POLITÉCNICO DE LEIRIA

Advanced Game Programming Topics

Project 1

Alexandra Madeira - 2221926

Pedro Ferreira - 2212613

Engine Classes

AIMovementComponent

The AIMovementComponent.hpp file defines a specialized component class, AIMovementComponent, that is used in the game engine to enable basic AI-driven movement behavior for game objects. This component inherits from a base Component class, making it a modular and reusable part of the game engine's architecture.

AnimationClass

The AnimationClass.cpp file implements the functionality for an Animation class, a key feature for managing sprite-based animations in the engine. This class is designed to handle frame-based animations, providing smooth transitions between frames and rendering the corresponding portion of a texture to the screen at the correct position.

AnimationComponent

The AnimationComponent.hpp file defines the AnimationComponent class, a key component in the engine architecture for managing animations associated with game objects. It inherits from a base Component class, making it an attachable module to any game object that requires animation functionality. This class builds upon the functionality of the Animation class, adding a system for managing multiple animations and switching between them dynamically during gameplay.

CollisionComponent

The CollisionComponent.hpp file defines the CollisionComponent class, an integral part of the game engine for detecting and managing collisions between objects. By inheriting from a base Component class. It provides functionality for defining collision behavior, including type classification, boundary calculations, collision detection, and callback execution.

The class introduces an enumeration, ColliderType, to distinguish between different kinds of objects in the game world. These types—such as PLAYER, ENEMY, PLAYER_BULLET, and ENEMY_BULLET—help categorize the purpose and interaction rules of game objects during collisions. Each CollisionComponent instance is initialized with a specific collider type, ensuring precise classification.

We were not able to make Box2D work, so we figured a way arround it. It was not the best but It was the solution we found.

CollisionManager

The CollisionManager.cpp file implements the CollisionManager class, which is responsible for handling collision detection among game objects. This class acts as a centralized system for managing all active CollisionComponent instances.

At its core, the CollisionManager maintains a list of shared pointers to CollisionComponent instances. These components represent the collision boundaries of game objects and are dynamically added or removed from the manager's control.

The AddCollider method allows new CollisionComponent instances to be registered with the manager. Before adding a collider, the method checks if it already exists in the list, preventing duplicates. This ensures that each collider is only tracked once, preserving the integrity of the system.

The RemoveCollider method removes a specific CollisionComponent from the manager. It uses the std::remove algorithm to locate the target collider and erase it from the list. This operation supports dynamic changes in the game world, such as removing objects that are destroyed or deactivated.

Component

The Component.hpp file defines the base Component class, a foundational part of this game engine's component-based architecture. This class provides a common interface and shared functionality for all specialized components that can be attached to game objects. By serving as the parent class for various components.

The Component class is structured to be lightweight and extensible. It includes the following key features:

1. **Core Lifecycle Methods:**
    - Start: A virtual method that can be overridden by derived classes. It is called when the component is first added to a game object, allowing initialization specific to the component.
    - Update: Another virtual method, called every frame, that allows components to perform time-dependent operations. It takes a deltaTime parameter, enabling frame-rate-independent logic.
    - Render: Called during the rendering phase, allowing components to contribute to the visual output of the game. It takes an SDL_Renderer pointer for graphics operations.
2. **Enable/Disable Functionality:**
   Components can be toggled on or off using the Enable, Disable, and SetEnabled methods. The isEnabled flag determines whether the component is active. This feature is particularly useful for temporary deactivations, such as pausing behavior or disabling interactions during cutscenes.
3. **Owner Management:**
   Each component is associated with a GameObject, its owner, which represents the entity the component is attached to. The SetOwner method assigns the owning game object, while the GetOwner method retrieves it. This association enables components to interact with their owner and access its properties and methods.

GameObject

The GameObject.cpp file implements the GameObject class, a central abstraction in the game engine responsible for representing entities in the game world. The class manages its position, dimensions, and a collection of associated components, which define its behavior and functionality.

The GameObject class begins with a constructor that initializes the entity's position, dimensions, and initial state. The x and y parameters specify the object's starting position in the game world, while width and height define its size. The constructor also records the initial position in initialX and initialY, potentially for reset or reference purposes. The components vector, which holds the various components attached to the game object, is implicitly initialized as empty.

The destructor of the GameObject clears the components vector, ensuring proper cleanup when the object is destroyed. This prevents memory leaks by removing references to dynamically allocated components.

