Unique Endless Horror FPS Full Game Template

Overview

The *Unique Endless Horror FPS Full Game Template* is a fully developed framework for creating an immersive and repayable first-person horror shooter. This system features dynamically generated tunnels that extend as players progress by eliminating zombies and interacting with keypads to unlock new sections.

Zombies present a unique challenge: they can only be killed when exposed to light. Players must strategically use their flashlight, throwable glow sticks, and ceiling lights to illuminate enemies before engaging in combat. Additionally, ceiling lights can be destroyed, affecting the environment and difficulty dynamically.

The player is equipped with a pistol, machine gun, and shotgun, all of which can be used to eliminate enemies. Resources such as ammo, health packs, and glow sticks can be obtained by breaking crates scattered throughout the environment.

Progress is tracked through distance travelled and total kills, which determine the final score when the player dies.

The template also includes a main menu with a login system, providing a structured starting point for game sessions.

This system provides a complete foundation for developing a horror FPS with dynamic environments, strategic combat, and an engaging scoring system.

Please note

This documentation will guide you through the largest and most vital scripts in the template. There are more than 44 scripts all in all, so I will only cover the vital ones.

Here is a list of all of the scripts :

EnemyAIManager.cs
EnemyHealthManager.cs
HitDamageManager.cs
LoginManager.cs
PickupManager.cs
PlayerVitalsManager.cs
WeaponManager.cs
EnemyAnimationManager.cs
FlashlightManager.cs
InstanciationManager.cs
MenuManager.cs
PlayerCameraManager.cs
ScoreManager.cs
EnemyAudioManager.cs
FollowClampBehaviour.cs
InterfaceAnimationManager.cs
ObjectHealthManager.cs
PlayerMovementManager.cs
ScreenShakeManager.cs
CheckMovementBehaviour.cs
DestroyAtDeathBehaviour.cs
DestroyTimerBehaviour.cs
DetachChildrenBehaviour.cs
DetachFromParentBehaviour.cs
DistanceTrackerBehaviour.cs
DoorTriggerBehaviour.cs
EndColliderDetector.cs
FlickerEmissionMaterialBehaviour.cs
FlyCameraManager.cs
ForceBehaviour.cs
FPSCounterBehaviour.cs
LightFadeBehaviour.cs
LightFlickerBehaviour.cs
LightReactionBehaviour.cs
LightTriggerBehaviour.cs
PickupIdentifyBehaviour.cs
ProjectileBehaviour.cs
RandomAudioBehaviour.cs
RandomForceBehaviour.cs
ScrollTextureBehaviour.cs
TriggerBehaviour.cs
WeaponAnimatorBehaviour.cs

WeaponSwayBehaviour.cs
ZoneColliderBehaviour.cs

Now, lets get started:

Player related setup:

PlayerMovementManager

Overview

The *PlayerMovementManager* handles all player movement mechanics, including walking, running, crouching, proning, sliding, and jumping. It also integrates gravity and smooth camera transitions for a fluid movement experience.

Features

- **Multi-State Movement:** Supports walking, running, crouching, proning, and sliding.
- **Dynamic Speed Adjustment:** Movement speed changes based on the player's stance and actions.
- **Jumping & Gravity:** Implements realistic jumping and gravity effects.
- **Smooth Camera Transition:** Adjusts the camera's position when crouching, proning, or sliding for immersion.
- **Sliding Mechanic:** Allows fast traversal for a short duration when transitioning from sprint to crouch.

How It Works

- **Movement Handling:** Processes input for directional movement, adjusting speed based on current state (walking, running, crouching, etc.).
- **Jumping & Gravity:** Applies gravity and allows jumping when grounded.
- **Crouching & Proning:** Toggles crouch or prone states, adjusting player height and speed accordingly.
- **Sliding:** Initiates a temporary high-speed movement when sprinting and crouching. Ends after a short duration or when movement stops.

- **Camera Adjustments:** The camera smoothly transitions to match movement states like crouching, proning, and sliding.

Setup Instructions

1. Attach the *PlayerMovementManager* script to the player GameObject.
2. Ensure the **CharacterController** component is attached.
3. The main camera should be set as **Camera.main** for smooth positional adjustments.
4. Configure movement values such as **walkSpeed, runSpeed, crouchSpeed**, and **jumpForce** to fit your game.

Integration Requirements

- The player GameObject must have a **CharacterController** component.
- Player input for movement (WASD or arrow keys), crouching (Left Ctrl or C), proning (Z), running (Left Shift), and jumping (Spacebar) must be assigned in Unity's Input Manager.

This system provides a complete movement foundation for a first-person horror shooter, allowing smooth and dynamic traversal through the environment.

