CSC581 FALL 2023 HOMEWORK4

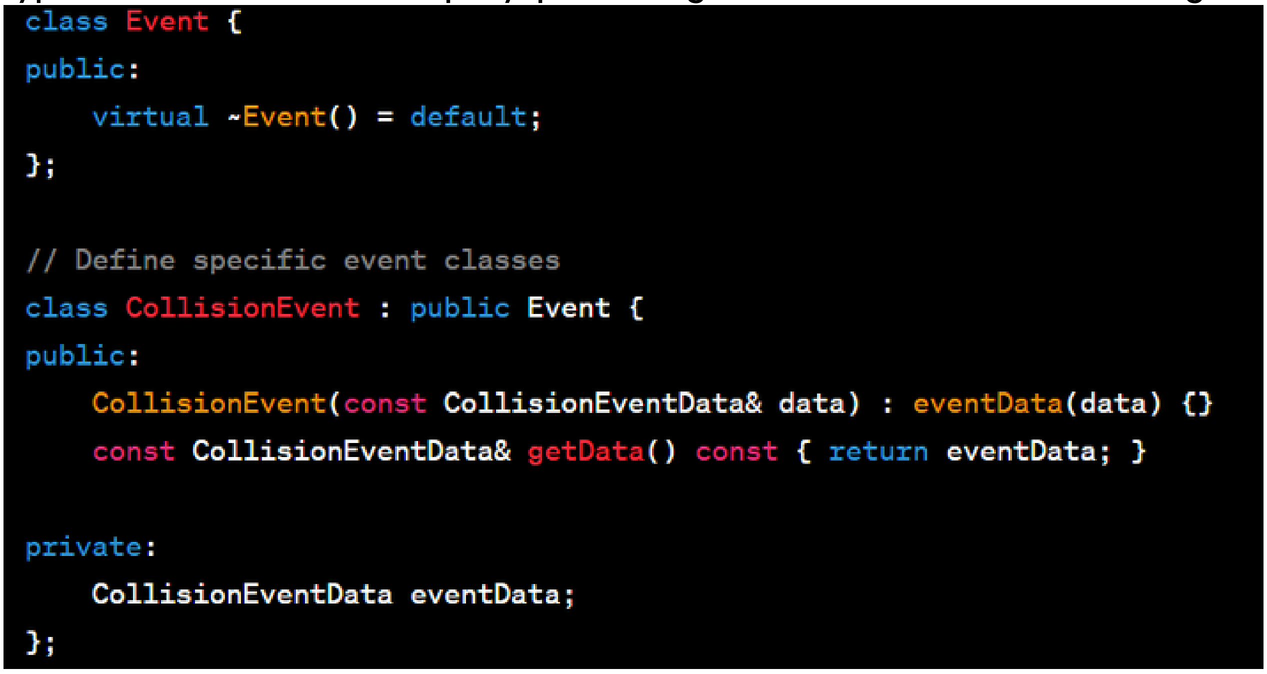
Submitted by:
Unnati bukhariya

Submitted to:
Dr. Alexander Card

1.

The conceptualization and development of an event-driven game engine represented a captivating journey, involving meticulous design decisions and their subsequent implementation. This comprehensive exploration provides an intricate breakdown of my decisions, accompanied by extensive code snippets to unveil the intricacies of my SFML platformer.