The Update method is responsible for advancing the state of the game object. It iterates through the components vector and calls the Update method of each enabled component, passing in the deltaTime parameter. This design allows each component to update its behavior independently, facilitating modular and reusable functionality.

Graphics

The Graphics.cpp file implements the Graphics class, which handles the initialization and management of the graphical rendering environment in the game engine. This class encapsulates SDL2's window and renderer functionality, providing a streamlined interface for graphics-related operations.

InputHandler

The InputHandler.cpp file defines the InputHandler class, which manages input from keyboards and game controllers in the game engine. This class simplifies the handling of various input events, providing a centralized and extensible system for processing player interactions.

Level

The Level.cpp file implements the Level class, which serves as a container for managing all game objects within a specific level or scene in the game engine. It integrates closely with the Graphics system to render objects and coordinates updates for the game logic, making it a foundational part of the gameplay structure.

Initialization and Cleanup

The Level constructor takes a pointer to a Graphics instance as its parameter, allowing it to access the rendering context. This ensures that game objects within the level can be rendered to the screen.

The destructor ensures proper cleanup by calling the ClearLevel method. This removes all references to game objects managed by the level, releasing memory and preparing the level for destruction.

Game Object Management

The Level class manages game objects using a std::vector of std::shared_ptr<GameObject>. This approach provides several advantages, including automatic memory management and the ability to share ownership of objects.

- AddGameObject: This method adds a new game object to the level. By storing shared pointers, the level ensures that objects remain accessible as long as they are part of the level.
- RemoveGameObject: This method removes a specific game object from the level. It uses std::find to locate the object in the vector and erases it if found. This design supports dynamic modification of the level's contents, such as spawning or destroying objects during gameplay.
- ClearLevel: This method clears all game objects from the level, effectively resetting it. This is particularly useful when transitioning between levels or restarting the current one.

Main Function

The main.cpp file serves as the entry point for the game, initiating the application and managing its execution flow. It leverages SDL2 as the underlying library and encapsulates the main game logic within a Game object, defined in the XennonGame module.

Movement Component

The MovementComponent.hpp file defines the MovementComponent class, a specialized component for handling dynamic movement of game objects in the game engine. This class extends the base Component class and introduces features like axis-based locking, screen-bound constraints.

Object Pool

The ObjectPool.cpp file implements the ObjectPool class, a resource management system designed to optimize object allocation and reuse in the game engine. This class is particularly beneficial for scenarios involving frequent creation and destruction of game objects, such as projectiles, enemies, or particle effects.

Initialization

The ObjectPool constructor takes two parameters: an initialSize, specifying the number of objects to preallocate, and an objectFactory, a function that produces new instances of game objects. During construction, the object pool preallocates the specified number of objects, initializes them using the factory, and marks them as inactive. These inactive objects are stored in the inactiveObjects list, ready for reuse.

Object Acquisition

The Acquire method retrieves an object from the pool:

1. If there are no inactive objects, it creates a new object using the factory function.
2. If the acquired object has a CollisionComponent, the method assigns the pool as its reference via SetObjectPool, enabling integration with collision management systems.
3. If inactive objects are available, the method retrieves one from the inactiveObjects list.
4. In all cases, the object is marked as active and added to the activeObjects list, ensuring it is ready for use.

This approach minimizes memory allocation overhead by reusing objects, enhancing performance in resource-intensive scenarios.

Object Release

The Release method deactivates an object and returns it to the pool:

1. The object is marked as inactive.
2. It is removed from the activeObjects list and added back to the inactiveObjects list. This process ensures that released objects are cleaned up and made available for future reuse without being destroyed, reducing the performance cost of dynamic memory management.

Sprite

The Sprite.cpp file defines the Sprite class, a fundamental component for handling 2D graphics in the game engine. This class is responsible for loading image files, creating textures, and rendering them to the screen using SDL2.

Game Classes

Background

The Background.cpp file implements the Background class, a specialized type of GameObject that handles scrolling backgrounds for the game. This class adds visual and dynamic depth to the game environment by creating seamless, continuously scrolling visuals.

Initialization

The constructor initializes a Background object with several parameters:

- Position (x, y): Defines the starting position of the background.
- Sprite (spritePath): Specifies the path to the image used for the background.
- Scrolling (ScrollDirection, scrollSpeed): Sets the direction and speed of the scrolling effect.
- Dimensions (width, height): Determines the size of the background.