PlayerVitalsManager

Overview

The *PlayerVitalsManager* handles the player's health, damage processing, and health UI updates. It ensures that the player can take damage, die, and trigger the end-game menu when health reaches zero.

Features

- **Health System:** Tracks and updates the player's current health.
- **Damage Handling:** Reduces health when the player takes damage and plays a random injury sound.
- **Death Handling:** Disables movement, hides the interface, and triggers a death animation when health reaches zero.
- **UI Integration:** Updates the health display dynamically.
- **End Menu Activation:** Automatically brings up the end-game menu upon death.

How It Works

- **Health Tracking:** The script updates the health UI and ensures health remains within its limits.
- **Damage Application:** When the player takes damage, health decreases, and a random pain sound plays.
- **Death Sequence:** If health reaches zero, the player loses control, a death animation plays, and after a delay, the end-game menu appears.

- **Game Pausing on Death:** The game pauses when the end menu is triggered.

Setup Instructions

1. Attach the *PlayerVitalsManager* script to the player GameObject.
2. Assign the **HealthText UI** element to display the player's current health.
3. Assign an **AudioSource** component to play damage sounds.
4. Link the **SceneManager**, **Weapons**, **Interface**, and **MainCamera** in the inspector.
5. Populate the **RandomTakeDamageSound** array with sound effects for damage feedback.

Integration Requirements

- The player GameObject must include a **Text UI element** named "HealthText" to display health.
- An **AudioSource** component is required for damage sound effects.
- The **SceneManager** must contain a **MenuManager** script to handle the end-game menu.

This system ensures the player's health mechanics are properly managed, including UI updates, death handling, and damage audio feedback.

PlayerCameraManager

Overview

The *PlayerCameraManager* controls first-person camera movement, allowing the player to look around smoothly using mouse input while enforcing vertical rotation limits.

Features

- **Mouse-Based Look Control:** Adjusts camera rotation based on mouse movement.
- **Customizable Sensitivity:** Separate sensitivity values for horizontal and vertical movement.
- **Vertical Look Limits:** Prevents the player from looking too far up or down.
- **Smooth Rotation:** Rotates the camera and player body independently for fluid movement.

How It Works

- **Mouse Input Processing:** Captures mouse movement along the X and Y axes.
- **Vertical Rotation:** Adjusts the camera's pitch while clamping the angle within defined limits.
- **Horizontal Rotation:** Rotates the player's body based on mouse movement along the X-axis.

Setup Instructions

1. Attach the *PlayerCameraManager* script to the **player's camera** GameObject.

2. Ensure the camera is parented to the player GameObject to allow body rotation.
3. Adjust **lookSpeedX** and **lookSpeedY** for preferred mouse sensitivity.
4. Set **upperLookLimit** and **lowerLookLimit** to define vertical rotation constraints.

Integration Requirements

- The camera GameObject must be a **child of the player GameObject** to allow proper body rotation.
- **Mouse input must be enabled** in Unity's Input Manager for "Mouse X" and "Mouse Y".

This system ensures smooth and responsive first-person camera control while maintaining natural player movement limits.

PickupManager

Overview

The *PickupManager* handles item pickups, updates the player's inventory, plays pickup sounds, and displays floating UI messages for collected items.

Features

- **Item Pickup Detection:** Detects and processes various pickup types (health packs, ammo, batteries, and glowsticks).
- **Dynamic UI Messages:** Displays pickup notifications that float and fade out.
- **Sound Effects:** Plays an audio cue when an item is picked up.
- **Queue-Based Messaging:** Ensures pickup messages display sequentially without overlap.

How It Works

- **Trigger Detection:** When the player enters a trigger zone with a "Pickup" tag, the system identifies the item type.
- **Item Collection:** Adds the item to the appropriate inventory system (health, ammo, flashlight battery, or glowsticks).
- **UI Updates:** Shows a temporary floating message to inform the player of the collected item.
- **Sound Playback:** Plays a pickup sound when collecting an item.

Setup Instructions

1. Attach the *PickupManager* script to the player GameObject.
2. Ensure items meant to be collected have the "Pickup" tag.
3. Assign the **PickupSound** audio clip for feedback.
4. Link the **pickupUIText** GameObject to display item messages.
5. Adjust **messageDuration, fadeDuration, and floatDistance** for UI animation preferences.

Integration Requirements

- Items must use the *PickupIdentifyBehaviour* script to define their type and pickup amount.
- The **WeaponManager, PlayerVitalsManager, and FlashlightManager** scripts must be present to store collected items.
- A UI **Text** object named "PickupText" should exist for displaying messages.
- The player must have an **AudioSource** component for pickup sounds.

