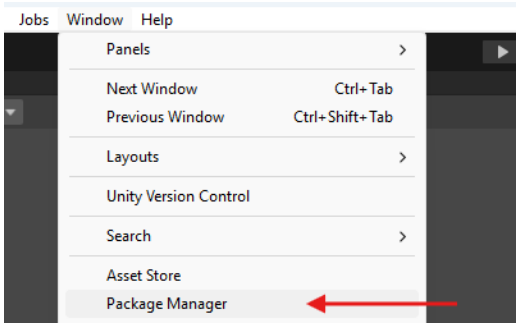
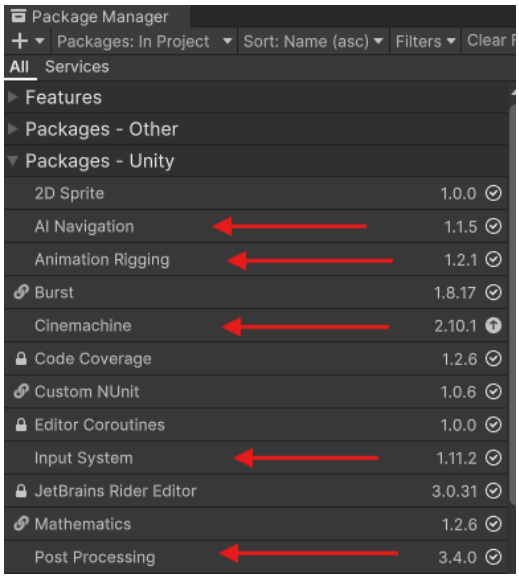

HORROR
SYSTEM

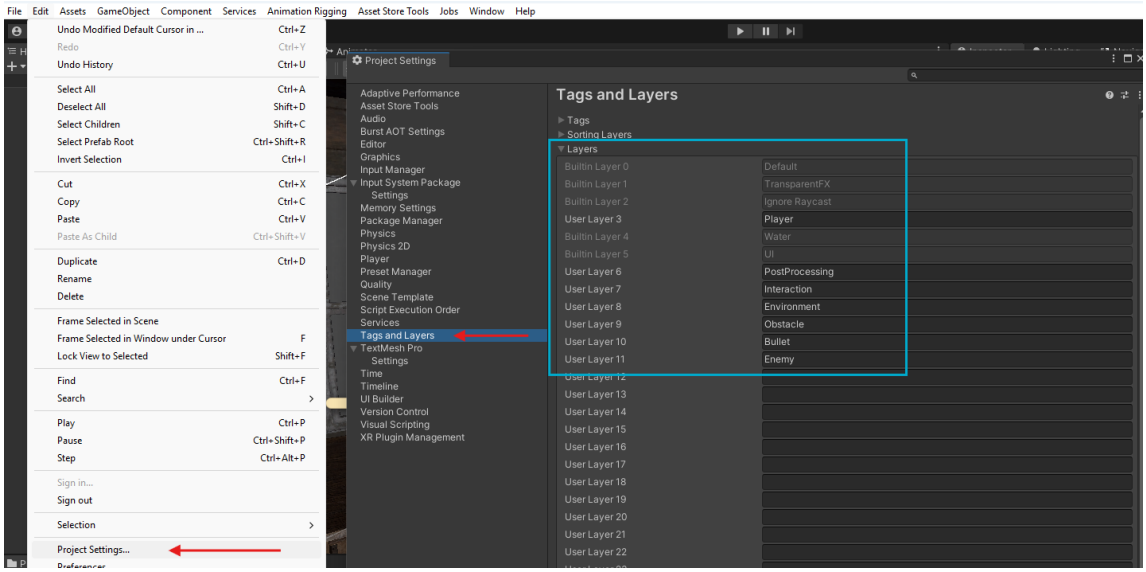
IMPORT ASSET

Create a new project. Open the Package Manager and import the following packages: **Animation Rigging, Input System, Post Processing, Cinemachine, AI Navigation.**



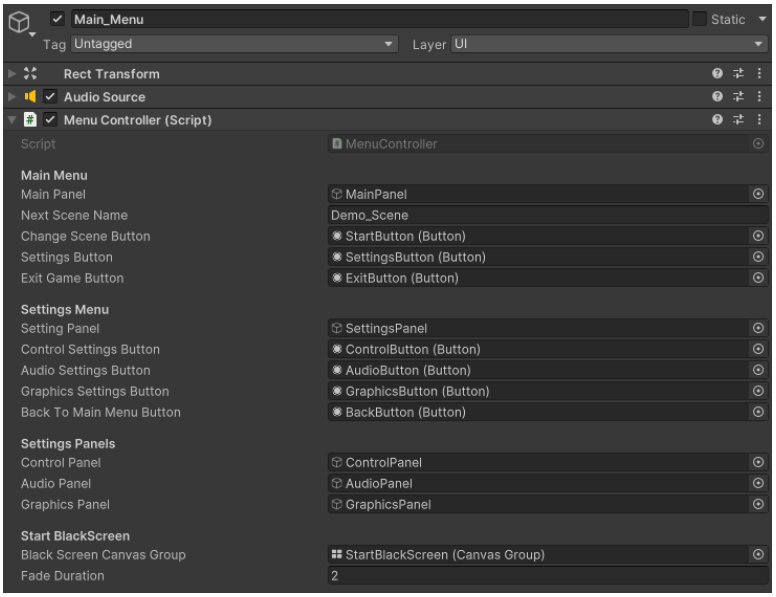


Navigate to **Edit → Project Settings → Tags and Layers** and verify the following layers exist: **PostProcessing, Interaction, Obstacle, Bullet, Enemy.**



MENU

Navigate to **Prefabs → UI → Menu** folder and open the **Main_Menu** prefab. The **MenuController** component manages the menu, enabling the correct panels when buttons are pressed.



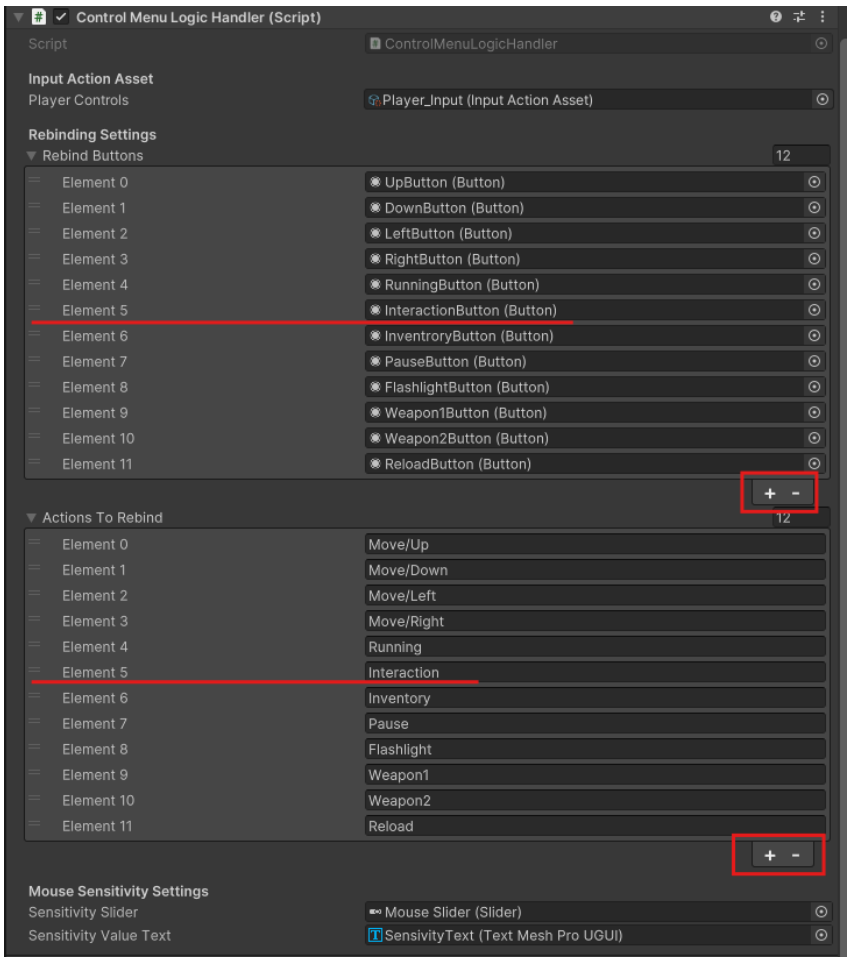
In the **Main_Menu** hierarchy, select **ControlPanel** and enable it.
Inside **ControlPanel**, locate the **Content** object. It contains two child objects:

- **Mouse Sensitivity** – contains a slider for adjusting mouse sensitivity.
- **Rebind Buttons** – holds buttons that allow key rebinding when clicked.

Adding a New Button:

1. **Duplicate** the last button in the **Rebind Buttons** list and adjust its position.
2. Select **ControlPanel** and in the **ControlMenuLogicHandler** component:
    - Add a new entry to **Rebind Buttons** and drag your newly created button into it.
    - Add a corresponding entry to **Actions To Rebind** and enter the **Input System Action name**.

Important: The index of the new **Rebind Buttons** entry must match the index in **Actions To Rebind**.



After making all the changes, exit the **Main_Menu** prefab – the changes will save automatically. Important: All modifications made to the **Main_Menu** prefab must also be applied to the **Pause_Menu** prefab.

Loading Screen

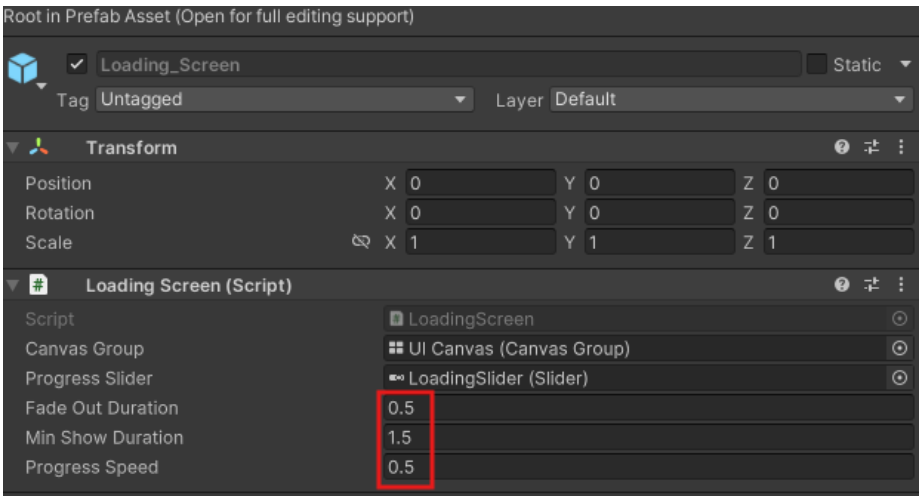
The **Loading_Screen** prefab is created once and persists between scenes (DontDestroyOnLoad). It's important that the **Loading_Screen** prefab is placed in the **Resources** folder and that the prefab name is correctly specified in the **SceneLoader** class.

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections;

2 references
public static class SceneLoader
{
    2 references
    public static void LoadScene(string sceneName)
    {
        if (LoadingScreen.Instance == null)
        {
            GameObject loadingScreenPrefab = Resources.Load<GameObject>("Loading_Screen");
            GameObject.Instantiate(loadingScreenPrefab);
        }

        LoadingScreen.Instance.Show();
        LoadingScreen.Instance.StartCoroutine(LoadSceneAsync(sceneName));
    }
```

SceneLoader is a static class and does not need to be attached to any GameObjects.
In **MenuController**, assign the correct action (SceneLoader.LoadScene) to trigger when a button is clicked

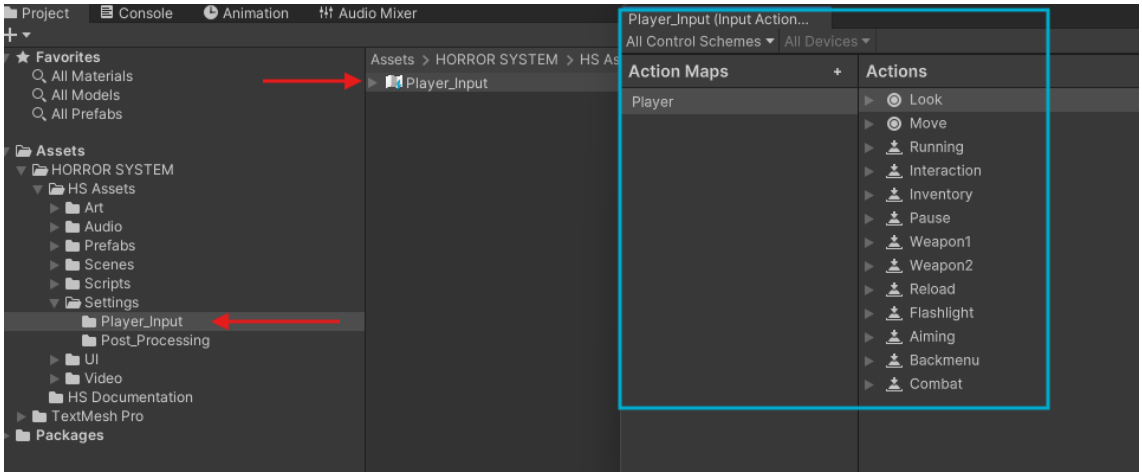
```
private void ChangeScene()
{
    changeSceneButton.onClick.AddListener(() =>
    {
        SceneLoader.LoadScene(nextSceneName);
        Time.timeScale = 1;
    });
}

1 reference
private void RestartScene()
{
    if (restartButton != null)
    {
        restartButton.onClick.AddListener(() =>
        {
            SceneLoader.LoadScene(SceneManager.GetActiveScene().name);
            Time.timeScale = 1;
        });
    }
}
```

Click on the **Loading_Screen** prefab, and in the **LoadingScreen** component, you can make the necessary adjustments.

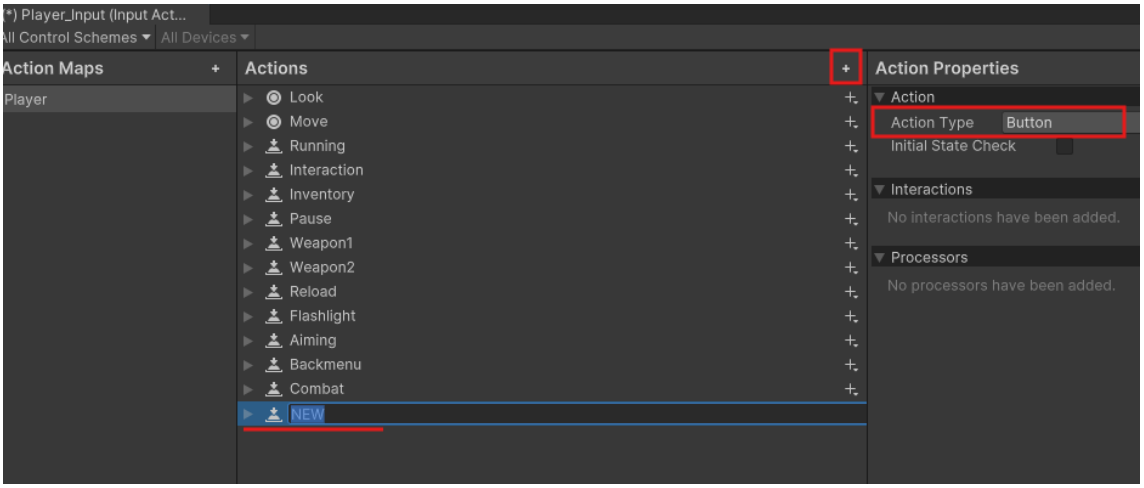


INPUT SYSTEM

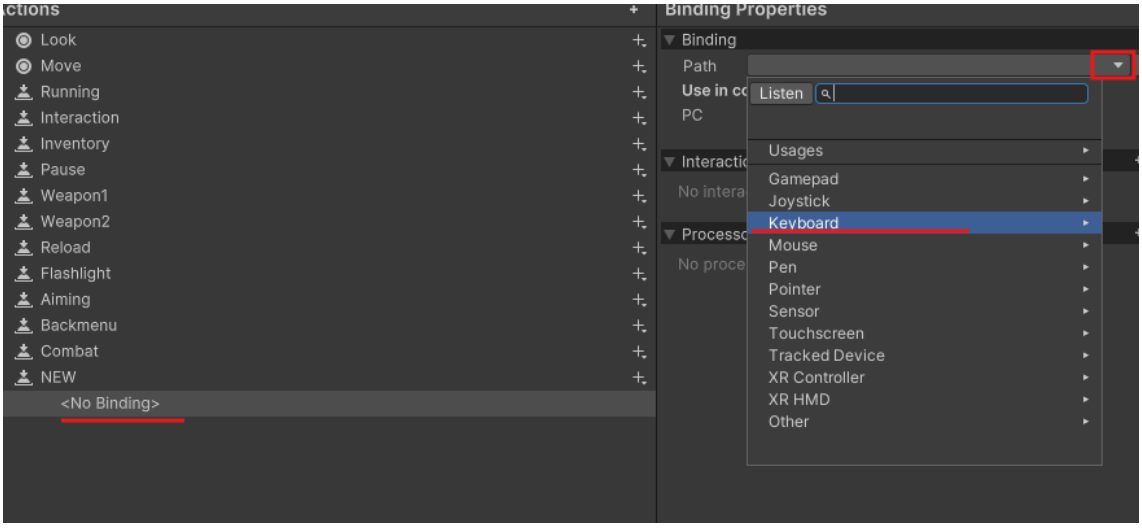
Navigate to **Settings → Player_Input** and open **Player_Input.**



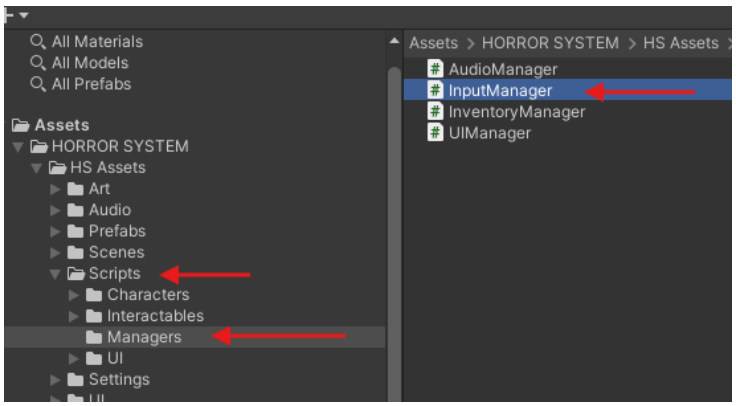
Click the **"+"** button to add a new action, select an **Action Type**, and specify the name for the new action.



Now select the binding path.



Navigate to **Scripts → Managers** and open **InputManager**. Add a new action following the same pattern as the existing ones.




