Lancaster University College
at Beijing Jiaotong University

Technical Report

Group 8

Zou Zeng 21722040 Zhao Zikuo 21722039
Yang Haotian 21722035 Xue WenTao 21722033
Xue Congran 21722032 Li Shaoxu 21722009

Table of Contents

1. Technology Overview **3**

1.1 Coding structure of whole game 3
1.2 Coding style 3
1.3 Unity's technical route selection 4

2.Players core Mechanics **5**

2.1 IK 6
2.2 Blend tree 10
2.3 Motion capture 13

3. Camera and Aim **16**

3.1 Overview effect of Camera 16
3.2 Implement detail of Camera. 17
3.3 Aim system 19

4. Weapon System **22**

4.1 Overview 22
4.2 Code structure 22
4.3 Details implementation in Unity 24
4.4 Optimization – Object pool 25

5. Interaction System **26**

5.1 Overview 26
5.2 Logical exposition 27

6. damage system **28**

6.1 Overview 28
6.2 Core Components 28
6.3 Key Technologies and Techniques 30
6.4 Conclusion 31

7. Enemy Melee and Enemy variant **31**

7.1 Overview 31
7.2 Unity Implementation Process 32
7.3 Code Implementation 33
7.4 Optimization Section 34

8. Enemy Range and Enemy Boss **35**

8.1 Enemy Range Overview 35
8.2 Process 35

8.3 Brief description of Enemy Range 37
8.4 Special Characteristics of Enemy Range 38
8.5 Boss Overview 41
8.6 Process 41
8.7 Brief description of Enemy Boss 42
8.8 Skills implementation of Enemy Boss 44

9.UI **47**

9.1 Overview 47
9.2 Implementation 47

10 Audio **49**

10.1 Audio Source and Audio mixer 49
10.1.1 Overview 49

11 Car and LogiSteeringWheel **52**

11.1 WheelColider 52
11.2 Logitech steering wheel external 57
11.3 The interaction between the car and the player 58
11.4 Other 60

12. Online part **61**

12.1 Understanding Client-Server Model 61
12.2 Core Concepts 62
12.3 Remote Procedure Calls (RPCs) 63
12.4 Client-Side Prediction Code 64
12.5 Listening to Network Events 65
12.6 Physics Prediction 67
12.7 Cost of Predicting PhysX (Rigidbody3D) 67

13. Task and dialog system **67**

13.1 Overview 67
13.2 Code diagram 68
13.3 The method of using the system. 70

1. Technology Overview

1.1 Coding structure of whole game

This FPS game features an extensive codebase with strong interdependencies, organized into specific packages to maintain manageability. Player functionalities, including weapon management, are grouped under the Player package, reflecting their nested relationships. Standalone functionalities, such as game boundaries (ZoneLimit.cs) and target locking (Target.cs), are placed outside any specific package for easy access. With over a hundred scripts and tens of thousands of lines of code, this modular approach helps manage the complexity of our most challenging project.

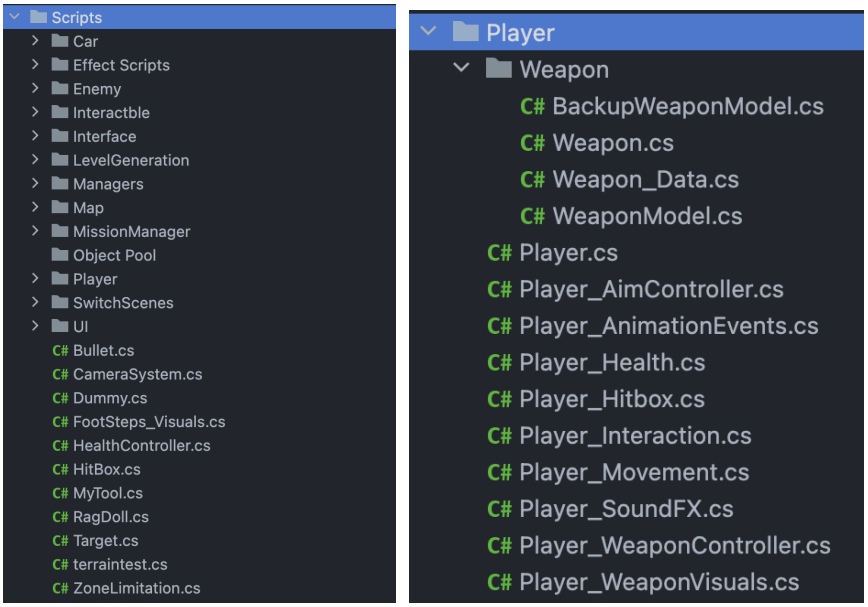


Figure 1. Code package

1.2 Coding style

Naming Conventions

- Class and Variable Names:
  - Class names and public variable names use PascalCase, e.g., Player, PlayerControls.
  - Private variables and method names use camelCase, e.g., controlsEnabled, SetControlsEnabledTo.

Class Structure

- Declaration Order of Class Members:
  - Public variables and properties are declared first, followed by private variables and methods.
  - Constants and static members are declared before instance members.

Access Modifiers

- Access Modifiers: Use access modifiers (public, private, protected, etc.) as explicitly as possible to define the scope of class members.

Properties and Auto-properties

- Auto-properties: Use auto-properties (e.g., public PlayerControls controls { get; private set; }) to simplify code.

Initialization and Assignment

- Initialization Order:
    - Initialize components and properties in the Awake method to ensure initialization is complete before the Start method.
    - Use the Awake method to get component references and initialize variables.

Event Handling

- Event Subscription and Unsubscription:
    - Subscribe to events in the OnEnable method and unsubscribe in the OnDisable method to avoid memory leaks caused by unremoved event subscriptions.

Method Design

- Method Names: Method names should be verb phrases that clearly describe the method's functionality, e.g., SetControlsEnabledTo.
- Method Parameters: Parameter names should be concise and meaningful, reflecting the purpose of the parameters.

Code Simplicity

- Code Simplicity: Avoid unnecessary comments and verbose code, maintaining simplicity and readability.
- Modular Design: Maintain modularity by dividing functionalities into different classes and methods, enhancing code maintainability and reusability.

Miscellaneous

- Event Delegates: Use event delegates to handle input and UI interactions, e.g., using performed to handle UI actions.
- Component Retrieval: Use the GetComponent method to retrieve component references and perform initialization in the Awake method to ensure components are properly initialized before use.

1.3 Unity’s technical route selection

We chose Unity for our project mainly due to its exceptional precision and accuracy in control handling, which is critical for delivering a smooth gameplay experience. Unity's input system is highly customizable and responsive, allowing us to fine-tune player controls to achieve the exact feel we want. This level of control is crucial for ensuring that movements and interactions in the game are intuitive and responsive.

Compared to Unreal Engine (UE), Unity's scripting environment in C# offers more straightforward and readable code, which simplifies the process of implementing precise control logic. In our experience, Unity's component-based architecture makes it easier to isolate and adjust specific aspects of player control without affecting other systems, enhancing our ability to deliver precise and accurate player inputs.

Additionally, Unity’s real-time debugging and fast iteration cycles are significant advantages. Being

able to quickly test and refine control mechanics directly in the editor saves a substantial amount of development time and helps us rapidly address any issues with input precision. Unity's extensive documentation and community support also provide valuable resources for troubleshooting and optimizing control schemes.

In contrast, while Unreal Engine offers powerful tools and visual scripting capabilities, we found that the learning curve and complexity of Blueprints can sometimes slow down the development process, especially for precise control tuning. Unity's more straightforward scripting approach in C# allows us to achieve high precision in control handling more efficiently.

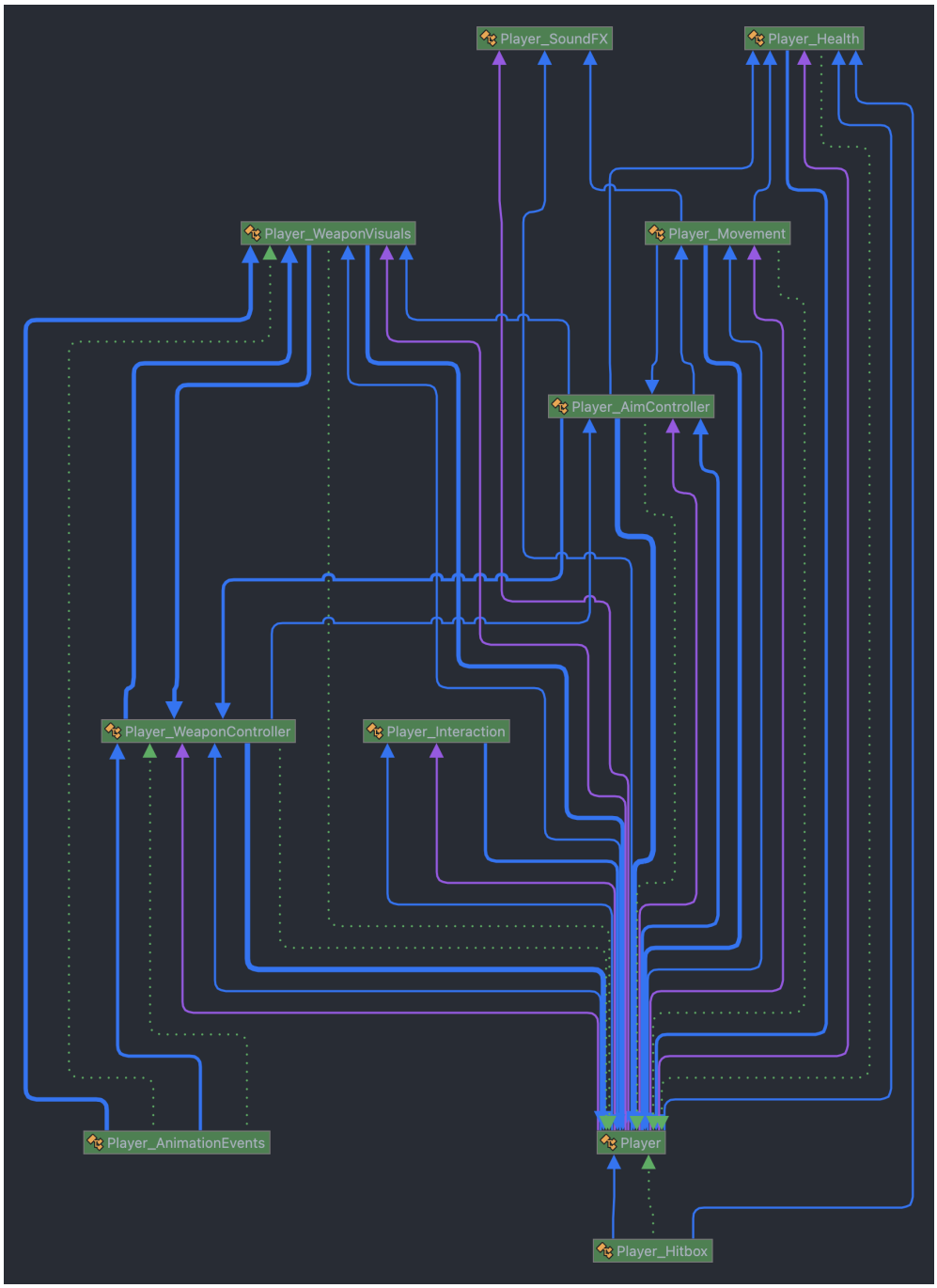
In summary, Unity's precise control capabilities, combined with its user-friendly scripting environment and robust debugging tools, make it an ideal choice for our project, ensuring that we can deliver a highly responsive and accurate gameplay experience.

2.Players core Mechanics

Before readers really delve into the technology we use, it is proper give the structure of the diagram and briefly explain the logic and relationship in it. The diagram only contains relevant C# classes about player.

Here is our explanation:

1. **Central Coordination:** The Player class acts as the central hub, coordinating the flow of information and actions between all other classes.
2. **Visual and Functional Synchronization:** Player_WeaponVisuals and Player_WeaponController work closely to ensure that visual representations of weapons match their functional states.
3. **Aiming and Movement:** Player_AimController and Player_Movement ensure that aiming and movement mechanics are accurately reflected in both visuals and functionality.
4. **Health and Damage:** Player_Health and Player_Hitbox work together to manage health and damage, ensuring that the player’s health status is accurately tracked and updated based on in-game events.
5. **Interactions and Sounds:** Player_Interaction and Player_SoundFX manage player interactions with the game world and ensure that corresponding sounds are played to enhance the immersive experience.
6. **Animation Events:** Player_AnimationEvents and Player_AnimationController ensure that animations are triggered correctly based on player actions, providing a seamless visual experience.



2.1 IK

2.1.1Overview

Inverse Kinematics (IK), also known as Animation Rigging, is a powerful technique used in Unity for creating realistic and deeply customized animations. By manipulating the IK components, I was able to design unique gun-holding postures for each player character in our project.

Key Features of IK Implementation:

1. Customized Shooting Actions:

- Each player character was given a unique shooting action tailored to their specific weapon, enhancing the gameplay experience and visual appeal. As shown in the picture below.

2. Real-Time Adjustment:
   - IK allows for real-time adjustments, making it possible to dynamically alter character poses based on the environment and gameplay scenarios. This flexibility is crucial for maintaining realistic and engaging animations.
3. Enhanced User Engagement:
   - By providing distinct and personalized animations for each character, user engagement was significantly improved. Players experienced a more immersive and responsive interaction with the game.
4. Technical Integration:
   - Utilizing Unity's IK tools, we integrated this technology seamlessly into our development pipeline. This included setting up the IK rig, defining constraints, and fine-tuning the animations to achieve the desired effect.

Implementing IK in Unity allowed for a high degree of customization in character animations, particularly in shooting mechanics. This not only improved the visual fidelity of the game but also contributed to a more immersive and enjoyable player experience. The ability to dynamically adjust and personalize animations based on gameplay contexts highlights the versatility and power of IK technology in modern game development.

2.1.2Coding Diagram

In our Unity project, we employ Inverse Kinematics (IK) to create realistic and dynamic animations for character actions, particularly focused on weapon handling and shooting mechanics. Here's a detailed explanation of how we achieve this:

Rigging Process:

Bone Binding:

- We start by binding various bones to specific joints within the character rig. Each joint is responsible for controlling the direction and movement of the corresponding bone. For instance, the HeadAim, GunAim, and LeftHand_IK are key components in our rig setup.

Joint Control:

- Each of these joints is manipulated to ensure precise control over the character's movements. The Multi-Aim Constraint and Two Bone IK Constraint components are configured to manage the direction and orientation of the character's head, gun, and hands.