This system ensures a smooth and informative item pickup experience, enhancing gameplay immersion.

FlashlightManager

Overview

The FlashlightManager oversees the flashlight's functionality, including battery usage, brightness adjustments, and toggling the light on and off during gameplay. It enhances player immersion through realistic battery management and light behavior.

Features

- Battery Consumption: Manages battery depletion when the flashlight is active.
- Light Brightness: Adjusts flashlight intensity dynamically based on remaining battery.
- Toggle Control: Allows players to turn the flashlight on/off using the "F" key.
- Battery Pickup Integration: Enables the player to recharge the flashlight with collected battery packs.
- UI Updates: Displays real-time battery status on the screen.

How It Works

1. Battery Display: Updates a UI Text object to show the current battery level.

2. Light Toggle: The flashlight's state switches between on and off when the "F" key is pressed.

3. Battery Drain: Reduces the battery level while the flashlight is active, with intensity tied to the remaining battery percentage.

4. Light Component: Enables or disables the flashlight's light source based on its state.

5. Boundary Management: Ensures battery values are within acceptable limits (0%–100%).

Setup Instructions

1. Attach the FlashlightManager script to the flashlight GameObject.

2. Ensure the flashlight GameObject has:

    - A Light component for illumination.
    - An AudioSource component for sound feedback (if used).

3. Create and name a UI Text object "FlashlightBatteryText" for battery display.

4. Assign values to these public variables in the Unity Editor:

    - BatteryDrainAmount for battery depletion rate.
    - BatteryPickupAmount for the recharge amount.
    - FlashlightBrightness for light intensity.

5. Optionally, set the FlashLightAudioClip for sound effects.

6. Link the LightTrigger GameObject to manage flashlight-related triggers.

Integration Requirements

- Ensure a UI Text object named "FlashlightBatteryText" exists in the scene.
- The Flashlight GameObject must have a Light component for proper functionality.
- Provide battery pickup items as collectable objects within the game.

WeaponManager

Overview

The WeaponManager controls all aspects of the player's weaponry, from weapon selection and firing mechanics to animations, ammunition handling, and damage application. It ensures smooth weapon transitions and immersive combat interactions with both enemies and the environment.

Features

- **Weapon Selection:** Allows switching between Pistol, Machine Gun, and Shotgun using mouse scroll or hotkeys.
- **Ammunition Management:** Tracks ammo count and magazine sizes for each weapon type.
- **Raycast Shooting:** Implements raycast-based hit detection with feedback for enemies, objects, and triggers.
- **Dynamic UI Updates:** Updates HUD elements for ammo count, weapon images, and damage indicators in real time.

- **Damage Application:** Handles specific damage types for limb, body, and head hits with visual effects.
- **Weapon Animations:** Includes animations for shooting, reloading, and weapon switching.
- **Glowstick Mechanics:** Enables glowstick throwing, with visuals and sound effects.
- **Interactive Hits:** Triggers additional behaviors for certain hit objects (e.g., enemy AI or door switches).

How It Works

1. **Weapon Switching:**
    - Weapons can be switched using the mouse wheel or numeric keys (1, 2, 3).
    - Handles animations during weapon transitions for a seamless experience.
2. **Raycast Shooting:**
    - Checks for hit objects within range and applies relevant effects such as:
        - Explosions for "Untagged" objects.
        - Damage and flinch animations for enemies.
        - Trigger-based interactions for objects like doors.
3. **Damage Application:**
    - Differentiates between hits (limb, body, head) for enemies.
    - Updates HUD damage indicators and invokes enemy AI responses.
4. **Ammunition Handling:**
    - Manages ammo depletion during firing and reloading.
    - Displays remaining ammo and magazines in the HUD.
5. **Glowstick Mechanics:**
    - Allows throwing glowsticks with associated animations and sound effects.
    - Tracks and updates remaining glowstick count in the UI.
6. **Weapon Feedback:**
    - Includes muzzle flashes, weapon-specific sounds, and hit effects for immersive combat interactions.

Setup Instructions

1. Attach the WeaponManager script to the player's weapon manager GameObject.
2. Ensure all required GameObjects are assigned in the Unity Editor:
    - Weapons (Pistol, Machine Gun, Shotgun)
    - UI elements (e.g., CrossHair, AmmunitionText, DamageImage)
    - Muzzle flash effects and explosion effects.

3. Add animations for:
    - Weapon switching (WeaponSwitch animation).
    - Flinch animations for enemies (via EnemyAnimationManager).
4. Define weapon-specific properties:
    - Damage, fire rate, and magazine sizes.
    - Muzzle points for firing effects.
5. Ensure the scene includes compatible components:
    - EnemyHealthManager for enemies.
    - ObjectHealthManager for destructible objects.
    - TriggerBehaviour for trigger interactions.