```
    [Header("Action Name References")]
    [SerializeField] private string look = "Look";
    [SerializeField] private string move = "Move";
    [SerializeField] private string running = "Running";
    [SerializeField] private string interaction = "Interaction";
    [SerializeField] private string inventory = "Inventory";
    [SerializeField] private string pause = "Pause";
    [SerializeField] private string flashlight = "Flashlight";
    [SerializeField] private string aiming = "Aiming";
    [SerializeField] private string backmenu = "Backmenu";
    [SerializeField] private string combat = "Combat";
    [SerializeField] private string weapon1 = "Weapon1";
    [SerializeField] private string weapon2 = "Weapon2";
    [SerializeField] private string reload = "Reload";
    [SerializeField] private string test = "New";

    private InputAction lookAction;
    private InputAction moveAction;
    private InputAction runningAction;
    private InputAction interactionAction;
    private InputAction inventoryAction;
    private InputAction pauseAction;
    private InputAction flashlightAction;
    private InputAction aimingAction;
    private InputAction backmenuAction;
    private InputAction combatAction;
    private InputAction weapon1Action;
    private InputAction weapon2Action;
    private InputAction reloadAction;
    private InputAction testAction;
```

```
4 references
public Vector2 LookInput { get; private set; }
4 references
public Vector2 MoveInput { get; private set; }
3 references
public float RunningValue { get; private set; }
4 references
public float AimingValue { get; private set; }
5 references
public float CombatValue { get; private set; }
4 references
public bool InteractionTriggered { get; private set; }
3 references
public bool BackMenuTriggered { get; private set; }
4 references
public bool InventoryTriggered { get; private set; }
4 references
public bool PauseTriggered { get; private set; }
4 references
public bool FlashlightTriggered { get; private set; }
4 references
public bool Weapon1Triggered { get; private set; }
4 references
public bool Weapon2Triggered { get; private set; }
4 references
public bool ReloadTriggered { get; private set; }
0 references
public bool TestTriggered { get; private set; }
```

```
85  private void InitializeInputActions()
86  {
87      lookAction = playerControls.FindActionMap(actionMapName).FindAction(look);
88      moveAction = playerControls.FindActionMap(actionMapName).FindAction(move);
89      runningAction = playerControls.FindActionMap(actionMapName).FindAction(running);
90      interactionAction = playerControls.FindActionMap(actionMapName).FindAction(interaction);
91      inventoryAction = playerControls.FindActionMap(actionMapName).FindAction(inventory);
92      pauseAction = playerControls.FindActionMap(actionMapName).FindAction(pause);
93      flashlightAction = playerControls.FindActionMap(actionMapName).FindAction(flashlight);
94      aimingAction = playerControls.FindActionMap(actionMapName).FindAction(aiming);
95      backmenuAction = playerControls.FindActionMap(actionMapName).FindAction(backmenu);
96      combatAction = playerControls.FindActionMap(actionMapName).FindAction(combat);
97      weapon1Action = playerControls.FindActionMap(actionMapName).FindAction(weapon1);
98      weapon2Action = playerControls.FindActionMap(actionMapName).FindAction(weapon2);
99      reloadAction = playerControls.FindActionMap(actionMapName).FindAction(reload);
00      testAction = playerControls.FindActionMap(actionMapName).FindAction(test);
01
02      RegisterInputActions();
03  }
```

```
105 private void RegisterInputActions()
106 {
107     lookAction.performed += context => LookInput = context.ReadValue<Vector2>();
108     lookAction.canceled += context => LookInput = Vector2.zero;
109
110     moveAction.performed += context => MoveInput = context.ReadValue<Vector2>();
111     moveAction.canceled += context => MoveInput = Vector2.zero;
112
113     runningAction.performed += context => RunningValue = context.ReadValue<float>();
114     runningAction.canceled += context => RunningValue = 0f;
115
116     aimingAction.performed += context => AimingValue = context.ReadValue<float>();
117     aimingAction.canceled += context => AimingValue = 0f;
118
119     combatAction.performed += context => CombatValue = context.ReadValue<float>();
120     combatAction.canceled += context => CombatValue = 0f;
121
122     interactionAction.started += context => InteractionTriggered = true;
123     interactionAction.canceled += context => InteractionTriggered = false;
124
125     inventoryAction.started += context => InventoryTriggered = true;
126     inventoryAction.canceled += context => InventoryTriggered = false;
127
128     pauseAction.started += context => PauseTriggered = true;
129     pauseAction.canceled += context => PauseTriggered = false;
130
131     backmenuAction.started += context => BackMenuTriggered = true;
132     backmenuAction.canceled += context => BackMenuTriggered = false;
133
134     flashlightAction.started += context => FlashlightTriggered = true;
135     flashlightAction.canceled += context => FlashlightTriggered = false;
136
137     weapon1Action.started += context => Weapon1Triggered = true;
138     weapon1Action.canceled += context => Weapon1Triggered = false;
139
140     weapon2Action.started += context => Weapon2Triggered = true;
141     weapon2Action.canceled += context => Weapon2Triggered = false;
142
143     reloadAction.started += context => ReloadTriggered = true;
144     reloadAction.canceled += context => ReloadTriggered = false;
145
146     testAction.started += context => TestTriggered = true;
147     testAction.canceled += context => TestTriggered = false;
148 }
149
```

```
1 reference
private void EnableInputActions()
{
    lookAction.Enable();
    moveAction.Enable();
    runningAction.Enable();
    interactionAction.Enable();
    inventoryAction.Enable();
    pauseAction.Enable();
    flashlightAction.Enable();
    backmenuAction.Enable();
    aimingAction.Enable();
    combatAction.Enable();
    weapon1Action.Enable();
    weapon2Action.Enable();
    reloadAction.Enable();
    testAction.Enable();
}

1 reference
private void DisableInputActions()
{
    lookAction.Disable();
    moveAction.Disable();
    runningAction.Disable();
    interactionAction.Disable();
    inventoryAction.Disable();
    pauseAction.Disable();
    flashlightAction.Disable();
    aimingAction.Disable();
    combatAction.Disable();
    backmenuAction.Disable();
    weapon1Action.Disable();
    weapon2Action.Disable();
    reloadAction.Disable();
    testAction.Disable();
}
```

```
public void SetReloadTriggered(bool value)
{
    ReloadTriggered = value;
}

0 references
public void SetTestTriggered(bool value)
{
    TestTriggered = value;
}
```

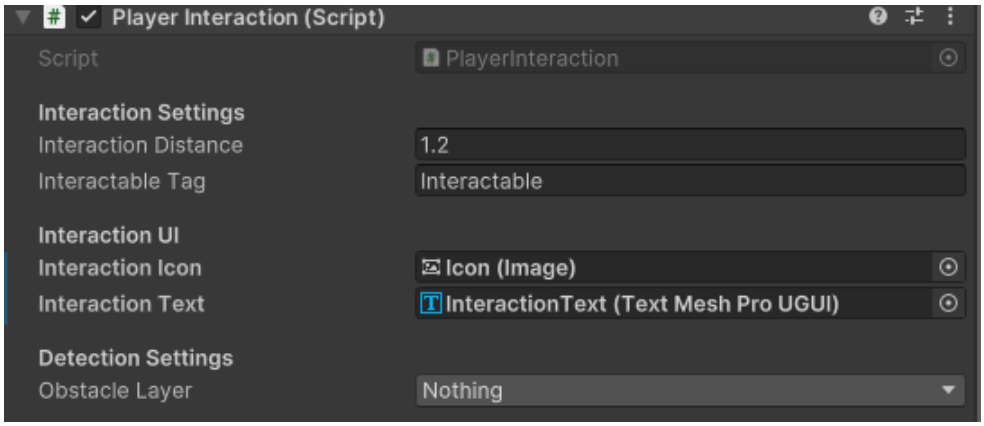
Input example: inventory toggle implementation

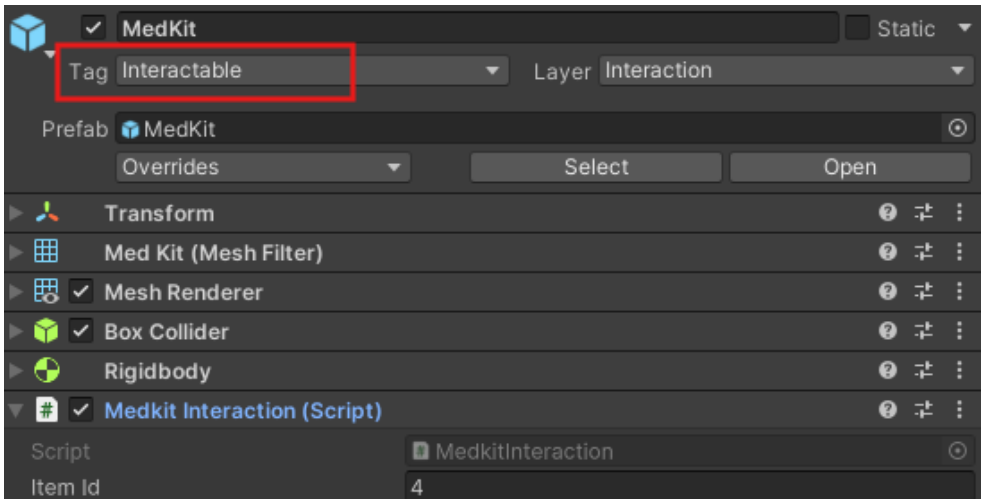
```
if (InputManager.instance.InventoryTriggered)
{
    SetInventoryPanel();
    InputManager.instance.SetInventoryTriggered(false);
}
```

INTERACTION

In the new version of the asset, a new interaction system with objects is implemented. Unlike the previous trigger-based interaction, the interaction now works based on distances from the object. Now, interaction occurs with an object that has a specific tag when the player is within a certain distance from it.

On the scene, click on the character and find the **PlayerInteraction** component. There, you can configure the interaction distance, the object tag, and the icon. At the bottom, you will see **ObstacleLayer**—this is the layer that blocks interaction. For example, if there is a wall in front of the object, the wall should be assigned to this layer to prevent interaction through it.



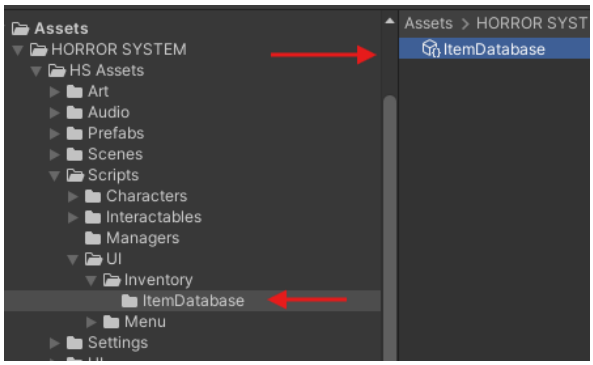


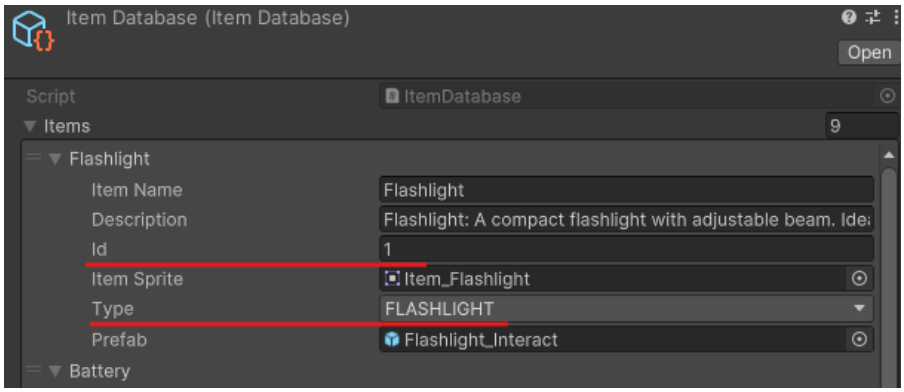
On the actual object we can interact with, in addition to the tag, there must also be an **InteractiveItem** component. When adding a new object to the project, a separate component should be written for it following the same approach.
When configuring the object, you need to assign it an ID, the pickup amount (if applicable), sounds, and how we can interact with it through the inventory.

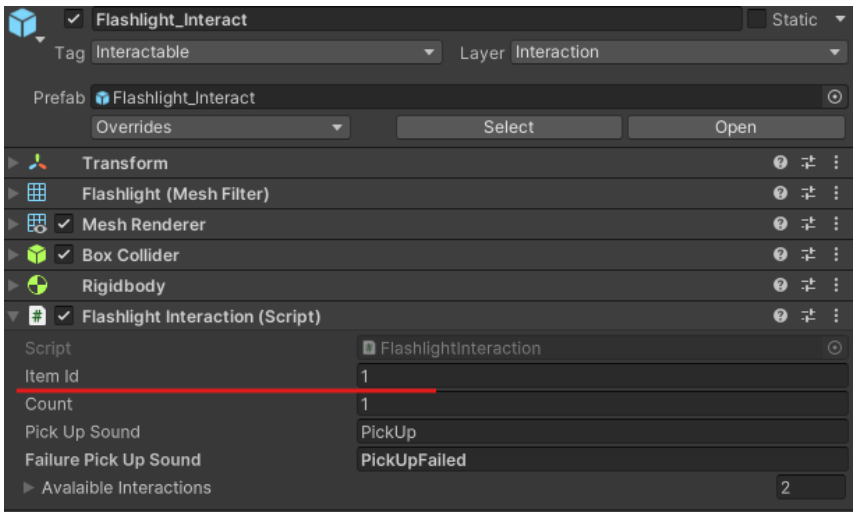
```
using UnityEngine;

Unity Script (1 asset reference) | 0 references
public class MedkitInteraction : InteractiveItem
{
    private PlayerHealth playerHealth;
    private InventorySystem playerInventory;