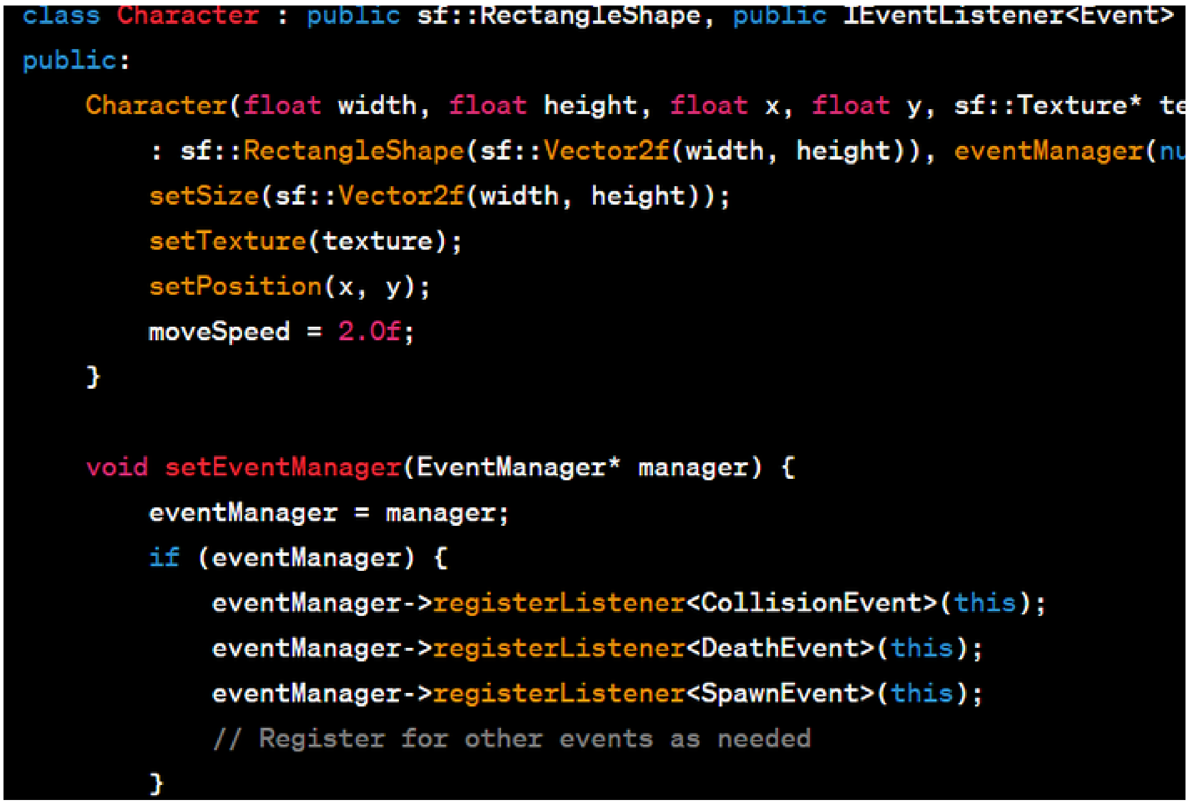
1. Event Representation:
The cornerstone of the system is the event representation. The Event class hierarchy, meticulously crafted, serves as a versatile container for in-game occurrences. The core class, depicted below, establishes a framework for derived events, ensuring uniformity across the board. Events, encapsulated in a versatile hierarchy, allowed for a structured and extensible system. The abstract Event class, with its virtual handle method, ensured that each event type could be handled uniquely, promoting a modular and scalable design:



```cpp
class Event {
public:
    virtual ~Event() = default;
};

// Define specific event classes
class CollisionEvent : public Event {
public:
    CollisionEvent(const CollisionEventData& data) : eventData(data) {}
    const CollisionEventData& getData() const { return eventData; }

private:
    CollisionEventData eventData;
};
```

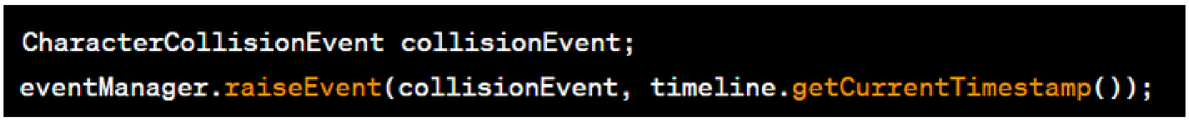
2. Event Registration
The Event Manager's role in dynamic event registration was pivotal. Enabling systems or game objects to dynamically register for events injected a high degree of customization. The system didn't just react; it allowed entities within the game to define their responsiveness dynamically. This design choice catered to the diverse needs of different components, promoting a flexible and adaptable architecture.

