Advanced Game Programming Topics

Project 2 Report

Simão Henriques Bercial 2221924

Daniel Gonçalves 2221927

Gonçalo Marques 2221930

Leiria, February of 2025

Index

Advanced Game Programming Topics

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, getting the renderer and getting the physics engine.

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.

```cpp
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;
}

Physics* JetEngine::GetPhysicsEngine()
{
    return &physics;
}
```

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, as well as a Debug method for collisions.

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);
}

void Renderer::Debug(SDL_Rect rect) {
    SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
    SDL_RenderDrawRect(renderer, &rect);
    SDL_RenderFillRect(renderer, &rect);
};


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
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, enemies, missiles, etc. It has functions that can be overridden by the child objects like Update, Render, GetBoundingBox, CreateRigidBody and SetPhysicsEngine.

It also has a number of protected variables that are inherited and used by the sub-classes.

```
class GameObject {
public:
    virtual ~GameObject() {};

    virtual void Update(float deltaTime) = 0;
    virtual void Render(Renderer* renderer) = 0;
    virtual SDL_Rect GetBoundingBox() const = 0;
    virtual void CreateRigidBody(Physics* physics) = 0;
    void SetPhysicsEngine(Physics* physics);

protected:
    SDL_Texture* texture;
    SDL_Rect position;
    SDL_Rect spriteRectObject;
    int frameWidth;
    int frameHeight;
    float frameTime;
    int currentFrame;
    int numFrames;
    float moveSpeed;
    float movementMagnitude;
    float posX;
    float posY;
    float velocityX;
    float velocityY;
    int textureWidth;
    int textureHeight;

    b2ShapeId rigidbodyId;
    b2Transform rigidbodyTransform;
};
```

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.

We also have functions to create Static Bodies and Dynamic Bodies, as well as to get the transform properties of a given Rigid Body. It is also the class where the collisions are being dealt with, so we also have a method that Detects Collisions, although its not really working. We have a Debug method for the collisions as well.

```cpp
Physics::Physics() {
    worldDef.gravity = {0.0f, 0.0f};
    timeStep = 1.0f / 60.0f;
    subStepCount = 4;
    b2SensorEvents sensorEvents = b2World_GetSensorEvents(worldId);
}

Physics::~Physics(){
    b2DestroyWorld(worldId);
}

b2WorldId Physics::GetWorld() {
    return worldId;
}

b2ShapeId Physics::CreateStaticBody(float posX, float posY, bool isSensor, float width, float height)
{
    b2BodyDef bodyDef = b2DefaultBodyDef();
    bodyDef.type = b2_staticBody;
    bodyDef.position = { posX, posY };

    b2BodyId bodyId = b2CreateBody(worldId, &bodyDef);

    b2Polygon dynamicBox = b2MakeBox(width / 2, height / 2);

    b2ShapeDef shapeDef = b2DefaultShapeDef();
    shapeDef.isSensor = isSensor;
    b2ShapeId shapeId = b2CreatePolygonShape(bodyId, &shapeDef, &dynamicBox);

    return shapeId;
}

b2ShapeId Physics::CreateDynamicBody(float posX, float posY, bool isSensor, float width, float height)
{
    b2BodyDef bodyDef = b2DefaultBodyDef();
    bodyDef.type = b2_dynamicBody;
    bodyDef.position = { posX, posY };

    b2BodyId bodyId = b2CreateBody(worldId, &bodyDef);

    b2Polygon dynamicBox = b2MakeBox(width / 2, height / 2);

    b2ShapeDef shapeDef = b2DefaultShapeDef();
    shapeDef.isSensor = isSensor;
    b2ShapeId shapeId = b2CreatePolygonShape(bodyId, &shapeDef, &dynamicBox);

    return shapeId;
}

b2Transform Physics::GetRigidBodyTransform(b2ShapeId shapeId)
{
    b2BodyId bodyId = b2Shape_GetBody(shapeId);
    return b2Body_GetTransform(bodyId);
}

void Physics::UpdatePhysics()
{
    b2World_Step(worldId, timeStep, subStepCount);

}

void Physics::DetectCollision(b2SensorBeginTouchEvent collision)
{
    b2ShapeId* sensorShape = &collision.sensorShapeId;
    b2ShapeId* visitorShape = &collision.visitorShapeId;

    std::cout << sensorShape << " detected " << visitorShape << std::endl;
}

void Physics::Debug(b2Transform* bodyTransform, b2ShapeId shapeId)
{
    b2Vec2 position = bodyTransform->p;
    b2Rot angle = bodyTransform->q;

    b2Polygon bodyPolygon = b2Shape_GetPolygon(shapeId);
    b2Vec2 vertex1 = bodyPolygon.vertices[0];
    b2Vec2 vertex2 = bodyPolygon.vertices[1];
    b2Vec2 vertex3 = bodyPolygon.vertices[2];
    b2Vec2 vertex4 = bodyPolygon.vertices[3];

    float width = vertex2.x - vertex1.x;
    float height = vertex3.y - vertex1.y;

    // Convert Box2D's position from meters to pixels (assuming 1 meter = 30 pixels)
    int posX = static_cast<int>(position.x );
    int posY = static_cast<int>(position.y );

    SDL_Rect rect = {
                posX - static_cast<int>(width),  // Adjust X for the center of the body
                posY - static_cast<int>(height), // Adjust Y for the center of the body
                static_cast<int>(width),         // Width scaled to pixels
                static_cast<int>(height)         // Height scaled to pixels
    };

    renderer->Debug(rect);
}

void Physics::SetRenderer(Renderer* renderer) // FOR DEBUGGING PURPOSES ONLY
{
    this->renderer = renderer;
}
```