    12 references
    public override void Interact(GameObject player = null)
    {
        base.Interact(player);

        this.playerHealth = player.GetComponent<PlayerHealth>();
        this.playerInventory = player.GetComponent<InventorySystem>();

        var itemHasBeenAdded = playerInventory.AddItem(itemId, count, this);
        if (itemHasBeenAdded)
        {
            AudioManager.instance.Play(pickUpSound);
            Destroy(0);
        }
        else
        {
            FailurePickUp();
        }
    }
```

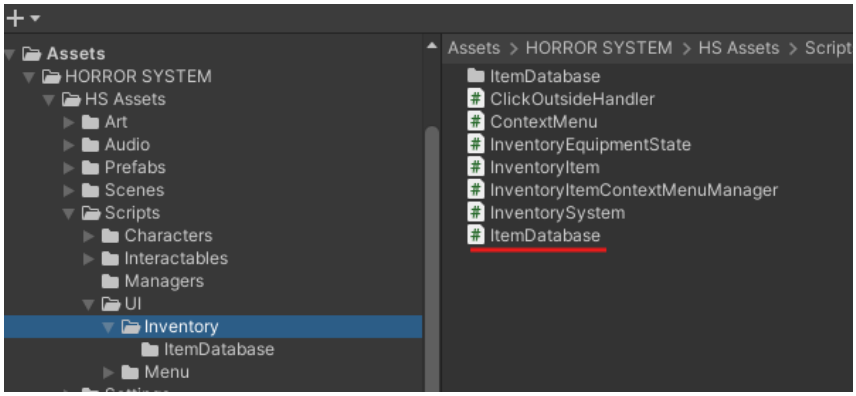
Adding Items to the Item Database

1. **Navigate to the Item Database**
   - Go to: **Scripts → UI → Inventory → ItemDatabase**
   - Open the **ItemDatabase**.
2. **Registering New Items**
   - Add all interactable and inventory-compatible items to the database.
   - Ensure each entry is correctly filled out with the required data.
3. **Important Notes**
   - **ID Consistency**: The item's ID must match the ID specified in the object's component.
   - **Item Types**: If needed, new item types can be defined within the **ItemDatabase script**.







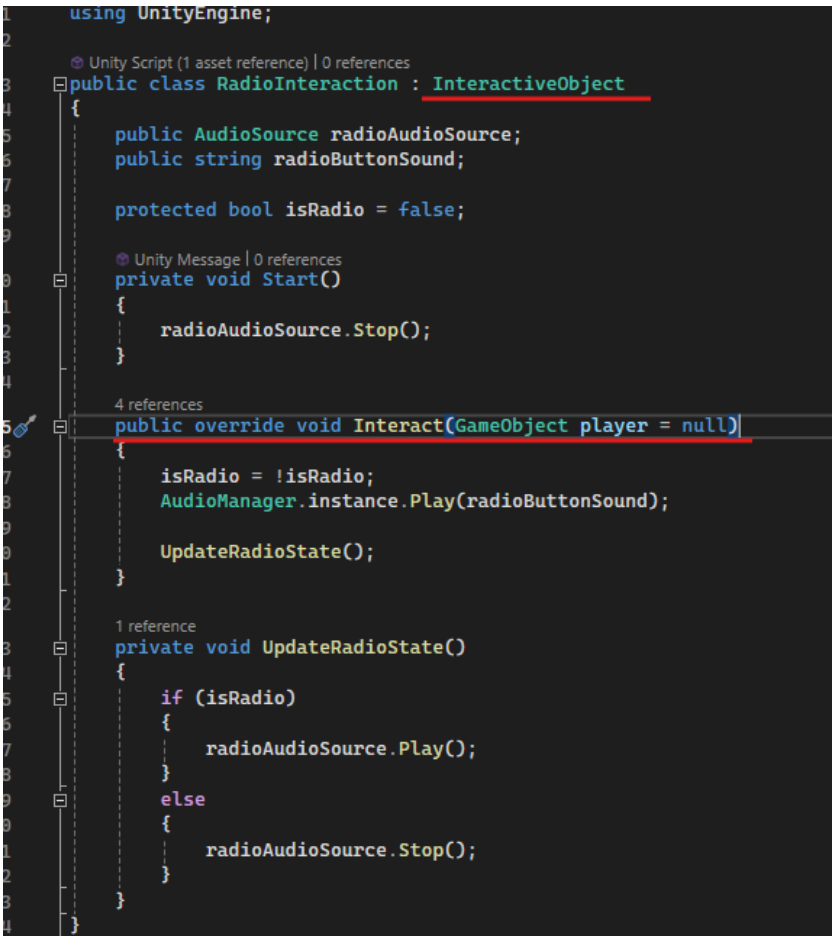



```
using UnityEngine;
using System.Collections.Generic;

[CreateAssetMenu(fileName = "ItemDatabase", menuName = "Inventory/Item Database")]
Unity Script | 26 references
public class ItemDatabase : ScriptableObject
{
    public List<Item> items = new();

    0 references
    public void AddItem(Item newItem)
    {
        items.Add(newItem);
    }

    2 references
    public Item GetItem(string itemId)
    {
        return items.Find(item => item.id == itemId);
    }

    [System.Serializable]
    11 references
    public class Item
    {
        public string itemName;
        public string description;
        public string id;
        public Sprite itemSprite;
        public Type type;
        public GameObject prefab;
    }

    18 references
    public enum Type
    {
        BATTERY,
        FLASHLIGHT,
        AMMO,
        KNIFE,
        MEDKIT,
        PISTOL,
        KEY
    }
```

Interactive Objects (Non-Inventory Interactions)

For objects that require direct interaction (e.g., opening/closing a door) but are **not** inventory items:

- **Required Component**: Attach the **InteractiveObject** script to the object.
- **Usage**: This component handles basic interactions without involving the inventory system.



```
using UnityEngine;

Unity Script (1 asset reference) | 0 references
public class RadioInteraction : InteractiveObject
{
    public AudioSource radioAudioSource;
    public string radioButtonSound;

    protected bool isRadio = false;

    Unity Message | 0 references
    private void Start()
    {
        radioAudioSource.Stop();
    }

    4 references
    public override void Interact(GameObject player = null)
    {
        isRadio = !isRadio;
        AudioManager.instance.Play(radioButtonSound);

        UpdateRadioState();
    }

    1 reference
    private void UpdateRadioState()
    {
        if (isRadio)
        {
            radioAudioSource.Play();
        }
        else
        {
            radioAudioSource.Stop();
        }
    }
}
```

INVENTORY

Inventory System Overview

The latest asset update introduces a **button-activated inventory system** with the following features:

Core Functionality

- **Item Management**: Move, use, or drop items directly from the inventory.
- **Item Descriptions**: View detailed information about each item.
- **Dynamic Counters**: Displays item quantities (e.g., ammo, flashlight battery, medkits) in the bottom-right corner of inventory slots.

Interactive Actions

- **Healing**: Use medical items to restore health.
- **Reloading**: Recharge the flashlight or pistol directly from the inventory.
- **Key System**:
    - Keys automatically remove from inventory when used on doors (no manual interaction required).

Additional UI Elements

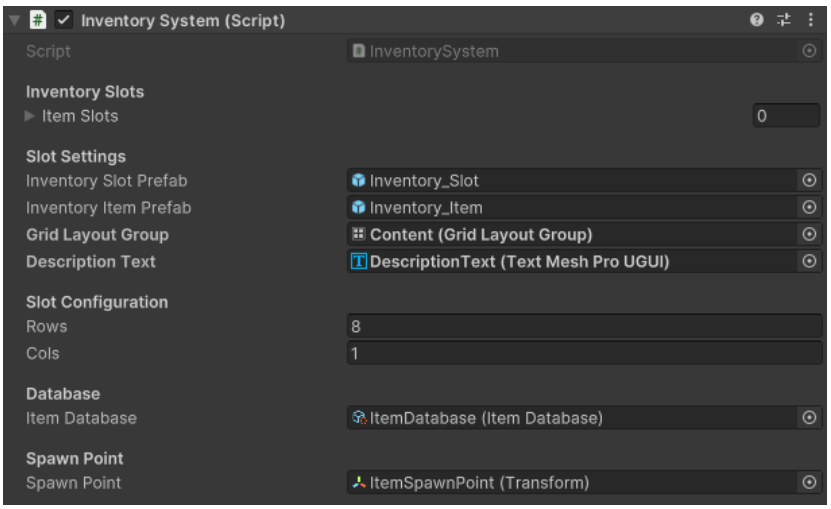
- **Health Display**: The player's current health is shown in the **top-left corner** of the inventory.

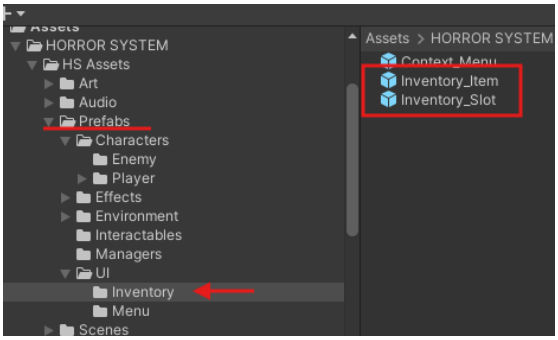
Inventory System Configuration

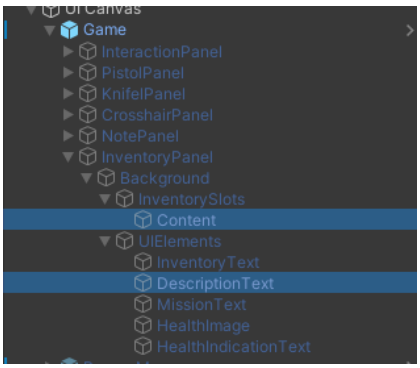
The inventory functionality is managed by the **InventorySystem** component attached to the player character. This system requires the following setup:

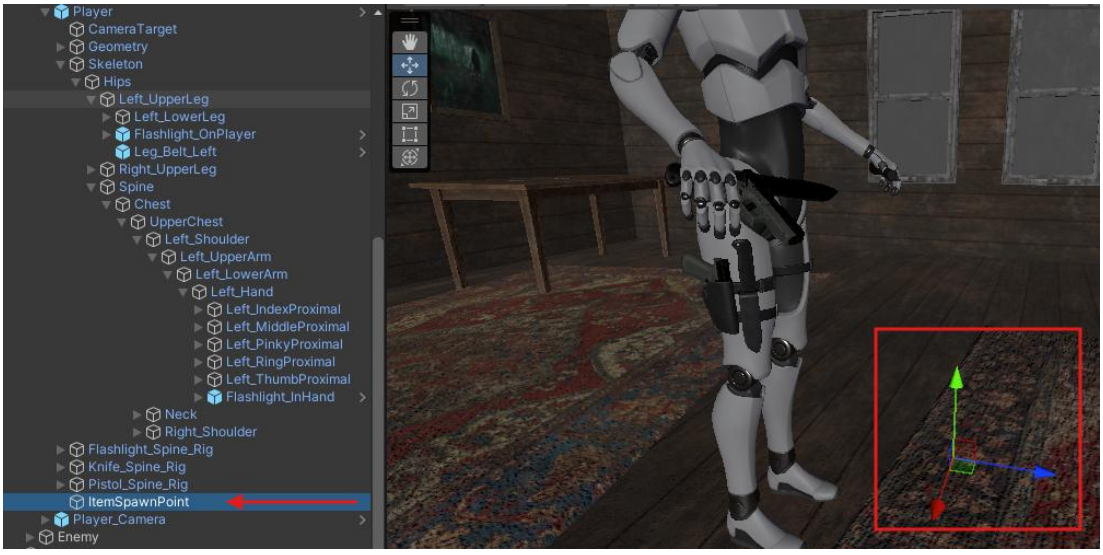
Essential Components

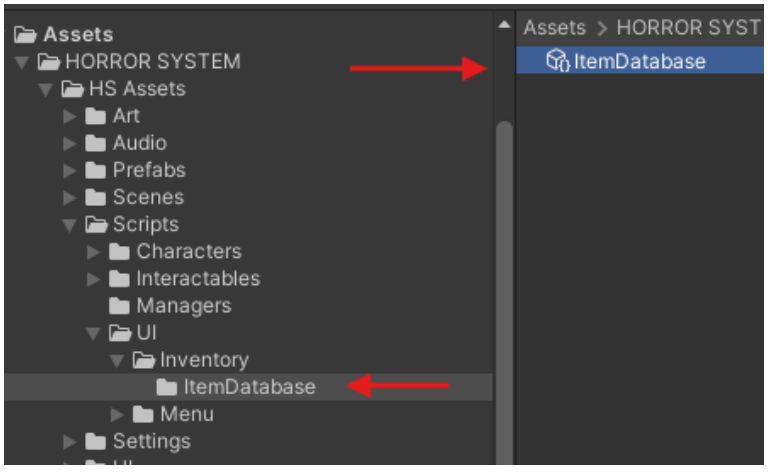
1. **Prefab Assignments**:
    - **Slot Prefab**: The UI element representing an inventory slot
    - **Item Prefab**: The visual representation of an item when placed in a slot
2. **Grid Configuration**:
    - Customize the inventory layout by adjusting rows/columns in the grid settings
3. **Database Reference**:
    - Must link to the **ItemDatabase** which contains all item data (IDs, types, properties)
4. **Drop Mechanics**:
    - Assign a **Spawn Point** transform where items will physically appear when dropped from inventory




HORROR SYSTEM
HS Assets
Art
Audio
Prefabs
Characters
Enemy
Player
Effects
Environment
Interactables
Managers
UI
Inventory
Menu
Scenes
Assets > HORROR SYSTEM
Inventory_Item
Inventory_Slot


Game
InteractionPanel
PistolPanel
KnifelPanel
CrosshairPanel
NotePanel
InventoryPanel
Background
InventorySlots
Content
UIElements
InventoryText
DescriptionText
MissionText
HealthImage
HealthIndicationText


Player
CameraTarget
Geometry
Skeleton
Hips
Left_UpperLeg
Left_LowerLeg
Flashlight_OnPlayer
Leg_Belt_Left
Right_UpperLeg
Spine
Chest
UpperChest
Left_Shoulder
Left_UpperArm
Left_LowerArm
Left_Hand
Left_IndexProximal
Left_MiddleProximal
Left_PinkyProximal
Left_RingProximal
Left_ThumbProximal
Flashlight_InHand
Neck
Right_Shoulder
Flashlight_Spine_Rig
Knife_Spine_Rig
Pistol_Spine_Rig
ItemSpawnPoint
Player_Camera
Enemy


Assets
HORROR SYSTEM
HS Assets
Art
Audio
Prefabs
Scenes
Scripts
Characters
Interactables
Managers
UI
Inventory
ItemDatabase
Menu
Settings
Assets > HORROR SYST
ItemDatabase

Configuring Inventory Items

Accessing Inventory Item Settings

1. Navigate to:
   Prefabs → UI → Inventory
2. Open the Inventory_Item prefab

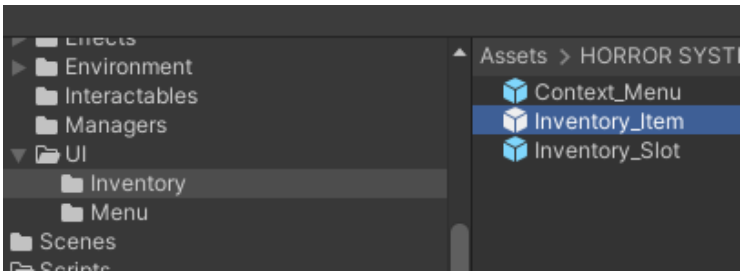
Component Overview

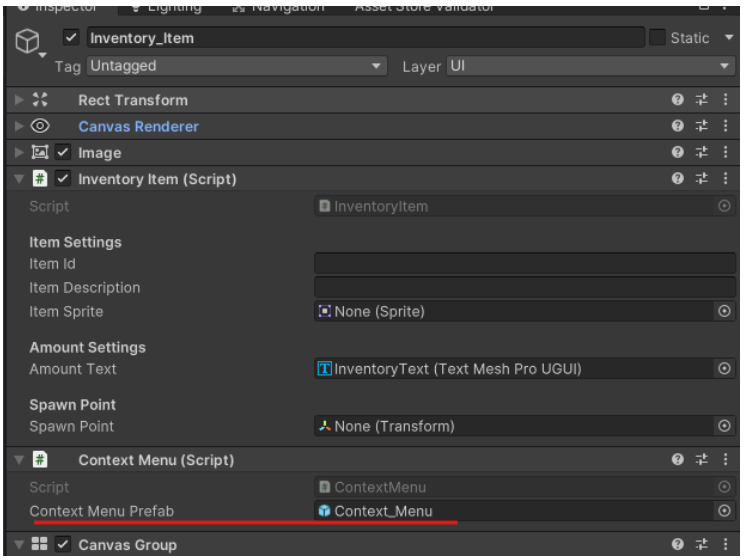
The prefab contains two key components:

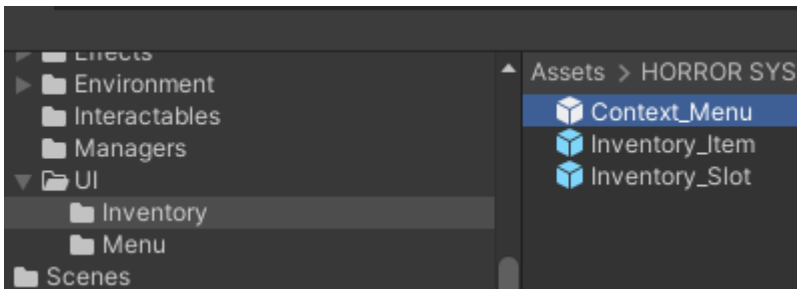
1. **InventoryItem**
   - *Do not modify* - all data populates automatically when items are added to slots
2. **ContextMenu**
   - Requires setup to enable item interactions

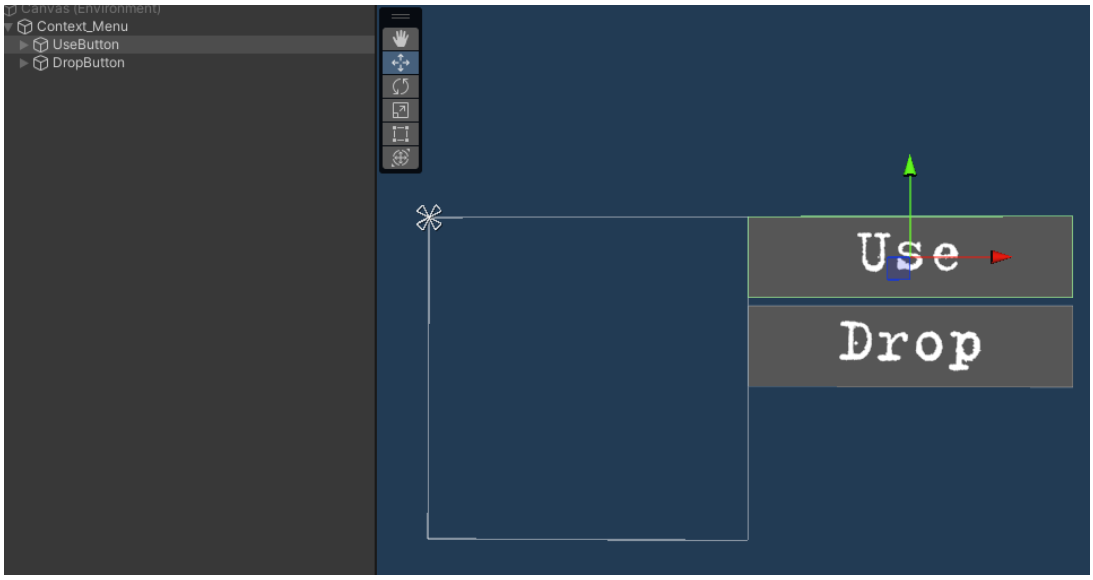
Context Menu Configuration

1. **Assignment**:
   - From the same folder (Prefabs → UI → Inventory), drag the Context_Menu prefab into the ContextMenu component's reference field
2. **Customization**:
   - Open the Context_Menu prefab to:
     - Modify button layouts
     - Configure interaction options
     - Adjust visual elements





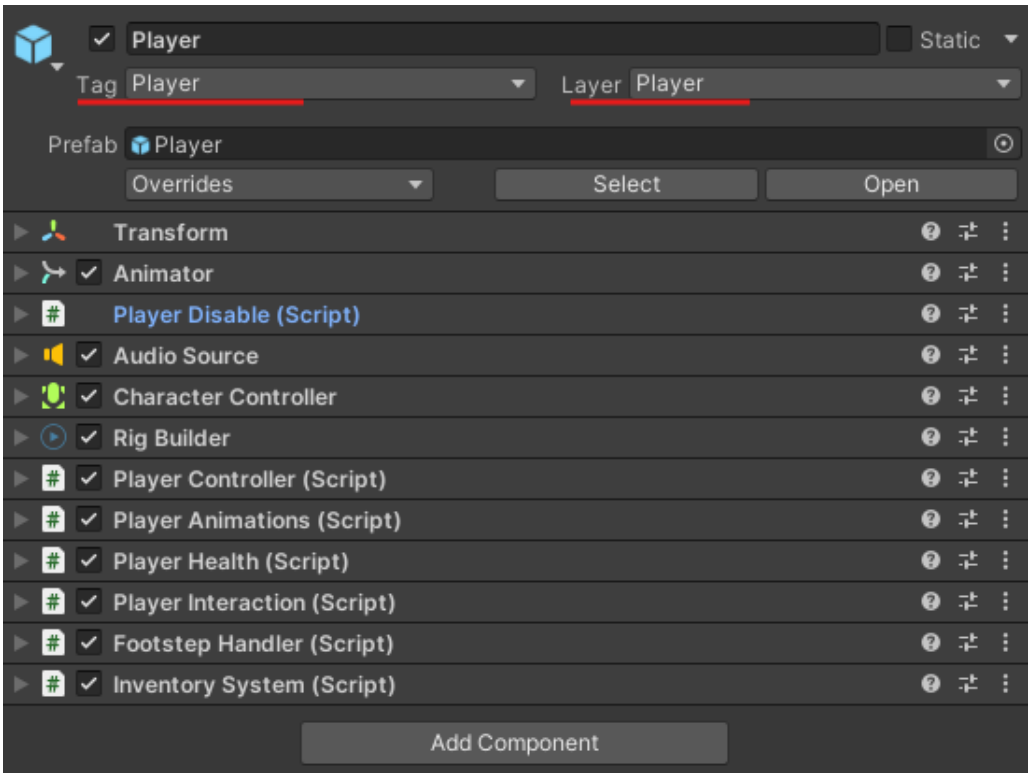

Environment
Interactables
Managers
UI
Inventory
Menu
Scenes
Assets > HORROR SYS
Context_Menu
Inventory_Item
Inventory_Slot


Context_Menu
UseButton
DropButton
Use
Drop

PLAYER

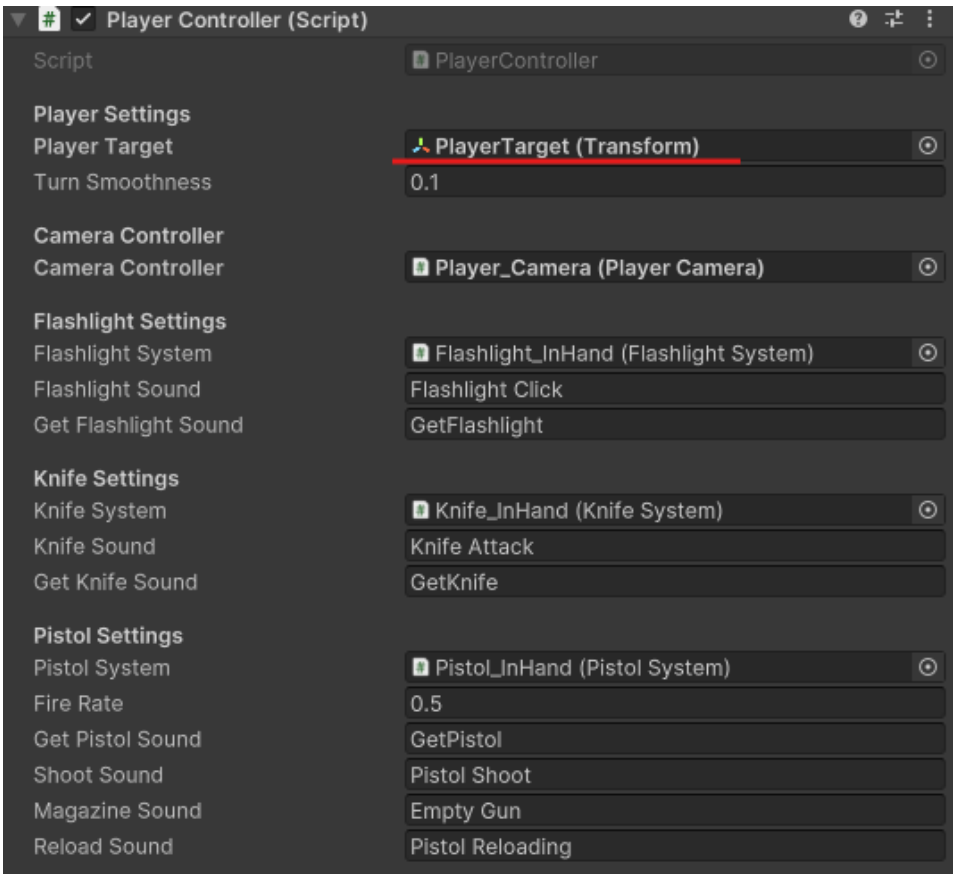
The player must have the following components: **Animator, AudioSource, CharacterController, RigBuilder, PlayerDisable, PlayerController, PlayerAnimations, PlayerHealth, PlayerInteraction, FootstepHandler, and InventorySystem.**

The player should also be assigned the "**Player**" tag and layer.



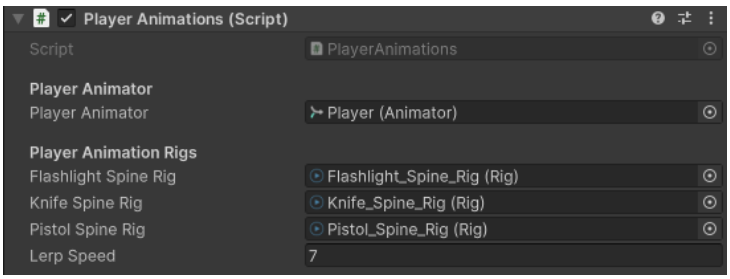
PlayerController

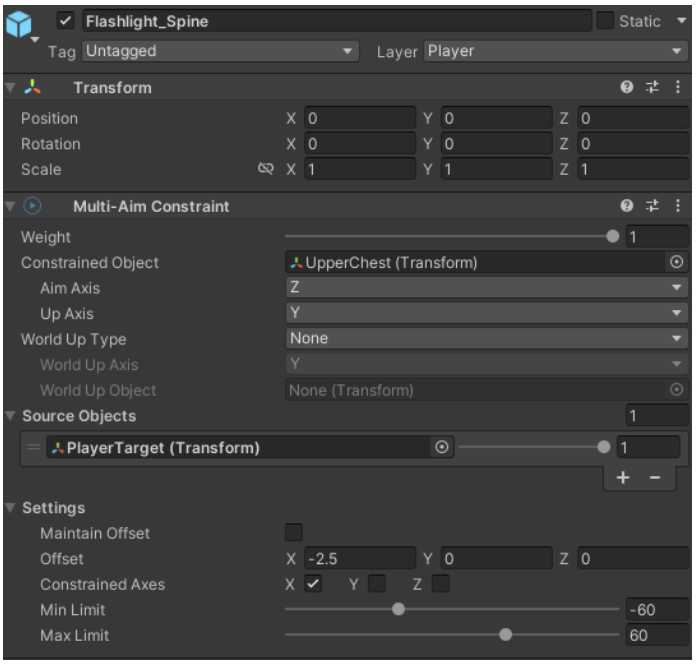
PlayerController handles the core player mechanics, including movement, weapon switching, weight adjustments, and sound effects. **PlayerTarget** is the object that the player rotates toward. This object is a child of the camera.



PlayerAnimations

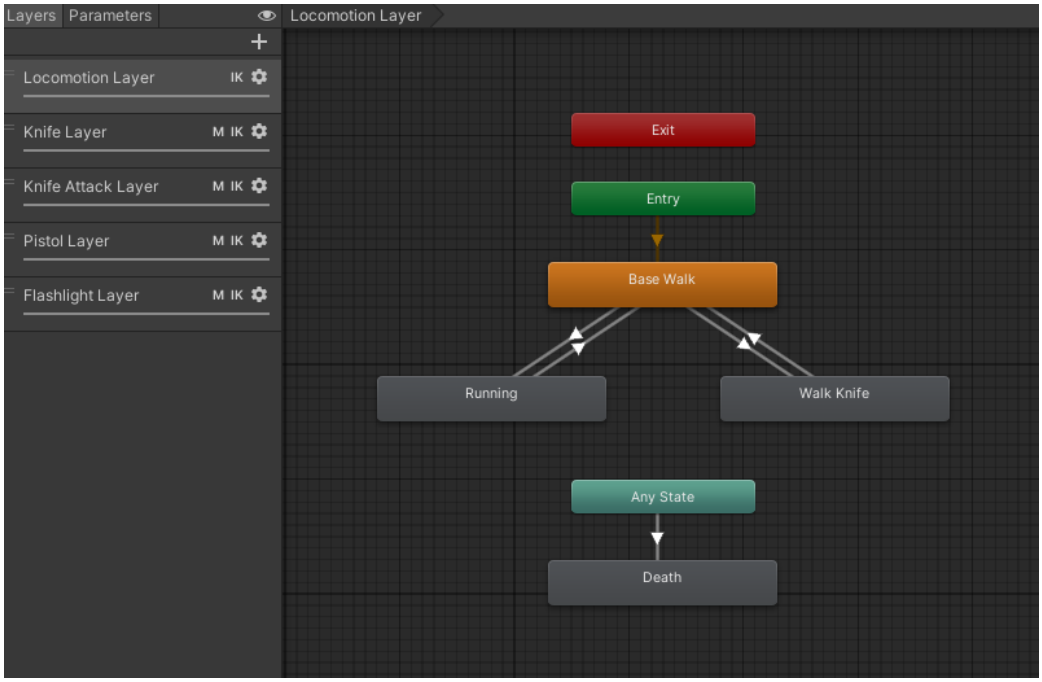
PlayerAnimations is responsible for playing the character's animations. It references the character's Animator and triggers the necessary animation. It also updates the character's weights. For more details on how **Animation Rigging** works, please refer to the official documentation.





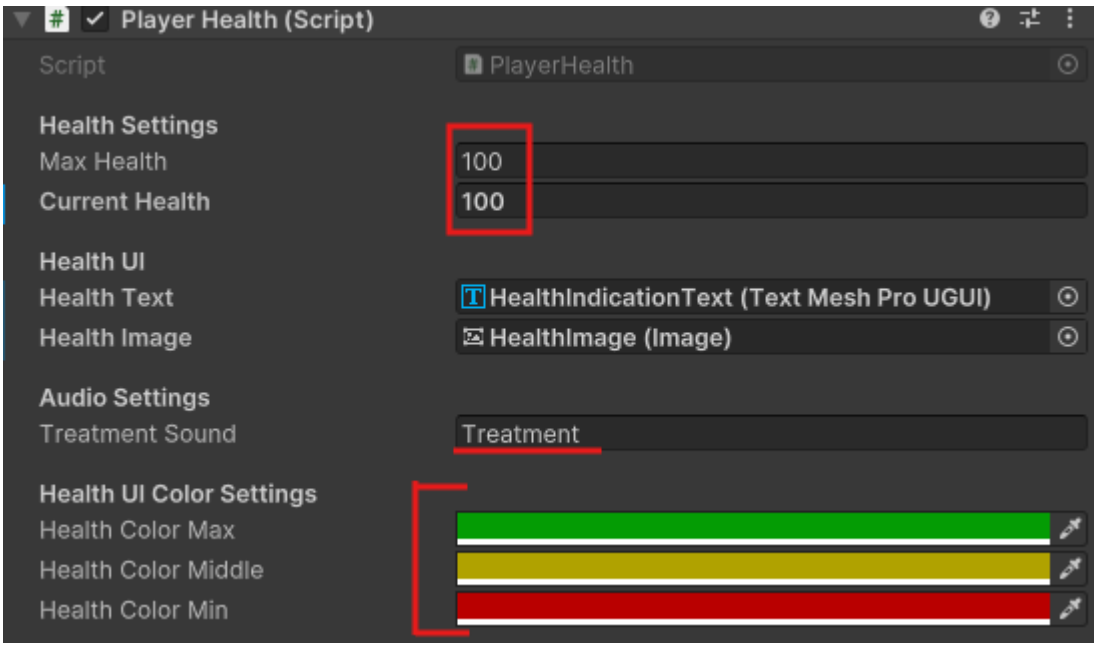