1.



```
class Character : public sf::RectangleShape, public IEventListener<Event>
public:
    Character(float width, float height, float x, float y, sf::Texture* te
        : sf::RectangleShape(sf::Vector2f(width, height)), eventManager(nu
        setSize(sf::Vector2f(width, height));
        setTexture(texture);
        setPosition(x, y);
        moveSpeed = 2.0f;
    }

    void setEventManager(EventManager* manager) {
        eventManager = manager;
        if (eventManager) {
            eventManager->registerListener<CollisionEvent>(this);
            eventManager->registerListener<DeathEvent>(this);
            eventManager->registerListener<SpawnEvent>(this);
            // Register for other events as needed
        }
```

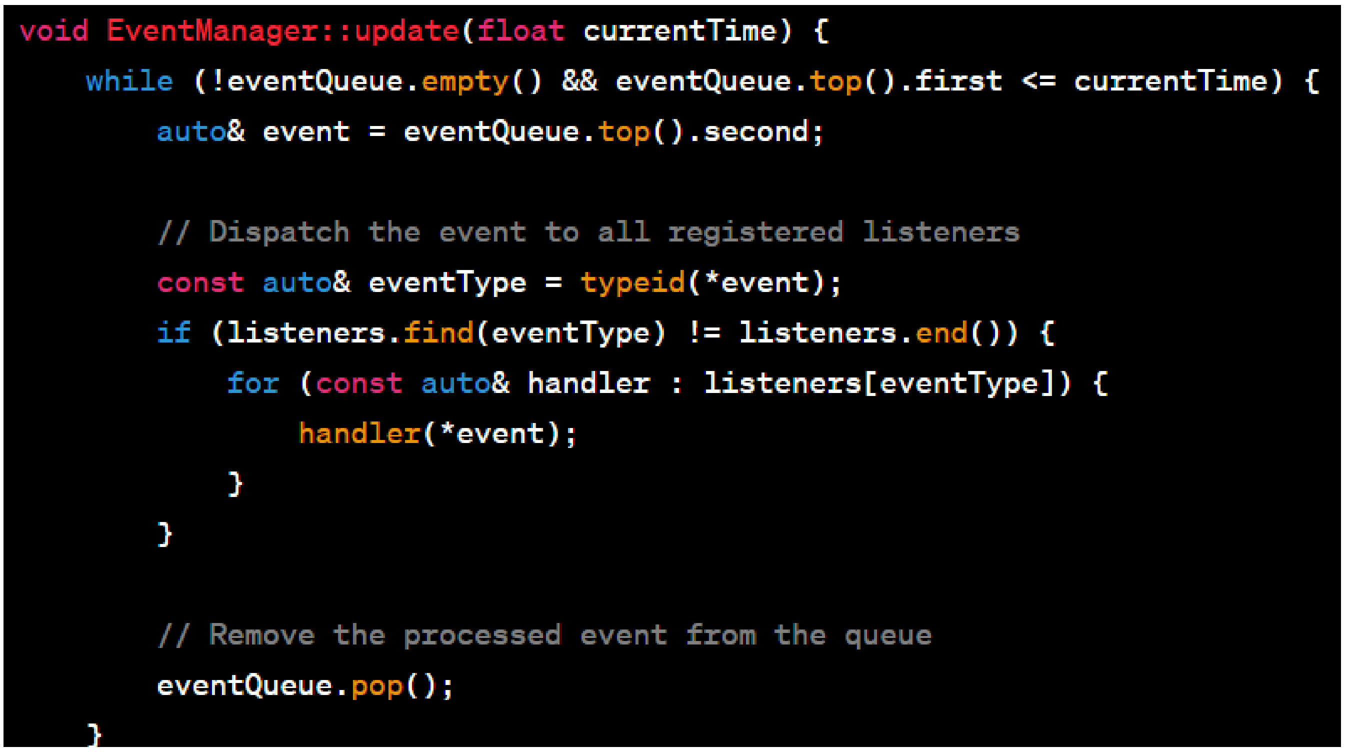
3. Event Raising
Integrating a Timeline class added a temporal dimension to events, an essential element for orchestrating engaging gameplay experiences. The ability to timestamp events and manage their order in the timeline not only facilitated chronological precision but also laid the groundwork for advanced features like replays. Events were raised through the Event Manager, where each event received a timestamp from the timeline:



```
CharacterCollisionEvent collisionEvent;
eventManager.raiseEvent(collisionEvent, timeline.getCurrentTimestamp());
```

4. Event handling
Event handling was achieved through the Event Manager, which dispatched events to registered systems or game objects. The timeline played a crucial role in determining when events should be handled, aligning with their timestamps.
The extensible nature of the event handling system facilitated the addition of new event types without requiring substantial modifications to existing code.