The constructor calls the base GameObject constructor to initialize position and dimensions. It also sets up scrolling-specific attributes:

- scrollSpeed: Controls how fast the background scrolls.
- xOffset and yOffset: Track the offset of the background's movement in each direction.
- direction: Determines the scrolling direction (e.g., left, right, up, or down).

The constructor then attaches an AnimationComponent to the Background object, using the provided sprite path to create a visual representation. It sets up a default animation named "scroll" with a single frame that matches the background's size and starts the animation immediately.

Bullet

The Bullet.hpp file defines the Bullet class, a specialized game object representing projectiles fired by either the player or enemies.

Initialization

The constructor initializes a Bullet instance with the following parameters:

renderer: The SDL renderer used for drawing the bullet.

x and y: The initial position of the bullet.

isPlayerBullet: A boolean indicating whether the bullet belongs to the player or an enemy.

Based on the isPlayerBullet flag, the constructor determines the bullet's dimensions, selects the appropriate sprite sheet, and configures animations, collisions, and movement.

Components

The Bullet class attaches three key components:

1. Animation Component:
    - The constructor adds an AnimationComponent to the bullet and configures it with different sprite sheets depending on the bullet type ("graphics/missilePlayer.bmp" for player bullets and "graphics/EnWeap6.bmp" for enemy bullets).
    - It defines and plays a looping animation named "bullet" using the relevant frames and sizes:

        Player bullets use a 4-frame animation with dimensions 8x16 pixels.

        Enemy bullets use an 8-frame animation with dimensions 16x16 pixels.

    - The animation frame time is set to 0.05f for smooth motion.
2. Collision Component:
    - The constructor adds a CollisionComponent and sets its type based on the bullet's origin (PLAYER_BULLET or ENEMY_BULLET).
    - A collision callback is assigned to handle interactions with other objects. Depending on the bullet's type, it checks for collisions with enemies (for player bullets) or the player (for enemy bullets). Upon collision, the bullet is returned to the object pool for reuse.
3. Movement Component:
    - The constructor adds a MovementComponent and sets the bullet's speed to 200.0f units per second.
    - The movement direction is determined by the bullet type:

Player bullets move upwards with a velocity of (0, -1.0f).

Enemy bullets move downwards with a velocity of (0, 1.0f).

Enemy Spawner

The EnemySpawner.cpp file implements the EnemySpawner class, which manages the spawning and updating of enemy entities in the game.

Initialization

The constructor initializes the EnemySpawner with several key components:

- renderer: Used to render newly spawned enemies.
- level: A reference to the current game level, allowing the spawner to add enemies to the level.
- collisionManager: Handles the registration of enemy collision components to enable interaction with other game objects.

It sets up spawn parameters, including screen dimensions (screenWidth and screenHeight) and the spawn height (spawnY), determining where enemies appear on the screen.

The constructor also creates two object pools:

1. Loner Pool: Preallocates 10 Loner enemy objects. These are initialized with a renderer and a collision manager.
2. Rusher Pool: Preallocates 10 Rusher enemy objects. These require only a renderer during instantiation.

This pooling mechanism optimizes performance by reusing enemy objects instead of dynamically allocating and deallocating them during gameplay.

Update Method

The Update method handles time-dependent logic for spawning enemies and updating active enemy objects:

1. Spawn Timers:

   The lonerSpawnTimer and rusherSpawnTimer track when to spawn new enemies. If a timer reaches zero, the corresponding spawn method (SpawnLoner or SpawnRusher) is called, and the timer is reset to a predefined interval (LONER_SPAWN_INTERVAL or RUSHER_SPAWN_INTERVAL).

2. Object Pools:

The Update method for both the lonerPool and rusherPool ensures that active enemies are updated every frame, maintaining their animations, movement, and other behaviors.

Spawning Logic

The SpawnLoner and SpawnRusher methods handle the instantiation and setup of enemies:

1. Randomized Spawn Position:

   Both methods use a uniform distribution to generate a random x-coordinate within the screen's bounds, ensuring varied spawn locations.

2. Acquire from Pool:

   Enemies are acquired from their respective pools, reactivating inactive objects or creating new ones as needed.

3. Position and Activation:

   The acquired enemy's position is set to the calculated spawn coordinates (spawnX, spawnY), and it is marked as active using SetActive(true).

4. Add to Level:

   The enemy is added to the level, allowing it to participate in game logic and rendering.