```
2 references
public void SetFlashlighState(bool state)
{
    this.isFlashlight = state;
    InventoryManager.instance.hasFlashlight = state;
    playerAnimations.SetFlashlightIdle(state);
}
```

```
using UnityEngine;
using UnityEngine.Animations.Rigging;

Unity Script (1 asset reference) | 4 references
public class PlayerAnimations : MonoBehaviour
{

    [Header("Player Animator")]
    public Animator playerAnimator;

    [Header("Player Animation Rigs")]
    public Rig flashlightSpineRig;
    public Rig knifeSpineRig;
    public Rig pistolSpineRig;
    public float lerpSpeed;

    Unity Message | 0 references
    public void Start()
    {
        playerAnimator = GetComponent<Animator>();
    }

    1 reference
    public void SetMovementParameters(float moveX, float moveZ)
    {
        playerAnimator.SetFloat("MoveX", moveX);
        playerAnimator.SetFloat("MoveZ", moveZ);
    }

    2 references
    public float GetMoveX()
    {
        return playerAnimator.GetFloat("MoveX");
    }

    2 references
    public float GetMoveZ()
    {
        return playerAnimator.GetFloat("MoveZ");
    }

    1 reference
    public void RunningAnimation(bool isRunning)
    {
        playerAnimator.SetBool("Running", isRunning);
    }
```



PlayerHealth

PlayerHealth is responsible for the player's health. You can adjust the health amount, the healing sound, as well as the health icon colors for 100%, 50%, and 25% health. Health decreases when the character enters a trigger with the **EnemyDamage** component.

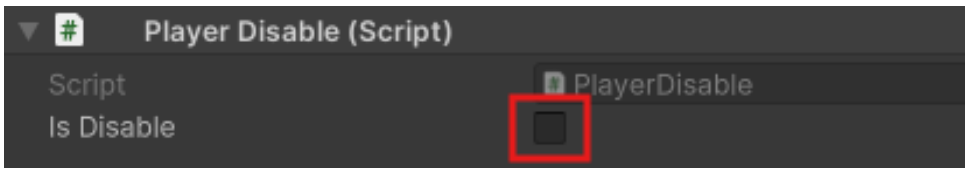



```
using UnityEngine;

Unity Script (1 asset reference) | 0 references
public class EnemyDamage : MonoBehaviour
{
    public int damage = 2;

    Unity Message | 0 references
    void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            PlayerHealth playerHealth = other.GetComponent<PlayerHealth>();
            if (playerHealth != null)
            {
                playerHealth.TakeDamage(damage);
            }
        }
    }
}
```

PlayerDisable

Function:

- Disables player input during UI interactions (inventory, pause menu, notes).
- Pauses all game actors and scripts by freezing the scene.




