POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Advanced Game Programming Topics / Computer Graphics

Project 2 Report

Simão Henriques Bercial 2221924

Daniel Gonçalves 2221927

Gonçalo Marques 2221930

Leiria, February of 2025

Index

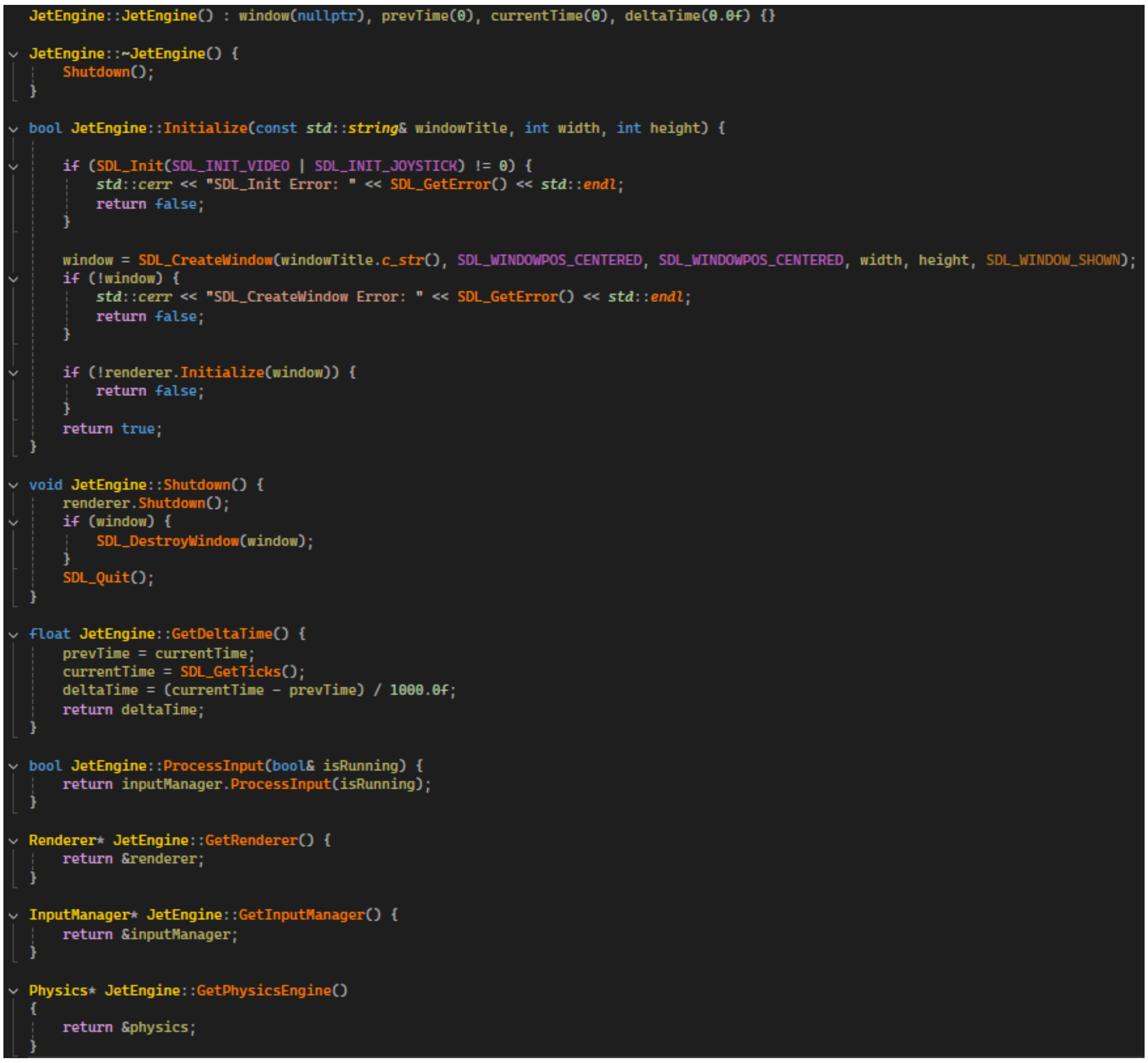
Advanced Game Programming Topics

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;
}
```

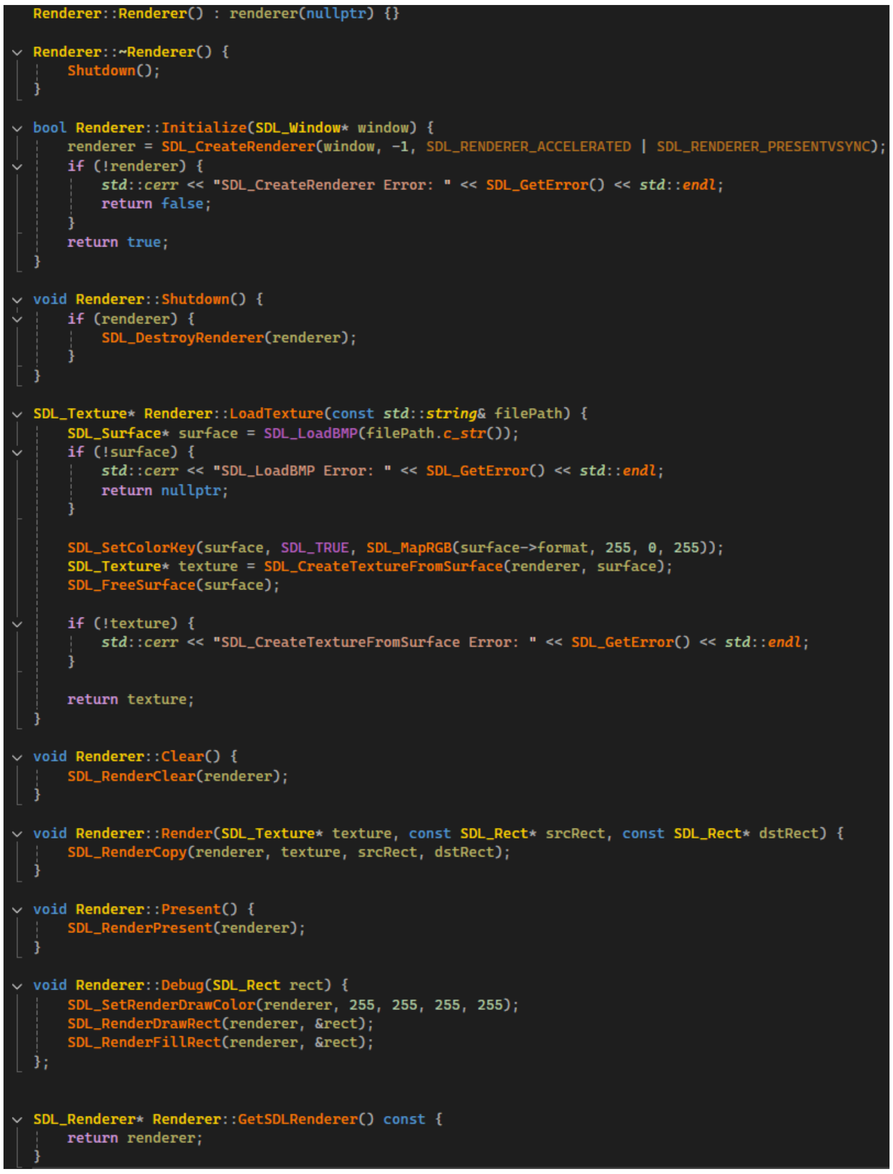
Renderer.cpp

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;
}
```

