Advanced Programming Topics

Leiria December 2023

Ricardo Brites 2202225

Table of Contents

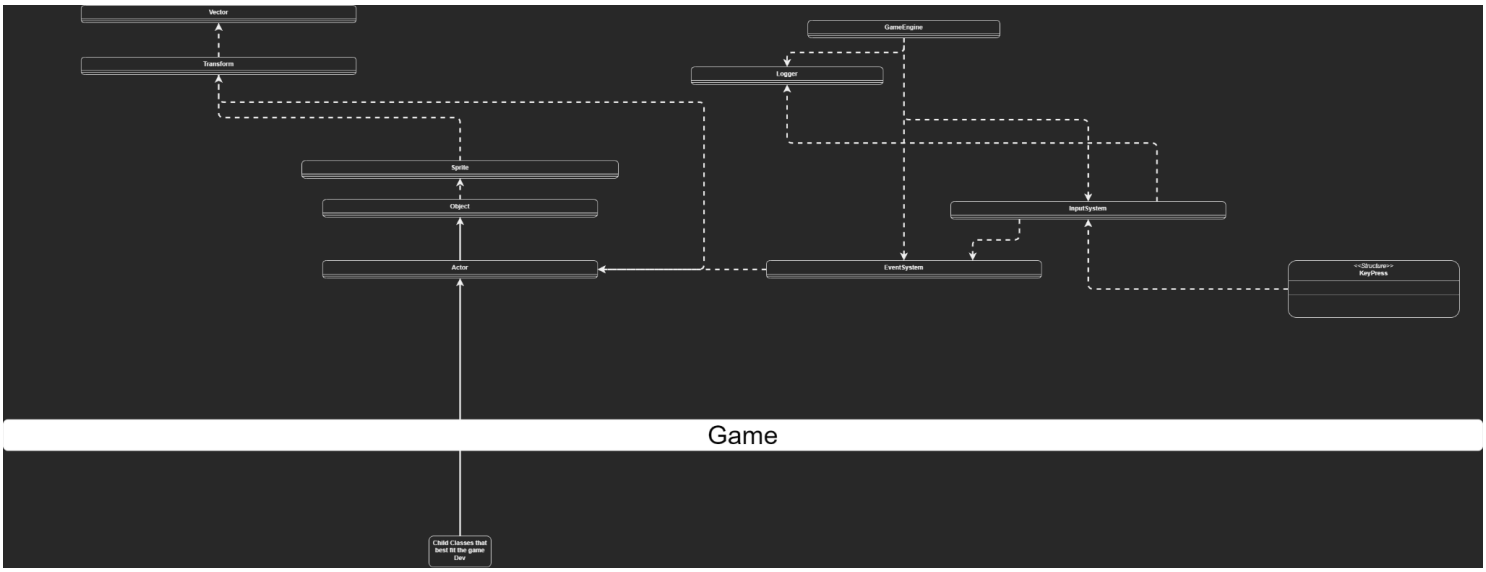
General Project Information

Configurations

The debug configuration incorporates logic for presenting debug information in the command line by utilizing configuration-dependent macro definitions. Conversely, the release configuration omits any logic for exhibiting debug messages.

This functionality is realized through the employment of the preprocessor, which defines distinct versions of the macros contingent upon the configuration specified manually in the project settings. This approach was selected for its simplicity, absence of runtime performance implications, and avoidance of the need for multiple preprocessor if statements to verify the presence of _DEBUG and _RELEASE symbols, thereby contributing to a more streamlined and cleaner codebase.

Engine Logic



The game engine starts in the GameEngine.cpp and GameEngine.h, the game engine executes the following logic in the order displayed (GameLoop Specifically):

- DeltaTime calculations;
- Check Input;
- Check Collisions;
- Trigger BeginPlay Function in level Actors
- Trigger Tick function in level actors
- Render
- Memory cleanup (could be called a garbage collector)

The engine is made specifically for single-player games so a variable that stores the reference for a player class that is derived from the base Actor class… This player reference is stored as a reference to the base class Actor to avoid using classes not present in the game engine project.