```
3 references
private void PlayerDisable()
{
    if (isInventoryPanelActive || isNotePanelActive || isPausePanelActive)
    {
        global::PlayerDisable.instance.DisablePlayer();
    }
    else
    {
        global::PlayerDisable.instance.EnablePlayer();
    }
}
```

PLAYER CAMERA

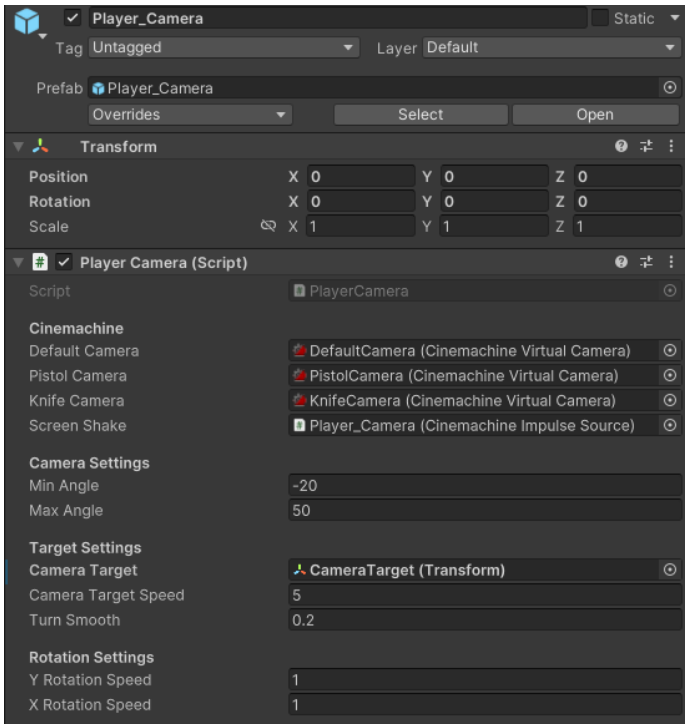
PlayerCamera handles camera rotation. In this asset version, we utilize Cinemachine for camera control. For detailed implementation, we recommend reviewing the official Cinemachine documentation.
Configurable parameters include:

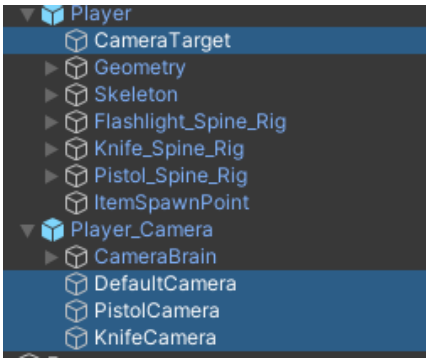
- Camera angle constraints
- Rotation speed

We employ separate **CinemachineVirtualCamera** instances with unique configurations for:

- Pistol aiming
- Knife combat

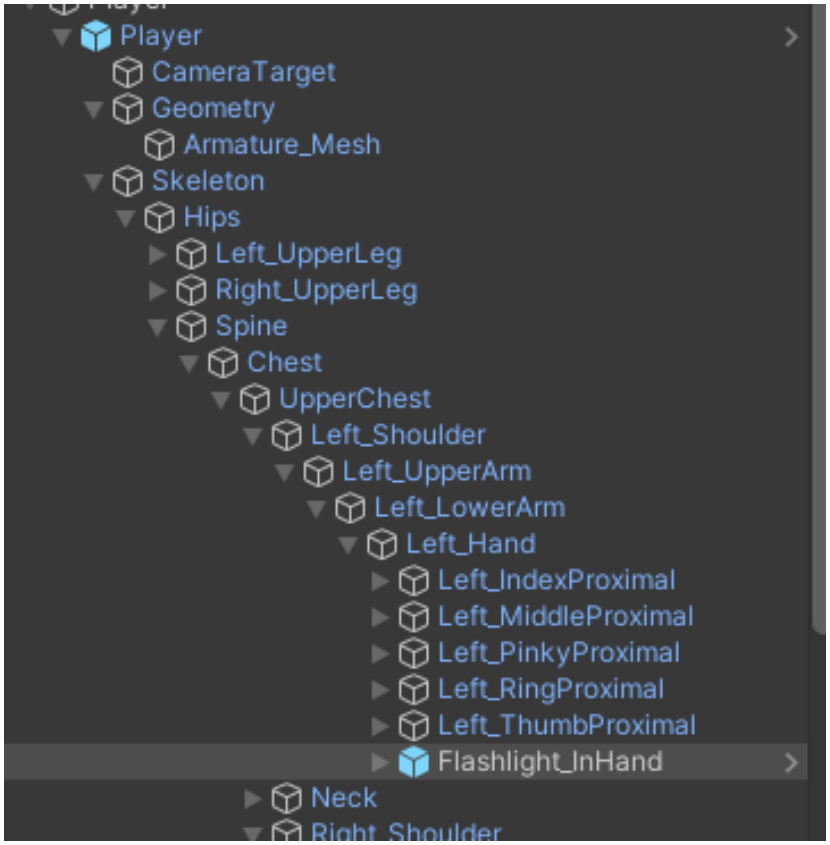
The **CameraTarget** (child object of the Player) serves as the rotation pivot point - while the camera follows its orientation.

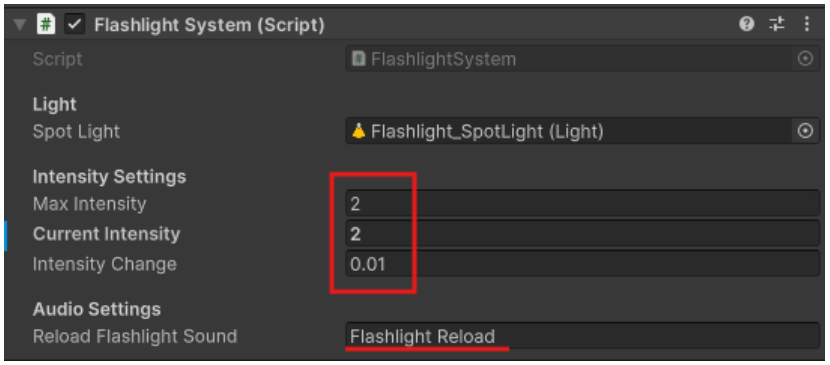




FLASHLIGHT

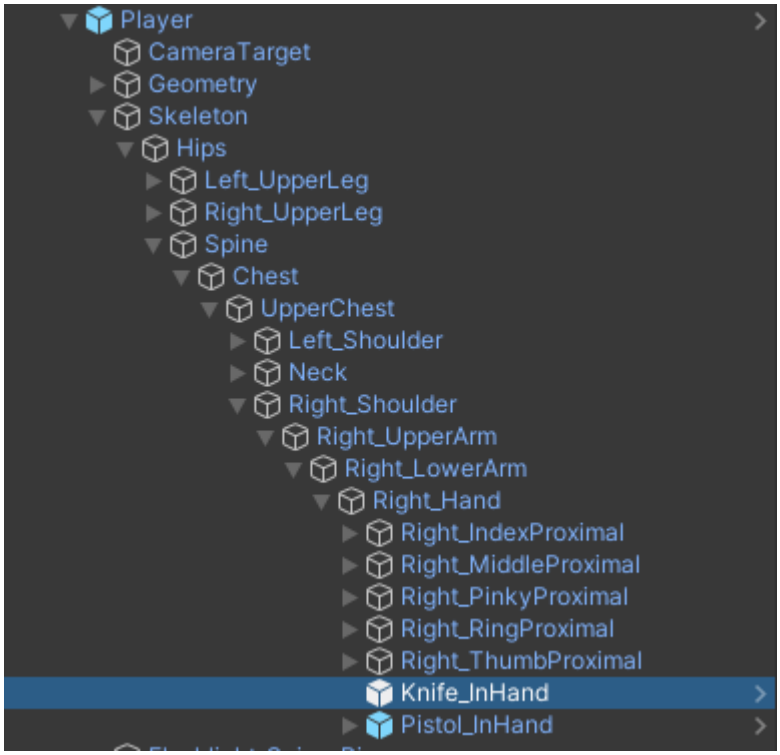
FlashlightSystem is a system for adjusting light intensity. This component is attached to **Flashlight_InHand**. You can configure the desired intensity, intensity change parameters, as well as the reload sound.

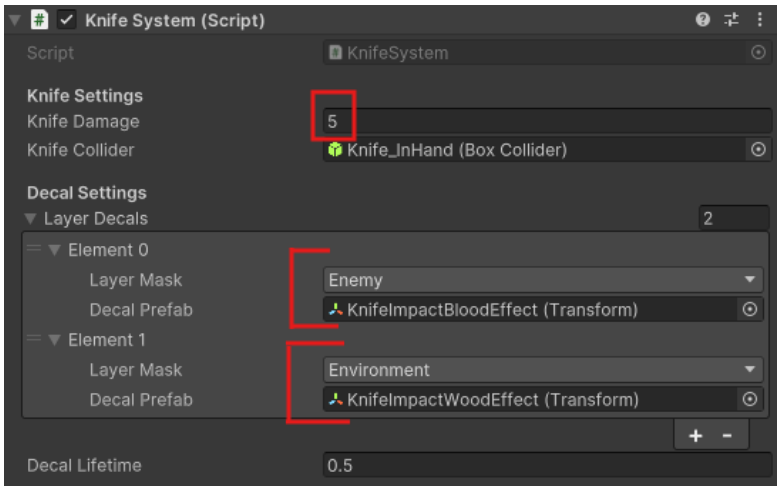




KNIFE

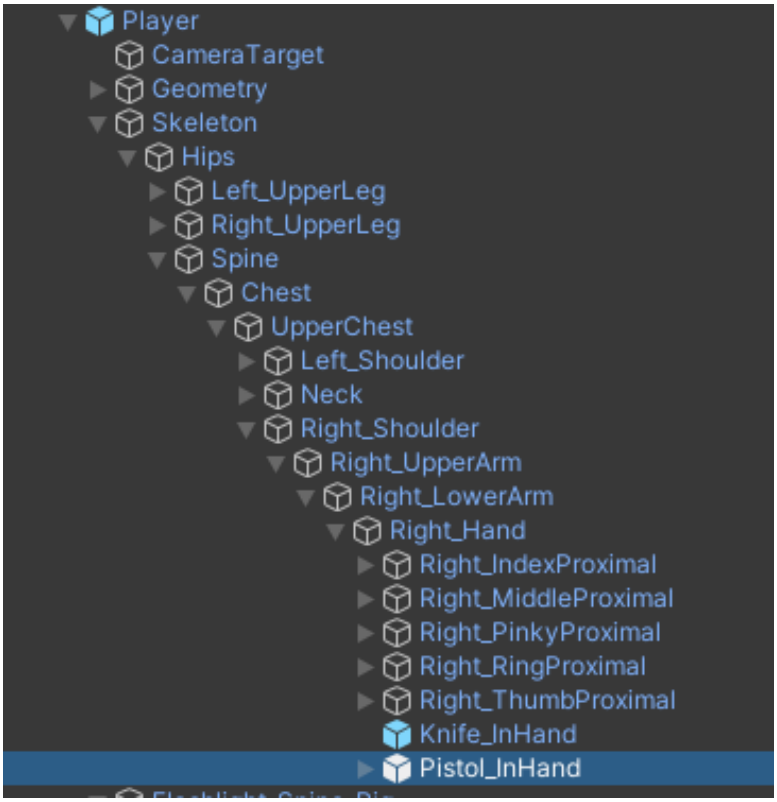
KnifeSystem is a component responsible for enabling the collider, applying damage, and spawning decals. This component is attached to **Knife_InHand**. You can configure the desired damage and set up decal spawning upon interaction with a specific layer.

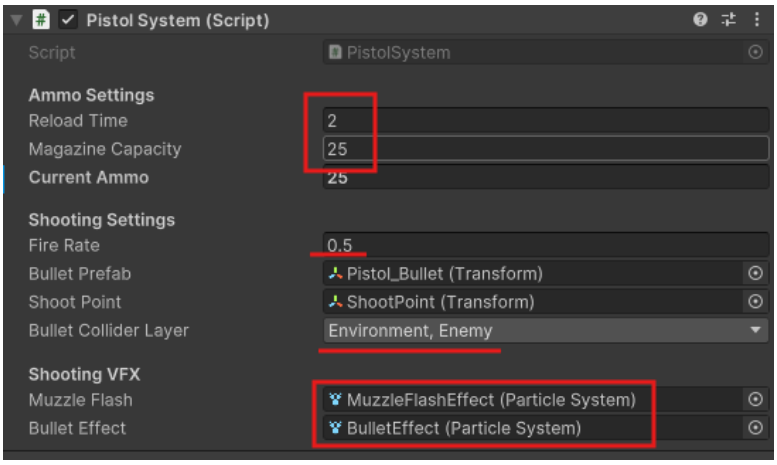




PISTOL

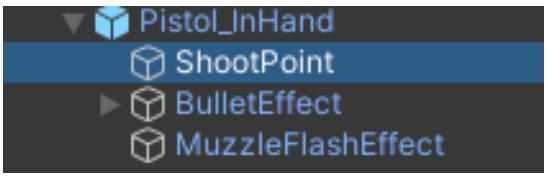
PistolSystem is a component responsible for handling shooting mechanics. This component is attached to **Pistol_InHand**. You can configure the magazine size, reload time, fire rate, bullet collision layers, as well as visual effects.

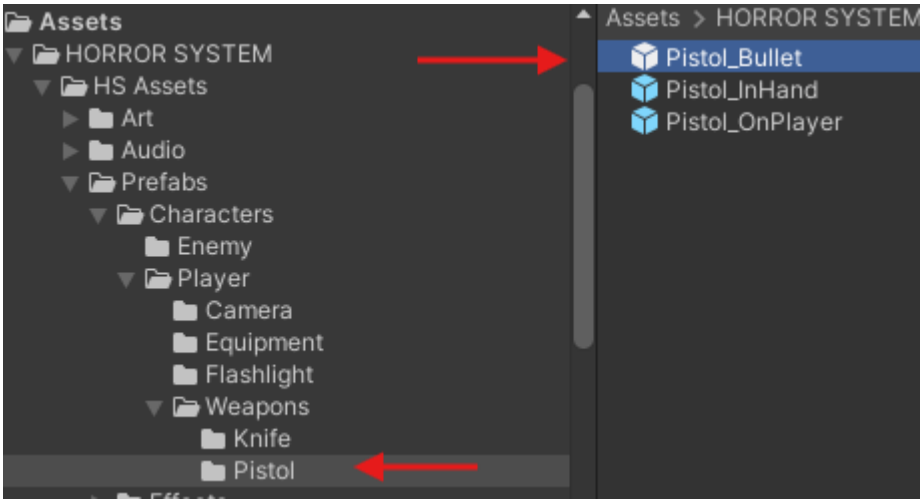


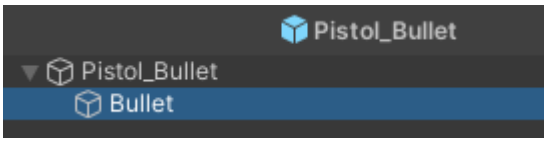


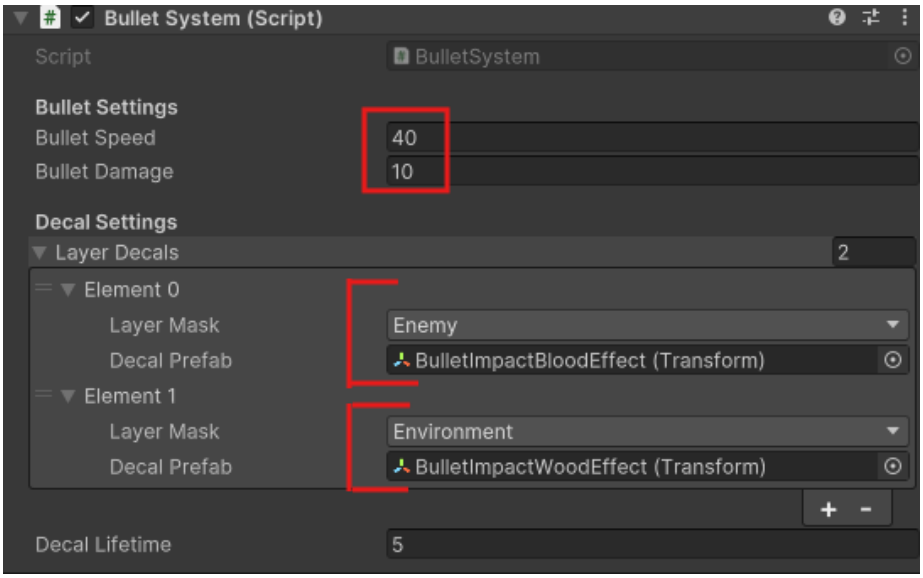
ShootPoint is the spawn location for bullets. As a child object of **Pistol_InHand**, it determines the projectile's origin point. The **Pistol_Bullet** prefab contains all bullet-related configurations (located at: Prefabs → Characters → Player → Weapons → Pistol).
The prefab includes a child object named **Bullet** featuring the **BulletSystem** component, where you can configure:

- Projectile velocity
- Damage output
- Layer-specific decal spawning upon impact


Pistol_InHand
ShootPoint
BulletEffect
MuzzleFlashEffect

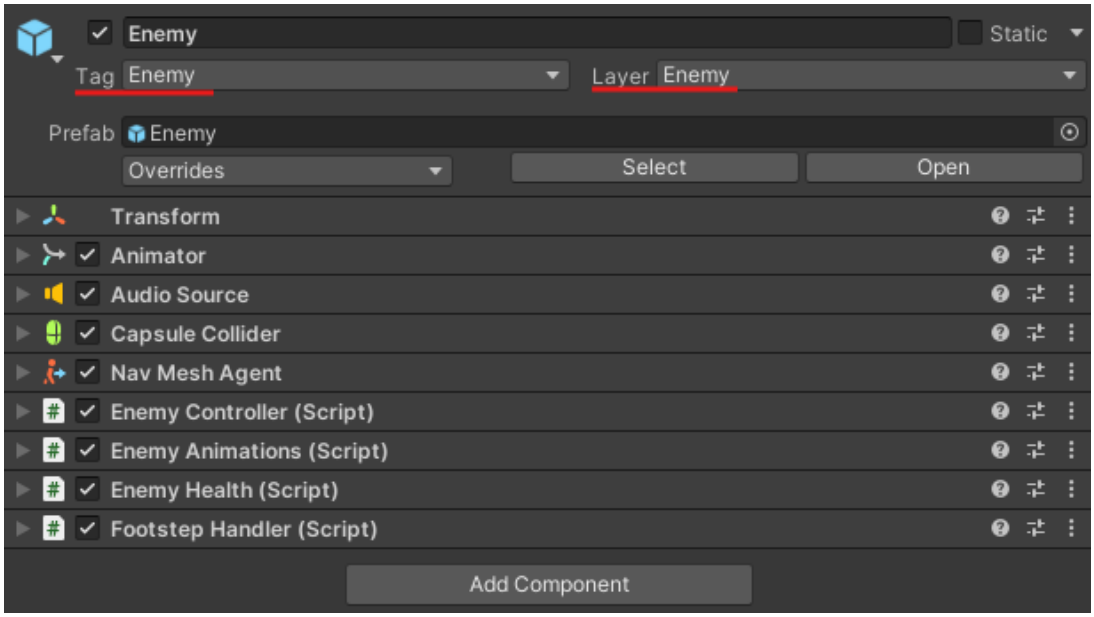

Assets
HORROR SYSTEM
HS Assets
Art
Audio
Prefabs
Characters
Enemy
Player
Camera
Equipment
Flashlight
Weapons
Knife
Pistol
Assets > HORROR SYSTEM
Pistol_Bullet
Pistol_InHand
Pistol_OnPlayer


Pistol_Bullet
Pistol_Bullet
Bullet


Bullet System (Script)
Script
BulletSystem
Bullet Settings
Bullet Speed
40
Bullet Damage
10
Decal Settings
Layer Decals
2
Element 0
Layer Mask
Enemy
Decal Prefab
BulletImpactBloodEffect (Transform)
Element 1
Layer Mask
Environment
Decal Prefab
BulletImpactWoodEffect (Transform)
Decal Lifetime
5

ENEMY

The enemy must have the following components: **Animator, AudioSource, CapsuleController, NavMeshAgent, EnemyController, EnemyAnimations, EnemyHealth, FootstepHandler.**
The enemy should also be assigned the "**Enemy**" tag and layer.



EnemyController

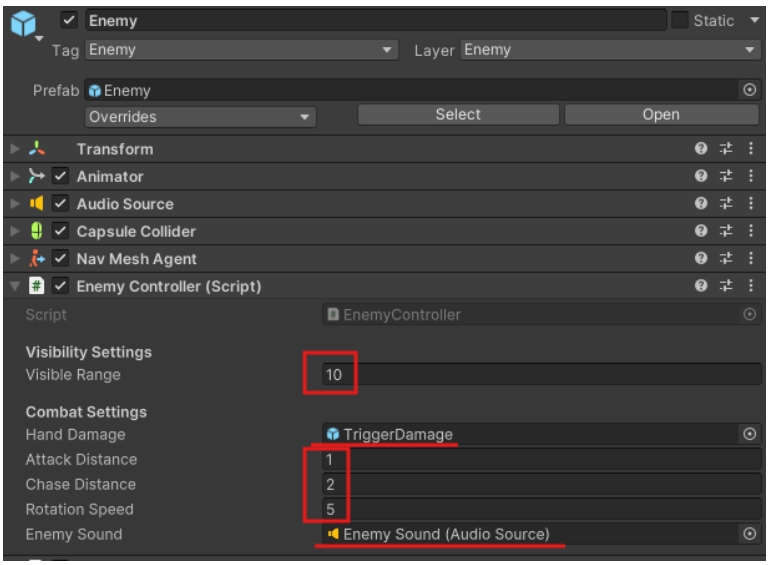
The **EnemyController** handles core enemy mechanics, including:

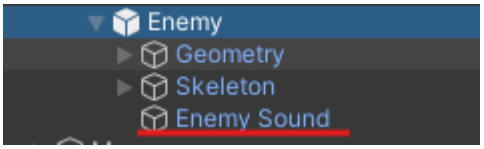
- Chase distance (pursuit range)
- Attack distance (engagement range)
- Sound effects
- Other behavioral parameters

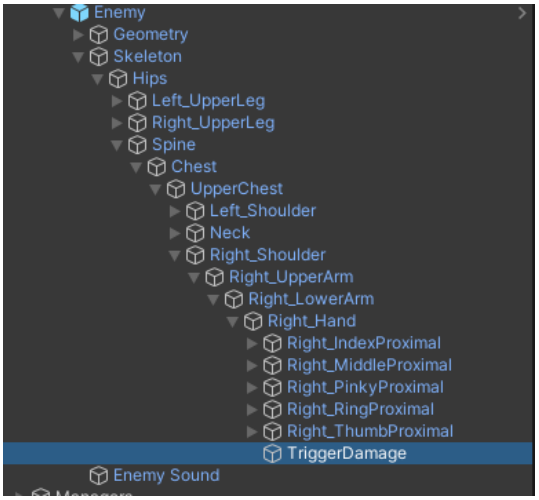
Damage Trigger Setup

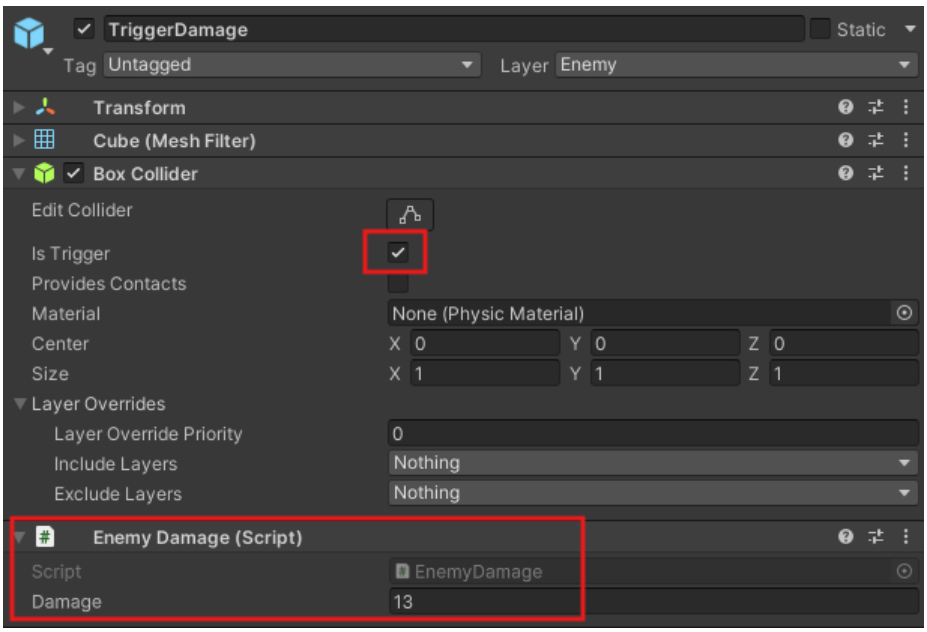
A **TriggerDamage** object must be added with:

1. A trigger collider (for hit detection)
2. The **EnemyDamage** component (handles damage logic)



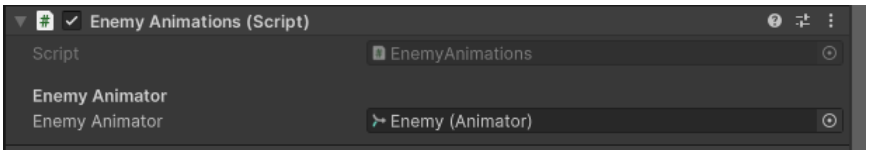





TriggerDamage
Static
Tag Untagged
Layer Enemy
Transform
Cube (Mesh Filter)
Box Collider
Edit Collider
Is Trigger
Provides Contacts
Material
None (Physic Material)
Center
X 0
Y 0
Z 0
Size
X 1
Y 1
Z 1
Layer Overrides
Layer Override Priority
0
Include Layers
Nothing
Exclude Layers
Nothing
Enemy Damage (Script)
Script
EnemyDamage
Damage
13

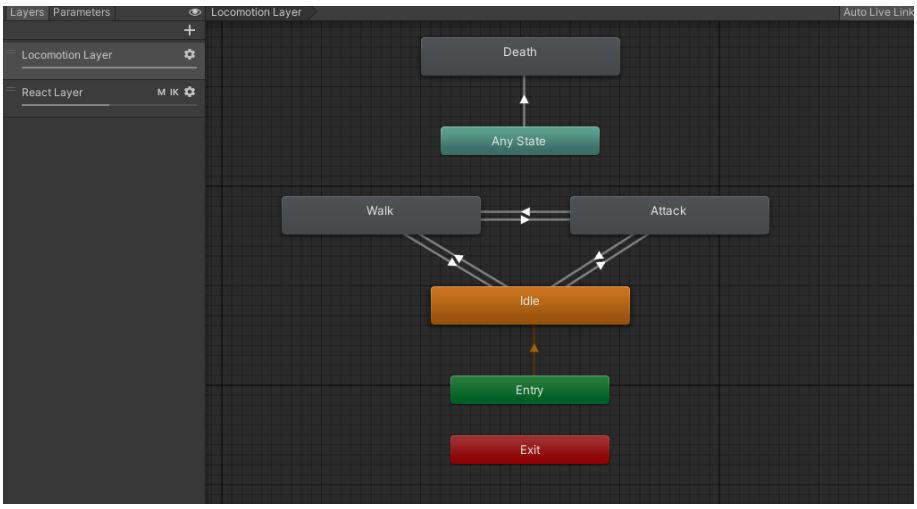
EnemyAnimations

EnemyAnimations is responsible for playing the character's animations. It references the character's Animator and triggers the necessary animation.

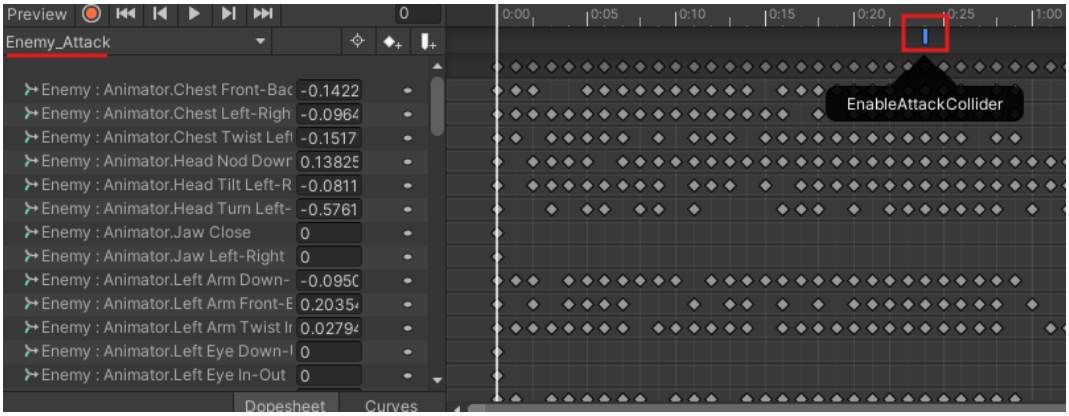


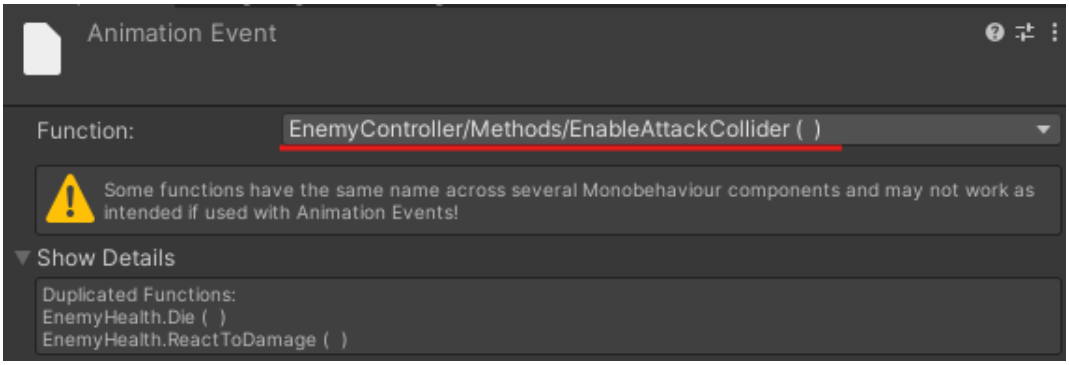