Integration Requirements

- **UI Elements:**
    - A damage indicator named "DamageImage" to display damage types.
    - Text objects for ammo count and glowstick amount.
    - Shooting helper text for player guidance.
- **Enemy Components:**
    - EnemyHealthManager for handling enemy health and damage.
    - EnemyAnimationManager for enemy response animations.
    - EnemyAIManager for AI behavior changes when hit.
- **Glowstick Objects:**
    - Models for thrown and fake glowsticks, along with a break sound effect.
- **Animations:**
    - Weapon animations for switching and reloading.
    - Aim positions for each weapon type.

Those are the most vital scripts for the players integration. Now, lets move onto enemies:

EnemyAIManager

Overview

The EnemyAIManager oversees enemy behavior, including movement, combat, and interactions with the player. It supports melee and ranged combat styles, offering dynamic challenges for players within customizable sight and attack ranges.

Features

Player Detection: Detects if the player is within sight or attack range using spherical checks.

Chase Mechanic: Makes the enemy pursue the player when detected.

Attack Behavior: Handles both melee and ranged attacks with customizable cooldowns and effects.

Animations: Triggers appropriate animations for idle, running, and attacking states.

Visual Range Indicators: Displays enemy sight and attack ranges in the Unity Editor for debugging.

Damage Application: Applies damage to the player and provides visual feedback through attack particles.

How It Works

1. **Player Detection:**
    - Uses **Physics.CheckSphere** to determine if the player is within sight or attack range.
    - Flags (playerInSightRange, playerInAttackRange) ensure proper behavior transitions.
2. **Chase Behavior:**
    - Enemy navigates towards the player using a **NavMeshAgent**.
    - Stops at the defined attack range for combat.
3. **Attack Mechanism:**
    - Executes melee or ranged attacks based on configuration (MeleeEnemy, RangedEnemy).
    - Muzzle effects or projectiles are instantiated at the **SpawnPoint** during attacks.
    - Attack cooldown prevents rapid attacks, controlled by timeBetweenAttacks and alreadyAttacked.
4. **Animations:**
    - Animation states are triggered using an **EnemyAnimationManager** (e.g., idle, running, attack).
5. **Damage Application:**
    - Melee: Uses raycast from the spawn point to detect and damage the player.
    - Ranged: Spawns projectiles that can collide with the player.

6. **Visual Gizmos:**
    - Sight and attack ranges are visualized in the Unity Editor using color-coded spheres (yellow for sight, red for attack).

Setup Instructions

1. Attach the EnemyAIManager script to enemy GameObjects.
2. Assign the following references in the Unity Editor:
    - **NavMeshAgent** for navigation.
    - **Player Transform** (e.g., "Player" object in the scene).
    - **LayerMasks** (whatIsGround, whatIsPlayer) for environment interaction.
    - **AttackParticle** prefab for visual feedback.
    - **Projectile** prefab for ranged attacks.
    - **SpawnPoint** GameObject for attack origin.
3. Configure properties:
    - timeBetweenAttacks (attack cooldown duration).
    - sightRange, attackRange (detection ranges).
    - Damage (attack strength).
    - Toggle MeleeEnemy or RangedEnemy to set attack type.

Integration Requirements

- The player GameObject should have a **PlayerVitalsManager** script to process damage.
- Ranged enemies require a **Projectile** prefab for proper attack functionality.
- Ensure the **NavMesh** is baked for the scene, and the enemy GameObject has a valid **NavMeshAgent** component.
- Attach an **EnemyAnimationManager** script to control animations.

Here's the documentation for your **EnemyAnimationManager** script:

EnemyAnimationManager

Overview

The EnemyAnimationManager controls enemy animations, such as idle, running, attacking, and flinching. It dynamically adjusts the enemy's appearance based on actions or states, adding polish and realism to gameplay interactions.

Features

- **Action-Based Animations:** Plays appropriate animations for attack, run, idle, or flinch states.
- **Smooth Transitions:** Ensures seamless animation transitions using Unity's **CrossFade**.

- **Customizable Animation Names:** Allows developers to define unique animation clips for each state.

How It Works

1. **Flinch Animation:**
    - Triggers a single flinch animation to convey enemy reaction when hit.
    - Uses GetComponent<Animation>().Play() for direct playback.
2. **Attack Animation:**
    - Initiates the enemy's attack animation with a crossfade effect to ensure smooth transition.
3. **Run Animation:**
    - Engages the running animation with a crossfade effect while chasing the player.
4. **Idle Animation:**
    - Sets the idle animation when the enemy is not interacting with the player or moving.