Input System

The input system is designed to receive an SDL_Event pointer containing the input events for the current frame, with a specific focus on input-related data, this pointer serves as the basis for invoking various behaviors according to the input type. Subsequently, these behaviors converge into the invocation of three distinct functions within the Object interface, contingent upon the nature of the input. The implementation of these functions in different child classes allows divergent behaviors unique to each class.

```
class Object
{

public:
    virtual void BeginPlay() = 0;
    virtual void Tick(float DeltaSeconds) = 0;
    virtual void OnKeyPressed(InputKeyCodes KeyCode) = 0;
    virtual void OnKeyReleased(InputKeyCodes KeyCode) = 0;
    virtual void OnInputAxis(InputKeyCodes KeyCode, Vector AxisValue) = 0;
};
```

Each of these functions accepts a KeyCode as a parameter. This KeyCode is defined as an enum, strategically created to facilitate streamlined input handling across any class that adheres to the Object interface. This design choice ensures a standardized approach to processing input across different classes, promoting consistency and ease of maintenance in the codebase.

Improvements

If I were to restart this project this would be an aspect I would change. The player is the only one that should use input, thus there is no reason for the base Actor class to use this behavior (input) it's unnecessary overhead that each object in the game could be free of.

An alternative approach would be to create another interface that implements the input only. By creating this secondary input interface I would be able to create two base classes Actor and Character… Much like Unreal Engine where ACharacter inherits from AActor which in turn inherits from UObject.

Collision System

Initially, the prescribed collision system for the classes involved the use of Box 2D, a recommendation from the instructor. However, multiple attempts to implement this library were unsuccessful due to inadequate documentation. Faced with this challenge, the developer tasked with the development of this project assumed responsibility for devising an alternative collision system.

The resulting collision system is intentionally designed to be as straightforward and lightweight as possible. Its operational principle involves iterating through all non-pending-kill objects, verifying their collision enablement by checking that the CollisionRadius float value is non-negative. Subsequently, the system calculates the distance between these objects. A collision is deemed to occur if this distance is less than the sum of the two radii.

Upon detecting a collision, facilitated by a collision interface implemented in all actors, a specific function, "OnCollisionStarted," is invoked in the actors where the collision was identified. This streamlined approach enhances the efficiency of collision detection and response, ensuring a simplified and effective system.

Improvements

While the current collision system performs effectively for detecting and responding to collisions through the "OnCollisionStarted" event, it becomes evident that it may not cover all scenarios, particularly those necessitating "OnCollisionLeave" events. Recognizing this limitation, it is reasonable to consider the refinement and expansion of the system to accommodate more complex behaviors.

To address this requirement, a more sophisticated approach could involve introducing states for each object. These states would enable the system to track the evolving nature of object interactions, allowing for the implementation of functions like "OnCollisionLeave" to handle events when collisions cease.

By incorporating state information for each object, the collision system gains the capacity to manage the lifecycle of collisions more comprehensively. This modification would provide a foundation for handling not only the initiation of collisions but also their termination, offering a more nuanced and adaptable solution to meet the specific demands of the game application.

Event System

The event system has been intentionally designed with a high degree of simplicity. Its operation involves a straightforward iteration through all non-pending-kill Actors, subsequently triggering the BeginPlay and Tick functions. In the case of the Tick function, the delta Time value is included as a parameter, providing essential time-related information for the function's execution.

This minimalist design ensures a streamlined and efficient process for handling actor events. The systematic invocation of BeginPlay and Tick functions for active Actors aligns with the fundamental principles of the event system, emphasizing simplicity and clarity in the execution of these essential functions. The inclusion of delta Time in the Tick function contributes to the system's responsiveness by providing real-time temporal context during the execution of actor updates.

Improvements

As indicated in the input system section of this report, a program optimization strategy recommends the division of the Object interface into two distinct interfaces. This segregation facilitates the creation of two base classes Actor and Character, emulating the implementation approach observed in Unreal Engine.

By bifurcating the Object interface, the aim is to enhance the program's optimization. The resultant separation into two interfaces aligns with best practices for code organization and facilitates a more modular and maintainable codebase. This optimization strategy draws inspiration from the structure employed in Unreal Engine, a proven framework in the field.

This division enables a clearer and more focused definition of responsibilities, allowing for the creation of base classes that are specialized and aligned with the specific functionalities of each interface. Such a restructuring contributes to code clarity, ease of maintenance, and potentially improved performance in the execution of the system.