Action Transitions:

Manual Adjustments:

- During action transitions, such as moving from idle to shooting or reloading, we manually adjust the rig values. This ensures a seamless blend between different IK actions and other animations, maintaining fluid and realistic character movements.

Rig Weight Control:

- To manage the influence of IK on the character's movement, we utilize a dedicated class to control the rig weight. This is done through code, as shown in the provided diagram, where methods like visualController.MaximizeRigWeight() and visualController.MaximizeLeftHandWeight() are called to adjust the rig's influence dynamically.

Technical Integration:

Real-Time Adjustments:

- Our implementation allows for real-time adjustments to the rig weights, which is crucial for responding to various gameplay scenarios. This flexibility enhances the overall player experience by providing responsive and adaptive character animations.

Code Implementation:

- The code snippet demonstrates how we integrate these adjustments programmatically. By encapsulating the rig weight control in a class, we ensure that the transitions are smooth and maintain high performance.

By leveraging advanced IK techniques and rigorous manual adjustments, we achieve a level of realism and responsiveness that sets our project apart. This approach showcases the power of IK in modern game development and underscores our commitment to delivering high-quality animations.

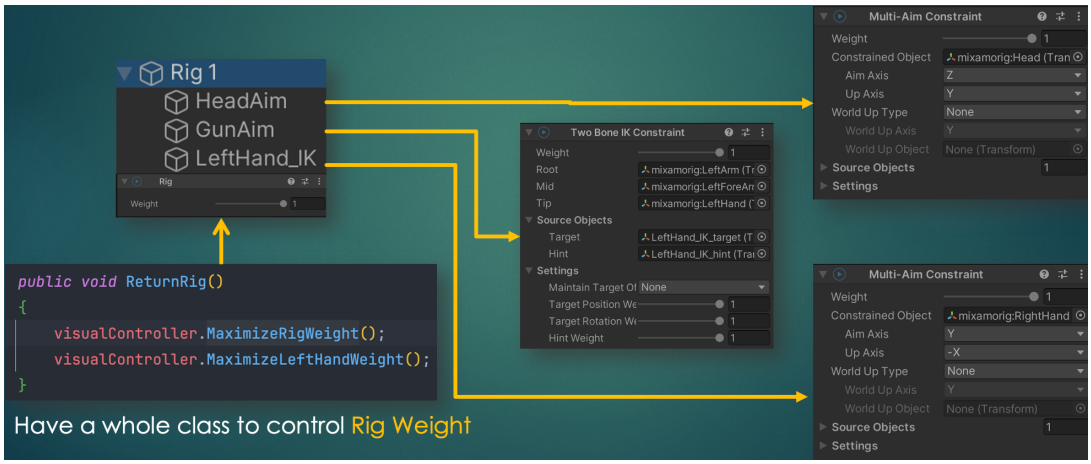


Figure 2. Shooting

2.1.3 Implement detail

The following explanation shows how me manipulate rig weight in Player_WeaponVisuals classes

In our Unity project, we utilize various techniques to adjust the rig weight and coordinate the IK components with character animations, particularly for weapon handling. Below is a summary of how we achieve this:

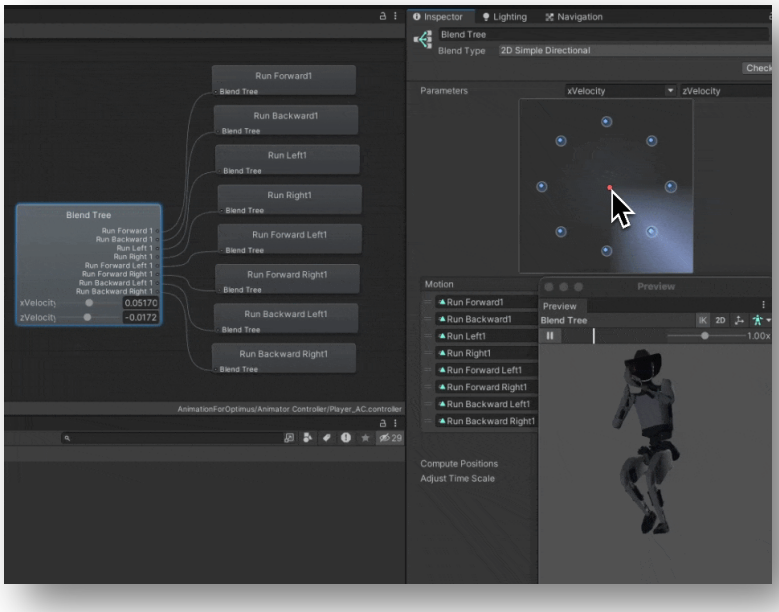
1. **Rig Initialization and Components:**
    - We initialize the player, animator, and rig components in the Start method. Weapon models and backup weapon models are also retrieved and set up at this stage.
    - The Rig, TwoBoneIKConstraint, and relevant target transforms are set up to manage the character's IK and animation rigging.
2. **Rig Weight and IK Weight Management:**
    - We dynamically adjust the rig weight and left-hand IK weight during the Update method by checking flags (shouldIncrease_RigWeight and shouldIncrease_LeftHandIKWieght).
    - The UpdateRigWigth and UpdateLeftHandIKWeight methods incrementally increase the weights based on predefined rates until they reach their maximum values.
3. **Action Transitions:**
    - **Firing and Reloading:**
        - PlayFireAnimation triggers the fire animation.
        - PlayReloadAnimation sets the reload speed, triggers the reload animation, and reduces the rig weight to ensure smooth transitions.
    - **Weapon Equip:**
        - PlayWeaponEquipAnimation handles the weapon equip animation, adjusting the IK weights and rig weight to facilitate the transition.
        - During equipment, the left-hand IK weight is set to zero before gradually increasing it again.
4. **Handling Weapon Models:**
    - Methods such as SwitchOnCurrentWeaponModel, SwitchOffWeaponModels, SwitchOffBackupWeaponModels, and SwitchOnBackupWeaponModel manage the activation states of different weapon models.
    - The AttachLeftHand method aligns the left-hand IK target with the current weapon’s hold point.
5. **Layer Management:**
    - SwitchAnimationLayer sets the appropriate animation layer based on the current weapon model to ensure that the correct animations are played.
6. **Code Implementation:**
    - The MaximizeRigWeight and MaximizeLeftHandWeight methods are used to set the flags that control the gradual increase of rig and IK weights.
    - ReduceRigWeight is called to lower the rig weight during specific actions to ensure smooth and coordinated animations.

By integrating these components and methods, we ensure that the character's movements are fluid and responsive, providing a seamless and immersive gameplay experience. The use of IK and rigging techniques allows for precise control over character animations, particularly during complex actions such as firing, reloading, and weapon switching.

2.2 Blend tree

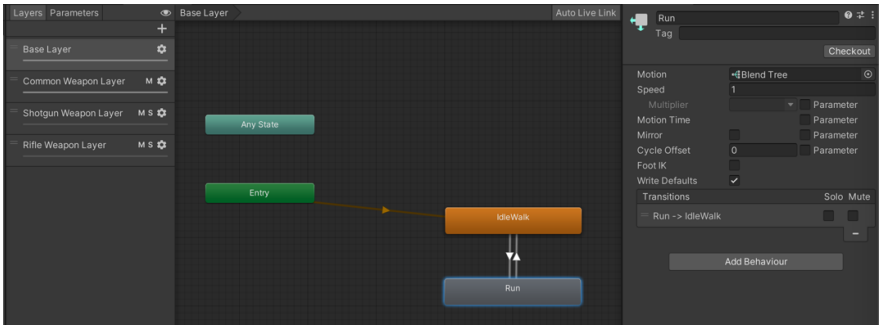
2.2.1 Overview

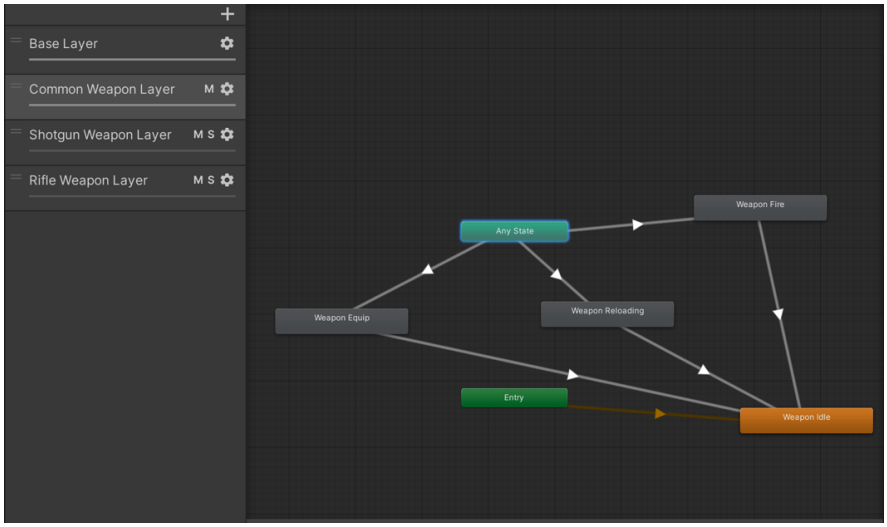
Using blend trees in Unity, we can create smooth and dynamic animations for characters by blending a few key actions. This technique allows us to generate movements of varying intensities in any direction, enhancing the realism and responsiveness of character animations. By adjusting parameters such as velocity, we can seamlessly transition between actions like running, walking, and idle states, ensuring fluid and natural motion in the game environment. This approach simplifies the animation process while providing extensive control over character behaviors.



2.2.2 Coding Diagram

Although blend tree does not have any specific diagram, it is appropriate to introduce our state machine of the main character.





The character's state machine in Unity is designed to control both the upper and lower body animations efficiently, ensuring fluid and coordinated movements. Here's an overview based on the provided images and descriptions:

Base Layer:

The Base Layer primarily controls the upper body animations. The Run state within this layer is managed by a blend tree, which allows for smooth transitions between different running directions and speeds.

Weapon Layers:

There are three specific weapon layers: Common Weapon Layer, Shotgun Weapon Layer, and Rifle Weapon Layer. These layers handle the animations related to different types of weapons.

Common Weapon Layer: This layer includes general weapon animations such as Weapon Equip, Weapon Fire, and Weapon Reloading.

Shotgun Weapon Layer & Rifle Weapon Layer: Both of these layers are synchronized with the Common Weapon Layer to ensure consistent animations across different weapon types. They control the specific behaviors related to shotguns and rifles, respectively.

Animation Synchronization:

The synchronization between the Common Weapon Layer and the specific weapon layers (Shotgun Weapon Layer and Rifle Weapon Layer) ensures that any changes or transitions in the common layer are reflected accurately in the specialized layers. This setup allows for modular and scalable animation management.

Blend Trees:

The use of blend trees in the Run state of the Base Layer enables the creation of dynamic and responsive animations. By adjusting parameters like xVelocity and zVelocity, the blend tree smoothly transitions between different running animations, creating a natural movement experience.

State Transitions:

The state machine includes transitions from Any State to key actions like Weapon Equip, Weapon Fire, and Weapon Reloading. This ensures that the character can respond quickly to player inputs, switching between different states as needed.

The Weapon Idle state serves as the default idle state for the character when no other actions are being performed.

2.2.3 Implement detail

The following function demonstrate how we get the xVelocity and zVelocity and then control the blend tree behaviours

```
private void AnimatorControllers()
{
    float xVelocity = Vector3.Dot(lhs: movementDirection.normalized, rhs: transform.right);
    float zVelocity = Vector3.Dot(lhs: movementDirection.normalized, rhs: transform.forward);

    animator.SetFloat(name: "xVelocity", value: xVelocity, dampTime: .1f, Time.deltaTime);
    animator.SetFloat(name: "zVelocity", value: zVelocity, dampTime: .1f, Time.deltaTime);

    bool playRunAnimation = isRunning & movementDirection.magnitude > 0;
    animator.SetBool(name: "isRunning", playRunAnimation);
}
```

Next, I will introduce more about Player_Movement.cs which include the content of how we run our state machine.

Components:

- Player player: Reference to the player script.
- CharacterController characterController: Handles movement physics.
- PlayerControls controls: Custom input controls.
- Animator animator: Manages animations.
- AudioSource walkSFX, runSFX: Plays footstep sounds.

Movement Parameters:

- walkSpeed, runSpeed, turnSpeed: Controls movement speeds.
- moveInput: Stores movement input values.
- movementDirection: Direction of movement.
- isRunning: Indicates if the player is running.
- canPlayFootsteps: Controls footstep sound playback.
- verticalVelocity: Handles vertical movement.

Core Methods:

- Start():
  - Initializes components, sets up variables, and assigns input events.
  - Example: Initializes player, characterController, animator, walkSFX, and runSFX.
- Update():
  - Main update loop that applies movement, rotation, and animation updates every frame.
  - Checks if the player is dead, then applies movement and rotation, and updates animations.
- AnimatorControllers():
  - Updates animation parameters like xVelocity, zVelocity, and isRunning.
  - Example: Calculates xVelocity and zVelocity and sets animator parameters.
- ApplyRotation():
  - Adjusts player rotation smoothly towards the aim direction.
  - Example: Uses Quaternion.Slerp to interpolate the player's rotation.
- ApplyMovement():
  - Ensures CharacterController is active, applies movement direction and gravity, and

moves the character.
    - Example: Calculates movementDirection based on moveInput and moves the character.
- PlayFootstepsSFX():
    - Plays appropriate footstep sounds based on the player's state.
    - Example: Checks isRunning and plays runSFX or walkSFX.
- StopFootstepsSFX():
    - Stops all footstep sounds.
    - Example: Stops both walkSFX and runSFX.
- AllowfootstepsSFX():
    - Enables footstep sounds to play after a delay.
    - Example: Sets canPlayFootsteps to true after 1 second.
- ApplyGravity():
    - Adjusts vertical velocity for gravity effects when the player is not grounded.
    - Example: Sets verticalVelocity based on whether the player is grounded.
- AssignInputEvents():
    - Binds input actions for movement and running, updating speed and stopping footstep sounds as necessary.
    - Example: Uses controls.Character.Movement.performed to update moveInput.

2.3 Motion capture

2.3.1 Overview

In our project, we implemented motion capture technology to enhance the realism and responsiveness of NPC (Non-Playable Character) animations.