Setup Instructions

1. Attach the EnemyAnimationManager script to the enemy GameObjects.
2. Assign the appropriate animation clips in the Unity Editor:
    - **FlinchAnimation** (e.g., "EnemyFlinch").
    - **AttackAnimation** (e.g., "EnemyAttack").
    - **RunAnimation** (e.g., "EnemyRun").
    - **IdleAnimation** (e.g., "EnemyIdle").
3. Ensure the enemy GameObjects have an **Animation** component.
4. Assign animation clips to the **Animation** component via the Unity Editor.

Integration Requirements

- Enemy GameObjects need an **Animation** component with animation clips for flinch, attack, run, and idle states.
- The EnemyAnimationManager script should be paired with other scripts managing AI behavior (e.g., EnemyAIManager) to synchronize animations with actions.

EnemyHealthManager

Overview

The EnemyHealthManager controls the health, damage, healing, and death mechanics of enemies. It integrates with other gameplay elements such as ragdoll effects, score tracking, and game state management to ensure smooth and immersive interactions.

Features

- **Health Management:** Tracks current, maximum, and minimum health levels for the enemy.
- **Damage Application:** Reduces health based on incoming damage and explosion impacts.
- **Healing:** Allows regeneration of health up to maximum limits.
- **Death Handling:** Manages enemy death through ragdoll instantiation and kill count tracking.
- **Game State Integration:** Updates player scores and game state using the SceneManager.
- **Damage Control:** Enforces conditional damage based on the enemy's current status.

How It Works

1. **Health Updates:**
    - Health is decreased when damage is applied via attacks or explosions.
    - Health is increased through healing mechanics.
2. **Death Mechanism:**
    - When health reaches zero, the enemy "dies" and a ragdoll effect is instantiated.
    - The enemy is destroyed after confirming its death state.
3. **Score and State Management:**
    - Increments the kill count in the **ScoreManager** upon enemy death.
    - Integrates with **SceneManager** to maintain gameplay progression.
4. **Conditional Damage:**
    - Damage is only applied when the CanTakeDamage flag is true, enabling controlled interactions.

Setup Instructions

1. Attach the EnemyHealthManager script to enemy GameObjects.
2. Configure the following properties in the Unity Editor:
    - **Health:** Set initial health values for the enemy.
    - **MaxHealth:** Define the upper limit for enemy health.
    - **MinHealth:** Specify the minimum allowable health.
    - **Ragdoll:** Assign a ragdoll prefab for death visuals.
3. Add a **ScoreManager** component to the SceneManager GameObject to track kills.
4. Ensure the player GameObject is tagged as "Player" and includes a **PlayerVitalsManager** for damage handling.

Integration Requirements

- **SceneManager:** Must include a **ScoreManager** to process enemy kills.
- **PlayerVitalsManager:** Required on the player GameObject to apply damage from enemies.
- **Ragdoll Prefab:** Represents the enemy upon death; must be assigned in the Unity Editor.

Now, the score manager to keep track of how the player is doing:

ScoreManager

Overview

The ScoreManager tracks player progress in terms of distance, score, and kills while updating related UI elements. It calculates high scores, saves them for future sessions using PlayerPrefs, and displays the results in both in-game and end-menu screens.

Features

- **Performance Tracking:** Monitors distance traveled, kills, and total score.
- **Dynamic UI Updates:** Displays real-time values for kills, distance, and score during gameplay.
- **High Score Management:** Saves and retrieves high scores for kills, distance, and total score.
- **End Menu Integration:** Updates the end menu with the player's performance and high scores.
- **Menu and Level Modes:** Supports score display in both menu and gameplay environments.

How It Works

1. **Game Modes:**
    - **Menu Mode:** Displays saved high scores using PlayerPrefs.
    - **Level Mode:** Calculates score based on distance and kills with a multiplier, updates UI, and checks for new high scores.
2. **High Score Management:**
    - Uses PlayerPrefs to persist high scores across sessions.
    - Updates saved values if the current score exceeds the high score.
3. **UI Updates:**
    - Dynamically updates kill count, distance traveled, and total score during gameplay.
    - Displays final scores and high scores in the end menu.
4. **Distance Tracking:**
    - Calculates distance using the **DistanceTrackerBehaviour** component on the player GameObject.

Setup Instructions

1. Attach the ScoreManager script to a designated GameObject in the scene.
2. Assign the following UI elements in the Unity Editor:
    - **KillsText**, **DistanceText**, and **ScoreText** for gameplay display.
    - **SavedKillsText**, **SavedDistanceText**, and **SavedScoreText** for menu high scores.
    - **EndMenuKillsText**, **EndMenuDistanceText**, and **EndMenuScoreText** for the end menu.
3. Set up **PlayerPrefs** keys for saving high scores:
    - "KillsSaved"
    - "DistanceSaved"
    - "ScoreSaved"
4. Ensure the player GameObject has a **DistanceTrackerBehaviour** component to calculate distance.