```
if (distanceToPlayer > chaseDistance)
{
    enemyAnimations.StopAttacking();
    enemyAnimations.Walk();
    navAgent.isStopped = false;
    navAgent.SetDestination(target.position);
}
```

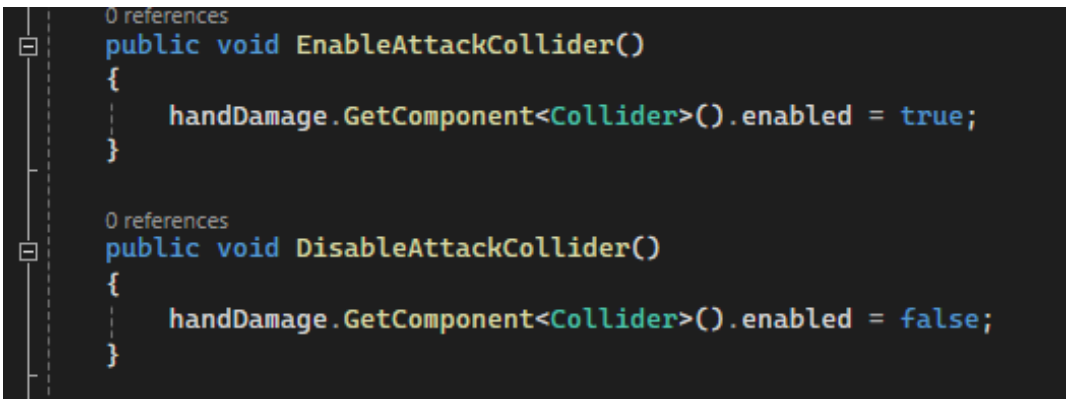
```
using UnityEngine;

Unity Script (1 asset reference) | 4 references
public class EnemyAnimations : MonoBehaviour
{

    [Header("Enemy Animator")]
    public Animator enemyAnimator;

    Unity Message | 0 references
    private void Start()
    {
        enemyAnimator = GetComponent<Animator>();
    }

    2 references
    public void Walk()
    {
        enemyAnimator.SetBool("Walk", true);
    }

    2 references
    public void StopWalking()
    {
        enemyAnimator.SetBool("Walk", false);
    }

    1 reference
    public void Attack()
    {
        enemyAnimator.SetBool("Attack", true);
    }