Here are some of the key benefits and advantages of using this technology:

1. **Realistic Animations:**
    - **High Fidelity Movements:** Motion capture allows us to capture real human movements and translate them directly into the game. This results in highly realistic and fluid animations for NPCs.
    - **Natural Interactions:** NPCs can interact with the environment and player in a more natural and believable manner, enhancing the overall immersion of the game.
2. **Efficiency and Consistency:**

- **Time-Saving:** Capturing complex animations through motion capture is significantly faster than manually animating each movement. This efficiency allows us to produce a large volume of high-quality animations in a shorter period.
- **Consistent Quality:** Motion capture ensures that animations are consistent across different NPCs and scenarios, maintaining a high level of quality throughout the game.

3. **Enhanced Player Experience:**
   - **Immersive Gameplay:** Realistic NPC animations contribute to a more immersive gameplay experience, making the game world feel more alive and dynamic.
   - **Emotional Engagement:** NPCs with lifelike movements can convey emotions and reactions more effectively, creating a deeper emotional connection with the player.
4. **Versatility:**
   - **Adaptable Animations:** Motion capture data can be easily adapted and reused for different characters and actions, providing versatility in animation design.
   - **Customizable Actions:** We can capture a wide range of movements, from everyday actions to complex combat sequences, and customize them to fit the specific needs of different NPCs.
5. **Improved Development Workflow:**
   - **Streamlined Pipeline:** Integrating motion capture into our development pipeline allows for a more streamlined and efficient workflow, reducing the time and effort required to create detailed animations.
   - **Real-Time Feedback:** Motion capture provides real-time feedback during the capture process, enabling immediate adjustments and refinements to ensure the highest quality animations.

2.3.2 Implement detail

Here we will introduce how we use this device. Since we are unity, our communication method is cumbersome than UE.

Here's how we utilize this equipment for our NPC animations, covering aspects such as wearing, calibration, and data export:

1. Wearing the Equipment

- **Full-Body Suit:**
  - The PN 3 Pro system includes a full-body compression suit with 17 inertial sensors that need to be worn by the actor. This suit ensures that sensors are securely positioned at key points on the body to accurately capture movements.
  - The suit is designed to be comfortable and allow for a full range of motion, ensuring that the actor can perform naturally without restrictions.
- **Hand Gloves:**
  - The system also includes motion capture gloves for capturing finger movements. These gloves are equipped with additional sensors to accurately record the fine details of hand and finger actions.
- **Strap System:**
  - For areas not covered by the compression suit, such as the head, additional straps

are used to secure sensors. These straps ensure that sensors remain in place during complex movements.

2. Calibration

- **Sensor Calibration:**
  - Before capturing motion data, each sensor must be calibrated. This process involves setting up the sensors to accurately detect their orientation and position relative to the actor's body.
  - Calibration typically involves the actor standing in a specific pose (e.g., T-pose) and performing a series of predefined movements to allow the system to adjust and synchronize the sensors.
- **Algorithm Adjustments:**
  - PN 3 Pro uses advanced sensor calibration algorithms that enhance data quality by correcting for any misalignments or discrepancies in sensor readings.
  - The system also features new human motion algorithms and post-processing algorithms that further refine the captured data to ensure high accuracy and realism.

3. Capturing Motion Data

- **Real-Time Recording:**
  - Once the system is calibrated, the actor can perform the desired movements. The PN 3 Pro captures these motions in real-time, allowing for immediate visualization and feedback.
  - This real-time capability helps in making quick adjustments to the performance or sensor placement if needed.
- **Finger Motion Capture:**
  - The inclusion of hand gloves enables detailed capture of finger and hand movements, adding a layer of realism to the animations that is particularly useful for actions requiring intricate hand gestures.

4. Data Export

- **Saving the Data:**
  - After recording the motion capture session, save the motion data in a format compatible with Unity, such as FBX or BVH. The PN 3 Pro software provides options to export the recorded sessions.
- **Data Export Settings:**
  - Ensure the export settings match the requirements for Unity. For example, selecting the appropriate frame rate and ensuring that the skeleton hierarchy aligns with Unity's rigging standards.

3. Camera and Aim

3.1 Overview effect of Camera

Our game features the ability to switch between a 60-degree top-down view and a third-person perspective. This dual-view setup offers several gameplay benefits that enhance the player experience:

1. Enhanced Tactical Awareness

- **Top-Down View:**
  - The 60-degree top-down view provides a broad overview of the environment. Players can see more of the battlefield or game area, which is particularly useful for planning strategies, navigating complex levels, and managing multiple objectives.
  - This perspective helps in spotting enemies, traps, and other important elements that might be missed in a more restricted view. It's ideal for tactical gameplay where situational awareness is crucial.

2. Immersive Action Experience

- **Third-Person View:**
  - Switching to a third-person perspective brings the player closer to the action, making the gameplay more immersive and engaging. Players can see their character's detailed animations, equipment, and immediate surroundings.
  - This view is perfect for intense combat situations, exploring intricate environments, and experiencing the game world more personally. It allows players to connect more with their character and enhances the emotional impact of the gameplay.

3. Versatility in Gameplay

- **Adaptive Play Styles:**
  - The ability to switch between views caters to different player preferences and play styles. Some players might prefer the strategic advantage of a top-down view, while others might enjoy the immersive experience of a third-person view.

- This flexibility allows players to adapt their perspective based on the current gameplay situation, enhancing their overall experience and satisfaction.

4. Better Environmental Interaction

- **Enhanced Exploration:**
  - The top-down view aids in navigation and exploration, making it easier to understand the layout of the environment, find hidden items, and solve puzzles that require a broader perspective.
  - The third-person view allows for more detailed interaction with the environment, such as inspecting objects closely, interacting with NPCs, and experiencing environmental storytelling elements.

5. Improved Combat Mechanics

- **Strategic Combat:**
  - In the top-down view, players can plan their combat strategies more effectively, seeing enemy positions and movement patterns. This is beneficial for managing multiple enemies and coordinating attacks.
  - The third-person view offers a more visceral combat experience, where players can aim and attack with precision, dodge, and engage in close-quarter combat, making the fights more dynamic and exciting.

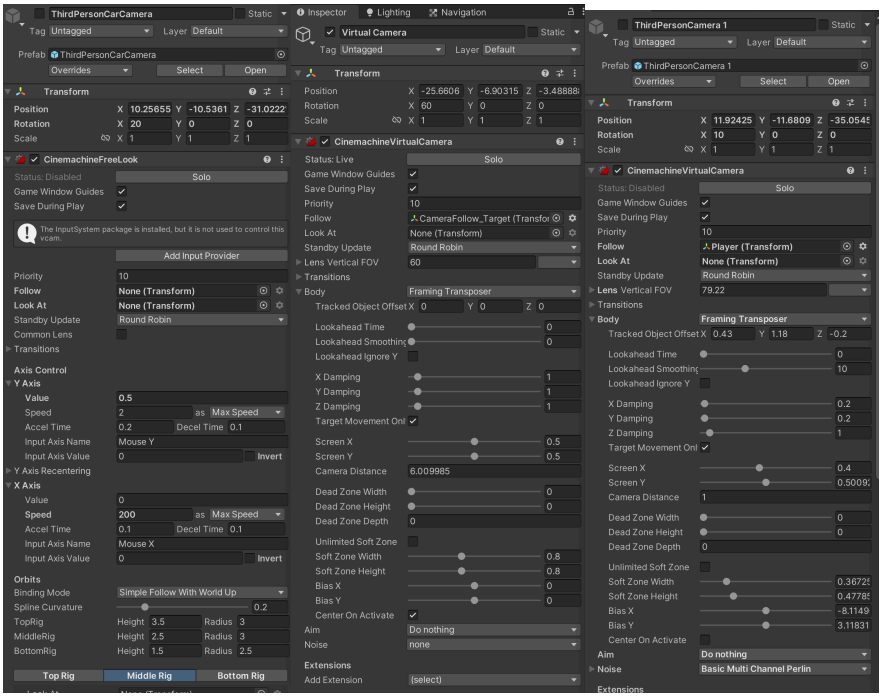
6. Dynamic Game Experience

- **Variety and Engagement:**
  - Switching between perspectives adds variety to the gameplay, preventing it from becoming monotonous. It keeps the player engaged by offering different ways to experience the game world and its challenges.
  - This dynamic approach can make the game feel more expansive and rich, as players can appreciate the game design from multiple angles.

3.2 Implement detail of Camera.

The whole camera switching system not only responsible for the switches of top-down view and third-person view, but it is also in charge of switching to our racing view.

Before we observe our code, here we introduce our unique setting of camera that makes a huge difference.



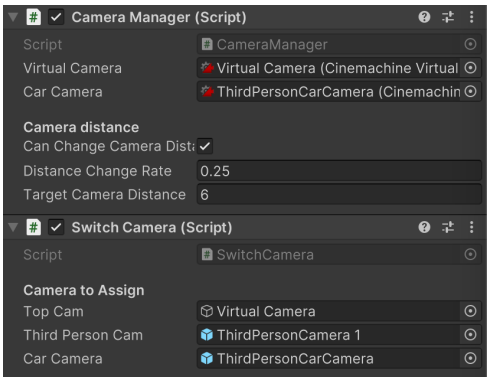
Besides these, we also have a main camera that has a CinemachineBrain to control all these camera

Camera Types:

1. **Third-Person Camera:**
    - **Component:** CinemachineVirtualCamera.
    - **Usage:** Provides a third-person perspective for general gameplay, allowing players to see their character from behind.
    - **Settings:**
        - **Follow and Look At:** Typically follows and looks at the player's transform.
        - **Framing Transposer:** Configured to maintain the player's position within a specific screen area while allowing smooth movement and rotation.
        - **Damping:** Adjusted to ensure smooth transitions and reduce jitter during character movements.
2. **Top-Down Camera:**
    - **Component:** CinemachineVirtualCamera.
    - **Usage:** Provides a 60-degree top-down view for tactical gameplay, offering a broad overview of the environment.
    - **Settings:**
        - **Follow and Look At:** Can follow and look at the player or a specific target area.
        - **Framing Transposer:** Adjusts the camera's position based on the target's movements, ensuring it stays within a defined screen area.
        - **Ortho Settings:** Often uses an orthographic projection to maintain consistent scale and view angle, useful for strategic planning and navigation.
3. **Racing Camera:**
    - **Component:** CinemachineFreeLook.

- **Usage:** Provides a specialized view for racing sequences, following the car from a dynamic third-person perspective.
- **Settings:**
  - **Follow and Look At:** Follows the car's transform, focusing on the vehicle to provide a dynamic racing experience.
  - **Rig Setup:** Configured with multiple rigs (Top, Middle, Bottom) to smoothly follow the car's movements and provide a cinematic feel.
  - **Axis Control:** Manages both X and Y-axis rotations, controlled by player input to offer full control over the camera angles during racing.

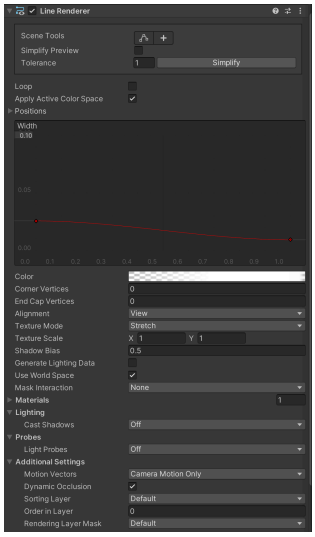
Also, we handle everything in our code, since the coding structure is much more simpler, you can see more detail in these two classes



3.3 Aim system

3.3.1 Laser-Assisted Aiming

To achieve a realistic laser beam effect emanating from the gun barrel, specialized settings are required. The desired effect includes the laser beam gradually fading out after a certain distance from the muzzle, as well as the beam's intensity decreasing both near the gun and as it extends away. These adjustments ensure the laser appears natural and enhances the overall visual experience in the game.



3.3.2 Shooting Mode Settings

For shooting sensation, we work very detail on following function and ideas

- Switching between precision shooting and parallel shooting modes
- Camera settings in precision shooting mode
- Camera settings in parallel shooting mode
- Different shooting modes targeting different layers
- The rate of camera adjustment
- The object the camera is aimed at (not the character itself)
- Whether to enable auto-aim assist to lock onto enemies (to accommodate the assistive function of physical firearms)
- The aiming reticle

In order to arrive at such different shooting patterns, we organize the behaviors of the camera and player body in Player_AimController.cs class. Now, we analyze together with our code.

Switching Between Precision Shooting and Parallel Shooting Modes

- Method: EnablePreciseAim(bool enable)
- Description: Toggles the precise aiming mode on and off. Adjusts camera distance and time scale based on the aiming mode.
- Code Snippet:

```
private void EnablePreciseAim(bool enable)
{
    isAimingPrecisly = !isAimingPrecisly;
    Cursor.visible = false;

    if (isAimingPrecisly)
    {
        cameraManager.ChangeCameraDistance(preicseAimCamDistance, camChangeRate);
        Time.timeScale = .9f;
    }
    else
    {
        cameraManager.ChangeCameraDistance(regularAimCamDistance, camChangeRate);
        Time.timeScale = 1f;
    }
}
```

Different Shooting Modes Targeting Different Layers

- Properties: LayerMask preciseAim and LayerMask regularAIm
- Usage: These layer masks are used to target different layers depending on the shooting mode. For example, the GetMouseHitInfo() method uses preciseAim layer mask for raycasting.
- Code Snippet:

```
public RaycastHit GetMouseHitInfo()
{
    Ray ray = Camera.main.ScreenPointToRay(mouseInput);

    if (Physics.Raycast(ray, out var hitInfo, maxDistance:Mathf.Infinity, preciseAim))
    {
        lastKnownMouseHit = hitInfo;
        return hitInfo;
    }

    return lastKnownMouseHit;
}
```

Whether to Enable Auto-Aim Assist to Lock onto Enemies

- Property: bool isLockingToTarget
- Usage: Toggles the auto-aim assist functionality.
- Code Snippet:

```
if (Input.GetKeyDown(KeyCode.L))
    isLockingToTarget = !isLockingToTarget;
```

The Aiming Reticle

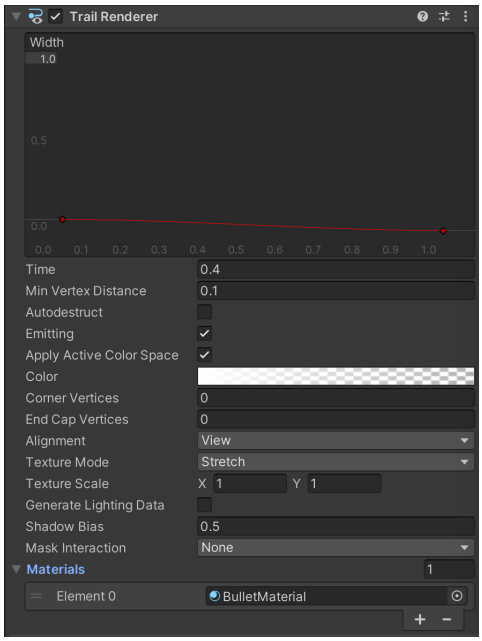
- Method: UpdateAimVisuals()
- Description: Updates the position and visibility of the aiming laser (reticle) based on the player's current aim.
- Code Snippet:

```
private void UpdateAimVisuals()
{
    aim.transform.rotation = Quaternion.LookRotation((Camera.main.transform.forward));

    aimLaser.enabled = player.weapon.WeaponReady();

    if (aimLaser.enabled == false)
        return;
    WeaponModel weaponModel = player.weaponVisuals.CurrentWeaponModel();
    weaponModel.transform.LookAt(aim);
    weaponModel.gunPoint.LookAt(aim);
    Transform gunPoint = player.weapon.GunPoint();
    Vector3 laserDirection = player.weapon.BulletDirection();
    float laserTipLenght = .5f;
    float gunDistance = player.weapon.CurrentWeapon().gunDistance;
    Vector3 endPoint = gunPoint.position + laserDirection * gunDistance;
    if (Physics.Raycast(origin:gunPoint.position, laserDirection, out RaycastHit hit, gunDistance))
    {
        endPoint = hit.point;
        laserTipLenght = 0;
    }
    aimLaser.SetPosition(index:0, gunPoint.position);
    aimLaser.SetPosition(index:1, endPoint);
    aimLaser.SetPosition(index:2, position:endPoint + laserDirection * laserTipLenght);
}
```

3.3.3 Bullets

Bullet Mechanics and Effects

- Bullets themselves have collision volumes and collision mass.
- The impact of a bullet is determined through its collision body.
- Bullet impacts trigger particle effects.
- Ensuring that the bullet's momentum remains constant ensures consistent collision effects with different objects.
- Bullets have a highlight and a unique trail effect, with specific tail renderer settings as follows.