1.



```cpp
void EventManager::update(float currentTime) {
    while (!eventQueue.empty() && eventQueue.top().first <= currentTime) {
        auto& event = eventQueue.top().second;

        // Dispatch the event to all registered listeners
        const auto& eventType = typeid(*event);
        if (listeners.find(eventType) != listeners.end()) {
            for (const auto& handler : listeners[eventType]) {
                handler(*event);
            }
        }

        // Remove the processed event from the queue
        eventQueue.pop();
    }
```

2.

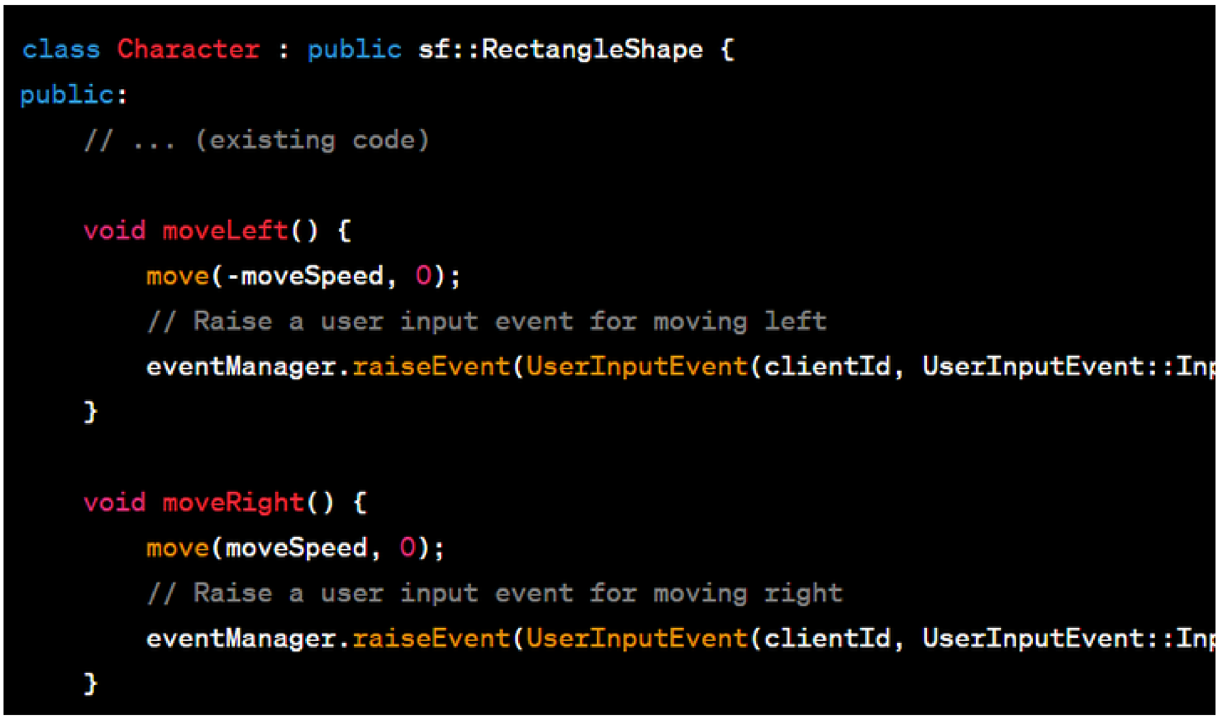
1. Server-Centric Approach:
Event Registration:

Event registration occurs on the server. Clients send requests to register their interest in specific event types.
Server maintains a registry of event listeners for each event type.
Event Queuing:

Events can be raised by both clients and the server.
When a client raises an event, it sends the event data to the server.
The server enqueues the event in its event queue with a timestamp.
Event Handling:

The server regularly checks its event queue.
If an event's timestamp is reached, the server dispatches the event to the appropriate listeners.
Clients' states are updated based on the results of event handling on the server.
Network Interaction:

Events raised by clients are sent to the server for processing.
The server sends the results of event handling (if any) back to the clients.