Memory Management

At the conclusion of each frame, the engine systematically iterates through all Actors, removing any instances that have been marked as pending kill. This cleanup process ensures the efficient management of memory resources and promotes a streamlined execution environment by eliminating objects that are no longer relevant or required in the application. The identification and removal of pending-kill Actors contribute to maintaining the system's overall performance and responsiveness.

This functionality has been implemented through the introduction of a singular boolean variable, serving as a marker to denote the pending kill state. This boolean variable effectively encapsulates the information needed to identify whether an Actor is slated for deletion. By employing this straightforward mechanism, the system can efficiently track and manage the removal of Actors at the end of each frame, contributing to a concise and effective approach to memory management within the application.

Upon reaching the specified kill zone, all actors undergo a status update, being marked as pending kill. This systematic identification and classification of actors within the defined kill zone allow for their subsequent removal during the cleanup phase at the end of each frame. The utilization of the pending kill state serves as an effective mechanism for signaling the engine to remove these actors, contributing to the overall maintenance of the game's state and ensuring efficient memory management.

Sprites

The rendering of sprites is facilitated through a dedicated sprite component, which is initialized with essential parameters, including the texture path, tiling information, animation duration in seconds, and the parent entity to which the component is attached.

Upon the creation of this sprite component, the PlayAnimation function becomes available. This function accepts a boolean parameter that dictates whether the animation should loop or not. By invoking PlayAnimation, the sprite component initiates the specified animation, taking into account the provided looping configuration.

This modular and parameterized approach to sprite management allows for flexible and dynamic control over animations, catering to various requirements within the application. The encapsulation of these features within a sprite component enhances code modularity and readability while providing a versatile toolset for animating entities in the game.

The sprite component is designed with extensibility in mind, offering the flexibility for customization through overrides. This means that developers have the option to implement custom logic by overriding specific methods or behaviors within the sprite component. This extensibility allows for the incorporation of specialized functionality or adjustments tailored to the unique requirements of individual entities or specific use cases.

By making the sprite component overrideable, the design promotes a modular and adaptable structure, empowering developers to augment or modify the component's behavior without the need for extensive changes to the existing codebase. This approach facilitates the accommodation of diverse scenarios and ensures that the sprite component can be fine-tuned to suit the evolving needs of the application.

Data Types

Vector

The vector is a fundamental data type within the game engine, encapsulating X, Y, and Z coordinates to represent spatial information. It is designed for simplicity and ease of use, providing a seamless experience for handling vector-related operations.

This data type incorporates a comprehensive set of operators, facilitating intuitive and efficient manipulation of vector values. The inclusion of these operators streamlines mathematical operations, transformations, and comparisons involving vectors. By ensuring a complete set of operators, developers can perform vector calculations with clarity and conciseness, contributing to a more fluid and expressive coding experience.

Transform

The Transform serves as a crucial component within the game engine, encompassing six essential vectors: location, relative location, rotation, relative rotation, scale, and relative scale. This comprehensive set of vectors empowers game developers with the flexibility to create and manipulate game entities in a manner aligned with their creative vision.

- Location
    - Represents the absolute position of the object in the game world.
- Relative Location
    - Denotes the object's position relative to its parent or origin, allowing for hierarchical transformations.
- Rotation
    - Specifies the absolute orientation of the object in the game world.
- Relative Rotation
    - Expresses the object's orientation relative to its parent or origin, facilitating hierarchical transformations.
- Scale
    - Defines the absolute scaling factors along the X, Y, and Z axes, influencing the object's size.
- Relative Scale
    - Indicates the scaling factors relative to the parent or origin, supporting hierarchical scaling.

Game Logic

Initially, a comprehensive background spanning the entirety of the window and the player are manually generated. Subsequently, through the utilization of a spawner system, the details of which will be expounded upon in a subsequent section of this report, all level entities including enemies such as Loner, Rusher, Drone, Asteroid, and power-ups like shield and weapon enhancements are dynamically spawned at random positions within the game environment. This spawner system contributes to the dynamic and evolving nature of the gameplay by introducing a varied array of challenges and enhancements for the player to interact with during the course of the game.

Enemies

Loner

The Loner enemy is characterized by its horizontal scrolling motion along a fixed Y-axis, complemented by periodic shooting directed towards the player at two-second intervals. The implementation of its horizontal movement involves adjusting the Actor's location during each frame, ensuring a seamless scrolling effect.