5. Collision Registration:

   The enemy's CollisionComponent is retrieved and registered with the collisionManager, enabling interactions with other entities like bullets or the player.

Loner

The Loner.cpp file defines the Loner class, a specific enemy type in the game.

Initialization

The constructor initializes the Loner enemy with its position, size, and required systems (renderer and collisionManager). The initialization process involves configuring its animations, movement, collision, and bullet pool. Each subsystem is encapsulated in its respective initialization method.

1. Animation Initialization:

An AnimationComponent is added to represent the Loner visually using a sprite sheet ("graphics/LonerA.bmp").

The animation setup includes extracting individual frames from the sprite sheet and adding an idle animation with a loop. Each frame corresponds to a 2D sprite of size SPRITE_SIZE.

2. Movement Initialization:

   The Loner uses an AIMovementComponent with a horizontal movement pattern. This component automatically handles directional movement and screen bounds constraints (0, 0, 800, 600).

3. Collision Initialization:

   A CollisionComponent is added to enable interaction with other entities. It is designated as an enemy collider type (CollisionComponent::ColliderType::ENEMY).

   The collision component is configured with a callback to handle collisions, such as deactivating the Loner when hit by a player bullet.

4. Bullet Pool Initialization:

   A bullet pool is established with a size of 10. The bullets are initialized as enemy bullets and share ownership of the renderer and position data.

   The pool system optimizes performance by reusing bullets, minimizing the overhead of frequent memory allocations and deallocations.

Update Method

The Update method oversees the Loner's behavior during each frame:

1. It decrements a shooting timer. When the timer reaches zero, the Shoot method is called, and the timer is reset to a cooldown value (SHOOT_COOLDOWN).
2. The bullet pool is updated, managing the movement and lifecycle of active bullets. Bullets that move off-screen are automatically released back into the pool.
3. All other game logic, including animations and movement, is handled through the base GameObject::Update call.

Shooting Mechanism

The Shoot method acquires a bullet from the pool, sets its position slightly below the Loner, and activates it. The bullet's collision component is registered with the collision manager, allowing interactions with other objects such as the player.

PlayerShip

The PlayerShip.cpp file defines the PlayerShip class, representing the player's controllable spaceship.

Initialization

The PlayerShip constructor initializes the spaceship with its position, size, and the required systems (an SDL renderer and a collision manager). The setup involves adding components for collision, movement, and animations, as well as preparing a bullet pool for shooting functionality.

1. Collision Component:

    A CollisionComponent is added to detect interactions with other game objects. It is assigned the PLAYER collider type.

    A callback is set up to handle specific collision scenarios, such as being hit by an enemy or projectile.

2. Movement Component:

    A MovementComponent is added to control the ship's movement. The movement is bounded within the screen dimensions, preventing the player from leaving the visible area.

3. Animation Components:

    Three AnimationComponents are added:

    - Idle animation using "graphics/Ship1_Idle.bmp".
    - Left bank animation using "graphics/Ship1_Left.bmp".
    - Right bank animation using "graphics/Ship1_Right.bmp".

    Each animation is configured with its frames and playback settings. For example, the idle animation consists of a single frame, while the left and right bank animations each use three frames.

4. Bullet Pool Initialization:

    A pool of 20 bullets is created to handle the player's shooting mechanism. Each bullet is instantiated with its position and type set as a player bullet.

Rusher

The Rusher.cpp file implements the Rusher class, which represents a specific type of enemy characterized by its vertical movement and aggressive behavior.

Initialization

The constructor initializes the Rusher enemy with its position and dimensions. It sets up the core functionality by delegating to three methods: InitializeAnimations, InitializeMovement, and InitializeCollision.

1. Animation Initialization:

   An AnimationComponent is attached to the Rusher, using the sprite sheet "graphics/rusher.bmp".

   The sprite sheet contains multiple frames representing the Rusher in different states. The idle animation is configured by extracting frames row-wise:

      Each frame has a width and height of SPRITE_WIDTH and SPRITE_HEIGHT, respectively.

      Frames are added to the idle animation with a frame time defined by ANIMATION_FRAME_TIME.

   The idle animation is set to loop continuously.

2. Movement Initialization:

   The Rusher uses an AIMovementComponent configured for vertical movement (MovementPattern::VERTICAL) with a speed defined by MOVEMENT_SPEED.

   Screen bounds are set to ensure that the Rusher remains within the visible game area (0, 0, 800, 600).

