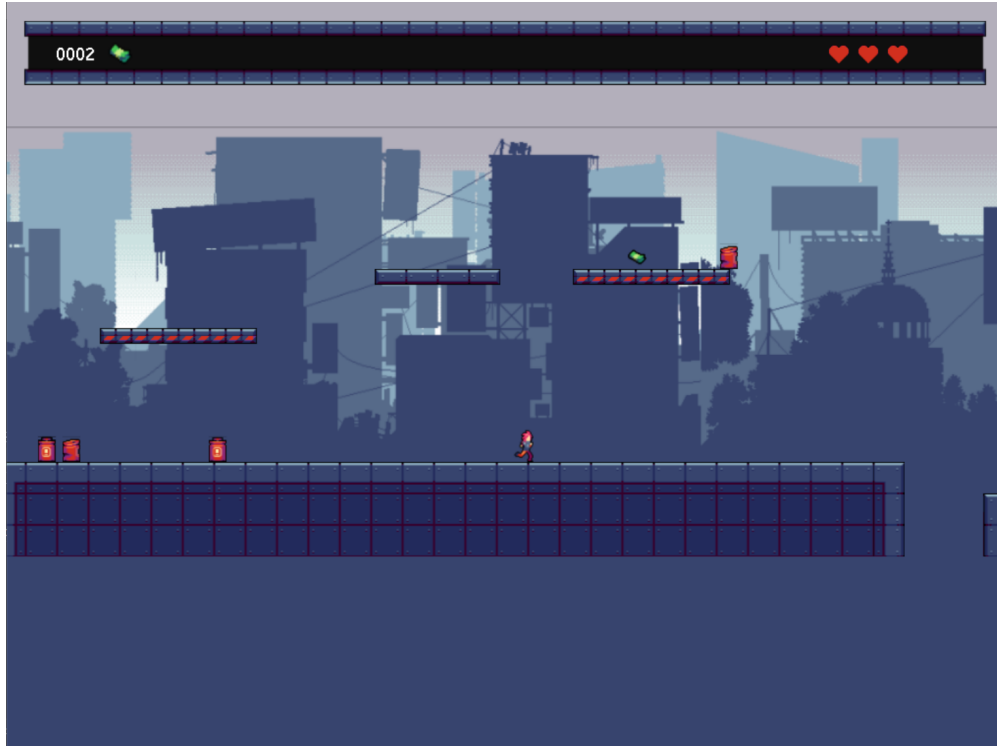


"City Runner" - Project Documentation

Welcome to the documentation for the "City Runner" project, an Infinite Scroller game developed as part of the Ubisoft Develop Mentorship program. In this documentation, you will find a detailed account of the design, development, and technical aspects of the game.



Project Overview




The "City Runner" project represents a technical challenge aimed at showcasing programming skills and challenging myself in the domain of game development. As a participant in the Ubisoft Develop Mentorship program, the task was to create an Infinite Scroller game using modern C++. The challenge extended beyond mere gameplay, emphasizing the importance of clear code structure, reusability principles, efficient memory management, ownership principles and the strategic use of modern C++ concepts.

Game overview

"City Runner" takes place in an urban environment where players must navigate obstacles, with the primary objective of collecting money and surviving as long as possible to maximize their score. The game is dynamically generated with random creation of city platforms and landscapes, ensuring a diverse and intriguing gaming experience. Furthermore, various pickups and power-ups are available, which will be detailed further in the subsequent sections of this document.





Character Actions

The main character possesses a set of three distinct actions:

-  **Run:** Default
-  **Jump:** Activated by the up arrow key
-  **Double Jump:** Executed by pressing the up arrow key after an initial jump

Collectibles and Power up

The game contains various pickable items that enhance the gaming experience:

-  **Money:** Collecting money adds to your final score. The more money you collect, the better you've played.
-  **Speed Boost:** Increases the player's speed by 50% for 5 seconds
-  **Jump Boost:** Enhances the player's jump height by 30% for 5 seconds
-  **Life Boost:** Adds an extra life to the player

Health System

The player starts with 3 lives and can accumulate up to 5 lives. Lives are gained through the Life Boost power-up, and they are lost by colliding with obstacle barrels. Colliding with an obstacle grants the player a 5-second invulnerability period, preventing consecutive collisions that could otherwise deplete the player's entire life pool at once. Obstacles are represented by red barrels of various types:



HUD



The HUD is designed to be simple and intuitive. On the left side, the player's score is displayed (representing the amount of money collected). The right side indicates

the current number of lives the player has. The center of the UI is dedicated to the player's status: *(in order from left to right)* invulnerability status, speed boost status and jump boost status.

Technical Overview

When the program is launched, a *Game Instance* entity is instantiated, marking the commencement of the game loop. It serves as an efficient controller, overseeing various *Game Modes* for specific phases: main menu, gameplay, and game over. Deliberately designed for simplicity, this choice guarantees seamless transitions and maintains a clear, manageable codebase. This entity always has a reference to the current Game Mode.

The game architecture revolves around two fundamental types of Game Modes, each playing a distinct role in shaping the player's experience. Firstly, the *Menu Game modes* are tailored to showcase static UI pages represented by *UIComponents*, guiding users through the interface and setting the stage for the upcoming gameplay. Secondly, the *Gameplay Game Mode* operates as a robust manager, efficiently handling all sub-managers and objects within the game scene.