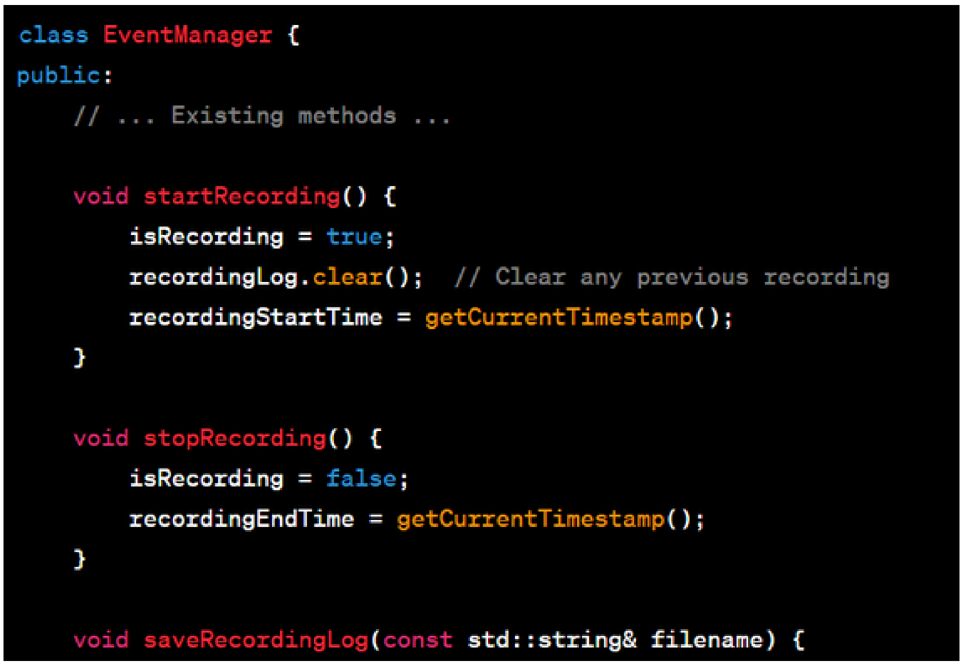


```
class Character : public sf::RectangleShape {
public:
    // ... (existing code)

    void moveLeft() {
        move(-moveSpeed, 0);
        // Raise a user input event for moving left
        eventManager.raiseEvent(UserInputEvent(clientId, UserInputEvent::Inp
    }

    void moveRight() {
        move(moveSpeed, 0);
        // Raise a user input event for moving right
        eventManager.raiseEvent(UserInputEvent(clientId, UserInputEvent::Inp
    }
```

3.

Replay System Design:

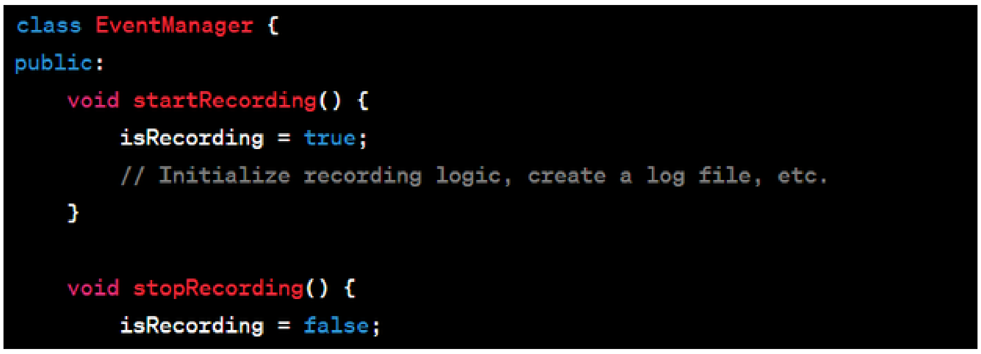
1. Event Types for Recording and Playback:
To initiate and conclude replay recording, we introduce two distinct event types - StartReplayEvent and StopReplayEvent. These events encapsulate the commands to begin and end recording, triggering the system to capture and store relevant events during the recording window.



```cpp
class EventManager {
public:
    // ... Existing methods ...

    void startRecording() {
        isRecording = true;
        recordingLog.clear();  // Clear any previous recording
        recordingStartTime = getCurrentTimestamp();
    }

    void stopRecording() {
        isRecording = false;
        recordingEndTime = getCurrentTimestamp();
    }

    void saveRecordingLog(const std::string& filename) {
```