3. Collision Initialization:

A CollisionComponent is added, identifying the Rusher as an enemy (ColliderType::ENEMY).

A collision callback is assigned to handle interactions with other objects, such as the player or bullets.

Main Game Loop

The XennonGame.cpp file defines the main game logic encapsulated within the Game class. This class serves for handling initialization, gameplay loop, and termination processes. It integrates all major systems of the game engine, including rendering, input handling, collision management, and gameplay mechanics.

Initialization

The Game class constructor sets up key attributes:

- graphics for rendering the game.
- level to manage game objects and their lifecycle.
- Various shared and unique pointers for the player, enemies, backgrounds, and an EnemySpawner.

The Initialize method handles system and game object setup:

1. Graphics Initialization:

    The graphics.Initialize method is called to set up SDL rendering. If it fails, the game exits with an error.

2. Background Setup:

    A static background and a parallax-scrolling background are created and added to the level.

3. Player Initialization:

A PlayerShip object is created and added to the level. Its bullet pool is initialized to manage shooting functionality.

4. Enemy Setup:

   Instances of Loner and Rusher enemies are created and configured. The Loner also initializes its bullet pool.

5. Collision Manager:

   The collision components of the player and enemies are added to the CollisionManager.

6. Enemy Spawner:

   An EnemySpawner is created to dynamically introduce enemies during gameplay.

Gameplay Loop

The Run method implements the game loop, which operates continuously until isRunning is set to false. The loop consists of:

1. Input Handling:

   The InputHandler processes player inputs. If the quit command is detected, the game exits.

   Player-specific inputs (movement and shooting) are handled by the PlayerShip instance.

2. Updating Game State:

   The level.Update method advances the state of all game objects.

   The collisionManager.Update method resolves collisions among active objects.

   The enemySpawner.Update method manages enemy spawning.

3. Rendering:

   The screen is cleared using graphics.Clear, and all game objects are rendered through level.Render.

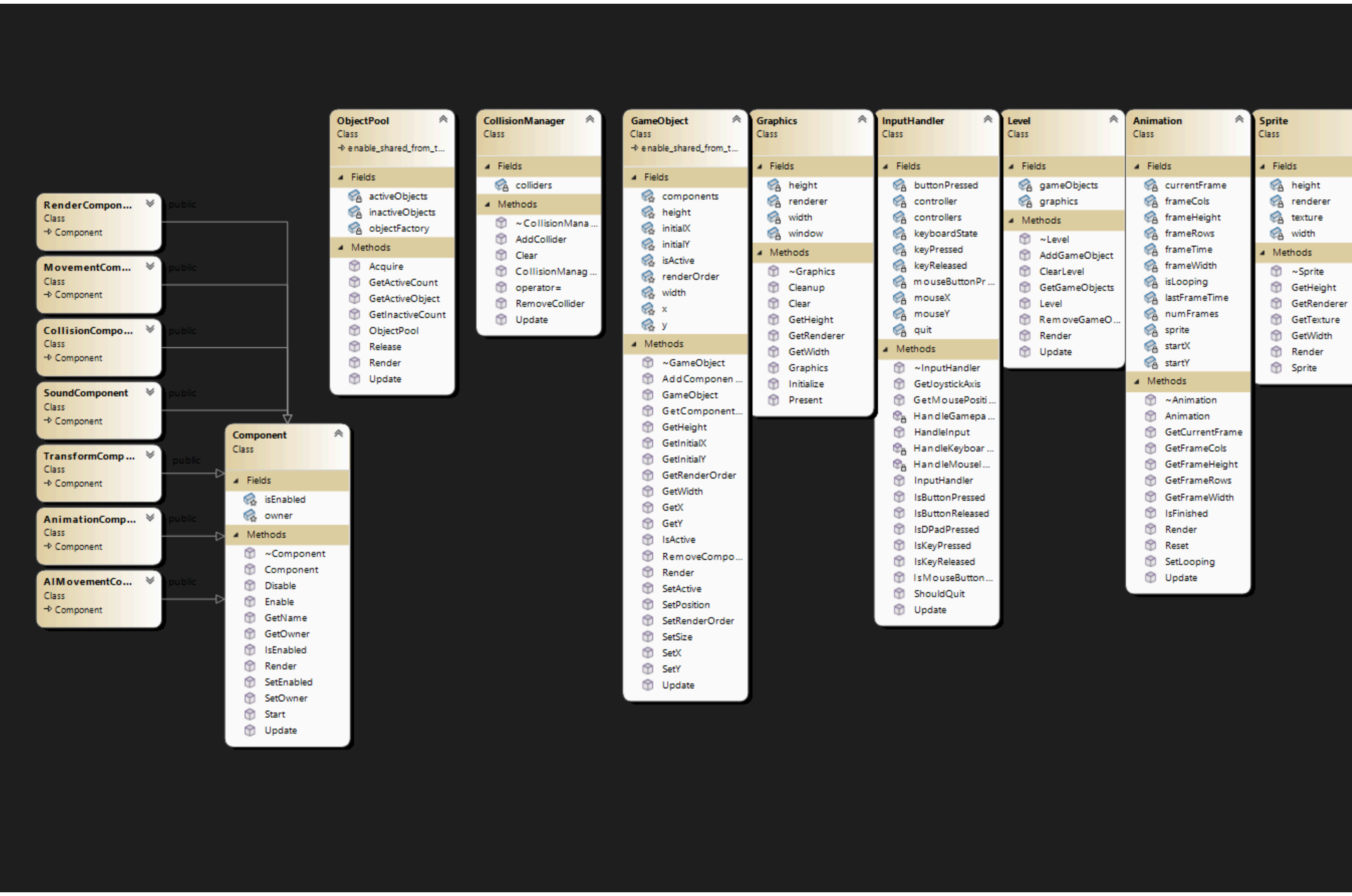
   The rendered frame is displayed using graphics.Present.