Core entities

In my project, Core Entities form the backbone of our game architecture. These entities encompass *Objects*, *Actors*, and *Components*, drawing inspiration from the foundational elements of the Unreal Engine.

- **Objects** serve as the fundamental building blocks, encapsulating data and functionality within the game.
- **Actors** represent dynamic entities within the game world. They take center stage, equipped with transforms (location, scale, rotation) and the ability to interact with other actors. In our project, Actors are capable of having renderable and visible components, contributing to the visual aspect of our gaming environment.
- **Components** act as modular, reusable units that can be attached to Objects and Actors, enhancing their functionality. Components contribute to the versatility of game entities, allowing for efficient customization and extension.

This trio of Core Entities collaborates harmoniously, providing a robust framework for the development and interaction of elements within the game.

Core components

Components encapsulate reusable behaviours that can then be assigned to Objects and Actors. The core gameplay components of my game are:

- **Camera Component:** Manages the position, rotation, and zoom of the in-game camera, providing a dynamic and responsive view of the game world.
- **Collision Component:** Handles collision detection and response, allowing game objects to interact with each other and the environment based on defined collision rules. It employs the AABB (Axis-Aligned Bounding Box) method for simplicity.
- **Mesh Component:** Represents the visual geometry or model of game objects, facilitating the rendering and visual representation of in-game entities, using

the provided `CSimpleSprite` implementation.

- **Movement Component:** Governs the movement and navigation of game actors, implementing a state machine-like algorithm for transitioning between running, jumping, falling and more.
- **Animation Component:** Controls the animation state and playback for characters or objects.
- **Input Component:** Manages user input, capturing and interpreting player commands to trigger specific actions or behaviors within the game.

Collision system

The collision system in this game is orchestrated by the *CollisionManager* and *CollisionComponent* classes, working in tandem to provide robust collision detection and response. The manager is responsible for handling collision presets, managing the registration and unregistration of collision components, and overseeing the collision resolution process. Upon initialization, the **Collision Manager** configures collision presets, defining how different collision profiles interact. These profiles encompass entities such as characters, enemies, obstacles, collectibles, floors, and more. During runtime, **Collision Components** register with the manager, allowing it to efficiently track interactions between game entities. The heart of the collision system lies in the `Update` method of the manager, where it iterates through *"dirty"* registered components, checks for potential collisions, and invokes collision responses. The *CheckCollision* function within the manager compares bounding boxes and, upon collision detection, triggers the appropriate response based on the collision presets. The collision responses include *blocking collisions*, where entities are prevented from overlapping, and *overlapping collisions*, signaling an event when entities intersect. The system further handles the initiation and conclusion of collisions, notifying relevant components through the event manager.

Game Aura system

In-game resources such as coins, health, speed, jump height and invulnerability are managed by a single system: the Game Aura System. The *GameAuraManager*, *GameAuraComponent*, and *GameAura* classes collectively form a comprehensive system for managing game auras and their impact on resources.

- The **GameAuraManager** serves as the central hub for registering and unregistering actors with their associated aura components. It manages the activation and termination of auras on different components.
- The **GameAuraComponent** class is designed to be attached to game actors, providing them with the ability to have resources and manage auras. During initialization, the component registers itself with the manager. It maintains resource values, both current and maximum. The component can activate, deactivate, and update auras during the game, ensuring that their effects are applied appropriately.
- The **GameAura** class represents individual auras objects with distinct properties. Auras can have different damage modes (flat, current amount percentage, or maximum amount percentage), associated resources, effects, and duration. The class implements methods for starting, updating, and ending auras, applying their effects to the associated Game Aura Component.

Together, these classes provide a flexible and modular system for managing auras and their impact on in-game resources. Actors can have multiple auras active

simultaneously, and the system ensures that the effects are applied and updated correctly, contributing to a dynamic and engaging gameplay experience.

UI system

The UI system follows a composition-based approach where **UIComponent** manages multiple **UIElement** instances. This modular design allows for the addition and removal of UI elements dynamically. The system considers camera movement and adjusts UI element positions accordingly. UI elements are designed maintaining the principles of Single Responsibility, managing one graphical entity at a time, such as sprites handled by the **UIElementSprite** or text by the **UIElementText**.

Event Management

The event management system is a crucial component of the application, facilitating the decoupling of various modules and enabling efficient communication between disparate parts of the codebase. At its core is the versatile **Event** class, designed as a template with *variadic arguments*, allowing flexibility in defining events with different parameter types and counts. This class provides methods for adding and removing event callbacks dynamically. Each event is associated with a unique instance ID, creating a mapping to a vector of callback information. The **EventManager** class, a practical implementation of the event system, demonstrates its usage in the game context. Events such as game over, movement status changes, and resource updates are seamlessly managed.

Object pooling

In order to manage background elements, game platforms, and various gameplay components like pickups, obstacles, and power-ups, an Object Pooling System has been implemented. This system optimizes resource usage and enhances performance by efficiently managing the instantiation and reclamation of objects during runtime.