Advanced Game Programming Topics

Project 1 Report

Simão Henriques Bercial 2221924

Daniel Gonçalves 2221927

Leiria, December of 2024

Index

Engine

JetEngine.cpp

The engine's main file handles initializing the engine, shutting the engine down, returning the delta time, sending information to the input manager to handle the controller inputs and returning that information to the game, getting the input manager and getting the renderer.

The initialize and shutdown functions are done like the ones done in class but divided into different functions, one creates a window and initializes SDL in order to have video and joystick functionalities and the other shuts down the renderer, destroys the current window and quits SDL.

```
JetEngine::JetEngine() : window(nullptr), prevTime(0), currentTime(0), deltaTime(0.0f) {}

JetEngine::~JetEngine() {
    Shutdown();
}

bool JetEngine::Initialize(const std::string& windowTitle, int width, int height) {
    if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK) != 0) {
        std::cerr << "SDL_Init Error: " << SDL_GetError() << std::endl;
        return false;
    }

    window = SDL_CreateWindow(windowTitle.c_str(), SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, width, height, SDL_WINDOW_SHOWN);
    if (!window) {
        std::cerr << "SDL_CreateWindow Error: " << SDL_GetError() << std::endl;
        return false;
    }

    if (!renderer.Initialize(window)) {
        return false;
    }
    return true;
}

void JetEngine::Shutdown() {
    renderer.Shutdown();
    if (window) {
        SDL_DestroyWindow(window);
    }
    SDL_Quit();
}

float JetEngine::GetDeltaTime() {
    prevTime = currentTime;
    currentTime = SDL_GetTicks();
    deltaTime = (currentTime - prevTime) / 1000.0f;
    return deltaTime;
}

bool JetEngine::ProcessInput(bool& isRunning) {
    return inputManager.ProcessInput(isRunning);
}

Renderer* JetEngine::GetRenderer() {
    return &renderer;
}

InputManager* JetEngine::GetInputManager() {
    return &inputManager;
}
```

Renderer.cpp

The engine's renderer has functions to initialize it, shut it down, load a texture, clear the renderer's queue, render the texture sent by the user onto the render target, present the loaded rendering and a function to get the renderer.

The initializer, such as the one for the engine, works like the one developed in class, creating a renderer with hardware acceleration and vsync. The shutdown function destroys the renderer.

The loadTexture function loads a BMP file onto a SDL Surface, sets the color key in order to make the background of any BMP file transparent, creates a texture from the created surface then frees the surface for further use. It then returns the texture if it isn't null.

The clear, render, present and GetSDLRenderer functions, clear the current render target, render the texture, present the texture and return a copy of the renderer respectively.

```cpp
Renderer::Renderer() : renderer(nullptr) {}

Renderer::~Renderer() {
    Shutdown();
}

bool Renderer::Initialize(SDL_Window* window) {
    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);
    if (!renderer) {
        std::cerr << "SDL_CreateRenderer Error: " << SDL_GetError() << std::endl;
        return false;
    }
    return true;
}

void Renderer::Shutdown() {
    if (renderer) {
        SDL_DestroyRenderer(renderer);
    }
}

SDL_Texture* Renderer::LoadTexture(const std::string& filePath) {
    SDL_Surface* surface = SDL_LoadBMP(filePath.c_str());
    if (!surface) {
        std::cerr << "SDL_LoadBMP Error: " << SDL_GetError() << std::endl;
        return nullptr;
    }

    SDL_SetColorKey(surface, SDL_TRUE, SDL_MapRGB(surface->format, 255, 0, 255));
    SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
    SDL_FreeSurface(surface);

    if (!texture) {
        std::cerr << "SDL_CreateTextureFromSurface Error: " << SDL_GetError() << std::endl;
    }

    return texture;
}

void Renderer::Clear() {
    SDL_RenderClear(renderer);
}

void Renderer::Render(SDL_Texture* texture, const SDL_Rect* srcRect, const SDL_Rect* dstRect) {
    SDL_RenderCopy(renderer, texture, srcRect, dstRect);
}

void Renderer::Present() {
    SDL_RenderPresent(renderer);
}

SDL_Renderer* Renderer::GetSDLRenderer() const {
    return renderer;
}
```

InputManager.cpp

The input manager handles controls by the player. The constructor accepts a gamepad and a deadzone value to avoid stick drift. The destructor closes the SDL Game Controller to disconnect the gamepad upon closing the game.