Integration Requirements

- **MenuManager:** The end menu must be managed by a **MenuManager** script to check active states.
- **Player GameObject:** Must have the "Player" tag for ScoreManager to locate it.
- **DistanceTrackerBehaviour:** Required for tracking distance traveled.

Now, let's see how we handle spawning in the next sections when we shoot at doors switches:

TriggerBehaviour, DoorTriggerBehaviour, and EndColliderDetector

Overview

These scripts manage interactions with environmental triggers to enhance gameplay:

- **TriggerBehaviour** enables dynamic environment interactions, including door states, section instantiation, and cleanup of unnecessary sections.
- **DoorTriggerBehaviour** automates door opening and closing based on player proximity.
- **EndColliderDetector** handles end-level trigger events, such as displaying the end menu.

Features

TriggerBehaviour

- **Door Interaction:** Switches door states (open/close) with animations.
- **Section Management:** Spawns new sections and removes sections behind the player for optimization.
- **Player Detection:** Triggers actions only when the player interacts with the collider.
- **Emission Effect:** Visually indicates an activated switch by changing its color to green.

DoorTriggerBehaviour

- **Automatic Door Control:** Opens or closes doors when the player enters or exits the trigger zone.
- **Animation Handling:** Prevents overlapping animations and ensures smooth transitions.
- **Multi-Door Support:** Differentiates logic for managing multiple types of doors.

EndColliderDetector

- **End-Level Trigger:** Displays the end menu when the player interacts with a designated collider.
- **Dynamic Scene Management:** Finds and interacts with the **SceneManager** for handling end-level functionality.

How It Works

TriggerBehaviour

1. **OnTriggerEnter and OnTriggerStay:**
    - Detects when the player enters the trigger zone.
    - Handles various trigger types, such as:
        - **Behind Door Trigger:** Closes doors when the player passes through.
        - **Section Destroy Trigger:** Deletes unnecessary sections once the player moves past them.
2. **SwitchDoor Method:**
    - Changes the door state to open by playing the assigned animation.
    - Spawns a new section at a designated spawn point via the **InstanciationManager**.
3. **Visual Feedback:**
    - Uses emissive material properties to indicate an activated switch with a green glow.

DoorTriggerBehaviour

1. **OnTriggerEnter:**
    - Detects player entry into the collider and opens the door with an animation.
2. **OnTriggerExit:**
    - Detects player exit and closes the door using the appropriate animation.

EndColliderDetector

1. **OnTriggerEnter:**

- Triggers the **ShowEndMenu** method from the **MenuManager** when the player interacts with the "EndCollider" object.

2. **Update Function:**
    - Continuously locates the **SceneManager** GameObject in the scene.

Setup Instructions

TriggerBehaviour

1. Attach the TriggerBehaviour script to the desired trigger zone GameObject.
2. Assign the following in the Unity Editor:
    - **IsDoorSwitch, IsBehindDoorTrigger, IsBehindSectionDestroyTrigger:** Set the appropriate flags for trigger functionality.
    - **DoorToSwitch:** Assign the door GameObject to control its state.
    - **SectionSpawnPoint:** Define the spawn location for new sections.
    - **SceneManager:** Assign the SceneManager GameObject or ensure it can be dynamically located.
3. Ensure doors have properly configured animations.

DoorTriggerBehaviour

1. Attach the DoorTriggerBehaviour script to the door trigger zone GameObject.
2. Assign door-specific parameters, such as animation clips and state flags.

EndColliderDetector

1. Attach the EndColliderDetector script to the end-level trigger zone GameObject.
2. Assign the **SceneManager** and ensure it includes the **MenuManager** component with the **ShowEndMenu** method.

Integration Requirements

- **SceneManager with MenuManager:** Required for both TriggerBehaviour and EndColliderDetector to manage scene transitions.
- **InstanciationManager:** Used by TriggerBehaviour to handle dynamic section spawning.
- **Animation Components:** Doors and switches must have assigned animations for smooth transitions.
- **Player GameObject:** Requires the "Player" tag for detection.

InstanciationManager

Overview

The InstanciationManager is responsible for dynamically spawning game sections at a designated location. It provides a straightforward and flexible way to randomize gameplay elements or environment setups by selecting and instantiating random sections from a predefined array.