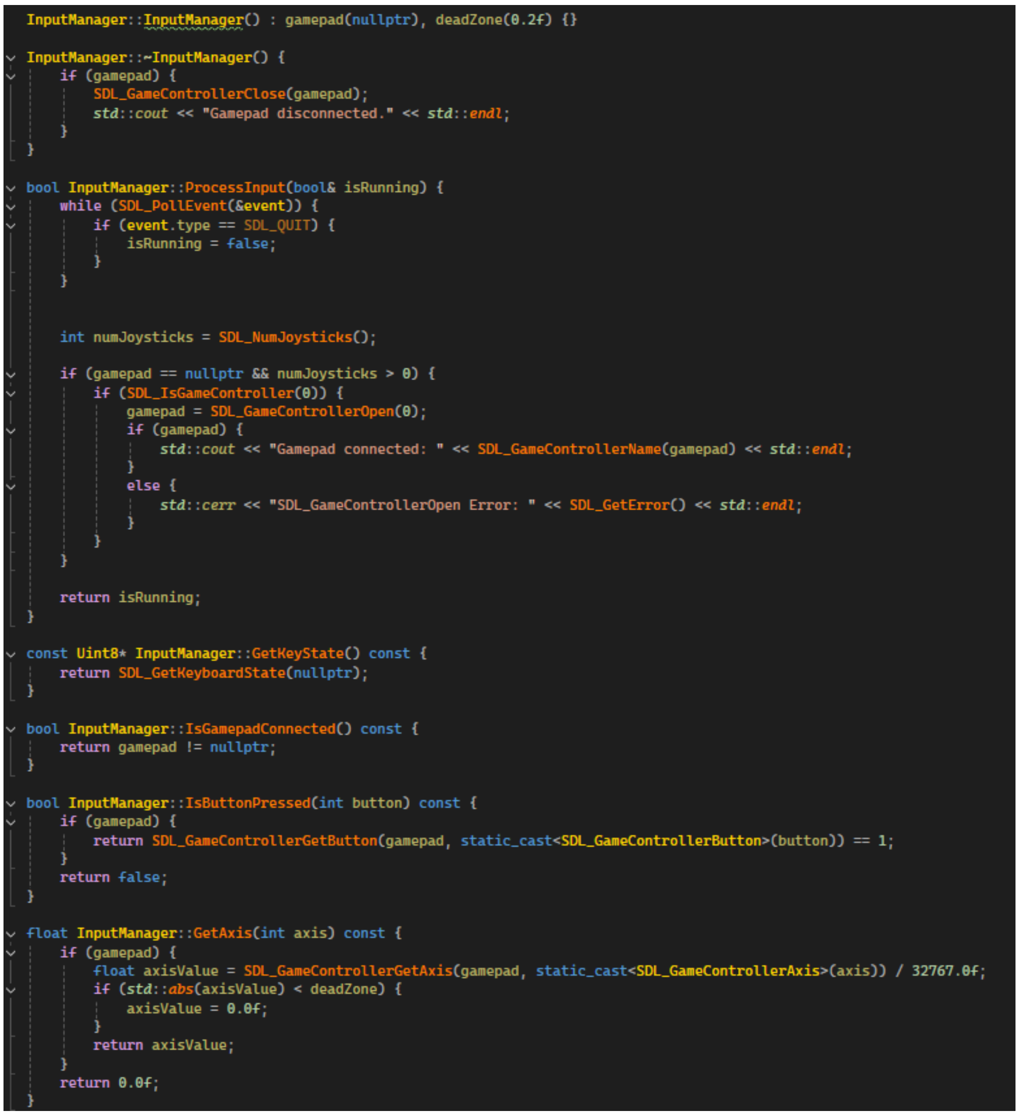
InputManager.cpp

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.

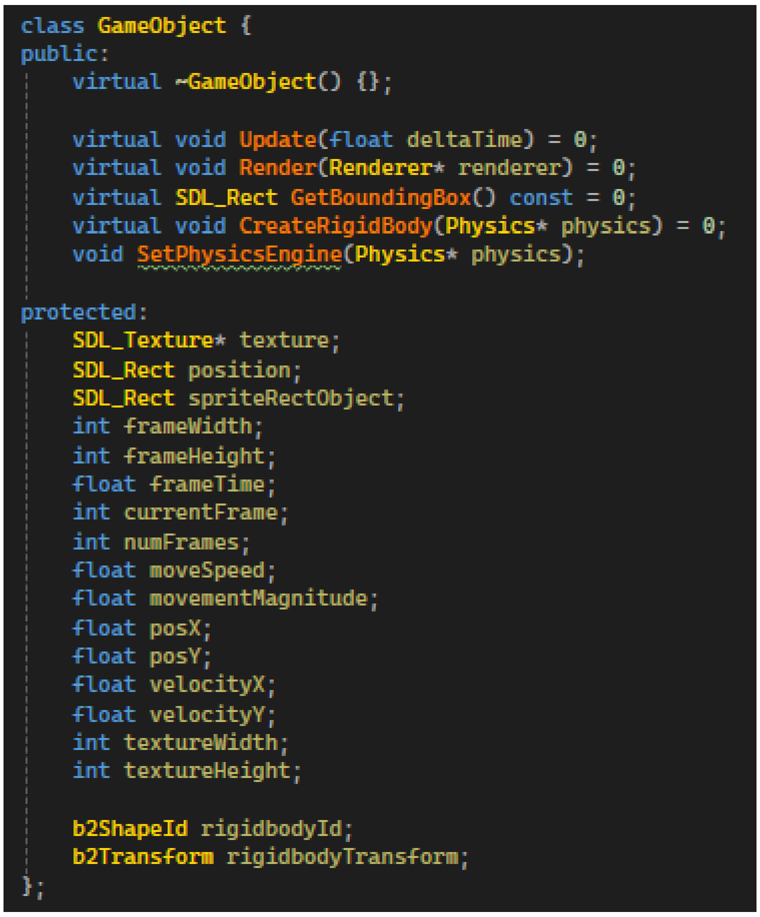


```cpp
InputManager::InputManager() : gamepad(nullptr), deadZone(0.2f) {}

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.

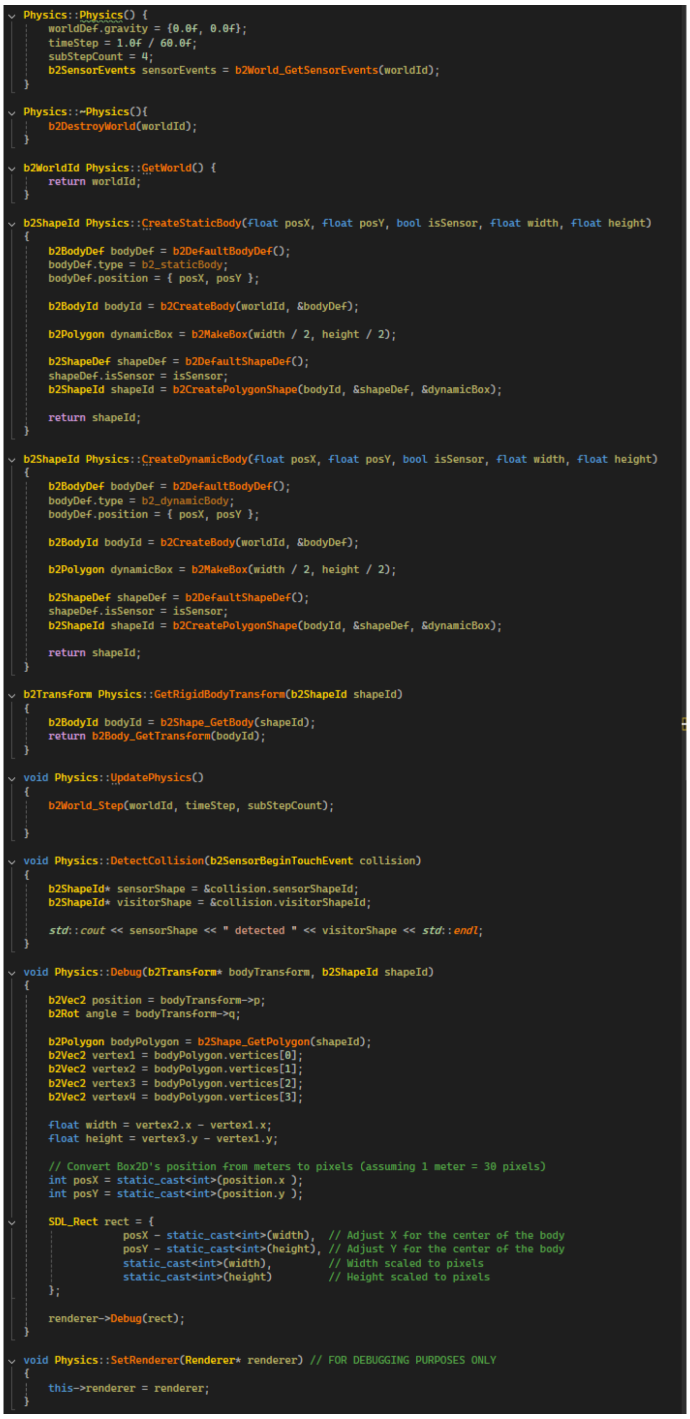


```
class GameObject {
public:
    virtual ~GameObject() {};

    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.