Cleanup

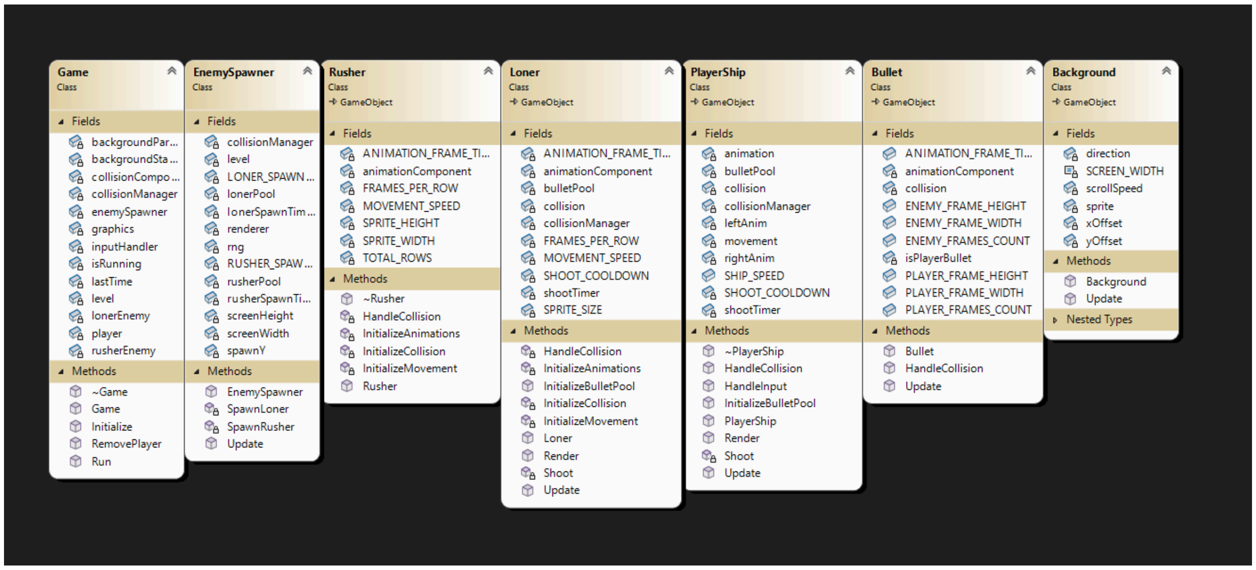
The destructor ensures a clean exit by clearing collision components and resetting the CollisionManager. This prevents memory leaks and ensures proper resource deallocation.

Class Diagrams

Engine:



Game:



Features

- Every feature was completed, excluding the Box2d Integration.

Bibliography

- https://wiki.libsdl.org/SDL2/FrontPage
- A Tour of C++ (Book)
- https://box2d.org/documentation/
- https://devdocs.io/cpp/
- https://foundationsofgameenginedev.com/
- https://www.youtube.com/@TheCherno