    3 references
    public void StopAttacking()
    {
        enemyAnimator.SetBool("Attack", false);
    }
```



Additionally, the **Enemy_Attack** animation has events that activate/deactivate **TriggerDamage**.

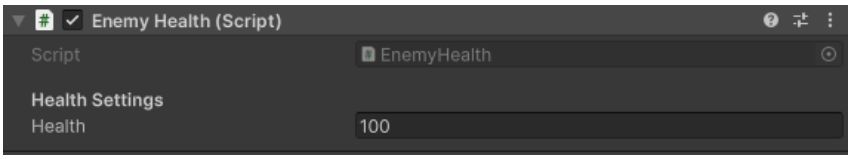






EnemyHealth

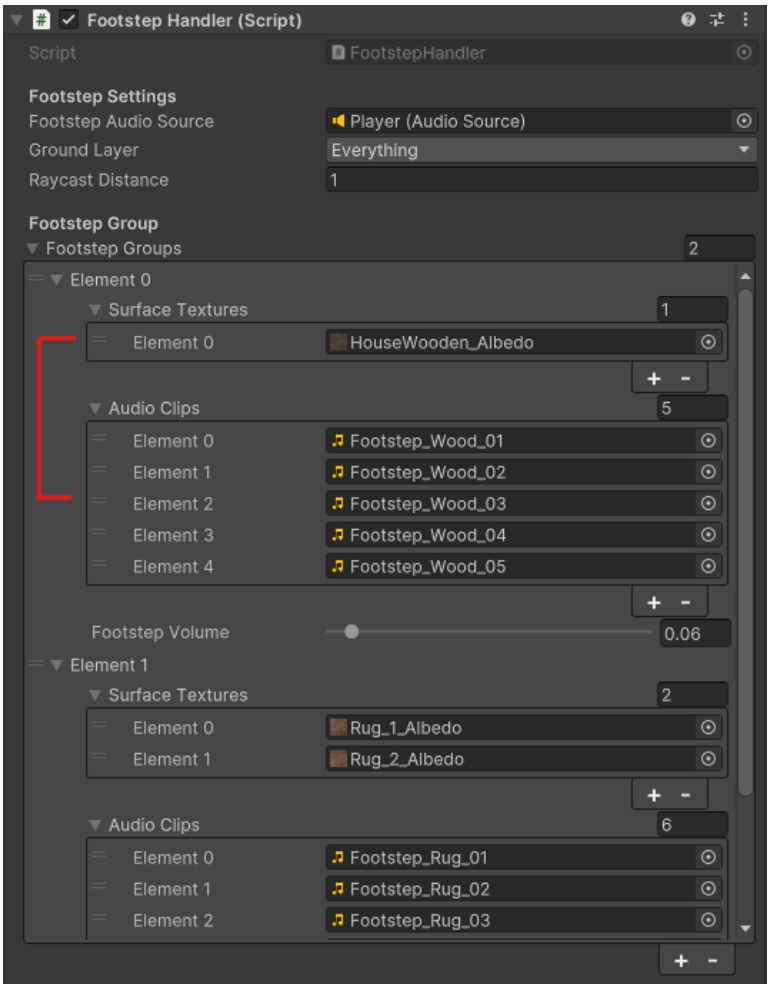
EnemyHealth is responsible for the enemy's health. You can configure the desired number of enemy health units. Health decreases when a bullet hits the enemy or when a knife strikes it via triggers.

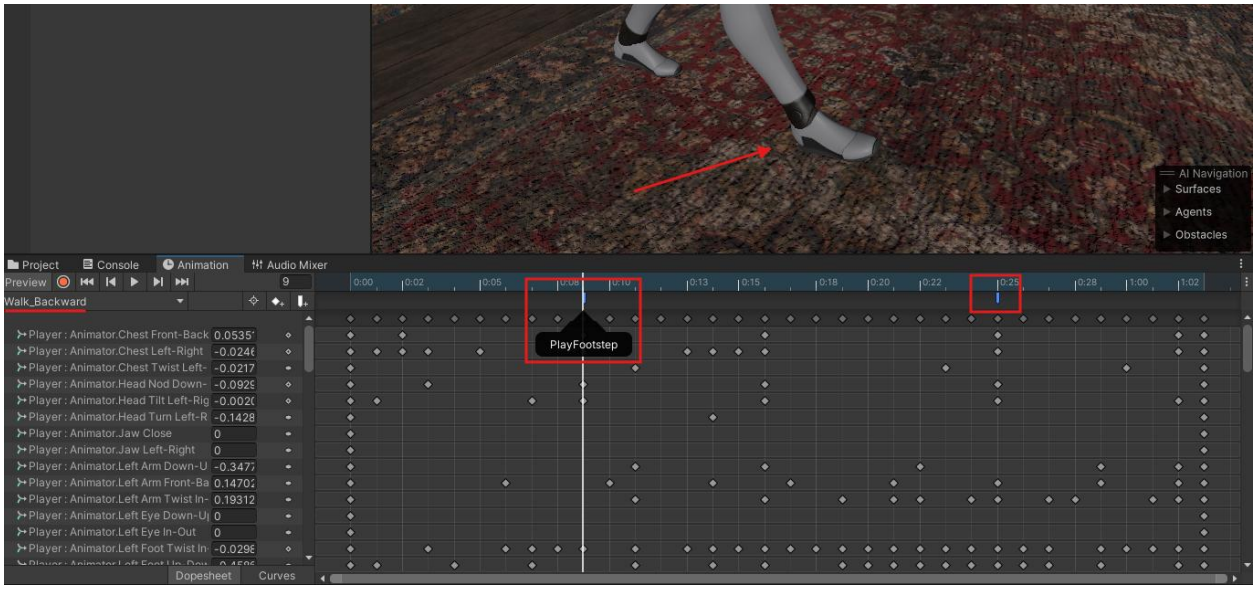


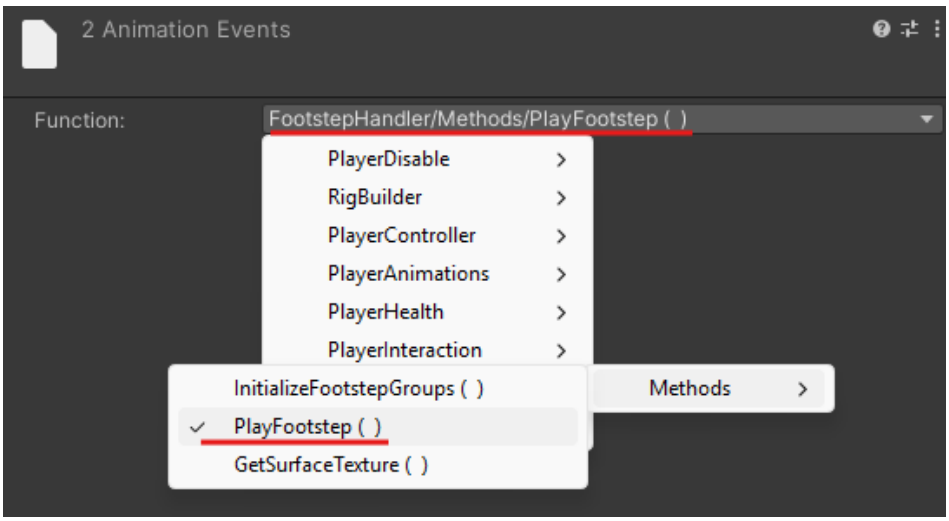

```
using UnityEngine;
using System.Collections.Generic;

Unity Script (1 asset reference) | 1 reference
public class KnifeSystem : MonoBehaviour
{
    [Header("Knife Settings")]
    public int knifeDamage = 5;
    public Collider knifeCollider;

    [Header("Decal Settings")]
    public List<LayerDecalKnife> layerDecals;
    public float decalLifetime = 5f;

    Unity Message | 0 references
    private void Start()
    {
        knifeCollider.enabled = false;
    }

    1 reference
    public void EnableCollider()
    {
        knifeCollider.enabled = true;
    }

    1 reference
    public void DisableCollider()
    {
        knifeCollider.enabled = false;
    }

    Unity Message | 0 references
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Enemy"))
        {
            EnemyHealth enemy = other.GetComponent<EnemyHealth>();
            if (enemy != null)
            {
                enemy.TakeDamage(knifeDamage);
            }
        }
```

FOOTSTEP

In this version of the asset, we have updated the footstep system. Now, the footstep sound is played via events, and the surface is determined using raycasting. The **FootstepHandler** component is used for both the player and the enemy. In it, you can configure the texture and the corresponding footstep sound for that texture.

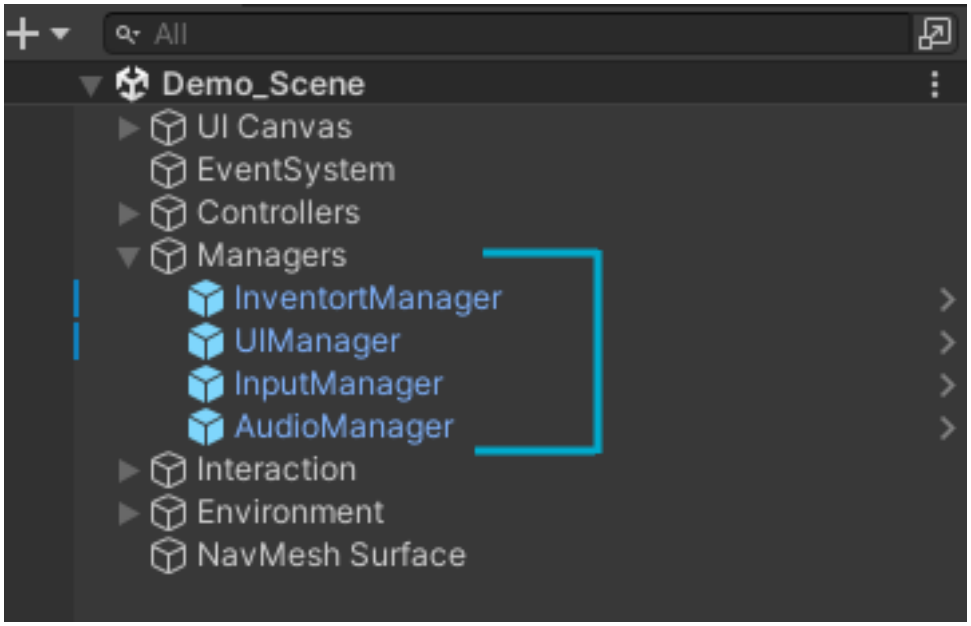





2 Animation Events
Function:
FootstepHandler/Methods/PlayFootstep ( )
PlayerDisable
RigBuilder
PlayerController
PlayerAnimations
PlayerHealth
PlayerInteraction
InitializeFootstepGroups ( )
PlayFootstep ( )
GetSurfaceTexture ( )
Methods

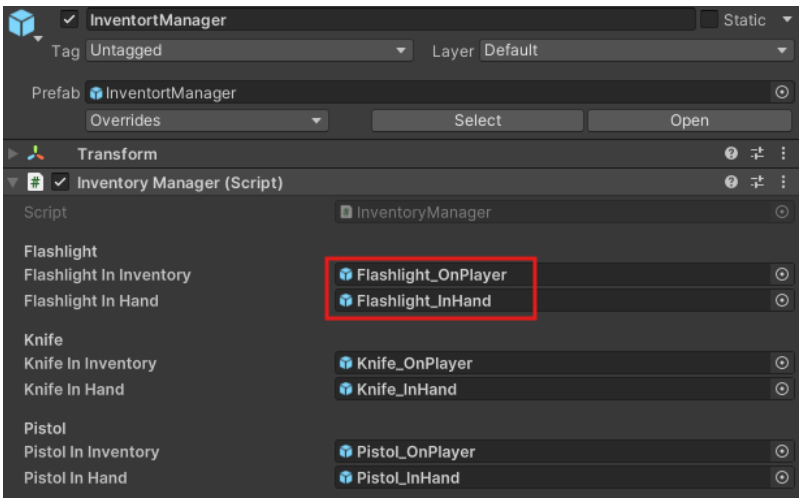
MANAGERS

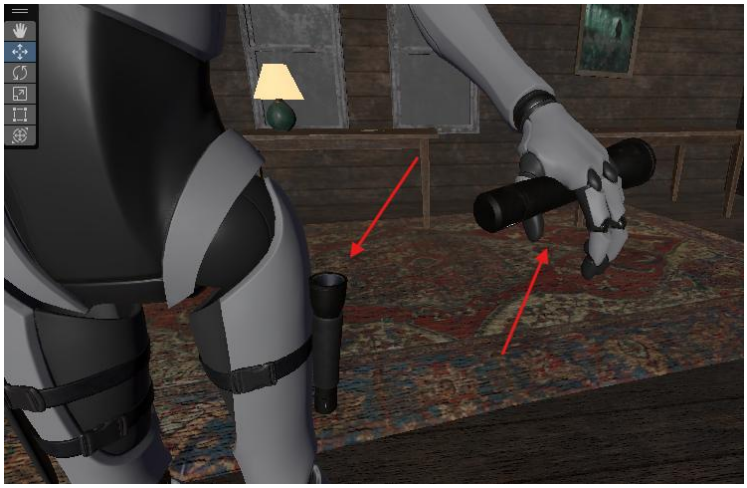
The following managers should be present in the scene: **InventoryManager, UIManager, InputManager, and AudioManager.**

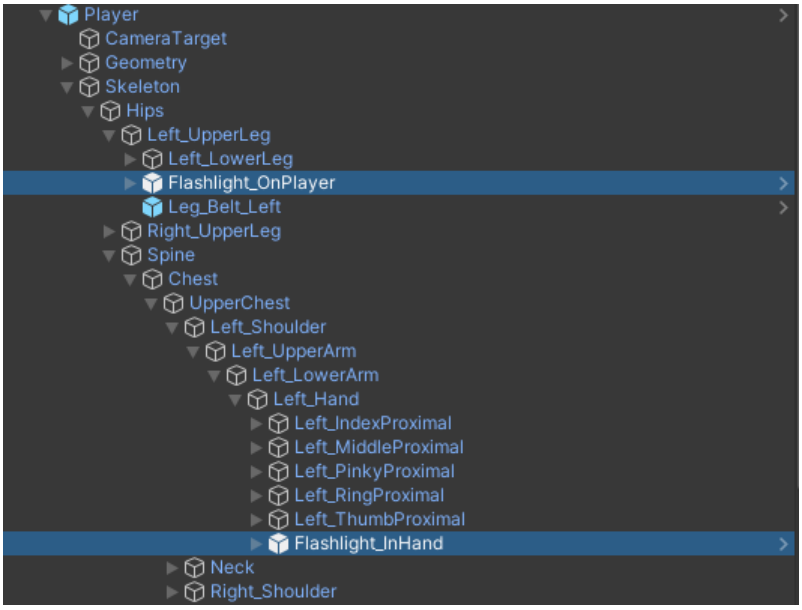


InventoryManager

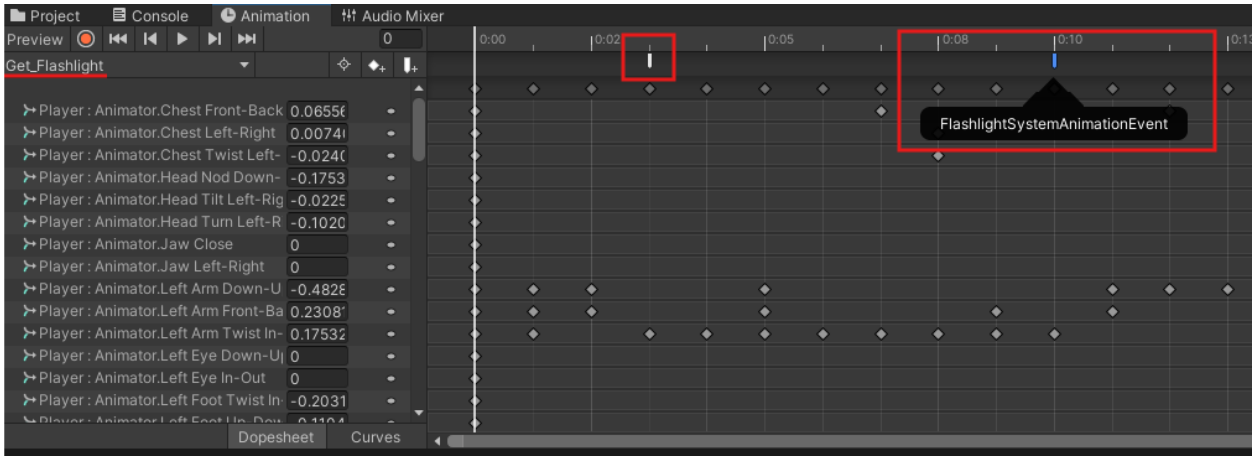
The **InventoryManager** handles object visibility—showing items either in the character's hand or inventory. Enabling a weapon or flashlight disables the body-mounted model and activates the hand-held version.

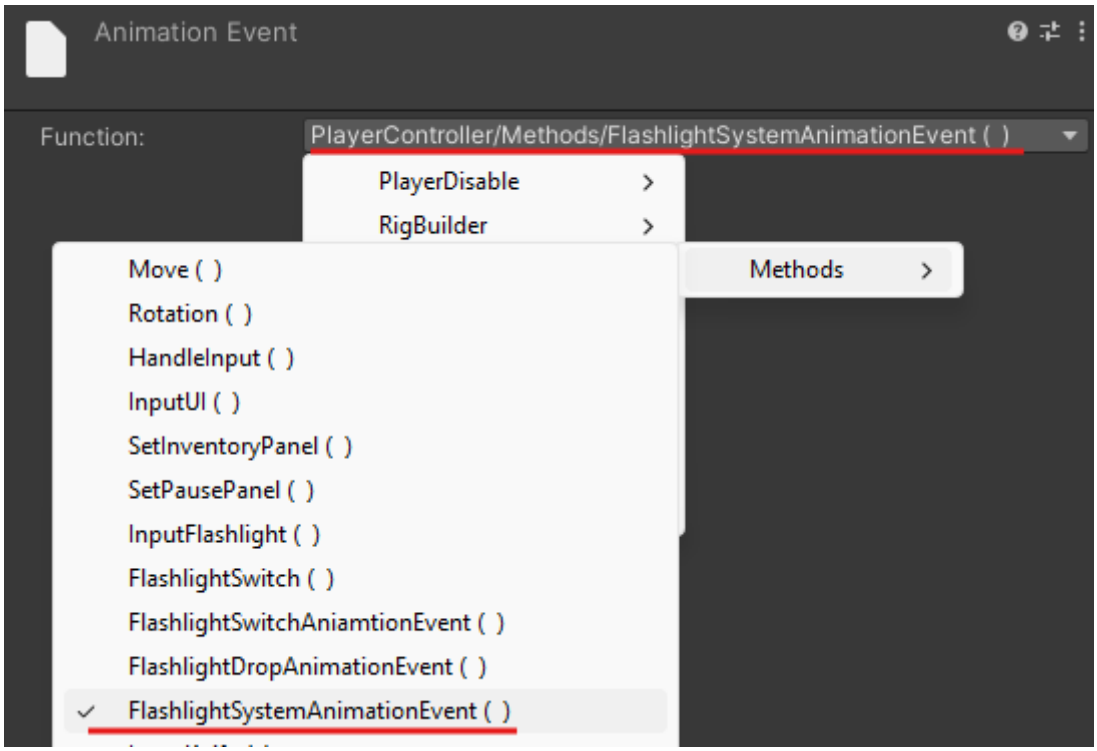






The flashlight and weapon are toggled via an animation event

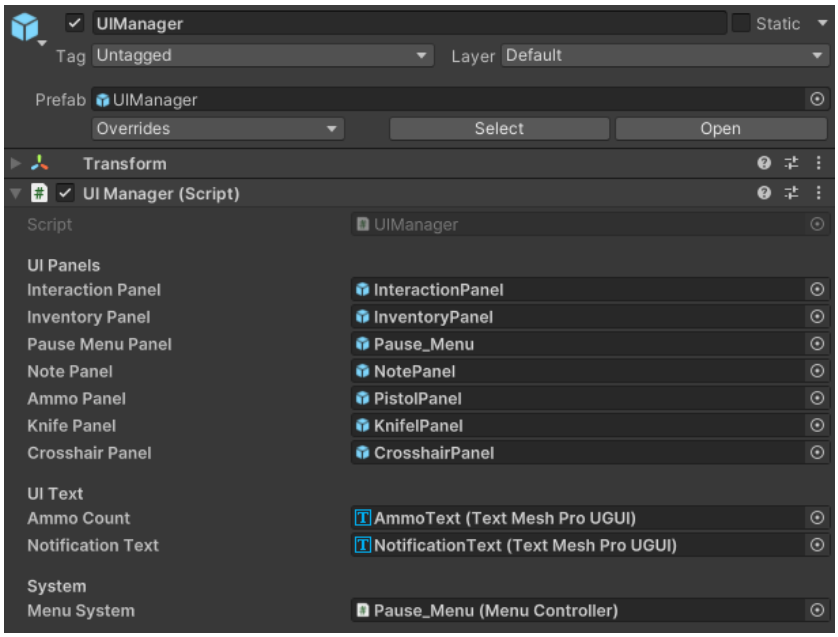


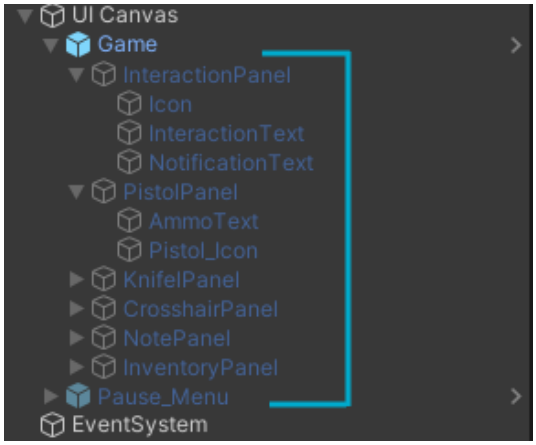


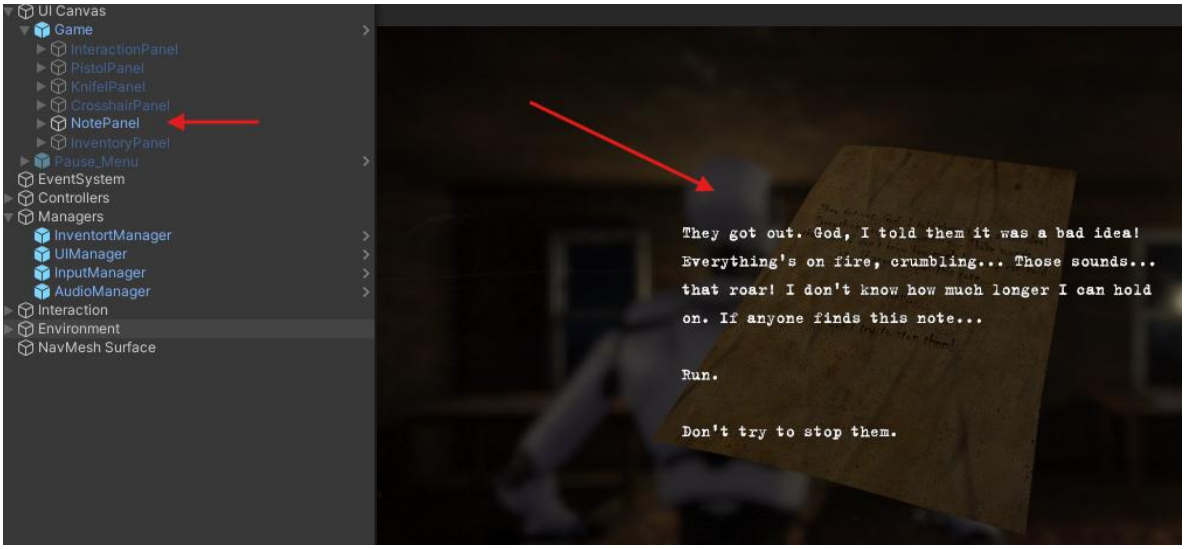