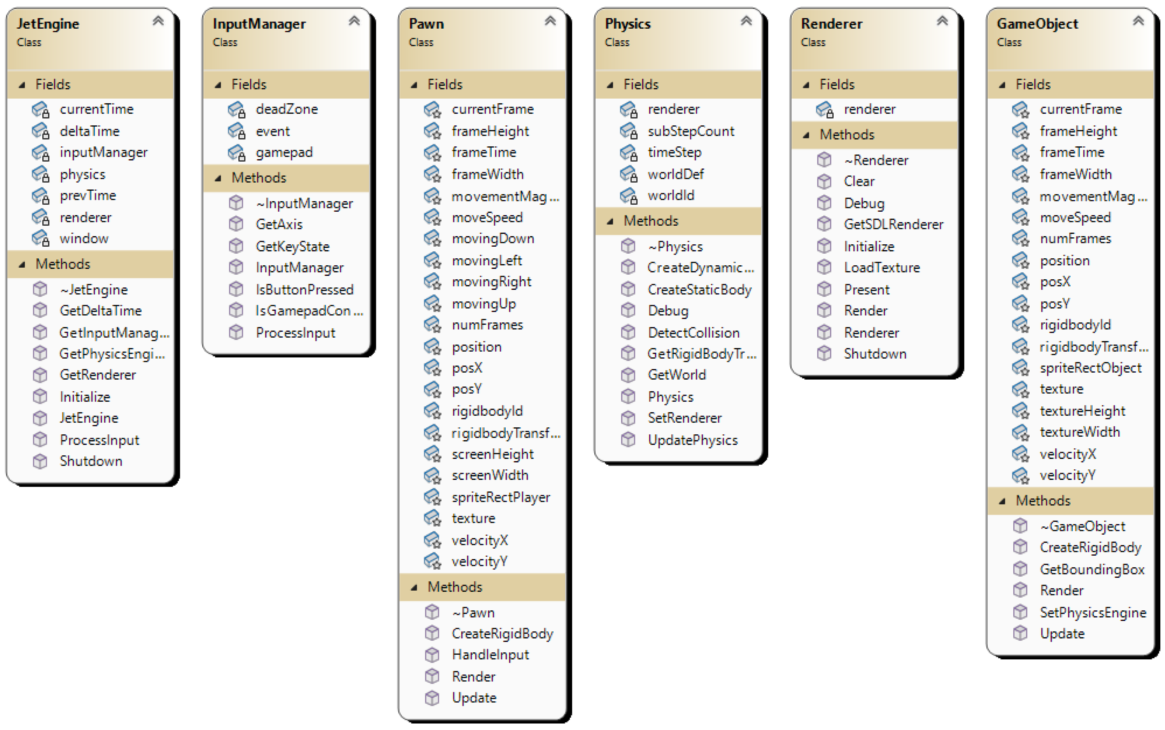
Companion

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/

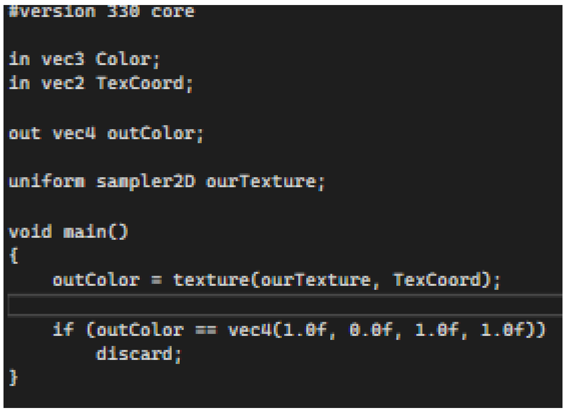
Technologies, U. (n.d.). *Unity - Manual: Execution Order of event functions*. https://docs.unity3d.com/560/Documentation/Manual/ExecutionOrder.html

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

Computer Graphics

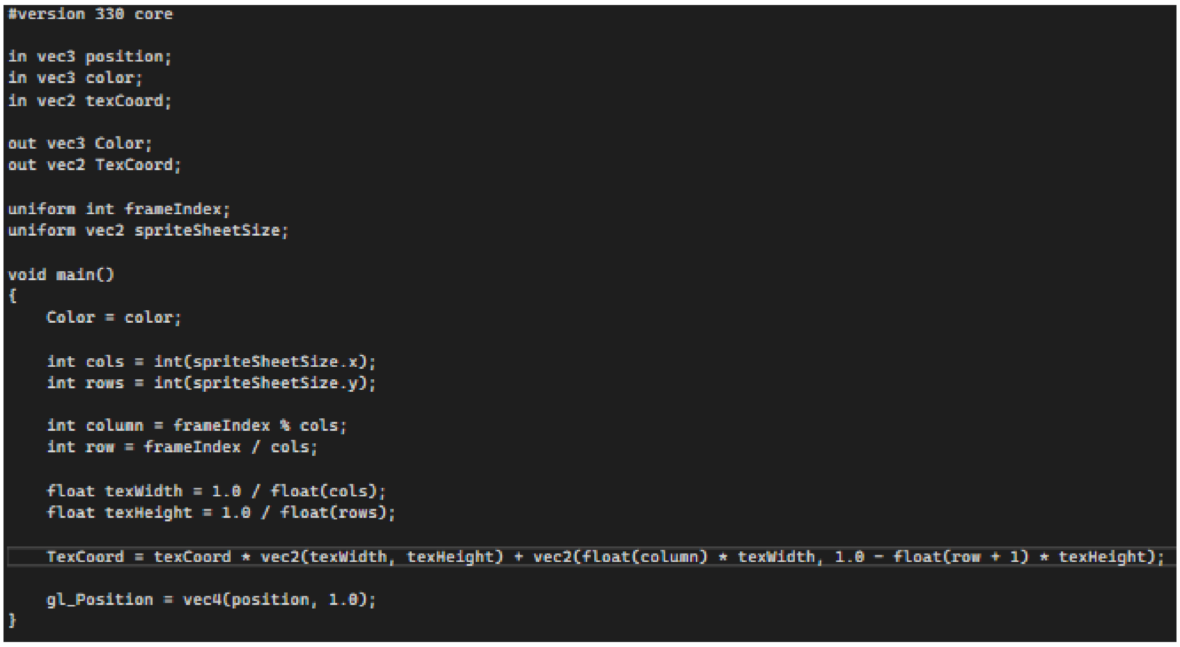
Shaders

The shaders used in the computer graphics part of the project implement a Fragment Shader similar to the one used in class, having the texture that is to be used as a uniform, the texture coordinates as an input and outputting the outColor as texture(ourTexture, TexCoord). Then, in order to apply chroma keying, if the outColor is magenta (the background for the BMP textures) the color is discarded.



```
#version 330 core

in vec3 Color;
in vec2 TexCoord;

out vec4 outColor;

uniform sampler2D ourTexture;

void main()
{
    outColor = texture(ourTexture, TexCoord);

    if (outColor == vec4(1.0f, 0.0f, 1.0f, 1.0f))
        discard;
}
```