4. Weapon System

4.1 Overview

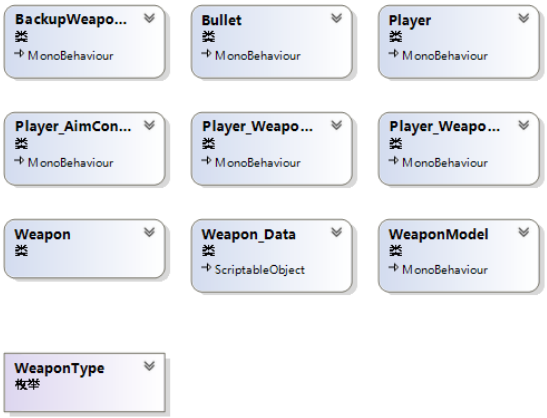
This section features an extensive and detailed weapon system. The classification of weapons is differentiated across six aspects: texture, damage, rate of fire, range, form, area of effect, reload speed, and switch speed. This categorization results in five distinct and stylistically diverse weapon types. In terms of weapon parameters, the system encapsulates various parameter options by integrating script parameters with state machine animations for both the character and the weapon itself. These parameters include weapon appearance, shooting modes, bullet count calculations, range control, and state machine management.

Regarding weapon interaction, behavior trees are employed to design two different gun-switching animation methods based on the player's keypress actions. The triggering conditions are embedded within the user interaction process, enabling functionalities such as switching, picking up, and discarding weapons, as well as changing weapon modes and camera shooting perspectives according to the user experience.

For bullet design, the weapon system particularly optimizes the bullet pool management to enhance game performance. By introducing object pool technology, it effectively stores and destroys fired bullets, preventing the game from slowing down due to an excess of bullets. Additionally, by calculating the runtime and range of bullets, the system extends the bullet disappearance effect when bullets miss the target. This optimization adds trailing effects to make the visual representation of disappearing bullets more natural.

4.2 Code structure

The class structure of weapon system:



1. Player

The Player class serves as the primary controller for the player character in the game. It manages various components and functionalities including player actions, weapon control,

interaction, health status, animations, and more. By referencing different sub-components such as Player_AimController, Player_Movement, Player_WeaponController, etc., the Player class coordinates and controls the behavior and state of the player character. Additionally, it handles enabling and disabling player controls, as well as event monitoring for UI interactions.

2. Player_WeaponController

The Player_WeaponController implements weapon control functionalities in a Unity game. It manages aspects such as equipping weapons, shooting, reloading, picking up, and dropping weapons. Through input-driven events, it facilitates complex interactions and management of the game's weapon systems.

3. Player_WeaponVisuals

Player_WeaponVisuals is responsible for managing visual effects and animations related to weapons for a player character in a Unity game. This includes playing animations for shooting, reloading, and equipping weapons, managing weapon model switching and activation of backup weapons, and using animation-guided techniques to optimize interactions and dynamic presentations between the character and weapons.

4. Player_AimController

Player_AimController oversees aiming control and associated visual effects for a player character. It involves updating aim and camera positions based on player input, managing toggling between precise and regular aiming modes, drawing aim lasers, and adjusting aiming offsets.

5. BackupWeaponModel

BackupWeaponModel manages the behavior of backup weapon models in a Unity game. It defines weapon types and mounting types, allowing activation or deactivation of corresponding weapon models as needed. It also provides methods to check for specific mounting types.

6. Weapon

Weapon describes the attributes and behaviors of weapons in the game. It includes various weapon-related data such as shoot type, ammo count, fire rate, bullet spread, etc. Moreover, it offers methods for shooting, reloading, and adjusting spread, facilitating weapon operations and management within the game.

7. Weapon_Data

Weapon_Data is a ScriptableObject used to store various data about weapons in a Unity game. It includes information like weapon name, bullet damage, magazine capacity, total reserve ammo, shoot type (single or automatic), fire rate, burst firing parameters (availability, bullet count, rate, delay), weapon spread, general properties (reload speed, equipment speed, gun distance, camera distance), and UI elements (weapon icon and description). This data is used to configure and define the behaviors and attributes of different weapons in the game for use and management.

8. WeaponModel

WeaponModel represents the weapon model in the game, managing various properties and related objects of the weapon. It includes information such as weapon type, equip animation type (side equip animation or back equip animation), hold type (common hold, low hold, high hold), gun point position (gunPoint), hold position (holdPoint), and associated audio effects related to the weapon (fire sound effects and reload sound effects). These definitions enable correct display, positioning, and playback of audio effects related to weapons in the game, enhancing the visual and auditory experience.

9. Bullet

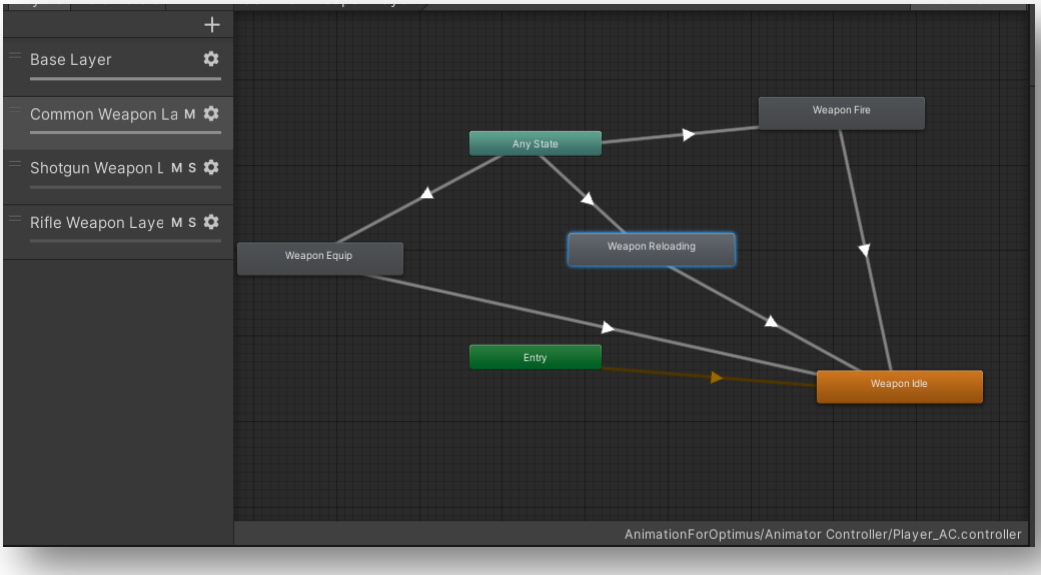
Bullet manages the behavior and effects of bullets in the game. It includes attributes such as bullet damage, flight distance, impact force, etc., and handles interactions between bullets and other objects in the game world through collision detection. Upon collision, the bullet may inflict damage on enemies, generate hit effects, or return itself to an object pool for reuse according to game settings. Additionally, it manages fading out of bullet trails and applying impact forces on enemies during collisions.

4.3 Details implementation in Unity

For weapon type， in Unity, the weapon system is implemented through a combination of Weapon, Weapon_Data, and other related classes, allowing for detailed customization across six key attributes: texture, damage, fire rate, range, mode, spread, reload speed, and equip speed. Each weapon, such as Pistols, Revolvers, AutoRifles, Shotguns, and Rifles, is configured using Weapon_Data to specify its unique parameters. The Weapon class then utilizes these parameters to define behaviors such as shooting (ShootType), burst firing, spread handling, and reloading, providing a robust framework for diverse and dynamic weapon interactions within the game.

In terms of weapon parameters, by passing in parameters from the Weapon_Data class, such as bulletDamage, fireRate, and burstFireRate, and integrating with the player character's state machine animations controlled in Player_WeaponVisuals, the system encapsulates various parameter options including weapon style, firing mode (single or burst), bullet count calculation, range control, and state machine management. Specifically, the Weapon class encapsulates methods for shooting, reloading, and spread handling, dynamically adjusting weapon behavior through parameterization. This, combined with the Animator component to control shooting and reloading animations, ensures that each weapon offers a unique visual and operational experience under different states.

In terms of weapon interaction, using a behavior tree, two distinct weapon-switching animations were designed based on different player input actions, facilitated by the Player_WeaponController and WeaponModel classes. These animations, controlled by the equipAnimationType in WeaponModel and managed through player input events captured in Player and PlayerControls, trigger under specific user interactions. The implementation includes methods for weapon switching, picking up, dropping weapons, and changing weapon modes or camera perspectives to enhance user experience. This dynamic interaction allows seamless transitions and realistic animations as defined in the Animator component, ensuring an immersive gameplay experience.



4.4 Optimization – Object pool

1. Overview:

Object pooling is a design pattern used to manage the reuse of objects in order to reduce the overhead of frequent allocation and deallocation. In the context of the Bullet class, object pooling optimizes performance by recycling bullet instances, which is particularly beneficial in a game where bullets are frequently created and destroyed.

```
public GameObject GetObject(GameObject prefab, Transform target)
{
    if (poolDictionary.ContainsKey(prefab) == false)
    {
        InitializeNewPool(prefab);
    }

    if (poolDictionary[prefab].Count == 0)
    {
        CreateNewObject(prefab);
    }

    GameObject objectToGet = poolDictionary[prefab].Dequeue();

    objectToGet.transform.position = target.position;
    objectToGet.transform.parent = null;

    // Traditional method: Instantiate(object)
    objectToGet.SetActive(true);

    return objectToGet;
}
```

```
private void ReturnToPool(GameObject objectToReturn)
{
    GameObject originalPrefab = objectToReturn.GetComponent<PooledObject>().originalPrefab;

    // Traditional method: Destroy(object)
    objectToReturn.SetActive(false);
    objectToReturn.transform.parent = transform;

    poolDictionary[originalPrefab].Enqueue(objectToReturn);
}
```

2. Optimization Strategy

(1) Initialization and Reuse:

a. Instead of instantiating new bullet objects every time a bullet is fired, the object pool maintains a collection of pre-instantiated bullet objects.

b. When a bullet is needed, it is fetched from the pool, initialized with necessary parameters, and used in the game.

c. Once the bullet has served its purpose (e.g., it hits a target or exceeds its range), it is returned to the pool for reuse.

(2) Reducing Garbage Collection:

a. Frequent instantiation and destruction of objects can lead to excessive garbage collection, which can cause performance spikes and frame drops.
b. By reusing objects, the object pool minimizes the creation of temporary objects, thereby reducing the load on the garbage collector.

(3) Efficient Memory Management:
a. Object pooling ensures that a fixed amount of memory is used for bullets, which can be advantageous for managing memory in a predictable manner.
b. The pool size can be adjusted based on the game's requirements to balance between memory usage and the number of available objects.

3. Technical Implementation

(1) Object Pool Class:
The ObjectPool class is responsible for managing the pool of bullet objects. It typically includes methods for fetching an object from the pool and returning it.

(2) Bullet Class Integration:
The Bullet class uses the object pool to manage its lifecycle. When a bullet is no longer needed, it is returned to the pool instead of being destroyed.

(3) Performance Optimization:
a. Efficient Retrieval: The object pool uses a queue data structure for efficient retrieval and return of bullet objects.
b. Trail Management: The bullet's trail renderer is managed to enhance visual effects without compromising performance. The FadeTrailIfNeeded method ensures the trail fades naturally, enhancing realism.

4. Effects of Optimization

(1) Performance Improvement:
a. Significantly reduces the number of allocations and deallocations, leading to smoother gameplay and lower CPU usage.
b. Minimizes garbage collection overhead, reducing frame drops and stuttering.

(2) Consistent Memory Usage:
Keeps memory usage stable by reusing objects, avoiding memory spikes that can occur with frequent instantiation and destruction.

(3) Enhanced Visuals:
The trail renderer management creates a visually appealing effect as bullets fade out, adding to the game's realism without additional performance costs.

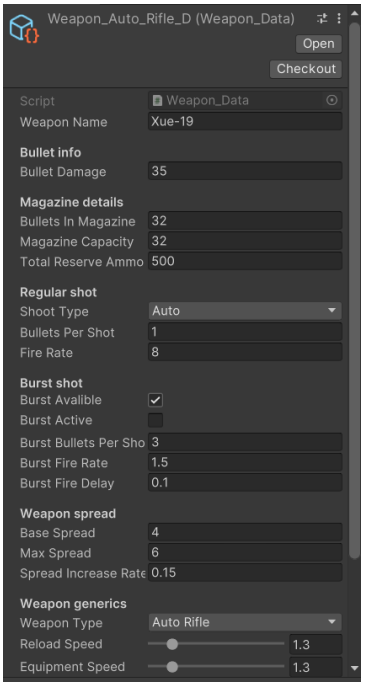
(4) Scalability:
The object pool can be easily scaled to handle varying game demands, ensuring that the system remains efficient even as the number of active bullets increases.