The process input checks if the loop is running and if so, gathers the number of joysticks connected, and if there isn't a gamepad already created, it sets the first gamepad connected as the one controlling the game.

GetKeyState, IsGamePadConnected and IsButtonPressed return the key pressed on the keyboard, if there's a gamepad connected and which key was pressed on the gamepad respectively.

GetAxis checks if there's a gamepad connected, and if so, uses SDL_GameControllerGetAxis dividing the axis value by 32767 because the value ranges from –32767 to 32767, so we get a value ranging from –1 to 1 to use in movement.

```cpp
#include "InputManager.h"
#include <iostream>

InputManager::InputManager() : gamepad(nullptr), deadZone(0.2f) {}

InputManager::~InputManager() {
    if (gamepad) {
        SDL_GameControllerClose(gamepad);
        std::cout << "Gamepad disconnected." << std::endl;
    }
}

bool InputManager::ProcessInput(bool& isRunning) {
    while (SDL_PollEvent(&event)) {
        if (event.type == SDL_QUIT) {
            isRunning = false;
        }
    }


    int numJoysticks = SDL_NumJoysticks();

    if (gamepad == nullptr && numJoysticks > 0) {
        if (SDL_IsGameController(0)) {
            gamepad = SDL_GameControllerOpen(0);
            if (gamepad) {
                std::cout << "Gamepad connected: " << SDL_GameControllerName(gamepad) << std::endl;
            }
            else {
                std::cerr << "SDL_GameControllerOpen Error: " << SDL_GetError() << std::endl;
            }
        }
    }

    return isRunning;
}

const Uint8* InputManager::GetKeyState() const {
    return SDL_GetKeyboardState(nullptr);
}

bool InputManager::IsGamepadConnected() const {
    return gamepad != nullptr;
}

bool InputManager::IsButtonPressed(int button) const {
    if (gamepad) {
        return SDL_GameControllerGetButton(gamepad, static_cast<SDL_GameControllerButton>(button)) == 1;
    }
    return false;
}

float InputManager::GetAxis(int axis) const {
    if (gamepad) {
        float axisValue = SDL_GameControllerGetAxis(gamepad, static_cast<SDL_GameControllerAxis>(axis)) / 32767.0f;
        if (std::abs(axisValue) < deadZone) {
            axisValue = 0.0f;
        }
        return axisValue;
    }
    return 0.0f;
}
```

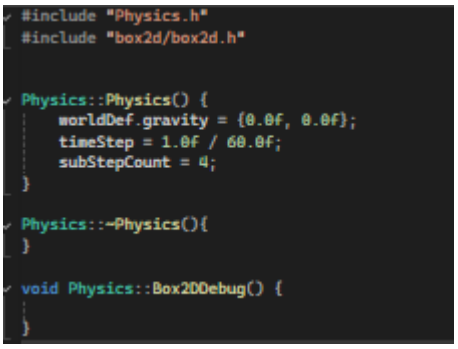
GameObject.h

The game object doesn't have a .cpp since it's a base class for the objects such as the player and the enemy. It has functions that can be overridden by the child objects like Update, Render and GetBoundingBox.

```
#ifndef GAMEOBJECT_H
#define GAMEOBJECT_H

#include "SDL.h"
#include "Renderer.h";

class GameObject {
public:
    virtual ~GameObject() = default;

    virtual void Update(float deltaTime, int textureWidth, int textureHeight, int frameWidth, int frameHeigth) = 0;
    virtual void Render() = 0;
    virtual SDL_Rect GetBoundingBox() const = 0;

protected:
    SDL_Renderer* renderer = nullptr;
};

#endif
```

Physics.cpp

The physics file only defines values like the gravity, which in our case is 0.0f, the timestep, which is defined as 1/60 since we are running the game at 60 frames per second and the substep count for 240hz handling of physics.



```cpp
#include "Physics.h"
#include "box2d/box2d.h"


Physics::Physics() {
    worldDef.gravity = {0.0f, 0.0f};
    timeStep = 1.0f / 60.0f;
    subStepCount = 4;
}

Physics::~Physics(){
}

void Physics::Box2DDebug() {

}
```

Game

A lot of functions and calculations that should have been done in the engine were done in the game due to inexperience and not knowing where to put what until after it was done, such as the player movement having an entire HandleInput function that we made that handles the movement, getting a reference to the engine's input manager and getting information from there instead of doing it all on the input manager.