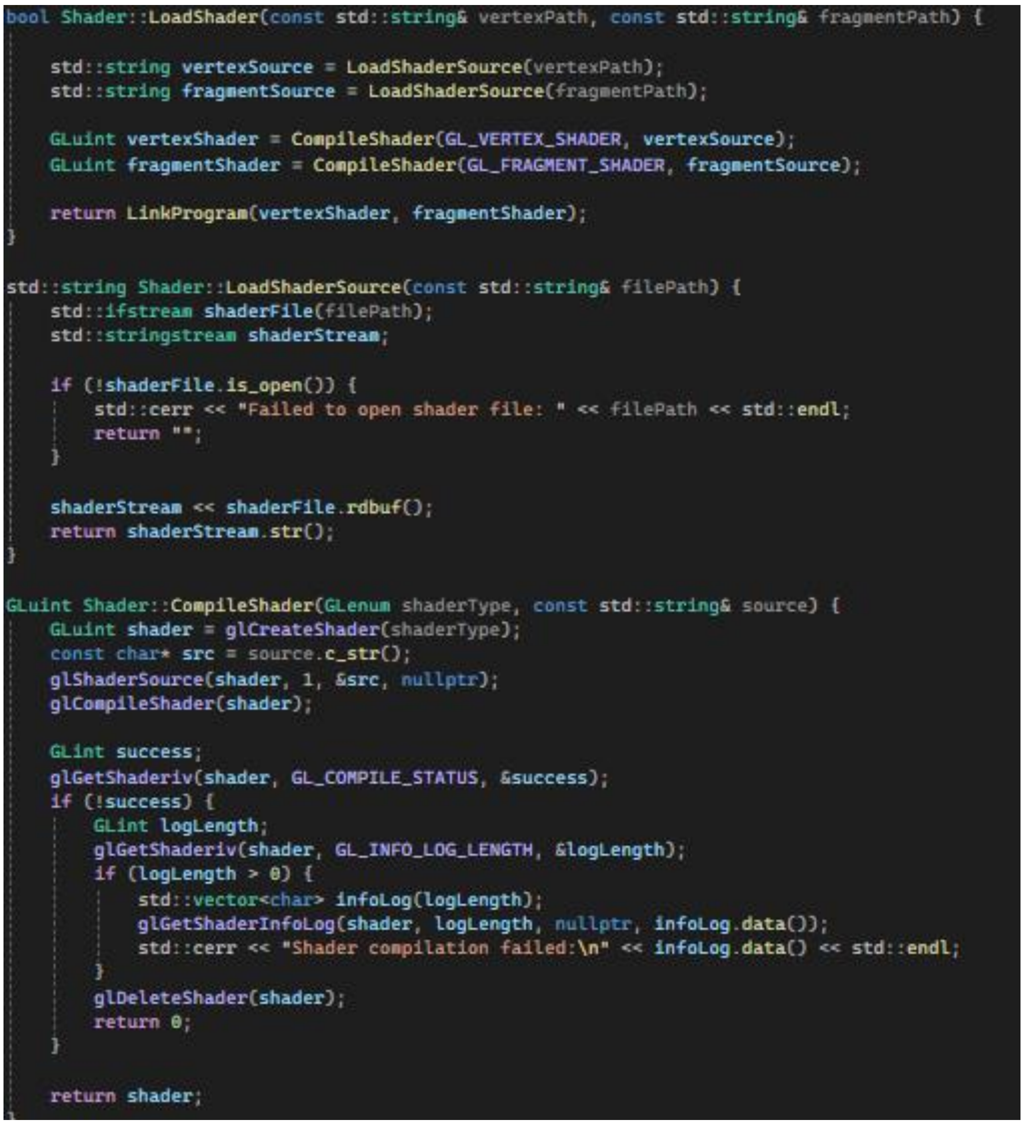
The vertex shader takes a frameIndex and a spriteSheetSize as inputs and iterates through the rows and columns until it finds the specific frame and sets the texture coordinates as the specific frame by getting the pixel values of the specific column and row and using that as the output coordinates.



```
#version 330 core

in vec3 position;
in vec3 color;
in vec2 texCoord;

out vec3 Color;
out vec2 TexCoord;

uniform int frameIndex;
uniform vec2 spriteSheetSize;

void main()
{
    Color = color;

    int cols = int(spriteSheetSize.x);
    int rows = int(spriteSheetSize.y);

    int column = frameIndex % cols;
    int row = frameIndex / cols;

    float texWidth = 1.0 / float(cols);
    float texHeight = 1.0 / float(rows);

    TexCoord = texCoord * vec2(texWidth, texHeight) + vec2(float(column) * texWidth, 1.0 - float(row + 1) * texHeight);

    gl_Position = vec4(position, 1.0);
}
```

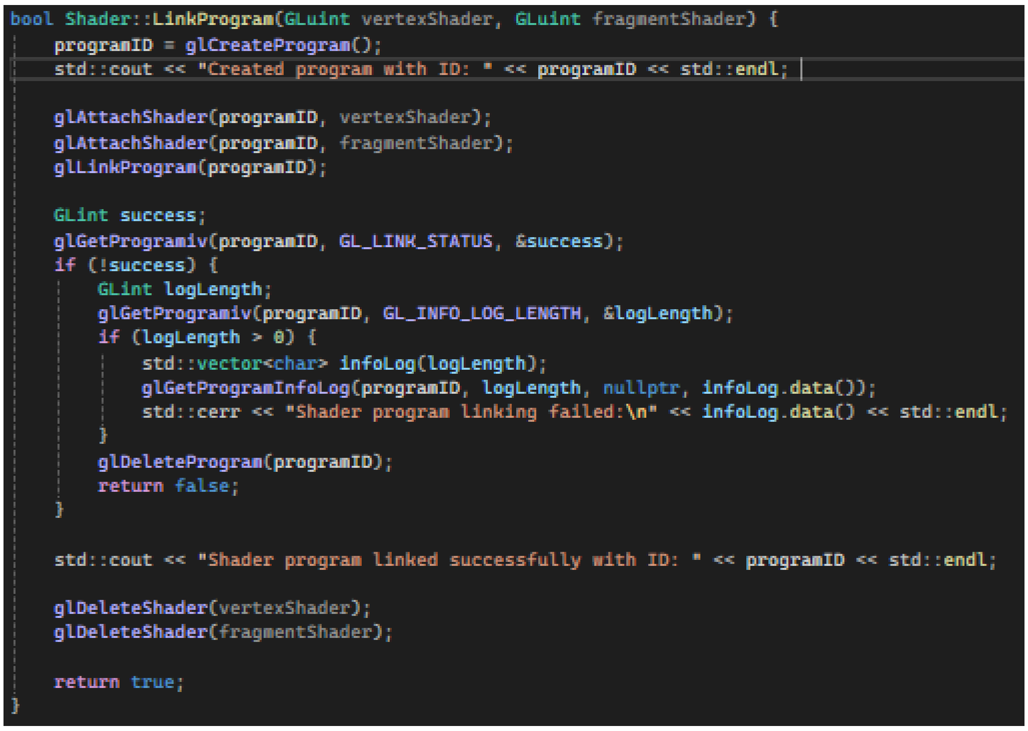
The Shader.cpp manages loading the shader, compiling it linking the program, using the shader, getting the location of a specific uniform, setting the Mat4 and returning the program ID.

Load Shader sets the vertex and fragment shader sources through a file path in the renderer by calling LoadShaderSource, which opens and reads the file and saves it to a shaderStream string, which is then returned as the value for vertexSource or fragmentSource. The loader then compiles both shaders and runs LinkProgram.



```
bool Shader::LoadShader(const std::string& vertexPath, const std::string& fragmentPath) {

    std::string vertexSource = LoadShaderSource(vertexPath);
    std::string fragmentSource = LoadShaderSource(fragmentPath);

    GLuint vertexShader = CompileShader(GL_VERTEX_SHADER, vertexSource);
    GLuint fragmentShader = CompileShader(GL_FRAGMENT_SHADER, fragmentSource);

    return LinkProgram(vertexShader, fragmentShader);
}

std::string Shader::LoadShaderSource(const std::string& filePath) {
    std::ifstream shaderFile(filePath);
    std::stringstream shaderStream;

    if (!shaderFile.is_open()) {
        std::cerr << "Failed to open shader file: " << filePath << std::endl;
        return "";
    }

    shaderStream << shaderFile.rdbuf();
    return shaderStream.str();
}

GLuint Shader::CompileShader(GLenum shaderType, const std::string& source) {
    GLuint shader = glCreateShader(shaderType);
    const char* src = source.c_str();
    glShaderSource(shader, 1, &src, nullptr);
    glCompileShader(shader);

    GLint success;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
    if (!success) {
        GLint logLength;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &logLength);
        if (logLength > 0) {
            std::vector<char> infoLog(logLength);
            glGetShaderInfoLog(shader, logLength, nullptr, infoLog.data());
            std::cerr << "Shader compilation failed:\n" << infoLog.data() << std::endl;
        }
        glDeleteShader(shader);
        return 0;
    }

    return shader;
```

LinkProgram then creates a shader program, attaches both shaders to it and links the program with the program ID. After debug to verify if the program was linked successfully, both shaders are deleted as they are no longer needed.