5. Interaction System

5.1 Overview

The interaction system is an integral and extended part of the weapon system. In a Unity game,

the weapon interaction system works through multiple scripts in unison to provide players with a rich and intuitive experience. Players can adjust initial weapon parameters in the Player and Assets -> Data sections, switch weapons using number keys 1-5, pick up weapons and ammo with the F key, reload with the R key, discard weapons with the G key, switch shooting perspectives with the P key, and toggle burst mode for the auto rifle with the T key. The camera can automatically adjust its distance and speed after a weapon change to ensure the view suits the new weapon, enhancing interactivity and immersion in the game.



5.2 Logical exposition

1. Initial Weapon Parameter Adjustment

Weapon_Data script allows developers to define and adjust weapon parameters such as damage, fire rate, reload speed, etc., through the Unity Inspector. The data is then used to instantiate and configure weapon instances at runtime.

2. Weapon Switching

Player_WeaponController script listens for number key inputs and changes the currently equipped weapon by enabling/disabling weapon models and updating relevant UI elements.

3. Picking Up Weapons and Ammo

Player_Interaction script detects collisions with weapons and ammo pickups. When the F key is pressed, it calls methods to add the item to the player’s inventory and update the UI.

4. Reloading

The Reload method in the Weapon class is called when the R key is pressed. It checks the ammo count and updates the weapon’s state.

5. Discarding Weapons

Player_WeaponController script includes logic to remove the current weapon from the player's inventory and drop it in the game world.

6. Switching Shooting Perspectives

Player_AimController script handles changing the camera view and updating the aim reticle based on the player's input.

7. Toggle Burst Mode
   Weapon class toggles the firing mode of applicable weapons, such as the auto rifle, when the T key is pressed.
8. Camera Adjustment After Weapon Change
   Player_WeaponVisuals script includes logic to adjust the camera's distance and speed after a weapon switch to maintain an optimal view.

6. damage system

6.1 Overview

In the development of our Unity-based TPS game, I was responsible for implementing the damage system. This system is critical for handling interactions between players, enemies, and projectiles, ensuring that each encounter is processed accurately and efficiently. The implementation leverages several key Unity technologies and design principles, including component-based architecture, event-driven programming, and object pooling.

6.2 Core Components

6.2.1 Health System

The health system is the backbone of the damage system, tracking the health of both players and enemies.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Enemy_Health : HealthController
6  {
7
8  }
9
```

Code Example: Enemy_Health.cs

The Enemy_Health class inherits from a base HealthController, which centralizes health management logic. This approach promotes code reuse and ensures consistent behavior across different types of entities.

6.2.2 Hitbox Detection

Hitboxes are used to detect collisions and apply damage to the appropriate entities.

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

public class Enemy_HitBox : HitBox
{
    private Enemy enemy;

    protected override void Awake()
    {
        base.Awake();

        enemy = GetComponentInParent<Enemy>();
    }

    public override void TakeDamage(int damage)
    {
        int newDamage = Mathf.RoundToInt(damage * damageMultiplier);

        enemy.GetHit(newDamage);
    }
}
```

Code Example: Enemy_HitBox.cs

The Enemy_HitBox class extends a base HitBox class, linking it to its parent Enemy component. This structure ensures that when a hit is detected, the corresponding enemy instance is notified and can process the damage.

6.2.3 Bullet Handling

Bullets are a primary method of dealing damage in the game. Their behavior is managed through a dedicated class that handles movement, collisions, and damage application.

```
using UnityEngine;

public class Bullet : MonoBehaviour
{
    private int bulletDamage;
    private float impactForce;

    private BoxCollider cd;
    private Rigidbody rb;
    private MeshRenderer meshRenderer;
    private TrailRenderer trailRenderer;


    [SerializeField] private GameObject bulletImpactFX;


    private Vector3 startPosition;
    private float flyDistance;
    private bool bulletDisabled;

    private LayerMask allyLayerMask;

    protected virtual void Awake()
    {
        cd = GetComponent<BoxCollider>();
        rb = GetComponent<Rigidbody>();
        meshRenderer = GetComponent<MeshRenderer>();
        trailRenderer = GetComponent<TrailRenderer>();
    }

    public void BulletSetup(LayerMask allyLayerMask, int bulletDamage, float flyDistance = 100, float impactForce = 100)
    {
        this.allyLayerMask = allyLayerMask;
        this.impactForce = impactForce;
        this.bulletDamage = bulletDamage;

        bulletDisabled = false;
        cd.enabled = true;
        meshRenderer.enabled = true;


        trailRenderer.Clear();
        trailRenderer.time = .25f;
        startPosition = transform.position;
        this.flyDistance = flyDistance + .5f; // magic number .5f is a Length of tip of the laser ( Check method UpdateAimV
    }

    protected virtual void Update()
```

Code Example: Bullet.cs

The Bullet class manages the bullet's lifecycle, including setup, flight, and collision detection. It also

handles interactions with the environment and characters, applying damage as necessary.

6.2.4 Animation Events

Animation events are used to synchronize actions with animations, ensuring smooth and realistic interactions.

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

public class Enemy_AnimationEvents : MonoBehaviour
{
    private Enemy enemy;
    private Enemy_Melee enemyMelee;
    private Enemy_Boss enemyBoss;

    private void Awake()
    {
        enemy = GetComponentInParent<Enemy>();
        enemyMelee = GetComponentInParent<Enemy_Melee>();
        enemyBoss = GetComponentInParent<Enemy_Boss>();
    }

    public void AnimationTrigger() => enemy.AnimationTrigger();

    public void StartManualMovement() => enemy.ActivateManualMovement(true);
    public void StopManualMovement() => enemy.ActivateManualMovement(false);

    public void StartManualRotation() => enemy.ActivateManualRotation(true);
    public void StopManualRotation() => enemy.ActivateManualRotation(false);

    public void AbilityEvent() => enemy.AbilityTrigger();
    public void EnableIK() => enemy.visuals.EnableIK(true, true, 1f);

    public void EnableWeaponModel()
    {
        enemy.visuals.EnableWeaponModel(true);
        enemy.visuals.EnableSecondaryWeaponModel(false);
    }

    public void BossJumpImpact()
    {
        enemyBoss?.JumpImpact();
    }

    public void BeginMeleeAttackCheck()
    {
        enemy?.EnableMeleeAttackCheck(true);

        enemy?.audioManager.PlaySFX(enemyMelee?.meleeSFX.swoosh, true);
    }

    public void FinishMeleeAttackCheck()
    {
```

Code Example: Enemy_AnimationEvents.cs

The Enemy_AnimationEvents class links animation events to specific methods in the enemy classes, allowing for precise control over actions like attacks and abilities.

6.3 Key Technologies and Techniques

6.3.1 Component-Based Architecture

Unity's component-based architecture allows for modular and reusable code. By breaking down functionalities into distinct components (e.g., HealthController, HitBox), the system remains flexible and maintainable.

6.3.2 Inheritance and Polymorphism

Utilizing inheritance and polymorphism, we create a hierarchy of classes that share common functionality while allowing for specific behaviors. For example, Enemy_Health inherits from HealthController, and Enemy_HitBox from HitBox.

6.3.3 Event-Driven Programming

Animation events and method calls triggered by specific actions (e.g., taking damage, triggering abilities) enable responsive and interactive gameplay. This approach ensures that animations and game logic are tightly synchronized.

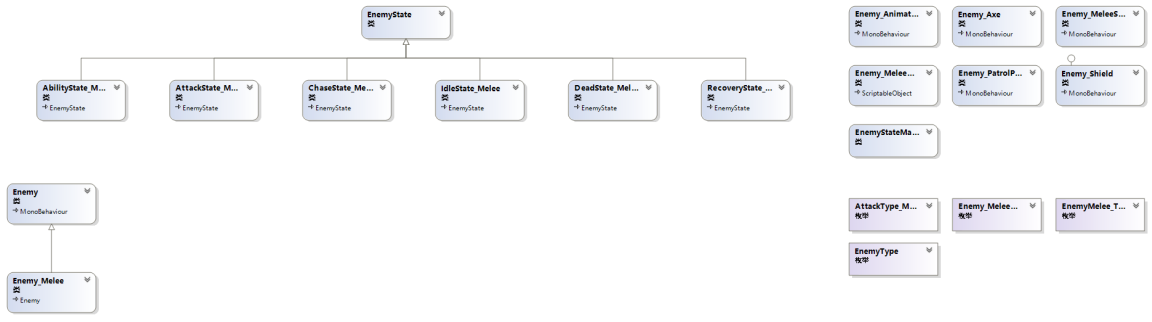
6.3.4 Object Pooling

Object pooling, as seen in the Bullet class, optimizes performance by reusing objects rather than creating and destroying them frequently. This technique is particularly effective for projectiles and visual effects.

6.4 Conclusion

The damage system in our Unity game is a robust and efficient implementation that leverages several advanced programming techniques and Unity's core features. By employing a component-based architecture, inheritance, event-driven programming, and object pooling, we have created a scalable and maintainable system that enhances the gameplay experience.

7. Enemy Melee and Enemy variant



7.1 Overview

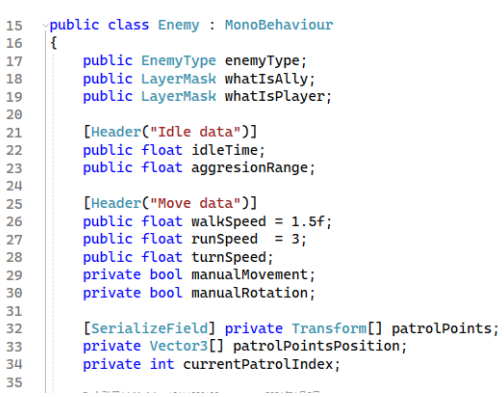
The Enemy_Melee section designs a sophisticated enemy AI system encompassing navigation, target acquisition, state management, and attack sets. This system utilizes map baking and code to achieve smooth pathfinding and rotation, rendering enemy movements more realistic. Enemies possess an automatic patrol function and transition to pursuit mode when the player enters their range or is attacked. A state machine class manages various enemy states such as movement, recovery, idle, skills, death, pursuit, and attack. The attack set is implemented via blend tree,

containing six melee attack animations, randomly selected each time. Death states employ the ragdoll component to produce randomized animations correlated with the point of attack. Enemies exhibit four unique attack methods and movements, including regular attacks, shielding, axe throwing, and dodging. Random generation of enemies involves variations in appearance, types, and weapons. Animator override technology is used to match different attack animations for different enemy types, avoiding redundant AnimationController creation. Finally, EnemyData allows for parameter adjustments for each enemy's attacks, such as attack speed, rotation speed, and displacement/rotation during attacks.

7.2 Unity Implementation Process

1. Enemy Model and Animation Import
    - Import designed enemy 3D models into Unity.
    - Import enemy animations including walking, running, attacking, etc.
2. Set up Animator Controller
    - Create an Animator Controller for the enemy.
    - Configure animation states like Idle, Move, Attack, Dead, and set transition conditions.
    - Use Blend Tree for diverse attack animations.
3. Programming AI Behavior
    - Create an enemy script with a C# state machine to control behavior.
    - Implement navigation and movement using NavMeshAgent.
    - Implement target acquisition and pursuit mechanisms.
    - Define state machine enumerations and state transition logic.
4. Attack and Animation Synchronization
    - Trigger attack animations in the enemy script when in attack state.
    - Add animation events to attack animations for logic execution like dealing damage, playing sounds.
5. Implement Diversity and Randomness
    - Create an enemy generator script for dynamic enemy spawning.
    - Implement a function to randomly select attack patterns.
6. Parameter Adjustment and Optimization
    - Create an EnemyData class to store enemy parameters.
    - Optimize performance by minimizing unnecessary calculations and optimizing pathfinding algorithms.
7. Testing and Debugging
    - Write unit tests for enemy AI components.
    - Test enemy behavior in-game and adjust as needed.

7.3 Code Implementation



```
15  public class Enemy : MonoBehaviour
16  {
17      public EnemyType enemyType;
18      public LayerMask whatIsAlly;
19      public LayerMask whatIsPlayer;
20
21      [Header("Idle data")]
22      public float idleTime;
23      public float aggresionRange;
24
25      [Header("Move data")]
26      public float walkSpeed = 1.5f;
27      public float runSpeed  = 3;
28      public float turnSpeed;
29      private bool manualMovement;
30      private bool manualRotation;
31
32      [SerializeField] private Transform[] patrolPoints;
33      private Vector3[] patrolPointsPosition;
34      private int currentPatrolIndex;
35
```

```
2 个引用 | Added by lishaoxu1234@gmail.com on 2024年5月10日
public AudioManager audioManager { get; private set; }
0 Unity 消息 | 0 个引用 | Changed by lishaoxu1234@gmail.com on 2024年5月7日
protected virtual void Awake()...

0 Unity 消息 | 0 个引用 | Changed by lishaoxu1234@gmail.com on 2024年5月10日
protected virtual void Start()...

0 Unity 消息 | 0 个引用 | Changed by 217220400luchjtu.ac.uk on 2024年4月9日
protected virtual void Update()...

4 个引用 | Added by 217220400luchjtu.ac.uk on 2024年4月13日
protected virtual void InitializePerk()...

1 个引用 | Added by 3527799240@qq.com on 2024年4月27日
public virtual void MakeEnemyVIP()...

2 个引用 | Changed by 217220400luchjtu.ac.uk on 2024年4月13日
protected bool ShouldEnterBattleMode()...

10 个引用 | Added by 18411800198@qq.com on 2024年4月6日
public virtual void EnterBattleMode()...

1 个引用 | Changed by lishaoxu1234@gmail.com on 2024年5月7日
public virtual void GetHit(int damage)...