2. Recording Mechanism in Event Manager:

The EventManager is extended to include methods for managing the recording state. During the recording phase, events are captured and stored in a log.

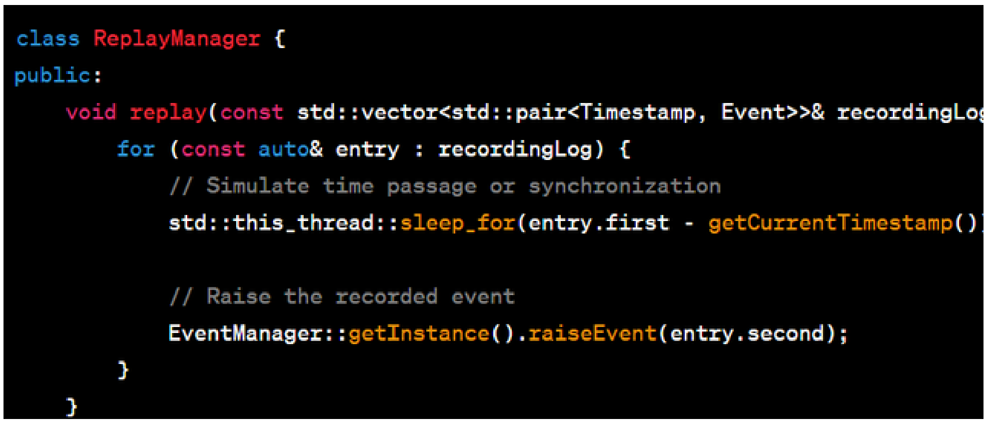


```cpp
class EventManager {
public:
    void startRecording() {
        isRecording = true;
        // Initialize recording logic, create a log file, etc.
    }

    void stopRecording() {
        isRecording = false;
```

3.

3. Replaying Events:

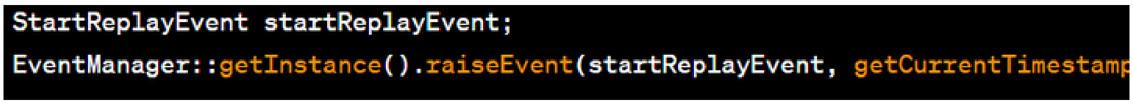
When it's time to replay, events are read from the recording log and raised at the corresponding timestamps. The ReplayManager orchestrates this process.



```
class ReplayManager {
public:
    void replay(const std::vector<std::pair<Timestamp, Event>>& recordingLo
        for (const auto& entry : recordingLog) {
            // Simulate time passage or synchronization
            std::this_thread::sleep_for(entry.first - getCurrentTimestamp()

            // Raise the recorded event
            EventManager::getInstance().raiseEvent(entry.second);
        }
    }
```

4. Integration with Gameplay:

During normal gameplay, events are raised as usual. The replay system seamlessly integrates with the existing event system.



```
StartReplayEvent startReplayEvent;
EventManager::getInstance().raiseEvent(startReplayEvent, getCurrentTimestam
```

In conclusion, every decision made in crafting this event-driven game engine was an investment in the future. The versatility of event representation, dynamic registration, temporal precision, networked gameplay, and replayability not only catered to the current assignment requirements but set the stage for future expansions. The engine, like a well-composed symphony, stands ready for new movements and harmonies, ready to adapt and evolve with the changing dynamics of game development.