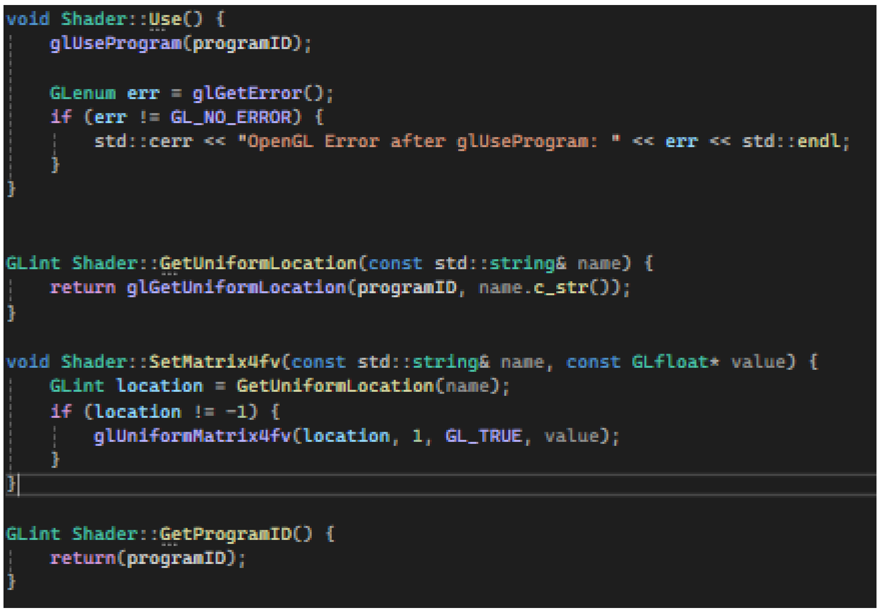


```cpp
bool Shader::LinkProgram(GLuint vertexShader, GLuint fragmentShader) {
    programID = glCreateProgram();
    std::cout << "Created program with ID: " << programID << std::endl;

    glAttachShader(programID, vertexShader);
    glAttachShader(programID, fragmentShader);
    glLinkProgram(programID);

    GLint success;
    glGetProgramiv(programID, GL_LINK_STATUS, &success);
    if (!success) {
        GLint logLength;
        glGetProgramiv(programID, GL_INFO_LOG_LENGTH, &logLength);
        if (logLength > 0) {
            std::vector<char> infoLog(logLength);
            glGetProgramInfoLog(programID, logLength, nullptr, infoLog.data());
            std::cerr << "Shader program linking failed:\n" << infoLog.data() << std::endl;
        }
        glDeleteProgram(programID);
        return false;
    }

    std::cout << "Shader program linked successfully with ID: " << programID << std::endl;

    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);

    return true;
}
```

The function Use() sets the program to be used as the current program by using its ID and then checks for any OpenGL errors. This line was added due to various problems with using the program as a lot of the times it was returning error 1821 (invalid value) because there were multiple shader programs created by accident.

The function GetUniformLocation returns the location of X uniform and this was done in order to set the Mat4 in SetMatrix4fv. This was also used for debugging as the location was returning as null a lot due to problems with Shader creation.

The function GetProgramID returns the ID of the current shader program.

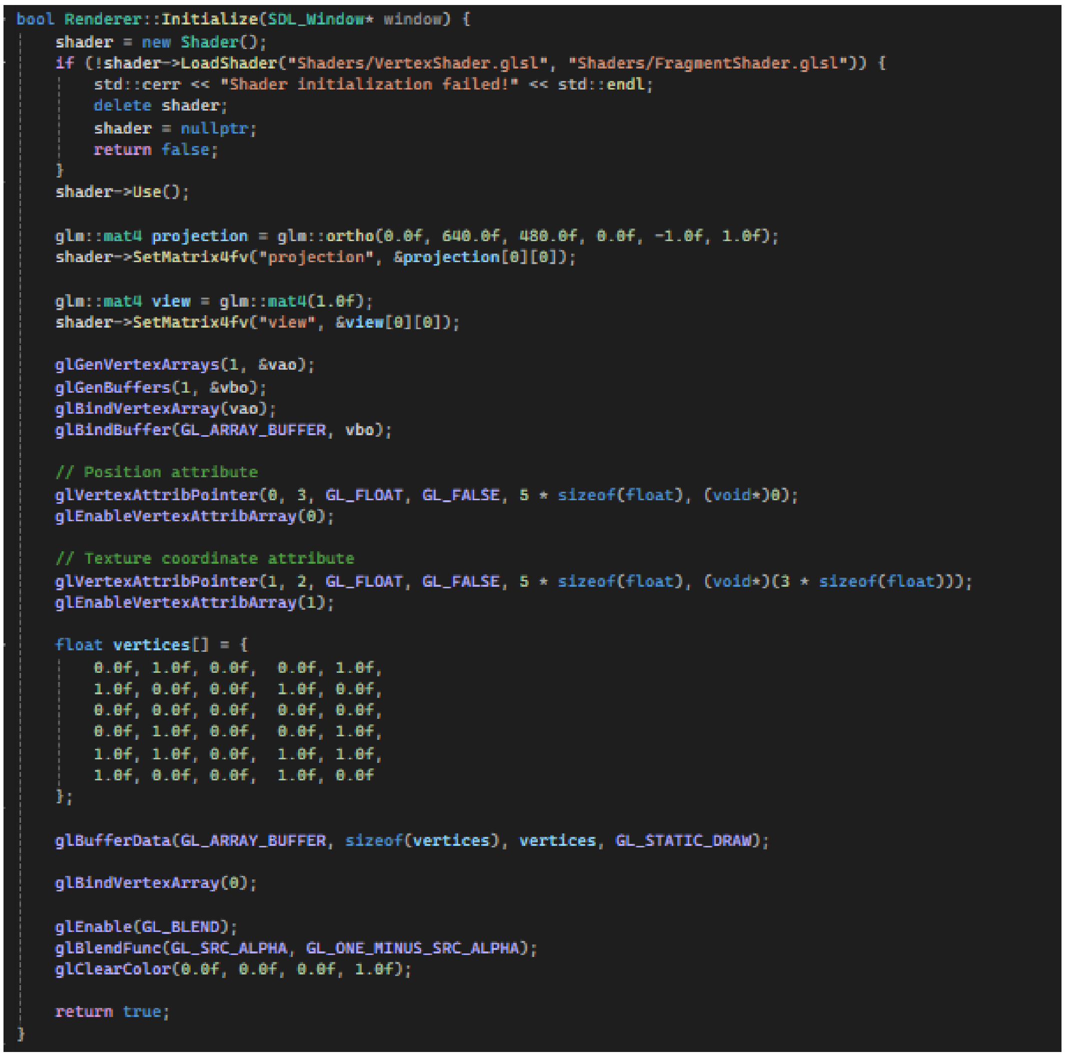


```cpp
void Shader::Use() {
    glUseProgram(programID);

    GLenum err = glGetError();
    if (err != GL_NO_ERROR) {
        std::cerr << "OpenGL Error after glUseProgram: " << err << std::endl;
    }
}


GLint Shader::GetUniformLocation(const std::string& name) {
    return glGetUniformLocation(programID, name.c_str());
}

void Shader::SetMatrix4fv(const std::string& name, const GLfloat* value) {
    GLint location = GetUniformLocation(name);
    if (location != -1) {
        glUniformMatrix4fv(location, 1, GL_TRUE, value);
    }
}

GLint Shader::GetProgramID() {
    return(programID);
}
```

Renderer.cpp

The Renderer is used to load and render textures on the screen. There were many problems with the renderer such as calling pointers that were deleted and initializing multiple shaders and GLAD which lead to a lot of issues. Because of this, the renderer is not fully functional as only a blue square is rendered on the screen. I could not pinpoint the issue due to lack of time but tried different debugging methods and all I managed to fix was the screen being completely black without rendering anything and the problem with accessing pointers that were already deleted.