7 个引用 | Changed by lishaoxu1234@gmail.com on 2024年5月7日
public virtual void Die()...
```

7.3.1 Enemy Class

Enemy Class Overview

It represents an enemy character in a Unity game, encapsulating various properties and behaviors such as movement, attacking, taking damage, and dying.

Enumeration and Fields

Firstly, the class defines an EnemyType enumeration to distinguish between different enemy types (e.g., melee, ranged, Boss, random). It then declares a series of public and private fields to store the enemy's state and configuration, including:

- enemyType: Stores the type of the enemy.
- idleTime: Represents the enemy's idle time.
- aggresionRange: Specifies the enemy's attack range.

Lifecycle Methods

- **Awake**: A Unity lifecycle method called when the object is instantiated. In this method, references to components like NavMeshAgent and Animator are obtained, and some fields are initialized.
- **Start**: Another Unity lifecycle method called before the first frame update. Here, patrol points and audio managers are initialized.
- **Update**: Invoked every frame to update the enemy's state. If combat mode conditions are met, the EnterBattleMode method is called.

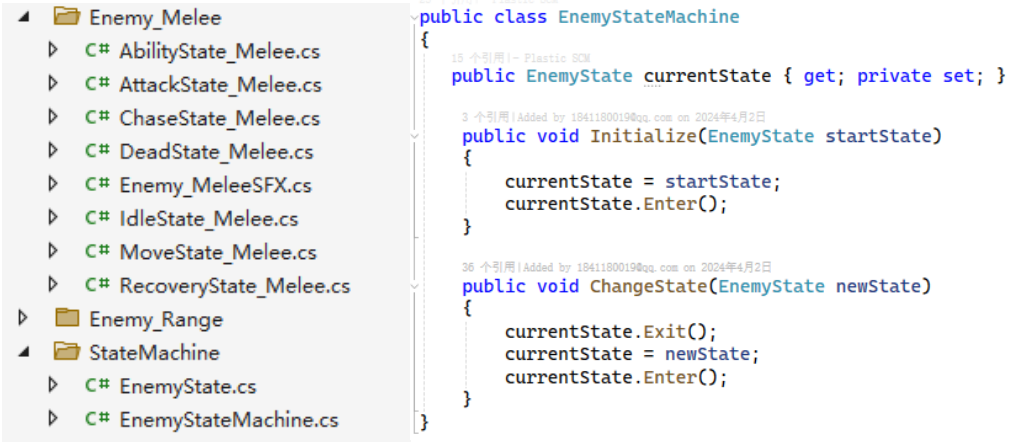
Behavioral Methods

- **EnterBattleMode**: Sets the enemy's state to combat mode.
- **GetHit**: Handles the enemy being attacked. It enters combat mode, reduces health, and calls the Die method if necessary.
- **Die**: Manages the enemy's death by dropping items, disabling animation and navigation, and activating the ragdoll system.
- **MeleeAttackCheck**: Checks for melee attacks and deals damage to nearby players if ready.
- **BulletImpact**: Handles the enemy being hit by a bullet, calling the DeathImpactCoroutine if fatal.
- **FaceTarget**: Orients the enemy towards a target.
- **GetPatrolDestination** & **InitializePatrolPoints**: Handle the enemy's patrol logic.
- **IsPlayerInAggressionRange**: Checks if the player is within the enemy's attack range.

Utility Method

- **OnDrawGizmos**: Used in the Unity editor to draw the enemy's attack range.

7.3.2 State Machine



State Machine Pattern for Enemies

EnemyState Base Class

The EnemyState serves as the base class for all enemy states. It defines the following basic methods, which may be overridden in specific state classes to implement specific behaviors:

- **Enter**: Called when entering a state and used to set the enemy's animation state.
- **Update**: Invoked every frame to update state timers.
- **Exit**: Called when exiting a state and resets the enemy's animation state.
- **AnimationTrigger**: May be called when specific animations are triggered.
- **AbilityTrigger**: May be called when specific abilities are triggered.
- **GetNextPathPoint**: Retrieves the enemy's next path point.

EnemyStateMachine

The EnemyStateMachine represents the main body of the state machine, managing the enemy's current state and providing methods to change it:

- **Initialize**: Initializes the state machine, sets the initial state, and calls its Enter method.
- **ChangeState**: Changes the current state by first calling the current state's Exit method, then updating the current state and calling the new state's Enter method.

7.4 Optimization Section

1. Simulating Death Using Ragdoll System

• Reason: Traditional animation methods for character death often appear mechanical and repetitive, failing to realistically reflect the dynamic response of a character after being attacked.

• Implementation: When the enemy's health drops to 0, activate the Ragdoll system. This system simulates the realistic falling effect of a character after death through the physics engine, making each death animation appear unique and realistic.

2. Animator Override Controller

• Reason: When a game has multiple similar enemy characters but their animations differ slightly, creating a complete animation controller for each enemy individually can be redundant and inefficient.

• Implementation: Through the Animator Override Controller, animation controllers can be quickly created for new enemies by only replacing the parts that differ from the base animation controller.

3. Manual Rotation for Enemy Movement

• Reason: Simple automatic pathfinding algorithms may not produce completely natural and smooth turning animations.

• Implementation: Add manual rotation logic to the enemy's movement script, dynamically adjusting its orientation based on the enemy's movement direction and speed.

4. Performance Optimization

• Reason: Complex AI logic and a large number of enemy instances can impact game performance.

• Implementation: Improve the efficiency of the AI system by reducing unnecessary calculations (such as enemy detection calculations only within the enemy's field of vision).

8. Enemy Range and Enemy Boss

8.1 Enemy Range Overview

The enemy range is partly same as the enemy melee, which are both derived mostly from two scripts—enemy and enemy_visuals. Enemy range is designed in order to diversify the enemy context. According to its name, enemy range could shoot the player from a distance with the weapon same as the player. In additional to the same characteristics as the enemy melee such as random appearance, random weapon type and some normal state like idle state, battle state, it has two important new characteristics—cover finding and grenade throwing. Moreover, there are also some works to configuration its shooting logic and intelligence.

8.2 Process

1. Model and animation setup

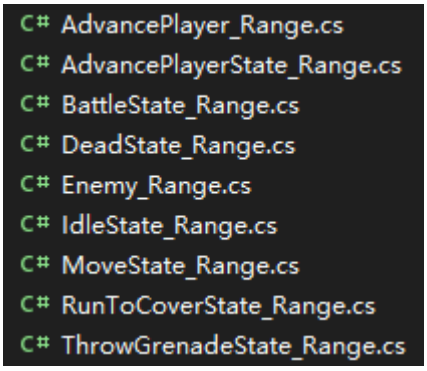
   The model is copied from enemy melee directly, but it has to hold different weapon from enemy melee, so we just hide all the weapon under the model and use its own script—Enemy_Range to control the choose of weapon model and use two script to control the visual of the chosen weapon, which are named Enemy_RangeWeaponData and Enemy_RangeWeaponModel. The weapon type and weapon hold manner are same as the Player and we use two Enum variant to hold these data.

   The animations of enemy range are imported as resources on a website including idle with a rifle, move with a rifle, shoot, etc.

   The animator controller setup is the same as enemy melee.

2. Weapon synchronization through IK configuration, which is the same as Player.
3. AI configuration of the range enemy

Create its own C# script and C# scripts for each state of range enemy, such as Idle state, move state and so on, which are showed in the follow picture.



Pic 1.1 The states of Enemy Range

4. Weapon Data configuration

   Each weapon has its own data, for example shotgun, it will have different bullet speed and bullets for each shoot. These data are stored through the Weapon Data under the Weapon System.

Pic 1.2 Weapon Data

5. Design of special characteristics of range enemy

   There are several characteristics of range enemy—the ability to find and change cover, the state of unstoppable, and the ability to throw grenade.

6. the smart aim design of range enemy

   In order to ensure the range enemy could aim correctly at the player, we set an empty object to follow the spine of the player and the range enemy would aim at this empty object with certain speed (not directly face the player, which may be weird), under the situation that the enemy see the player. If the player disappears from its sight, the enemy will aim at the last position it saw the player and slowly back to idle state.

```
public bool IsSeeingPlayer()
{
    Vector3 myPosition = transform.position + Vector3.up;
    Vector3 directionToPlayer = playersBody.position - myPosition;

    if (Physics.Raycast(myPosition, directionToPlayer, out RaycastHit hit, Mathf.Infinity, ~whatToIgnore))
    {
        if (hit.transform.root == player.root)
        {
            UpdateAimPosition();
            return true;
        }
    }

    return false;
}
```

Pic 1.3 Seeing Player

7. Design dead state, which is the same as enemy melee using ragdoll.
8. Testing and code improvement.

8.3 Brief description of Enemy Range

```
public class Enemy_Range : Enemy
{
    [Header("Enemy perks")]
    public Enemy_RangeWeaponType weaponType;
    public CoverPerk coverPerk;
    public UnstoppablePerk unstoppablePerk;
    public GrenadePerk grenadePerk;

    [Header("Grenade perk")]
    public int grenadeDamage;
    public GameObject grenadePrefab;
    public float impactPower;
    public float explosionTimer = .75f;
    public float timeToTarget = 1.2f;
    public float grenadeCooldown;
    private float lastTimeGrenadeThrown = -10;
    [SerializeField] private Transform grenadeStartPoint;

    [Header("Advance perk")]
    public float advanceSpeed;
    public float advanceStoppingDistance;
    public float advanceDuration = 2.5f;
```

Pic 1.4 Characteristics of Enemy Range 1

The range enemy could have five types of weapons—pistol, revolver, auto rifle, rifle, and shotgun.

The **CoverPerk** defines whether this range enemy could find cover or could find and change cover.

The **UnstoppablePerk** defines whether this range enemy is unstoppable. If it is unstoppable, it would advance the player through walking instead of normal running, indicating that it is an elite and doesn't care the player.

The **GrenadePerk** defines whether this range enemy could throw grenade. If so, it would

throw grenade toward the player, with the configuration parameters under the Header “Grenade perk”, such as the damage of grenade, the cooldown of each throwing and so on.

The **Advance perk** manages the speed of the enemy chasing the player, when to stop chasing and cooldown of each chasing time.

The **Cover system** manages the minimum time the range enemy stay at one cover point, the safe distance from it to the player, which requires it to change cover if player is too close. In order to implement the change of cover, we use two **CoverPoint** variant to hold the current cover point and last cover point. Which will be detailed in the following context.

Pic 1.5 Characteristics of Enemy Range 2

The **Weapon details** define the weapon data of the weapon held by this enemy. We also use a Transform variant to note down the position where the bullets are created and fired.

The **Aim details** define the aim speed and aim target of the enemy which is mentioned before.

8.4 Special Characteristics of Enemy Range

8.4.1 Cover

The range enemy, if configurated to be able to find cover, will automatically find cover point around it. In order to find cover, first there should be covers and cover points in the scene. This has been realized with the scripts—**Cover** and **CoverPoint**.

```
public class CoverPoint : MonoBehaviour
{
    public bool occupied;

    2 个引用
    public void SetOccupied(bool occupied) => this.occupied = occupied;
}
```

Pic 1.5 Script of CoverPoint

The **CoverPoint** is a class describing the points available for covering the enemy around a cover with a bool attribute **occupied** to indicate whether this cover point is occupied by other range enemy. If so, another range enemies couldn't take this cover point.

```
public class Cover : MonoBehaviour
{
    private Transform playerTransform;

    [Header("Cover points")]
    [SerializeField] private GameObject coverPointPrefeb;
    [SerializeField] private List<CoverPoint> coverPoints = new List<CoverPoint>();
    [SerializeField] private float xOffset = 1;
    [SerializeField] private float yOffset = .2f;
    [SerializeField] private float zOffset = 1;

    Unity 消息 | 0 个引用
    private void Start()
    {
        GernerateCoverPoints();
        playerTransform = FindObjectOfType<Player>().transform;
    }
```

Pic 1.5 Script of Cover 1

In the **Cover** script, if an object contains this Cover script, it will automatically generate four cover points around it as mentioned.

Moreover, in additional to just find a cover point, we prefer the enemy to find a valid cover point, which is not occupied by the other enemy, is furthest away from player, is not too close to player, is not behind the player and not too close to the last cover point. These are realized mostly through distance check from this cover point and the player.

```
private bool isValidCoverPoint(CoverPoint coverPoint, Transform enemy)
{
    if (coverPoint.occupied) return false;

    if(IsFurtherestFromPlayer(coverPoint) == false)
        return false;

    if (IsCoverCloseToPlayer(coverPoint))
        return false;

    if (isCoverBehindPlayer(coverPoint, enemy))
        return false;

    if (IsCoverCloseToLastCover(coverPoint, enemy))
        return false;


    return true;
}
```

Pic 1.6 Script of Cover 2

Above are all just preparation work for creating cover points. There are necessary works to do in the script of range enemy. First, a range enemy will collect nearby cover points and hold them under a list.

```
private List<Cover> CollectNearByCovers()
{
    float coverRadiusCheck = 30;
    Collider[] hitColliders = Physics.OverlapSphere(transform.position, coverRadiusCheck);
    List<Cover> collectedCovers = new List<Cover>();

    foreach (Collider collider in hitColliders)
    {
        Cover cover = collider.GetComponent<Cover>();

        if(cover !=null && collectedCovers.Contains(cover) == false)
            collectedCovers.Add(cover);

    }
    return collectedCovers;
}
```

Pic 1.7 Codes of Cover in Enemy_Range script 1

Then the range enemy will attempt to find a cover and check its validity through the methods mentioned above in the **Cover** script. After that, if this range enemy is configurated to be able to change cover if player is approaching, it will try to find a new and nearest cover point. If no such cover point is found, it will stay at the same cover point.

```
private Transform AttemptToFindCover() {

    List<CoverPoint> collectedCoverPoints = new List<CoverPoint>();

    foreach (Cover cover in CollectNearByCovers())
    {
        collectedCoverPoints.AddRange(cover.GetValidCoverPoints(transform));
    }

    CoverPoint closestCoverPoint = null;
    float shortestDistance = float.MaxValue;

    foreach (CoverPoint coverPoint in collectedCoverPoints)
    {
        float currentDistance = Vector3.Distance(transform.position, coverPoint.transform.position);
        if (currentDistance < shortestDistance)
        {
            closestCoverPoint = coverPoint;
            shortestDistance = currentDistance;

        }


    }

    if (closestCoverPoint != null)
    {
        lastCover?.SetOccupied(false);
        lastCover = currentCover;

        currentCover = closestCoverPoint;
        currentCover.SetOccupied(true);

        return currentCover.transform;
    }

    return null;

}
```

Pic 1.8 Codes of Cover in Enemy_Range script 2

Moreover, if the range enemy is failed to get a cover, it will stop finding a cover and enter the battle state.