Features

- **Random Section Instantiation:** Spawns a randomly selected section from an array of GameObjects.
- **Dynamic Placement:** Positions the instantiated section precisely at the specified spawn point.
- **Debug Safeguards:** Warns developers if the sections array or spawn point is not properly configured.

How It Works

1. **Section Selection:**
    - A random index is generated using Unity's Random.Range method.
    - The corresponding section in the **sections** array is selected.
2. **Instantiation:**
    - The chosen section is instantiated at the position and rotation of the assigned **spawnPoint**.
3. **Debug Checks:**
    - Validates that the **sections** array and **spawnPoint** are assigned to prevent runtime errors.
    - Logs a warning if configuration issues are detected.

Setup Instructions

1. Attach the InstanciationManager script to a GameObject in your scene (e.g., a manager GameObject).
2. Assign the following in the Unity Editor:
    - **sections:** Populate this array with the GameObjects to be instantiated (e.g., environment sections or gameplay elements).
    - **spawnPoint:** Specify the Transform where sections should be instantiated.
3. Ensure that the GameObjects in the **sections** array are properly configured with required components and positioning.

Integration Requirements

- The **sections** array should contain prefabs of GameObjects to be instantiated dynamically.
- The **spawnPoint** Transform must be properly assigned to define the location of instantiation.
- External scripts or triggers must call the SpawnSection method to initiate spawning.

Then, there are also other scripts that are vital to the function of the game:

ObjectHealthManager

Overview

The ObjectHealthManager script is responsible for managing the health of game objects. It supports destruction mechanics for explosive objects, optional pickup drops, and visual feedback effects such as explosions. This enhances gameplay with interactive and dynamic environmental elements.

Features

- **Health Management:** Tracks and updates the health of an object.
- **Destruction Effects:** Handles destruction visuals like explosions when health reaches zero.
- **Pickup Drops:** Supports optional random pickup generation upon object destruction.
- **Explosive Objects:** Differentiates between explosive and non-explosive objects to control effects upon destruction.

How It Works

1. **Health Tracking:**
    - Monitors the health variable of the object.
    - When Health is zero or below, the object is destroyed.
2. **Destruction Mechanism:**
    - For explosive objects, instantiates the **PiecesExplosion** effect at the object's position.
    - Optionally generates a random pickup if the IsPickups flag is enabled.
3. **Pickup Generation:**
    - Chooses a random pickup from the **Pickups** array.
    - Instantiates the selected pickup slightly above the object's position.
4. **Damage Handling:**
    - Reduces the object's health based on the damage received via the ApplyDamage method.

Setup Instructions

1. Attach the ObjectHealthManager script to destructible objects in the scene.
2. Assign the following in the Unity Editor:
    - **Health:** Set the initial health value for the object.
    - **IsExplosive:** Enable this flag if the object is explosive.
    - **IsPickups:** Enable this flag if the object should drop pickups.

- **PiecesExplosion:** Assign the prefab for the explosion effect.
- **Pickups:** Populate the array with possible pickup prefabs.

3. Ensure that each pickup prefab is configured correctly in the scene.

Integration Requirements

- **PiecesExplosion Prefab:** The explosion effect used when the object is destroyed must be assigned.
- **Pickups Array:** Includes prefabs of potential pickups dropped after destruction.
- **ApplyDamage Calls:** Ensure external scripts or interactions call ApplyDamage to reduce the object's health.

LightTriggerBehaviour

Overview

The LightTriggerBehaviour script dynamically manages interactions between light sources and enemies within a specified radius. It enables or disables enemy vulnerability to damage based on their proximity to the light, adding a layer of strategy to gameplay mechanics.

Features

- **Enemy Detection:** Continuously tracks enemies within a specified light radius.
- **Damage Enable/Disable:** Makes enemies damageable while inside the light range and reverts them to non-damageable when outside the range.
- **Dynamic Reaction:** Triggers light-related behaviors in enemies via the **LightReactionBehaviour** and **EnemyHealthManager** components.
- **Debug Visualization:** Displays the light radius in the Unity Editor using Gizmos for easier debugging.

How It Works

1. **Continuous Detection:**
    - Scans for nearby colliders within the **lightRadius** using Physics.OverlapSphere.
    - Identifies enemies based on the **enemyTag** (e.g., "Enemy").
2. **Light Interaction:**
    - **Enable Damage:** Marks enemies within the light radius as damageable by updating their **LightReactionBehaviour** and **EnemyHealthManager** components.

- **Disable Damage:** Reverts enemies to non-damageable once they leave the light radius or the light source is destroyed or disabled.

3. **Cleanup:**
    - When the light object is destroyed or disabled, all previously detected enemies are set to non-damageable.
4. **Debugging:**
    - Uses Gizmos.DrawWireSphere to visually represent the light radius in the Unity Editor.