Initially, the shooting behavior employed a worker thread to launch projectiles at two-second intervals, with the thread subsequently entering a dormant state. However, in the course of optimizing the project, this approach underwent refinement. The new implementation utilizes a float variable that acts as a timer, tracking elapsed time. When this timer reaches the two-second mark, the Loner initiates the firing of a projectile and resets the timer, achieving the desired shooting behavior without the need for a dedicated worker thread. This optimization streamlines the code and enhances the efficiency of the Loner enemy's shooting mechanism.

LonerProjectile

The Loner projectile is designed as a straightforward object, encompassing essential attributes such as animation, collision enablement, and linear movement along a predetermined direction set at the time of spawn. Its simplicity is a deliberate choice, ensuring efficiency and clarity in its functionality.

Rusher

The Rusher, upon spawning, randomly selects an X-coordinate within the screen and proceeds to move vertically at a predetermined speed. The speed parameter is configurable in the GameRules file, offering a static value that can be adjusted to fine-tune the Rusher's movement dynamics.

This design emphasizes simplicity and configurability, allowing for easy adjustments to the Rusher's behavior by modifying the speed parameter in the GameRules file. The randomized X-coordinate selection adds an element of unpredictability to its initial positioning, contributing to a varied and dynamic gameplay experience.

Drone

The spawner system, prior to spawning drones, employs a dynamic approach. It first selects a random Y-coordinate and subsequently spawns a variable number of drones within a defined limit, enhancing the variability of the game environment.

The drones, once spawned, exhibit a distinctive behavior. They traverse the screen in a sinusoidal wave pattern horizontally. This behavior is orchestrated through the Tick function, where the object's location is continuously updated. The Y-axis movement follows a sine wave pattern, introducing undulating motion, while the X-axis motion is maintained linearly at a constant travel speed.

This implementation of a sinusoidal wave in the Y axis and linear motion in the X axis imparts a visually engaging and dynamic movement pattern to the drones, contributing to the overall diversity and challenges presented in the gameplay.

Asteroid

The asteroid is characterized by a straightforward behavior in which it descends along the Y-axis to simulate a falling motion. This descent is achieved by updating the Y location of the object within the Tick function.

Upon collision, the asteroid undergoes a unique transformation. It separates into three smaller asteroids, effectively simulating a fragmentation event. This behavior repeats twice, resulting in multiple successive separations until the asteroid ceases to exist.

The asteroid system introduces three distinct types, each with varying durability and resilience to player hits. The differentiation is as follows:

Type 1: Breaks upon receiving one hit from the player's projectiles.
Type 2: Requires two hits from the player's projectiles to break.
Type 3: Possesses invincibility and cannot be broken by the player's projectiles.

The visual diversity of the asteroid types further enhances the gaming experience, providing clear visual cues to players about the characteristics of each type. The distinct visual appearances for each asteroid type are designed to align with their respective durability and behavior. These visual differences aid players in quickly identifying and reacting to the various asteroid types during gameplay. The customized visuals for each type contribute to the overall immersion and visual storytelling within the game, creating a more visually interesting and engaging gaming environment.

Power-Ups

In the game project, two distinct power-ups enhance the player's capabilities:

- Weapon Power-Up:
    - Grants the player an additional companion.
    - The player can have a maximum of two companions.

- Shield Power-Up:
    - Restores the player's health.

These power-ups contribute to the overall strategy and dynamics of the game. The Weapon Power-Up offers offensive advantages by introducing companions, while the Shield Power-Up focuses on defense by restoring the player's health. The combination of offensive and defensive power-ups adds strategic depth to the gameplay, allowing players to tailor their approach based on the available power-ups and the evolving challenges within the game environment.

The implementation of power-ups, specifically the Weapon Power-Up and Shield Power-Up, is integrated into the collision system. When a collision occurs between the player character and a power-up, the collision system triggers specific value changes in the player character or introduces new game elements. Here's how the power-ups are handled:

- Weapon Power-Up:
    - The player character gains an additional companion, enhancing the offensive capabilities.

- Shield Power-Up:
    - The player character's health is incremented, providing a defensive advantage by replenishing health.