```
    if (CanGetCover())
        stateMachine.ChangeState(runToCoverState);
    else
        stateMachine.ChangeState(battleState);
}

#region Cover System

2 个引用
public bool CanGetCover()
{
    if (coverPerk == CoverPerk.Unavailable)
        return false;

    currentCover = AttemptToFindCover()?.GetComponent<CoverPoint>();

    if (lastCover != currentCover && currentCover != null)
        return true;

    Debug.LogWarning("No cover found!");
    return false;
}
```

Pic 1.9 Codes of Cover in Enemy_Range script 3

8.4.2 Throw Grenade

The state of throwing grenade is not just adding a new animation to the state machine, because the enemy would have its left hand holding the gun naturally and its right hand throwing the grenade. Additional works need to do to adjust the weapon.

Therefore, we hide the other set of weapon models under the left hand of the enemy prefab with another script—**Enemy_SecondaryRangeWeaponModel**. these weapons will be activated when the enemy is throwing grenade and be hidden again when the motion is finished. Besides, the original IK constraints for controlling the weapon holding motion are disabled, because the right hand of the enemy is doing the throwing animation.

Also, the same as the gunpoint where the bullets are generated, there should be a point to generate grenade, so we set a position under the right hand of the enemy.

Moreover, the characteristics of this state could be adjusted through parameters mentioned above to control the damage of grenade and cooldown of each throwing.

8.5 Boss Overview

Bosses are of great importance in the difficulty configuration of a game. We complete two types of bosses with new visual scripts because the visual of boss have little to do with the normal enemy. Apart from the normal state of enemy, enemy boss can do two skills—jump attack and special ability. These skills require special handle because they are not normal animation implementations.

8.6 Process

1. Model and animation setup

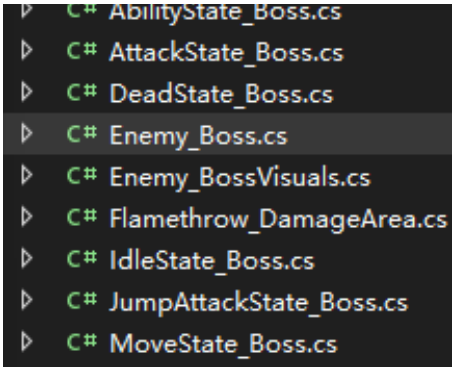
   The model of enemy boss should be different from normal enemy but with the same style.

The animations of enemy boss are imported as resources on a website as well.

The animator controller setup is the same as normal enemy.

2. Weapons setup, which include the flame thrower and hammer, and the weapon trail, which is an effect on the weapon to make it cool.
3. AI configuration of the range boss

   Create its own C# script and C# scripts for each state of enemy boss, such as Idle state, move state and so on, which are showed in the follow picture.



Pic 2.1 The states of Enemy Range

4. Skills logic configuration.

   For each boss, it has its own special skill and a same jump attack skill. What's more, the boss will speed up after some period to attack the player madly.
5. Design dead state, which is the same as enemy melee using ragdoll.
6. Testing and code improvement.

8.7 Brief description of Enemy Boss

```
public class Enemy_Boss : Enemy
{
    [Header("Boss details")]
    public BossWeaponType bossWeaponType;
    public float actionCooldown = 10;
    public float attackRange;

    [Header("Ability")]
    public float minAbilityDistance;
    public float abilityCooldown;
    private float lastTimeUsedAbility;

    [Header("Flamethrower")]
    public int flameDamage;
    public float flameDamageCooldown;
    public ParticleSystem flamethrower;
    public float flamethrowDuration;
    3 个引用
    public bool flamethrowActive { get; private set; }

    [Header("Hummer")]
    public int hummerActiveDamage;
    public GameObject activationPrefab;
    [SerializeField] private float hummerCheckRadius;
```

Pic 2.2 Characteristics of Enemy Boss 1

The range enemy could have two types—Hammer and Flamethrower.

The **Boss details** define the attack range of each boss and the cooldown of their action of each type of attack.

The **Ability** defines the shortest distance that the boss can do ability and the cooldown of doing ability.

The **Flamethrower** defines the damage of the flame ability of the boss with a flame thrower and the duration of the casting of flame.

The **Hammer** manages the damage of the lightning strike ability of the boss with a hammer, the effect prefab of lightning or other types of effect, and the damage radius of hammer.

The **Jump attack** defines the damage of jump attack, the cooldown that the boss can do next jump attack., the time to jump to the player and the shortest distance that the boss can do jump attack. Moreover, because jump attack is an “AOE (Area of effect)” attack from the sky to the ground, it should contain an impact radius and impact power that could blow up objects in the landing area. Besides, in order to remind the player that the boss will do jump attack, we choose to place a landing range effect at the position the boss will jump to.

```
[Header("Jump attack")]
public int jumpAttackDamage;
public float jumpAttackCooldown = 10;
private float lastTimeJumped;
public float travelTimeToTarget;
public float minJumpDistanceRequired;
[Space]
public float impactRadius = 2.5f;
public float impactPower = 5;
public Transform impactPoint;
[SerializeField] private float upforceMultiplier = 10;
[Space]
[SerializeField] private LayerMask whatToIgnore;

[Header("Attack")]
[SerializeField] private int meleeAttackDamage;
[SerializeField] private Transform[] damagePoints;
[SerializeField] private float attackCheckRadius;
[SerializeField] private GameObject meleeAttackFx;
```

Pic 2.3 Characteristics of Enemy Boss 2

The **Attack** defines the point on the weapon where would cast damage to player if hit the player.

8.8 Skills implementation of Enemy Boss

8.8.1 Speed up

The boss will periodically get into mad and speed up to attack the player to indicate that it is mad. If it should speed up, it will change it speed to run speed.

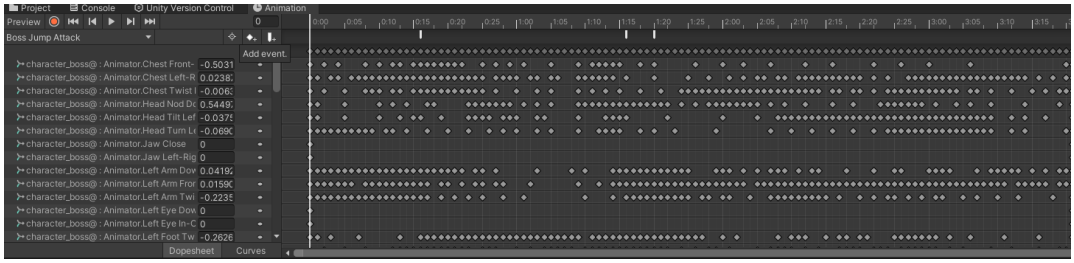
```
private bool ShouldSpeedUp()
{
    if (speedUpActivated)
        return false;


    if (Time.time > enemy.attackState.lastTimeAttacked + timeBeforeSpeedUp)
    {
        return true;
    }
    return false;
}
```

```
private void SpeedUp()
{
public override void Update()
{
    base.Update();

    actionTimer -= Time.deltaTime;
    enemy.FaceTarget(GetNextPathPoint());

    if (enemy.inBattleMode)
    {
        if (ShouldSpeedUp())
            SpeedUp();


        Vector3 playerPos = enemy.player.position;

        enemy.agent.SetDestination(playerPos);

        if (actionTimer < 0)
        {
            PerformRandomAction();
        }
        else if (enemy.PlayerInAttackRange())
            stateMachine.ChangeState(enemy.attackState);
    }
    else
    {
        if (Vector3.Distance(enemy.transform.position, destination) < .25f)
            stateMachine.ChangeState(enemy.idleState);

    }
}
```

Pic 2.4 Codes of Speed Up

8.8.2 Jump Attack

If the player is too far away from boss, the boss will do jump attack to close to the player. The jump attack is different from normal attack animation that is attacking player in the same position. There is a position changing during the jump attack. Therefore, we apply a new way to handle these movement via animation event.



Pic 2.5 Animation Event

Above is the animation of jump attack, at the first several frame, it will normally do the jump preparation motion and moving by its own navigation mechanism. When the animation event (shown above as a white strip) is activated, at the same time the boss jump to the air, we will control it movement manually. The velocity of the boss will set to be zero and then we place a landing effect at the position the boss will land. Then a new velocity is given to the boss through the calculation of dividing distance to player with the time needed to jump to the player. When the boss jump to the position, at the same time the jump animation comes to the last frame, an animation event will be activated to recover the boss to normal navigation.

```
public override void Enter()
{
    base.Enter();
    lastPlayerPos = enemy.player.position;
    enemy.agent.isStopped = true;
    enemy.agent.velocity = Vector3.zero;


    enemy.bossVisuals.PlaceLandingZone(lastPlayerPos);
    enemy.bossVisuals.EnableWeaponTrail(true);

    float distanceToPlayer = Vector3.Distance(lastPlayerPos, enemy.transform.position);

    jumpAttackMovementSpeed = distanceToPlayer / enemy.travelTimeToTarget;

    enemy.FaceTarget(lastPlayerPos, 1000);


    if (enemy.bossWeaponType == BossWeaponType.Hummer)
    {
        enemy.agent.isStopped = false;
        enemy.agent.speed = enemy.runSpeed;
        enemy.agent.SetDestination(lastPlayerPos);
    }
}
```

Pic 2.5 Animation Event

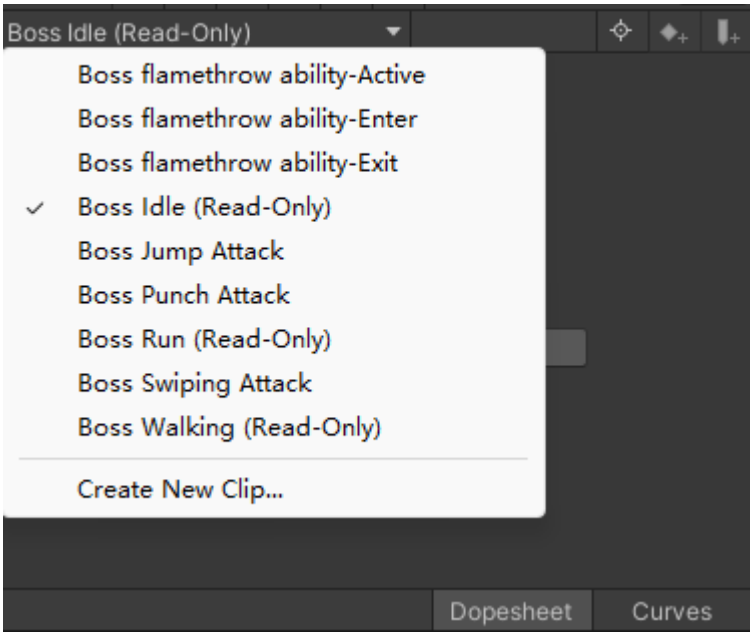
8.8.3 Do ability

Each boss has its own ability, the flame thrower boss will throw flame to the player and the hammer boss will heavily strike the player with lightning. The ability animation needs to be specially handled as well but in the same way as jump attack via animation events.

For the hammer lightning strike, the first several frame will be the normal strike motion, but when an animation event is activated, at the same time the boss will strike the player in front of

him, the lightning effect will be shown.

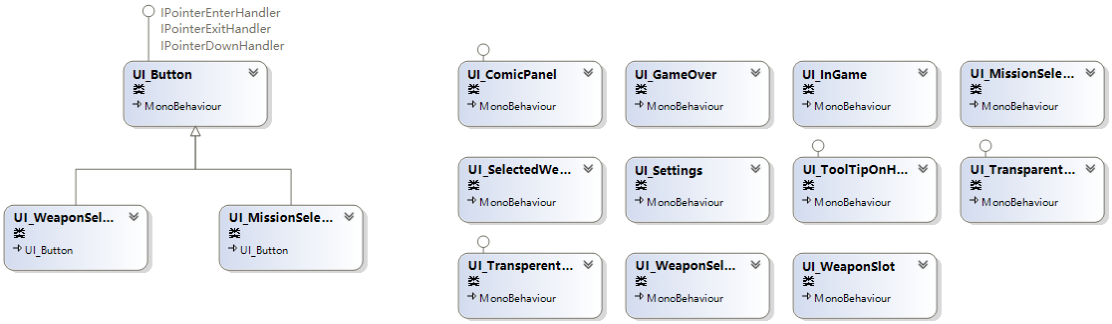
For the flame thrower ability, we cut it into three stages—Enter, Active, Exit, because we want to control the flame throwing duration. The basic principle is the same. When the boss enters the ability state, it will first play the animation “Boss flamethrow ability-Enter”, at the last frame of the Enter animation, there is an animation event to show the effect of flame and play the animation “Boss flamethrow ability-Active”, which is a loop animation and will exit only when the flame duration time is over. At the last frame of Active animation, the flame effect will be closed and the animation “Boss flamethrow ability-Exit” will be played and the boss will exits from the ability state.



Pic 2.6 Animation of Flame Thrower Boss

Besides, there is a small detail of flame thrower. In the weapon, there is a small cylinder, which is applied as a battery, indicating the duration of flame throwing. When the boss is throwing flame, the cylinder will decrease with the speed of its height dividing the duration time. After the boss threw the flame, the cylinder will charge back to the normal size with the speed of its height dividing the cooldown of doing ability.

9.UI

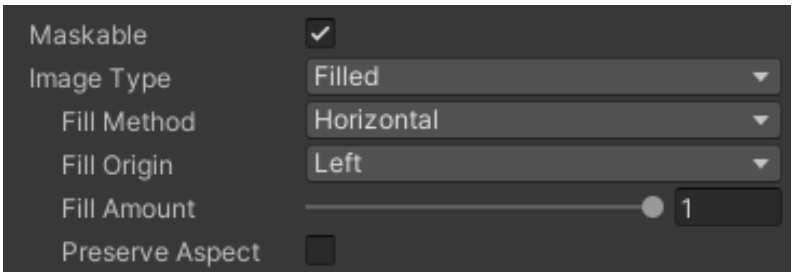


9.1 Overview

This section covers various aspects of user interface (UI) design, from basic elements to specific functional implementations. It includes acquiring UI elements like health bars, weapon interfaces, mission goal UIs; transparent UI elements on mouse hover; main menu layout and button design; mission and weapon selection interfaces; play, pause buttons, and time management; game over, victory screens, and settings interfaces. It also incorporates screen fading effects and comic panel UIs. Finally, the UI system undergoes cleanup and optimization.

9.2 Implementation

1. **Getting UI Elements**
   • Implementation: Utilize Unity's UGUI system to create various UI elements such as Text, Image, Button, etc., under the Canvas.
2. **Health Bar**
   • Implementation: Create an Image as the background for the health bar and use another Image (fill Image) to represent the current health. Adjust the width of the fill Image to reflect the character's health.




```
public void UpdateHealthUI(float currentHealth, float maxHealth)
{
    healthBar.fillAmount = currentHealth / maxHealth;
}
```

3. **Weapon UI**
   • Implementation: Design an interface to display the current equipped weapon and ammunition count, which may include weapon icons and ammo counters.