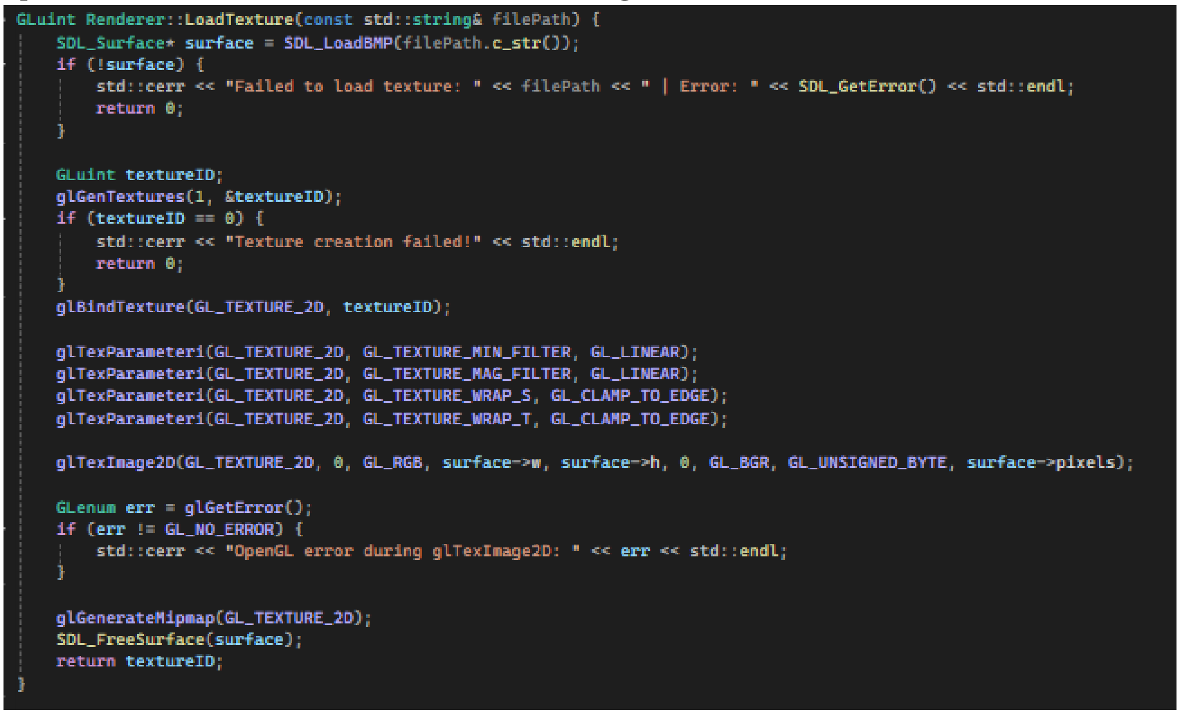
The Initialize function starts the renderer, by first creating a new shader, loading the glsl files and then using the shader. After this, it sets the projection and view matrix to a flat 2D using the window measurements. After setting the matrixes, the buffers are set up and then the Position and Texture coordinate attributes are set. It then sets up the vertices and rest of the buffer data.



```cpp
bool Renderer::Initialize(SDL_Window* window) {
    shader = new Shader();
    if (!shader->LoadShader("Shaders/VertexShader.glsl", "Shaders/FragmentShader.glsl")) {
        std::cerr << "Shader initialization failed!" << std::endl;
        delete shader;
        shader = nullptr;
        return false;
    }
    shader->Use();

    glm::mat4 projection = glm::ortho(0.0f, 640.0f, 480.0f, 0.0f, -1.0f, 1.0f);
    shader->SetMatrix4fv("projection", &projection[0][0]);

    glm::mat4 view = glm::mat4(1.0f);
    shader->SetMatrix4fv("view", &view[0][0]);

    glGenVertexArrays(1, &vao);
    glGenBuffers(1, &vbo);
    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);

    // Position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

    // Texture coordinate attribute
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);

    float vertices[] = {
        0.0f, 1.0f, 0.0f,  0.0f, 1.0f,
        1.0f, 0.0f, 0.0f,  1.0f, 0.0f,
        0.0f, 0.0f, 0.0f,  0.0f, 0.0f,
        0.0f, 1.0f, 0.0f,  0.0f, 1.0f,
        1.0f, 1.0f, 0.0f,  1.0f, 1.0f,
        1.0f, 0.0f, 0.0f,  1.0f, 0.0f
    };

    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    glBindVertexArray(0);

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    return true;
}
```

The shutdown function deletes the shader, sets it to a nullptr (this was necessary because it was what lead to the errors with accessing shaders that no longer existed) and then deletes the VAO and the VBO.

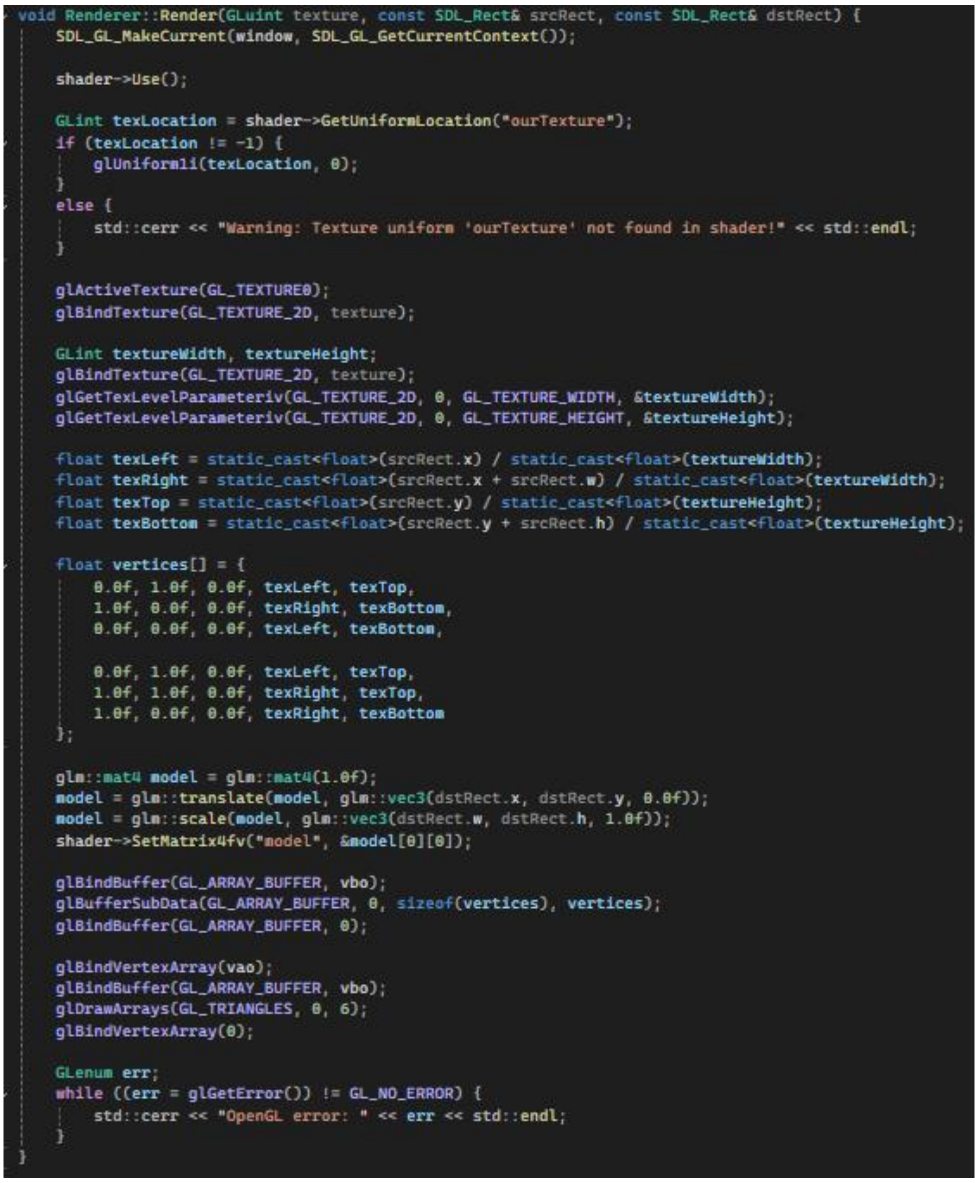
The LoadTexture function creates a surface and attaches the texture sent in the main to that surface. After that, it creates a texture using a texture ID and binds that ID. After setting the texture parameters, it generates the image using the surface's information. It then checks for OpenGL errors for debugging purposes then generates mipmaps for optimization and then frees the surface, returning the ID of the loaded texture.



```
GLuint Renderer::LoadTexture(const std::string& filePath) {
    SDL_Surface* surface = SDL_LoadBMP(filePath.c_str());
    if (!surface) {
        std::cerr << "Failed to load texture: " << filePath << " | Error: " << SDL_GetError() << std::endl;
        return 0;
    }

    GLuint textureID;
    glGenTextures(1, &textureID);
    if (textureID == 0) {
        std::cerr << "Texture creation failed!" << std::endl;
        return 0;
    }
    glBindTexture(GL_TEXTURE_2D, textureID);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, surface->w, surface->h, 0, GL_BGR, GL_UNSIGNED_BYTE, surface->pixels);

    GLenum err = glGetError();
    if (err != GL_NO_ERROR) {
        std::cerr << "OpenGL error during glTexImage2D: " << err << std::endl;
    }

    glGenerateMipmap(GL_TEXTURE_2D);
    SDL_FreeSurface(surface);
    return textureID;
}
```