Game

We improved the game logic from the last the project, although its still not project. After the fact, we realized that we could have had another, more modular approach, like Unity does, where an object is created and, if needed, calls and attaches other properties, like, transformations, collisions, sounds, etc. This way, a lot of the "passing around references" would be avoided. We probably should've had a Level class, where the level logic would be created.

We improved the parts that should belong on the engine and the ones that should belong to the game itself, so everything is ordered and optimized better.

Objectives Achieved

Spaceship

We achieved most of the Spaceship related things, except for the collisions.

Missiles

We achieved everything related to the Missiles shot from the Spaceship. Most of the logic for the missile types is implemented, although we can only see it in game if we change the initializing values on the code, given that the Power Ups were not implemented in the game, so we have no way of incrementing the value that changes the missile types.

Loner

We achieved the behaviour of the Loner, except for the firing projectiles and the collisions.

Rusher

We achieved all the behaviour relating to the Rusher enemies, except for the collisions.

Drones

We achieved all the behaviour relating to the Drone enemies, except for the collisions. Also, their movement is too fast, we tried to figure out a way of making it slower, but given how we have the movement velocity implemented, they wouldn't move at all.

Asteroids

The asteroids were not implemented.

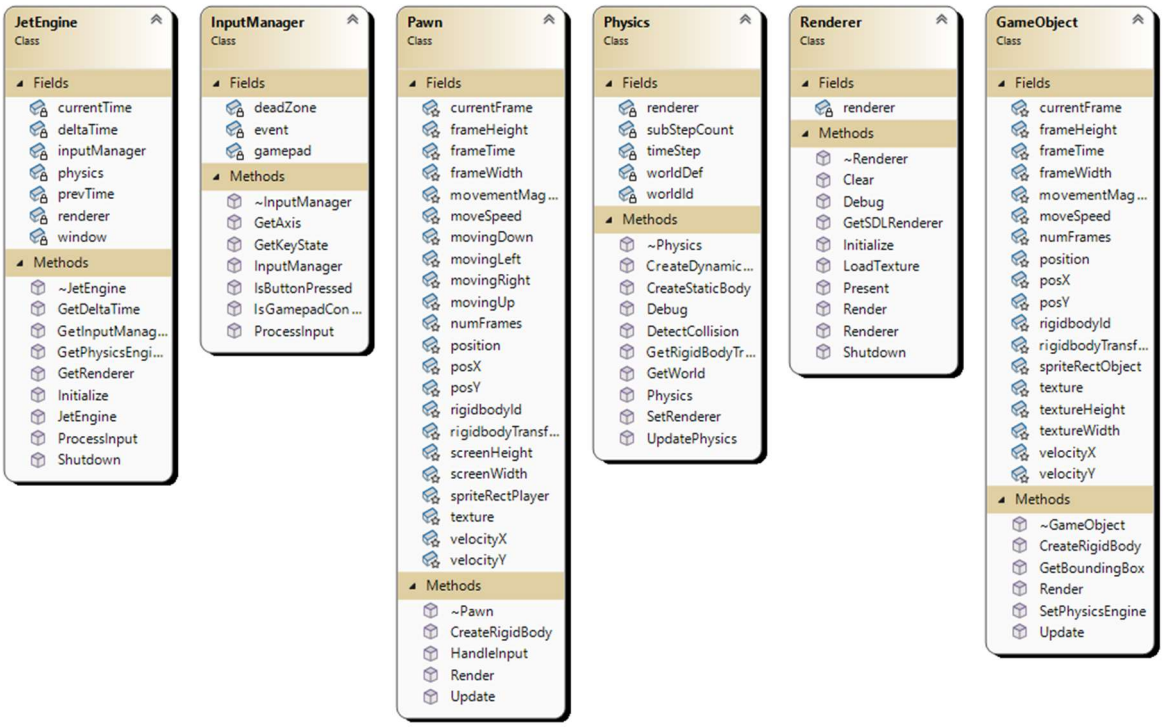
Companion

The companions were implemented, but, as we lack the Power Ups, we can only trigger them by pressing the E key for now.

Power Ups

The power Ups were not implemented.

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

Box2D: Overview. (n.d.). https://box2d.org/documentation/

Computer Graphics