CSCI 522 – Game Engine - Fall 2024 - HW 1

USC ID: 1154164561
NAME: Srija Madarapu
EMAIL: madarapu@usc.edu

Explain how the camera gets moved (on PC with keyboard):

a) Which events are used?

Events are used to capture and process user inputs for camera control. When a Key pressed on the controller(keyboard or Mouse) a event is generated. We have different queues for different type of events. InputEventQueue and GeneralEventQueue. As all the camera events are control events they have higher priority and are added to the InputEventQueue.

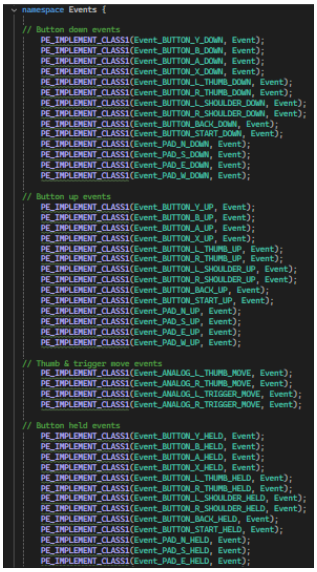
- **Game Events:**
    - Event_FLY_CAMERA: This event likely triggers camera movement based on the player's input, such as moving the camera forward, backward, or in other directions.
    - Event_ROTATE_CAMERA: This event handles camera rotation, allowing the camera to turn or pitch based on user input.

```
namespace PE {
namespace Events {

    PE_IMPLEMENT_CLASS1(Event_FLY_CAMERA, Event);
    PE_IMPLEMENT_CLASS1(Event_ROTATE_CAMERA, Event);
};
};
```

- **Keyboard Events:**

  All the controls for the event are defined in PE->Events->StandardControllerEvents.h.



b) Where do events originate?

Events originate from the keyboard input. When a key is pressed, the following occurs:

```
void Component::handleEvent(Event *pEvt)
{
    if (!m_enabled)
        return;
    // we have just received this event
    // need to push the distributor stack
    // and set myself as last distributor

    Handle cachedPrevDistributor = pEvt->m_prevDistributor;
    pEvt->m_prevDistributor = pEvt->m_lastDistributor;
    pEvt->m_lastDistributor = m_hMyself;

    distributeEvtToQueue(pEvt);
```

1. **Key Press Detection**: The operating system detects the key press and generates an appropriate event.
2. **Event Creation**: An event object representing the key press is created (e.g., Event_KEY_A_HELD).
3. **Event Queue**: The event is added to an event queue or event manager. This queue stores events that need to be processed.

c) How are events passed from one component to another?

1. **Event Queue Manager**: The EventQueueManager retrieves events from the event queue.
2. **Event Dispatching**: The EventQueueManager dispatches events to registered components using global registry. This is often done by calling specific functions or methods that handle the event.
3. **Component Handling**: Components such as the Camera Manager listen for and respond to specific events. For instance, when a Event_KEY_W_HELD is detected, the Camera Manager might update the camera's position to move forward.

```
// Constructor ----------------------------------------------------
EventQueueManager::EventQueueManager(PE::GameContext &context, PE::Me
{
    Handle dh = Handle("EVENT_QUEUE", sizeof(EventQueue));
    generalEvtQueue = new(dh) EventQueue();
    m_map.add("general", dh);

    Handle ih = Handle("EVENT_QUEUE", sizeof(EventQueue));
    inputEvtQueue = new(ih) EventQueue();
    m_map.add("input", ih);
}
```

d) Which components are involved?

Inputs: The Input component is designed to be flexible and supports various input devices depending on the platform. It initializes platform-specific input handlers and ensures they are properly added to the component.

```
void Input::addDefaultComponents()
{
    Component::addDefaultComponents();

#if APIABSTRACTION_D3D9 | APIABSTRACTION_D3D11
    Handle xin("DX9_XINPUT", sizeof(DX9_XInput));
    DX9_XInput *pxin = new(xin) DX9_XInput(*m_pContext, m_arena, xin);
    pxin->addDefaultComponents();
    addComponent(xin);
#elif PE_PLAT_IS_PS3 || PE_PLAT_IS_PSVITA
    Handle xin("PS3_PADINPUT", sizeof(PS3_PadInput));
    PS3_PadInput *pxin = new(xin) PS3_PadInput(*m_pContext, m_arena, xin);
    pxin->addDefaultComponents();
    addComponent(xin);
#endif

#if APIABSTRACTION_D3D9 | APIABSTRACTION_D3D11 | APIABSTRACTION_GLPC

    Handle kmin("DX9_XINPUT", sizeof(DX9_KeyboardMouse));
    DX9_KeyboardMouse *pkmin = new(kmin) DX9_KeyboardMouse(*m_pContext, m_arena, kmin);
    pkmin->addDefaultComponents();
    addComponent(kmin);
```

GameControls: Converting raw input events into game-specific actions, allowing for debugging and testing of camera controls via keyboard, controller, or touch input.

```
void DefaultGameControls::do_UPDATE(Events::Event *pEvt)
{

    // Process input events (controller buttons, triggers...)

    Handle iqh = Events::EventQueueManager::Instance()->getEventQueueHandle("input");

    // Process input event -> game event conversion

    while (!iqh.getObject<Events::EventQueue>()->empty())
    {
        Events::Event *pInputEvt = iqh.getObject<Events::EventQueue>()->getFront();

        m_frameTime = ((Event_UPDATE*)(pEvt))->m_frameTime;

        // Have DefaultGameControls translate the input event to GameEvents

        handleKeyboardDebugInputEvents(pInputEvt);

        handleControllerDebugInputEvents(pInputEvt);

        handleIOSDebugInputEvents(pInputEvt);
```

Event Creation: When a Key pressed on the controller(keyboard or Mouse) a event is generated. We have different queues for different type of events.

EventQueue: Events are created in response to user input or other system actions. For instance, when a key is held down, an Event_KEY_A_HELD is created.

EventDispatch: Events are added to the appropriate queue (general or input) using the EventQueueManager class. This separation ensures that different types of events are processed in a structured manner.

GlobalRegistry: The Register function sets up Lua bindings for various components and events within the PrimeEngine framework

```
namespace PE {

bool setLuaMetaDataOnly = 0;

void Register(PE::Components::LuaEnvironment *pLuaEnv, PE::GlobalRegistry *pRegistry)
{
    pLuaEnv->StartRootRegistrationTable();
    // start root
    {
        pLuaEnv->StartRegistrationTable("PE");
        // start root.PE
        {
            pLuaEnv->StartRegistrationTable("Components");
            // start root.PE.Components
```

Camera Manager: The CameraManager class manages various types of cameras in the game environment. It provides functionality to set, select, and retrieve camera Instances.

CameraOps(EventHandlingCode): The camera component updates its position and/or orientation based on the data from the Event_FLY_CAMERA event.

Render: After updating the camera's position, the new camera view is used to render the scene from the updated perspective.

Loop(ClientGame): The game loop continues to process input, update game state, and render the scene each frame based on the EventQueue.

```
while(m_runGame)
{
    runGameFrame();
} // while (runGame) -- game loop
```