The Render function went through a lot of changes and is what caused the most problems, as it was not able to render due to problems with shader creation, then problems with deleted pointers, and finally it is functional but I still cannot pinpoint why it is not rendering the textures correctly.

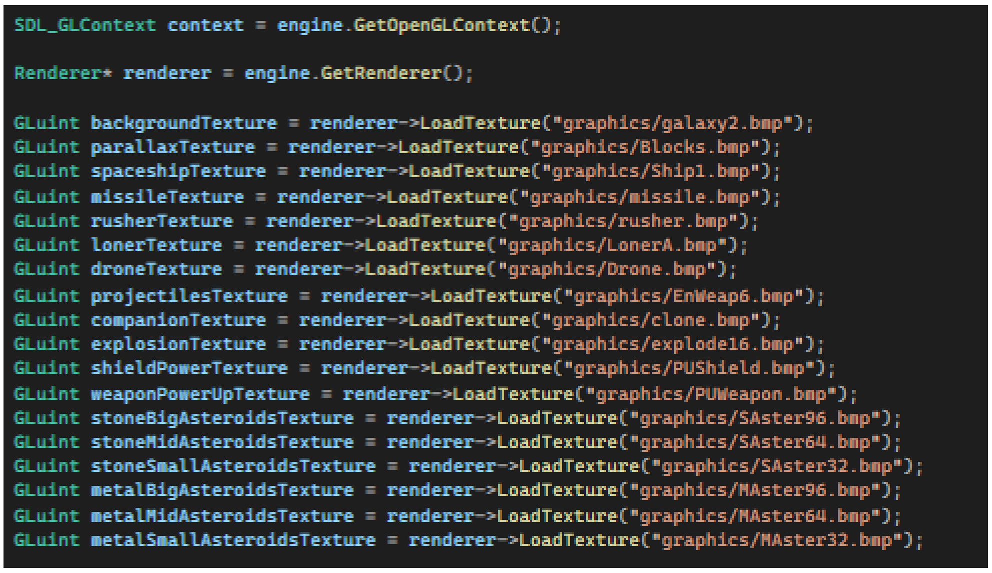
It starts by making the created context the one being currently used in order to avoid problems with multiple contexts being created accidentally. It then uses the Shader and sets the texture location as the uniform in the Fragment shader. After this, it sets the active texture and gets the width and the height of the texture. The sides of the texture are then set by dividing the x and y of the rect by the width and height to know which ones to use for the vertices, which are then set. The function then sets the model matrix and translates and scales it according to the texture size, then setting the Mat4 to the changed model matrix. After this, the vbo and vao are set and the arrays are drawn, or at least supposed to be drawn. I could not understand why the textures were not being appropriately rendered even after hours of debugging various functions.



```cpp
void Renderer::Render(GLuint texture, const SDL_Rect& srcRect, const SDL_Rect& dstRect) {
    SDL_GL_MakeCurrent(window, SDL_GL_GetCurrentContext());

    shader->Use();

    GLint texLocation = shader->GetUniformLocation("ourTexture");
    if (texLocation != -1) {
        glUniform1i(texLocation, 0);
    }
    else {
        std::cerr << "Warning: Texture uniform 'ourTexture' not found in shader!" << std::endl;
    }

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture);

    GLint textureWidth, textureHeight;
    glBindTexture(GL_TEXTURE_2D, texture);
    glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_WIDTH, &textureWidth);
    glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_HEIGHT, &textureHeight);

    float texLeft = static_cast<float>(srcRect.x) / static_cast<float>(textureWidth);
    float texRight = static_cast<float>(srcRect.x + srcRect.w) / static_cast<float>(textureWidth);
    float texTop = static_cast<float>(srcRect.y) / static_cast<float>(textureHeight);
    float texBottom = static_cast<float>(srcRect.y + srcRect.h) / static_cast<float>(textureHeight);

    float vertices[] = {
        0.0f, 1.0f, 0.0f, texLeft, texTop,
        1.0f, 0.0f, 0.0f, texRight, texBottom,
        0.0f, 0.0f, 0.0f, texLeft, texBottom,

        0.0f, 1.0f, 0.0f, texLeft, texTop,
        1.0f, 1.0f, 0.0f, texRight, texTop,
        1.0f, 0.0f, 0.0f, texRight, texBottom
    };

    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(dstRect.x, dstRect.y, 0.0f));
    model = glm::scale(model, glm::vec3(dstRect.w, dstRect.h, 1.0f));
    shader->SetMatrix4fv("model", &model[0][0]);

    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glBindVertexArray(vao);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glBindVertexArray(0);

    GLenum err;
    while ((err = glGetError()) != GL_NO_ERROR) {
        std::cerr << "OpenGL error: " << err << std::endl;
    }
}
```

