EvenManager` object.

```
// Character input
std::string inputs = "";
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
{
    inputs += "Left,";
}
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
{
    inputs += "Right,";
}
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space) &&
    !character->getIsJumping())
{
    inputs += "Space,";
}
eventManager->registerEvent(new EventObject(EventObject::CHARACTER_INPUT,
                                            EventManager::CHARACTER_INPUT_PRIORITY,
                                            character->getHeader(),
                                            inputs));
```

Event Raising

There is no explicit “raising” of the event as my implementation is not using a Listener-Handler event system. The implemented `EventManager` has a `pollEvent(std::string eventType, std::string header)` function that can be called from anywhere as long as there is access to the `EventManager` instance similar to the registration. The `eventType` and `header` parameters identify if the next event on the queue matches the given information. If it matches, it will pop the event from the queue and return the event instance, if not, it will not pop the event and simply return `NULL`. Using this return response, `CharacterObject` and `DeathZoneObject` can have “check event” functions that can be called in every iteration to detect any character input, collision, or death, spawn events to react on.

```
// Trigger event listeners
character->checkInputEvent();
character->checkPlatformCollideEvent();
deathZone->checkDeathEvent();
deathZone->checkSpawnEvent();
```

Event Handling

The event handling is also nothing explicit and is manually implemented after detecting and raising an event from a `GameObject`. Using the pollEvent() function, the `GameObject` will look for a specific event and if the function returns rather than a `NULL` value, it will make certain actions appropriate to the event.

```
void CharacterMovableObject::checkInputEvent()
{
    EventObject *currentEvent =
        MovableObject::getEventManager()->pollEvent(EventObject::CHARACTER_INPUT,
                                                    MovableObject::getHeader());

    if (currentEvent != NULL && currentEvent->getContent() != "")
    {
```