Spawner

The Spawner class operates as an independent entity, running on a dedicated worker thread. Its primary function is to spawn a random actor from a specified list at regular intervals, specifically once per second. The variety of actors that can be spawned includes Loners, Rushers, Drones, Asteroids, Weapon Power Ups, and Shield Power Ups.

The use of a dedicated worker thread for spawning enhances performance and responsiveness, allowing the main game loop to focus on other critical tasks while the Spawner handles the actor generation asynchronously.

In summary, the Spawner class contributes to the diversity and challenge of the game by introducing a variety of actors, each with its unique characteristics, at regular intervals.

The decision to implement the Spawner class on an independent worker thread reflects an optimization strategy aimed at addressing potential performance bottlenecks associated with frequent rand() function calls. By offloading the spawning logic to a separate thread, the main game loop is freed from the computational overhead of generating random numbers at a high frequency.

The rand() function can be computationally expensive, and its repeated usage in a performance-critical context may impact the overall responsiveness of the game. Utilizing a separate worker thread for spawning allows the main game loop to continue executing essential tasks without being hindered by the potential computational cost of random number generation.

This thread-based optimization demonstrates a thoughtful approach to balancing performance and responsiveness in the game engine, ensuring that the generation of random actors occurs independently and efficiently, contributing to an overall smoother gaming experience.

Spaceship

The spaceship serves as a comprehensive integration of all fundamental systems within the game engine, seamlessly incorporating the input, collision, sprites, and event systems.

Input System

The spaceship's input system leverages the Tick function to dynamically adjust the ship's location based on a boolean state.
Additionally, the system allows the spawning of missiles, introducing an interactive element to the ship's behavior.

Collision System

Health management is intricately tied to the collision system. The spaceship's health is defined by its ability to sustain damage or gain health through interactions such as picking up health power-ups. The collision system plays a pivotal role in regulating the ship's health status.

Sprite Component:

A custom Sprite component has been specifically crafted for the player. This component empowers the spaceship to dynamically alter its visual representation based on user input. This behavior is realized through a function that manipulates the animation state, enabling the ship's sprite to adapt in response to different inputs.

By integrating these systems, the spaceship embodies a holistic representation of the game's core functionalities. This design approach ensures a cohesive and interconnected gameplay experience, where user inputs, collisions, sprite animations, and events collectively contribute to the overall dynamics and responsiveness of the spaceship within the game environment.

Objectives

All outlined objectives have been successfully accomplished, with the exception of the following:

-> Additional credits section

-> The different missile power levels

-> Vertical Moving background rocks (the vertical movement is applied to every object in the game through a variable that delimits the “falling speed” per say).

Bibliography

https://www.gamedeveloper.com/programming/writing-a-game-engine-from-scratch---part-1-messaging

https://en.cppreference.com/w/cpp/chrono/high_resolution_clock/now

https://www.geeksforgeeks.org/dynamic-_cast-in-cpp/

https://www.geeksforgeeks.org/passing-a-function-as-a-parameter-in-cpp/

https://www.geeksforgeeks.org/object-delegation-in-cpp/

https://www.geeksforgeeks.org/how-to-call-function-within-function-in-c-or-c/

https://www.programiz.com/cpp-programming/library-function/cmath/sqrt

https://en.cppreference.com/w/cpp/thread/thread/thread

https://learn.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-170

https://en.cppreference.com/w/cpp/utility/functional/bind

https://en.cppreference.com/w/cpp/container/list/insert

https://en.cppreference.com/w/cpp/thread/sleep_for

https://www.positioniseverything.net/cpp-optional-parameters/

https://en.cppreference.com/w/cpp/language/operator_arithmetic

https://www.simplilearn.com/tutorials/cpp-tutorial/abstract-class-in-cpp

https://stackoverflow.com/questions/4920783/combining-enum-value-using-bitmask
https://www.youtube.com/watch?v=JnTa9Xtvmfl&t=7892s&ab_channel=freeCodeCamp.org

https://leanpub.com/linalgebra

https://www.geeksforgeeks.org/macros-and-its-types-in-c-cpp/#:~:text=A%20macro%20is%20a%20piece,the%20definition%20of%20the%20macro.

https://ocw.mit.edu/courses/2-s998-marine-autonomy-sensing-and-communications-spring-2012/resources/mit2_s998s12_lab02/