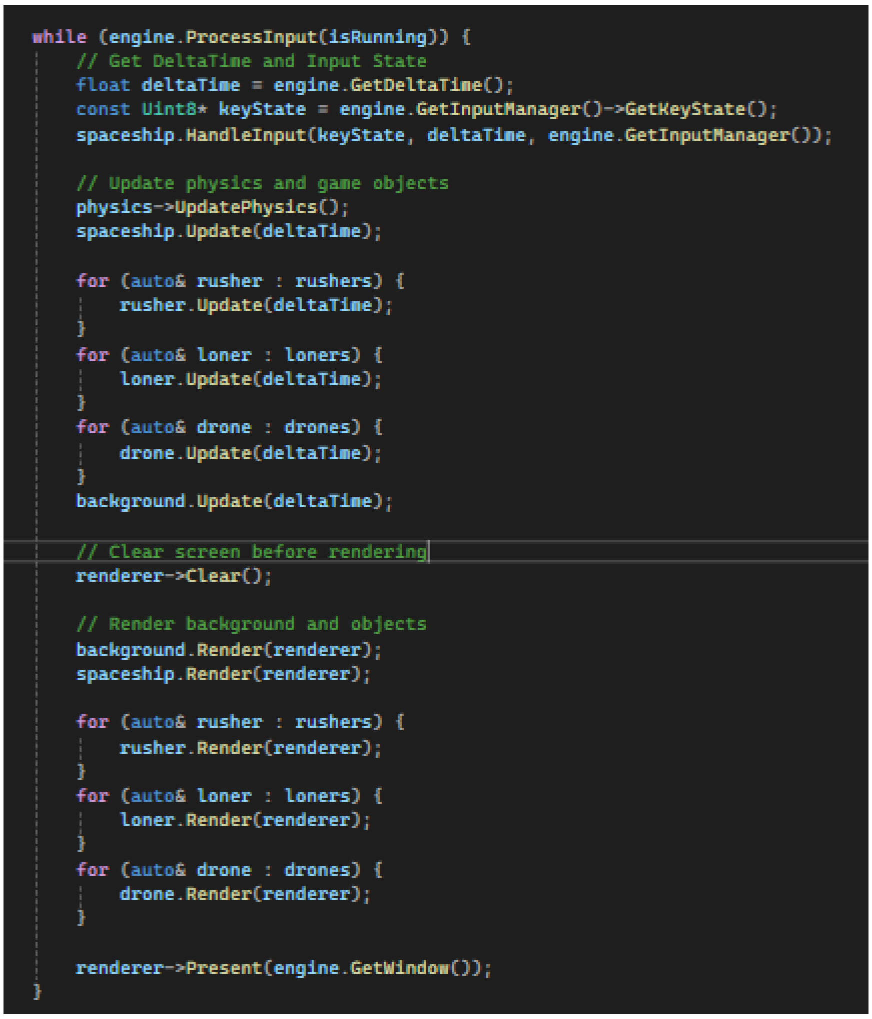
Xenon Clone Game.cpp

The main was changed to incorporate OpenGL, changing the start to create an OpenGL context, get the renderer and then load each texture.



```cpp
SDL_GLContext context = engine.GetOpenGLContext();

Renderer* renderer = engine.GetRenderer();

GLuint backgroundTexture = renderer->LoadTexture("graphics/galaxy2.bmp");
GLuint parallaxTexture = renderer->LoadTexture("graphics/Blocks.bmp");
GLuint spaceshipTexture = renderer->LoadTexture("graphics/Ship1.bmp");
GLuint missileTexture = renderer->LoadTexture("graphics/missile.bmp");
GLuint rusherTexture = renderer->LoadTexture("graphics/rusher.bmp");
GLuint lonerTexture = renderer->LoadTexture("graphics/LonerA.bmp");
GLuint droneTexture = renderer->LoadTexture("graphics/Drone.bmp");
GLuint projectilesTexture = renderer->LoadTexture("graphics/EnWeap6.bmp");
GLuint companionTexture = renderer->LoadTexture("graphics/clone.bmp");
GLuint explosionTexture = renderer->LoadTexture("graphics/explode16.bmp");
GLuint shieldPowerTexture = renderer->LoadTexture("graphics/PUShield.bmp");
GLuint weaponPowerUpTexture = renderer->LoadTexture("graphics/PUWeapon.bmp");
GLuint stoneBigAsteroidsTexture = renderer->LoadTexture("graphics/SAster96.bmp");
GLuint stoneMidAsteroidsTexture = renderer->LoadTexture("graphics/SAster64.bmp");
GLuint stoneSmallAsteroidsTexture = renderer->LoadTexture("graphics/SAster32.bmp");
GLuint metalBigAsteroidsTexture = renderer->LoadTexture("graphics/MAster96.bmp");
GLuint metalMidAsteroidsTexture = renderer->LoadTexture("graphics/MAster64.bmp");
GLuint metalSmallAsteroidsTexture = renderer->LoadTexture("graphics/MAster32.bmp");
```

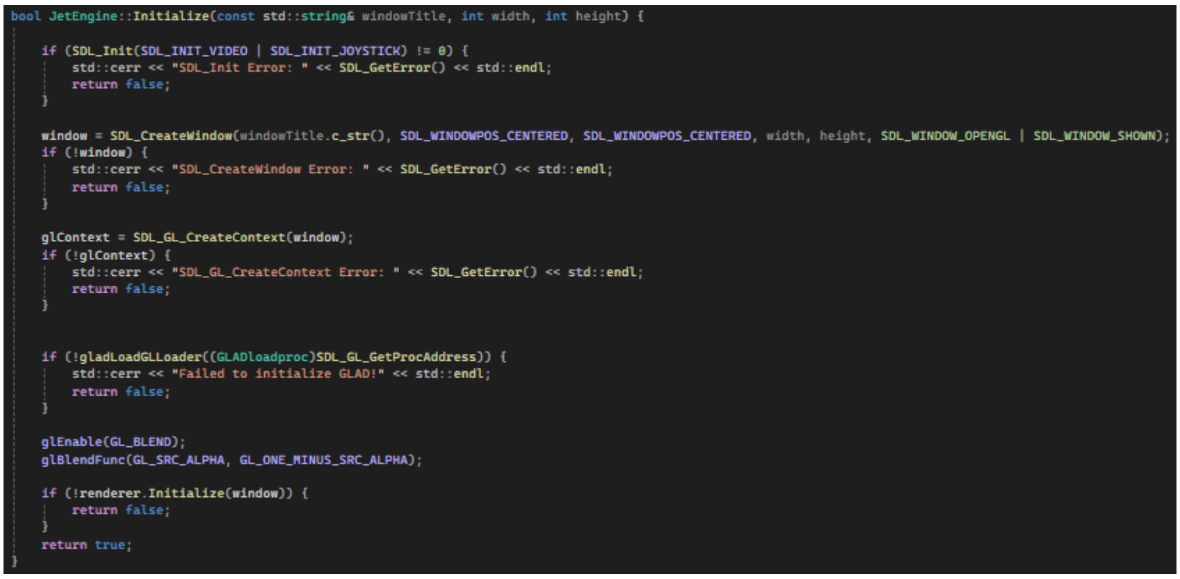
The main loop was kept intact as the rendering was done in the renderer class, just calling the render methods.



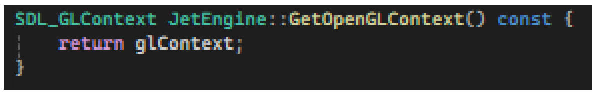
```cpp
while (engine.ProcessInput(isRunning)) {
    // Get DeltaTime and Input State
    float deltaTime = engine.GetDeltaTime();
    const Uint8* keyState = engine.GetInputManager()->GetKeyState();
    spaceship.HandleInput(keyState, deltaTime, engine.GetInputManager());

    // Update physics and game objects
    physics->UpdatePhysics();
    spaceship.Update(deltaTime);

    for (auto& rusher : rushers) {
        rusher.Update(deltaTime);
    }
    for (auto& loner : loners) {
        loner.Update(deltaTime);
    }
    for (auto& drone : drones) {
        drone.Update(deltaTime);
    }
    background.Update(deltaTime);

    // Clear screen before rendering
    renderer->Clear();

    // Render background and objects
    background.Render(renderer);
    spaceship.Render(renderer);

    for (auto& rusher : rushers) {
        rusher.Render(renderer);
    }
    for (auto& loner : loners) {
        loner.Render(renderer);
    }
    for (auto& drone : drones) {
        drone.Render(renderer);
    }

    renderer->Present(engine.GetWindow());
}
```

JetEngine.cpp

The engine class was altered to create a OpenGL context, load GLAD and set up the blending settings for OpenGL. A GetOpenGLContext function was added to return the current created Context.



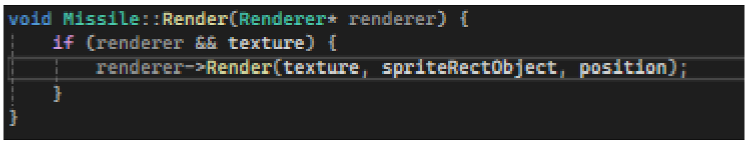
```cpp
bool JetEngine::Initialize(const std::string& windowTitle, int width, int height) {

    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_OPENGL | SDL_WINDOW_SHOWN);
    if (!window) {
        std::cerr << "SDL_CreateWindow Error: " << SDL_GetError() << std::endl;
        return false;
    }

    glContext = SDL_GL_CreateContext(window);
    if (!glContext) {
        std::cerr << "SDL_GL_CreateContext Error: " << SDL_GetError() << std::endl;
        return false;
    }


    if (!gladLoadGLLoader((GLADloadproc)SDL_GL_GetProcAddress)) {
        std::cerr << "Failed to initialize GLAD!" << std::endl;
        return false;
    }

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

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



```cpp
SDL_GLContext JetEngine::GetOpenGLContext() const {
    return glContext;
}
```

GameObjects

For the GameObjects such as the Enemies, Companion and Spaceship, the rendering was not altered as all it does it call functions inside the renderer, therefore all the rendering methods are left unchanged and still sent the texture, object and position.



```cpp
void Missile::Render(Renderer* renderer) {
    if (renderer && texture) {
        renderer->Render(texture, spriteRectObject, position);
    }
}
```

Objectives not Achieved

The objectives which were not achieved were the same ones as in the AGPT part of the project, along with incomplete rendering as I could not figure out the cause behind it.