Setup Instructions

1. Attach the LightTriggerBehaviour script to a light source GameObject.
2. Configure the following properties in the Unity Editor:
    - **lightRadius:** Define the range within which enemies are affected.
    - **enemyTag:** Specify the tag used to identify enemies (e.g., "Enemy").
3. Ensure enemies in the scene have:
    - The **LightReactionBehaviour** script to handle light-triggered reactions.
    - The **EnemyHealthManager** script to manage damageability.

Integration Requirements

- **Enemy Setup:** Enemies should have the appropriate components:
    - **LightReactionBehaviour** for custom light-triggered logic.
    - **EnemyHealthManager** for enabling/disabling damageability.
- **Light Source:** Ensure the light source is placed correctly to affect nearby enemies.
- **Debugging Needs:** Use the light radius Gizmo visualization for placement and range adjustment.

Those were the most vital scripts for gameplay functionality and interaction. Now, lets move onto the menu scripts :

LogInManager and MenuManager

Overview

The **LogInManager** and **MenuManager** scripts manage core menu functionalities:

- **LogInManager:** Handles user input for usernames and enables or disables the Start button based on input validity.
- **MenuManager:** Oversees game menu transitions, pause/resume functionality, and scene navigation, creating a smooth and user-friendly experience.

Features

LogInManager

- **Username Persistence:** Saves the entered username in PlayerPrefs for future sessions.
- **Dynamic Button Activation:** Enables or disables the Start button based on username input validity.
- **User-Friendly Input:** Auto-populates the username field with the saved value when available.

MenuManager

- **Game State Control:** Manages pause, start, and end menus during gameplay.
- **Scene Navigation:** Allows transitions between different scenes (e.g., Main Menu, levels).
- **Pause and Resume:** Handles pausing and resuming gameplay, including cursor visibility and player control states.
- **Menu Integration:** Coordinates visibility of menus based on gameplay state.
- **URL Handling:** Opens specific URLs in the user's default browser.
- **Quit Functionality:** Exits the application when requested.

How It Works

LogInManager

1. **Start Function:**
    - Retrieves the saved username from PlayerPrefs and sets it in the input field.
2. **Update Function:**
    - Continuously monitors the input field.
    - Updates the PlayerPrefs with the current username.
    - Enables or disables the Start button based on whether the input field is empty.

MenuManager

1. **Start Function:**
    - Initializes the game state and menu visibility.
    - Ensures player controls are disabled during menus.
    - Locks or unlocks the cursor depending on the menu state.
2. **Update Function:**

- Monitors for Escape key presses to toggle pause/unpause functionality.

3. **Scene and Menu Management:**
    - Switches between menus (Pause, Start, End) based on game events.
    - Provides methods for restarting the level, switching to the Main Menu, or opening new scenes.
4. **Additional Features:**
    - Implements methods for opening browser URLs and quitting the game.

Setup Instructions

LogInManager

1. Attach the **LogInManager** script to a GameObject in the menu scene.
2. Assign the following in the Unity Editor:
    - **UserName:** Link to the InputField component for username input.
    - **StartButton:** Reference the Start button GameObject to enable/disable interactability.

MenuManager

1. Attach the **MenuManager** script to a GameObject in the menu or gameplay scene.
2. Assign the following in the Unity Editor:
    - **PauseMenu, StartMenu, EndMenu:** Reference the respective menu GameObjects.
    - **Player:** Link the player GameObject for enabling/disabling player controls.
    - **URL1, URL2:** Add the desired URLs for redirection.
3. Configure scenes to include the required menus and UI elements:
    - Pause, Start, End menus should be properly linked.

Integration Requirements

- **UI Elements for LogInManager:**
    - The username input field must be configured with the **InputField** component.
    - The Start button must have an **Button** component for interactivity.
- **SceneManager Setup for MenuManager:**
    - Ensure all scenes (e.g., MainMenu, levels) are included in the build settings.
    - The player GameObject must have scripts for camera and weapon management to toggle controls effectively.

Conclusion

This template is a versatile and robust start for any fps or horror-based game and provides you with everything you need to make your own game. With robust yet flexible scripts that are fully documented so you can understand the way they work and alter them to suit your needs. The template also comes with many art and audio assets as well as an example main menu and premade game scene. This package truly offers great value to anyone looking to make an FPS game!

Contact information

For any queries or questions as well as suggestions please reach me at Johannesnienaber@gmail.com for email or through the Aligned Games studio website here https://alignedgames.mailchimpsites.com/

Thank you for purchasing the Template package, I hope you enjoy using it and make an awesome game with it!