This function takes the movement from player's keyboard or gamepad and then normalizes it so the issue of moving left or upwards faster than right or downwards is resolved. Instead of this, the player should only get values from the engine and apply them to the player.

The player's animation is also done inside the player instead of on the engine because we were having trouble with engine communication and how to handle moving left or right and changing the animation accordingly.

We also were unsure if we should have done the parallax effect inside the engine or the game, and ended up doing it inside the game. The parallax effect is done by taking a background texture, a parallax texture, the scrolling speed, the width and the height of the images and then changing the y of the parallax texture on the update, and if it reaches the screenHeight it will remove the screen height from the y. The rendering first renders the background layer and then the parallax layer above it.

Objectives Achieved

Spaceship moving with movement error fixed: Achieved

Spaceship and enemy animation: Achieved (In game instead of engine)

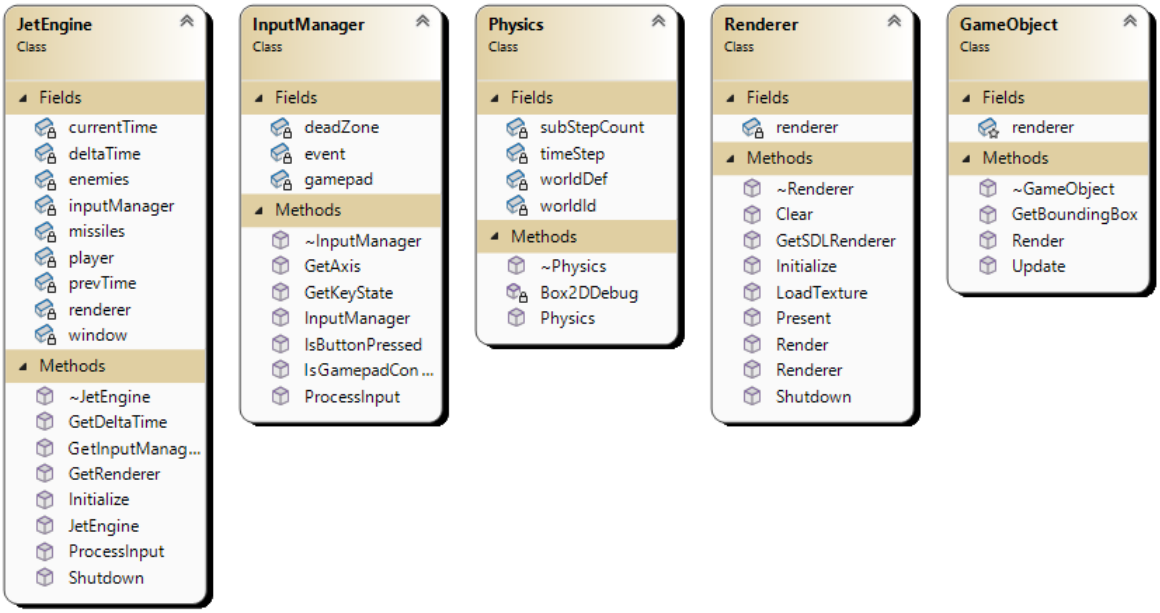
Parallax effect: Achieved (In game instead of engine)

Enemy behaviour: Achieved

Missile firing and missile animation: Achieved

Missile and enemy collisions: Not achieved

Class Diagram



Webgraphy

SDL2/FrontPage. (n.d.). SDL Wiki. https://wiki.libsdl.org/SDL2/FrontPage

Mike Shah. (2023, January 4). *[EP. 48] Parallax Scrolling in Simple DirectMedia Layer | Introduction to SDL2* [Video]. YouTube. https://www.youtube.com/watch?v=VHMjJ9v--EM

Game programming patterns. (n.d.). https://gameprogrammingpatterns.com/

Lazy Foo' Productions - Beginning Game Programming v2.0. (n.d.). https://lazyfoo.net/tutorials/SDL/

Cppreference.com. (n.d.). https://en.cppreference.com/w/

Technologies, U. (n.d.). *Unity - Manual: Execution Order of event functions*. https://docs.unity3d.com/560/Documentation/Manual/ExecutionOrder.html