```
public void FlashlightSwitchAniamtionEvent()
{
    InventoryManager.instance.SetFlashlightActive(isFlashlight);
    AudioManager.instance.Play(getFlashlightSound);
}
```

UIManager

The **UIManager** is responsible for managing UI elements on the screen—such as enabling/disabling the pause menu when its button is pressed or showing relevant HUD elements when a weapon is equipped.

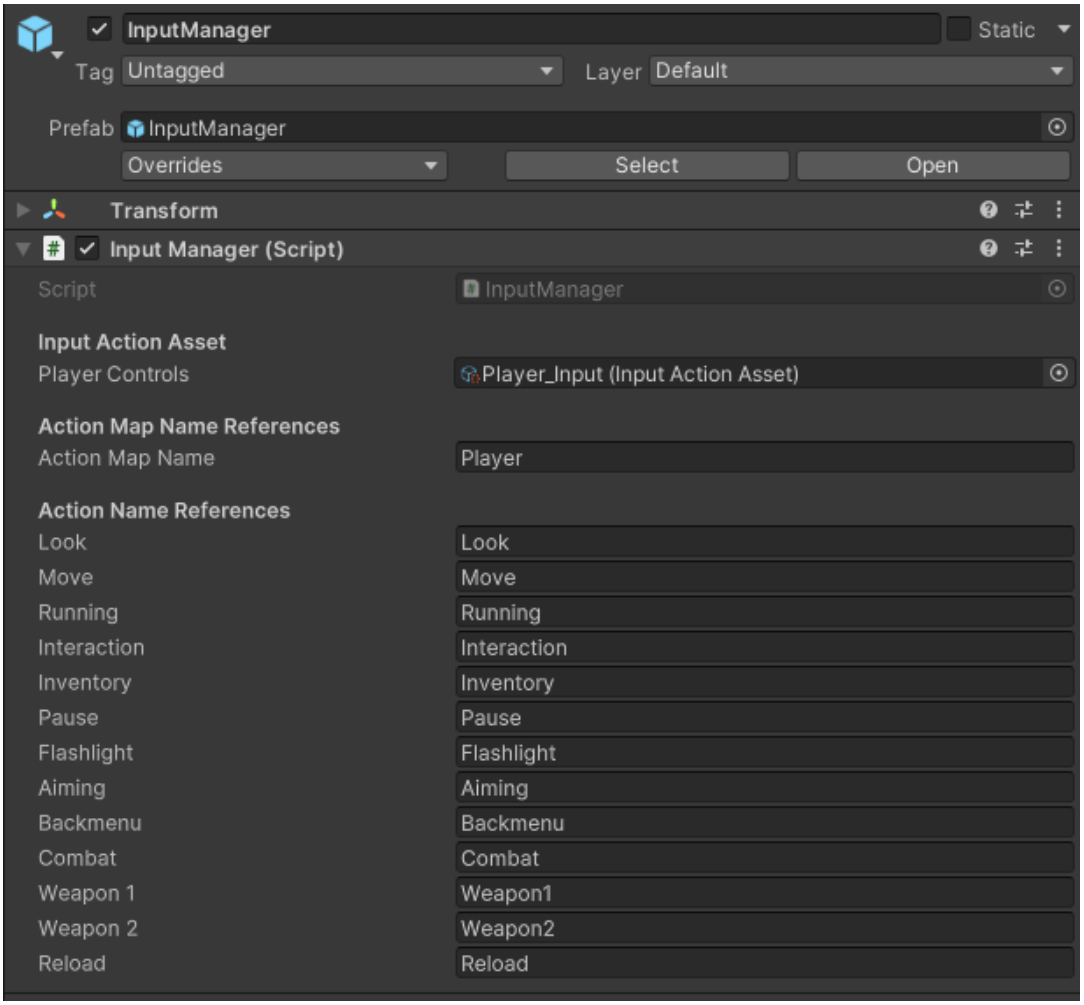






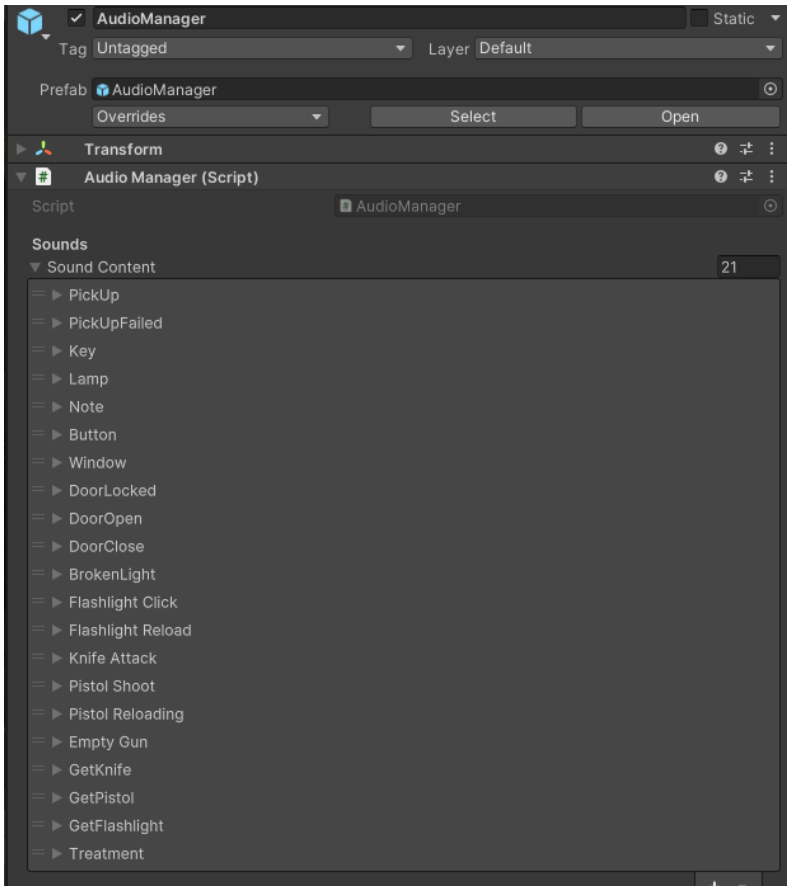
InputManager

The **InputManager** handles all input processing in the game. It acts as an intermediary between player inputs (keyboard, mouse, gamepad, etc.) and game actions. For a more comprehensive understanding of the input system implementation, please refer to the dedicated **Input System** section mentioned earlier in this documentation.



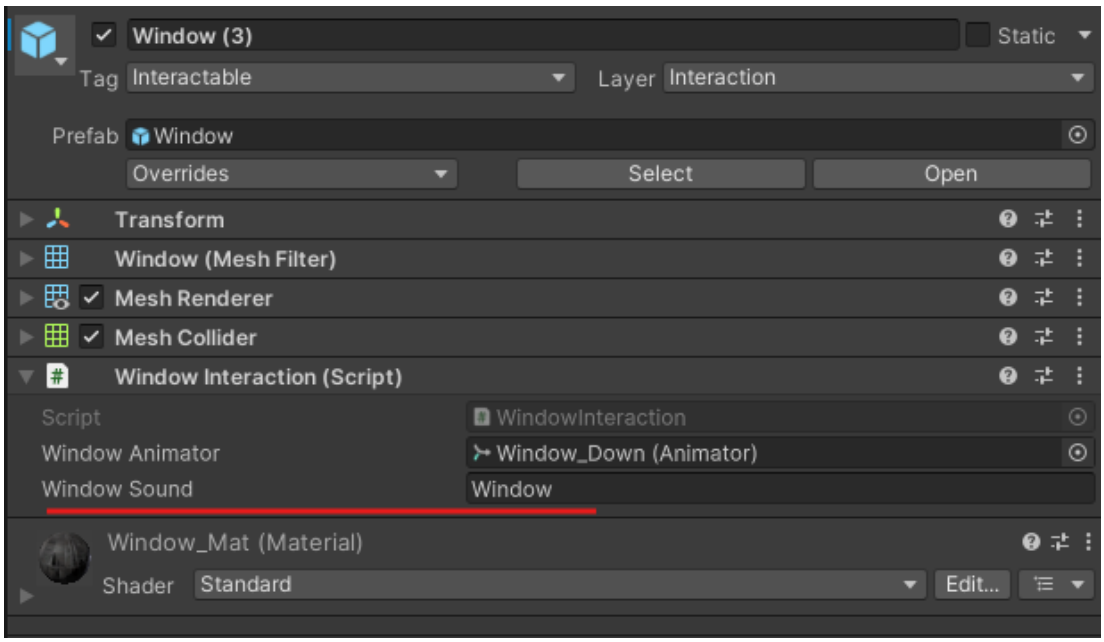
AudioManager

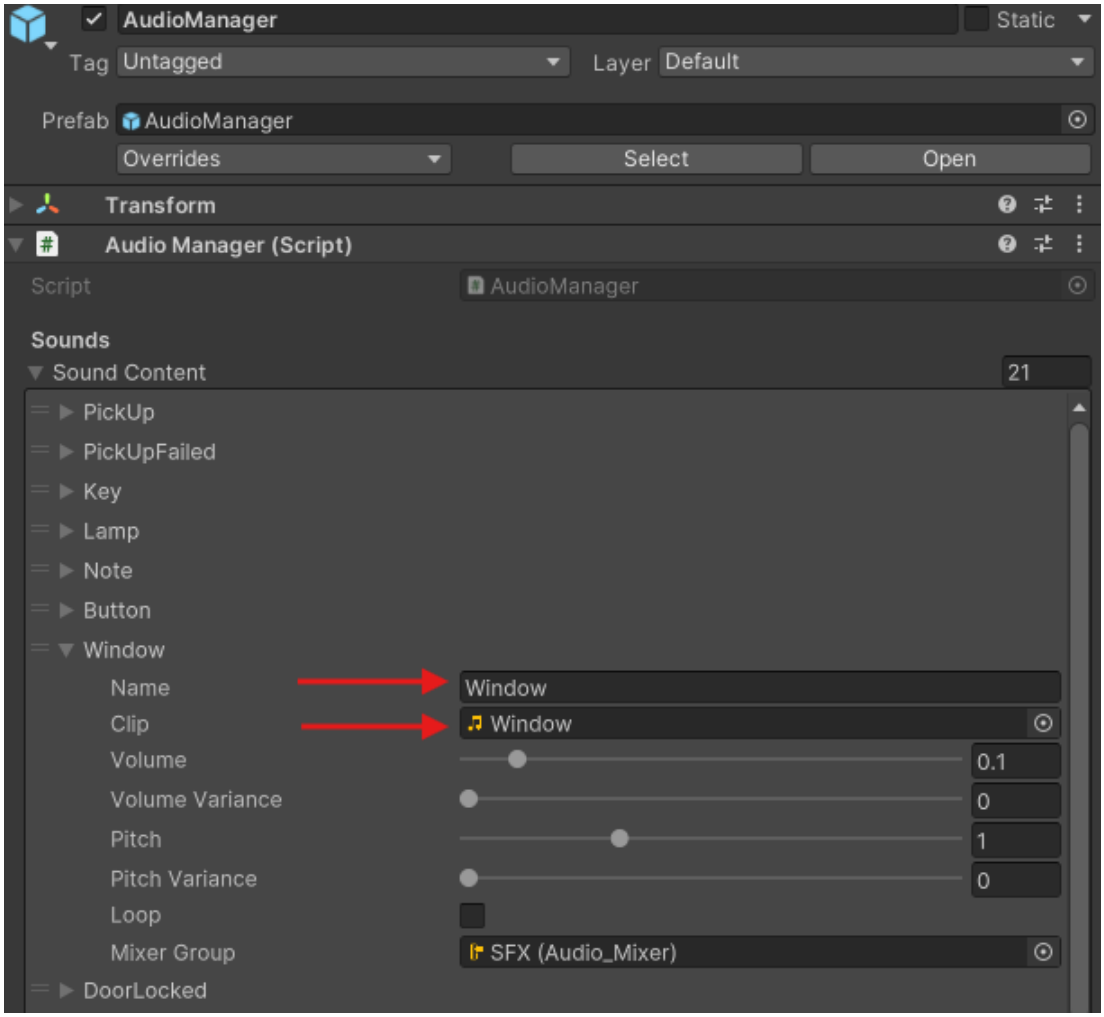
The **AudioManager** serves as the centralized sound management system that controls all audio playback within the game scene.



Example of triggering a sound effect through code:

```
using UnityEngine;

Скрипт Unity (1 ссылка на ресурсы) | Ссылок: 0
public class WindowInteraction : InteractiveObject
{
    public Animator windowAnimator;
    public string windowSound;

    protected bool isWindowOpen = false;

    Ссылок: 4
    public override void Interact(GameObject player = null)
    {
        isWindowOpen = !isWindowOpen;
        AudioManager.instance.Play(windowSound);

        windowAnimator.SetBool("WindowOpen", isWindowOpen);
        windowAnimator.SetBool("WindowClose", !isWindowOpen);
    }
}
```


Window (3)
Static
Tag Interactable
Layer Interaction
Prefab Window
Overrides
Select
Open
Transform
Window (Mesh Filter)
Mesh Renderer
Mesh Collider
Window Interaction (Script)
Script WindowInteraction
Window Animator Window_Down (Animator)
Window Sound Window
Window_Mat (Material)
Shader Standard
Edit...


AudioManager
Static
Tag Untagged
Layer Default
Prefab AudioManager
Overrides
Select
Open
Transform
Audio Manager (Script)
Script AudioManager
Sounds
Sound Content 21
PickUp
PickUpFailed
Key
Lamp
Note
Button
Window
Name Window
Clip Window
Volume 0.1
Volume Variance 0
Pitch 1
Pitch Variance 0
Loop
Mixer Group SFX (Audio_Mixer)
DoorLocked

IMPORTANT!

This documentation covers the basics of the asset and does not explain every mechanic in minute detail. If you encounter any difficulties or discover an error/bug, we recommend joining our Discord server, where you can ask questions and we'll be happy to help. Additionally, you can suggest your ideas for future updates and share your projects.

Good luck with your development!