```
public void Update(List<Weapon> weaponSlots,Weapon currentWeapon)
{
    for (int i = 0; i < weaponSlots_UI.Length; i++)
    {
        if (i < weaponSlots.Count) //只有一把武器
        {
            bool isActiveWeapon = weaponSlots[i] == currentWeapon ? true : false;
            weaponSlots_UI[i].UpdateWeaponSlot(weaponSlots[i], isActiveWeapon);
        }
        else
        {
            weaponSlots_UI[i].UpdateWeaponSlot(null, false);
        }
    }
}
```

4. **Mission Goal UI**
   • Implementation: Create an interface to showcase the current mission's objectives, which can be presented as text descriptions, progress bars, or other forms.
5. **Transparent UI Element on Mouse Hover**
   • Implementation: Utilize UGUI's Event Trigger component to add a mouse hover event to UI elements. Change the transparency of the element when the mouse hovers over it.

```
private Dictionary<Image,Color> originalImageColors = new Dictionary<Image, Color>
private Dictionary<TextMeshProUGUI, Color> originalTexColors = new Dictionary<Text
```

6. **Main Menu Layout**
   • Implementation: Design the layout of the main menu, potentially using Grid Layout Group or Vertical/Horizontal Layout Group to organize buttons and other elements.
7. **UI Buttons**
   • Implementation: Create multiple Button elements and add click event listeners to them.
8. **Main Menu Buttons Functionality**
   • Implementation: Program the corresponding functional logic for each main menu button, such as start game, settings, exit, etc.
9. **Mission Selection UI**
   • Implementation: Create an interface listing selectable missions, allowing players to choose the mission they want to undertake.
10. **Weapon Selection UI**
    • Implementation: Design a weapon selection interface where players can choose different weapons to equip.
11. **Play Button**
    • Implementation: Create a "Start Game" button that initializes the game scene and character states upon clicking.
12. **Pause UI**
    • Implementation: Design a pause interface with options such as resume game, return to main menu, etc.
13. **Time Manager**
    • Implementation: Create a time manager script to control the passage of time in the game, potentially including functions like pause and resume.
14. **Game Over UI**
    • Implementation: Design a game over interface displaying scores, rankings, and options to restart or exit the game.

15. **Fade Screen**
    • Implementation: Use UI animations or shaders to achieve a screen fading effect for scene transitions or UI element display/hiding.
16. **Comic Panel UI**
    • Implementation: Design a comic panel to showcase the game's background story or significant events, potentially using Scroll View to display multi-page content.
17. **Win Screen UI**
    • Implementation: Create a victory interface displaying the player's victory information and rewards.
18. **Settings UI**
    • Implementation: Design a settings interface allowing players to adjust game options such as volume and difficulty.
19. **Clean up**
    • Implementation: After project completion, clean up and optimize the UI system by deleting unnecessary resources and optimizing performance

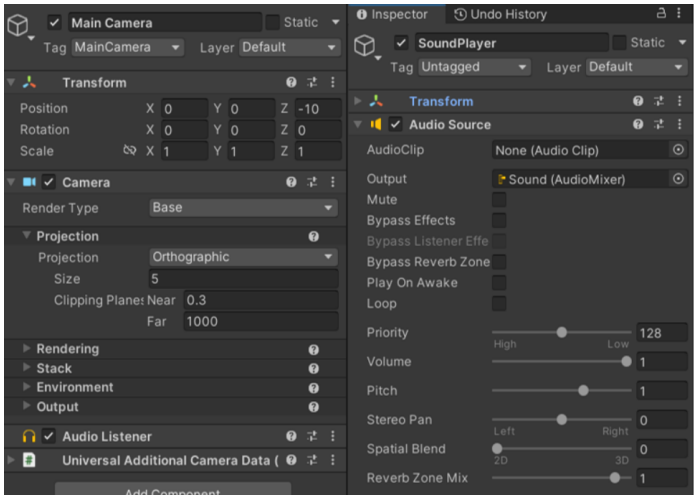
10 Audio

10.1 Audio Source and Audio mixer

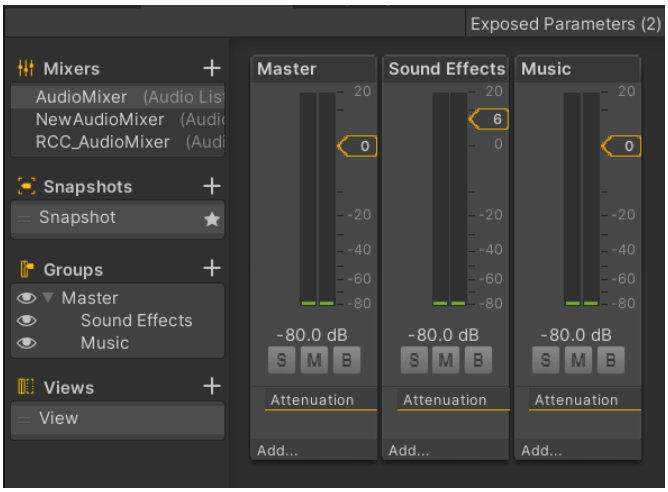
10.1.1 Overview

Two prerequisites are required for sound effects to play successfully: an Audio Source and an Audio Listener.

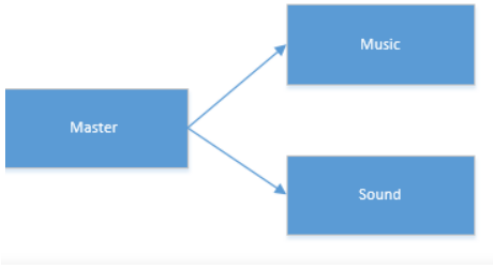
An Audio Source plays Audio in the scene, while an Audio Listener acts as a device similar to a microphone. It receives input from any given audio source in the scene and plays the sound through computer speakers.



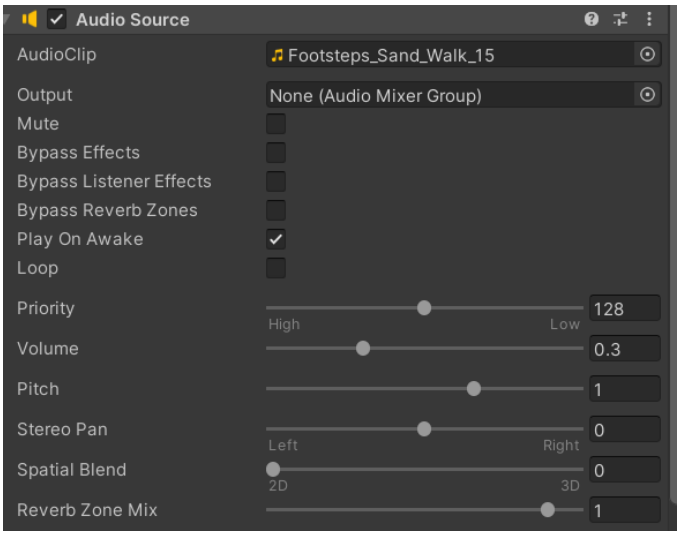
An Audio Mixer allows you to mix various audio sources and apply effects to the audio sources. After mounting the AudioMixer Group Controller file to the Output of the Audio Source script, the audio source emits audio with the AudioMixer effect applied.



The audio in the game can be roughly divided into two kinds: Sound and Music, so here we only achieve the setting of sound and music, and achieve the setting of the Master volume.



10.1.2 Audio Source Properties

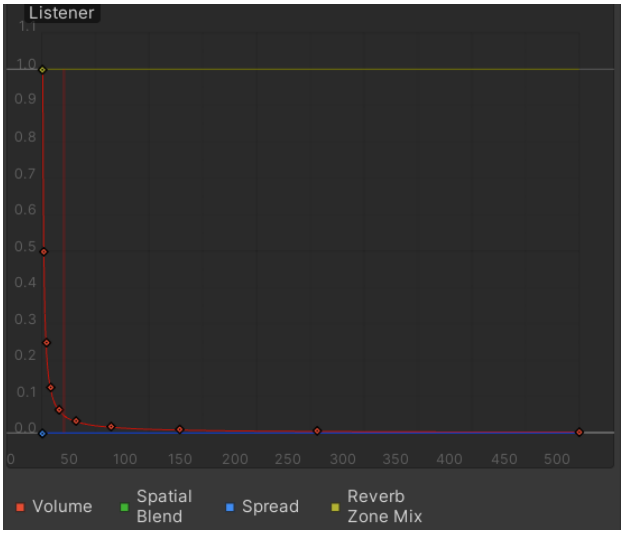


clip	Default AudioClip to play.
dopplerLevel	Set the Doppler scaling of this AudioSource.
gamepadSpeakerOutputType	Gets or sets the gamepad audio output type of the audio source.
ignoreListenerPause	AudioSource is allowed to play even if AudioListener.pause is set to true. This is useful for pausing menu element sounds or background music in the menu.
ignoreListenerVolume	This allows the audio source to take no account of the audio listener's volume.
isPlaying	Is the clip currently playing? (Read only)
isVirtual	true if all the sounds played by the AudioSource (the main sounds started by Play() or playOnAwake and the single play) are knocked out by the audio system.
loop	Do you want to loop this audio clip?
maxDistance	(logarithmic attenuation) MaxDistance is the distance at which the sound stops fading.
minDistance	At a minimum distance, AudioSource will stop increasing the volume.
mute	Mute/unmute the AudioSource. Set the volume to 0 when you mute it, and unmute it to return to the original volume.
outputAudioMixerGroup	AudioSource should route its signal to the target group.
panStereo	Pan the sound being played in stereo (left or right channel). Works in mono or stereo only.
pitch	The pitch of the audio source.
playOnAwake	If set to true, the audio source will automatically start playing when it wakes up.
priority	Set the priority of AudioSource.

Some of the properties below specify how sound is played in a 3D environment. The main properties include how sound decays with distance, the speed at which sound travels, and the maximum and minimum distance at which sound travels.

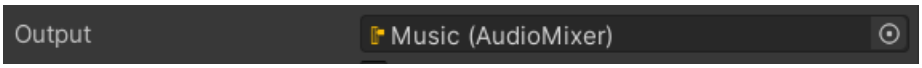
rolloffMode	Sets/gets how the AudioSource decays with distance.
spatialBlend	Set the degree to which 3D spatial calculations (attenuation, Doppler effect, etc.) affect the AudioSource. 0.0 makes the sound full 2D effect and 1.0 makes it full 3D.
spatialize	Enable or disable the spatialization.
spatializePostEffects	Determine whether the spatial sound effect is inserted before or after the effect filter.
spread	Sets the diffusion Angle (in degrees) of 3D stereo or multi-channel sound in the speaker space.
time	Play position (in seconds) .
timeSamples	Play position in the PCM sample.
velocityUpdateMode	Should the audio source be updated in a fixed or dynamic update mode?
volume	Audio source volume (0.0 to 1.0) .

Unity provides specific editing methods for sound propagation. By changing the curve in the figure below in relation to distance, we can dynamically edit the sound changes. For example, for the explosion sound, the collision of bullets, we can change the size of the sound through the distance to indicate the distance between the player and the object, and simulate the simulation effect.



10.1.2 Audio mixer

You can manage the volume of different sound sources with mixer.



Furthermore, the Mixer volume can be edited in-game via the UI interface.


```
public void SFXSliderValue(float value)
{
    sfxSliderText.text = Mathf.RoundToInt(value * 100) + "%";
    float newValue = Mathf.Log10(value) * sliderMultiplier;
    audioMixer.SetFloat(sfxParametr, newValue);

}
```

This is done by changing the value of the slider and returning to audioMixer.

11 Car and LogiSteeringWheel

11.1 WheelColider

11.1.1 Properties of the wheelColider

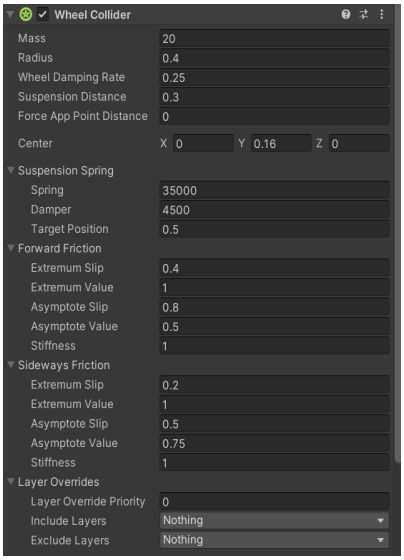
The Wheel collider is a collider for ground vehicles. It has built-in collision detection, wheel physics, and a slip-based tire friction model.

It contains the following properties:

Property	Description
Mass	Set the mass of the Wheel collider (in kilograms). The default value is 20.
Radius	Set the distance (in meters) from the center of the Wheel collider to the edge. Use this property to adjust the size of the Wheel collider. The default value is 0.5.
Wheel Damping Rate	Set the rate at which the wheel's rotational movement slows down when no forces are present (for example, when there is no acceleration). Higher values apply more damping, and cause the wheel to slow down more quickly. Lower values apply less damping, and the wheel takes longer to come to a stop. Use this property to fine-tune the responsiveness of wheels on a simulated vehicle. The default value is 0.25.
Suspension Distance	Set the maximum distance along the vertical Y axis that the Wheel collider can move from its target position (the position when equal or no forces are present). When the wheel encounters an uneven surface or an obstacle, it can move up or down within this defined range of vertical movement, simulating a suspension system compressing or extending. The default value is 0.3. A larger value provides more room for the wheel to move, allowing it to handle larger bumps or obstacles. A smaller value restricts the wheel's travel, making it less capable of dealing with rough terrain or obstacles with significant height differences.
Force App Point Distance	Set the point on the wheel where Unity should apply wheel forces. The value expresses the point as distance in meters along the vertical Y axis from the base of the wheel at rest. The default value is 0, which places the point at the wheel's base. For vehicle simulation, the ideal value is one which applies forces slightly below the vehicle's center of mass.
Center	Position the center of the wheel relative to the GameObject's Transform. The default value for each axis is 0.

Property	Description
Spring	Set the stiffness of the simulated spring (in newtons per meter). The default value is 35000 N/m, which simulates a normal vehicle.
Damper	Set the strength of the simulated damper or shock absorber (in newton-seconds per square meter). The default value is 4500 N-s/m2, which simulates a normal vehicle.
Target Position	Define the Wheel collider's rest point (that is, the location of the center of the Wheel collider when there are no forces or equal forces acting upon it) along the Suspension Distance. A value of 0 indicates the point of fully extended suspension (the bottom of the suspension distance). A value of 1 indicates the point of fully compressed suspension (the top of the suspension distance). By default the target position is 0.5. For most vehicle simulations, a typical value is between 0.3 and 0.7.

The displayed physical wheel can be simulated by setting and changing the relevant values of the wheel collider and calling the API of the collider.

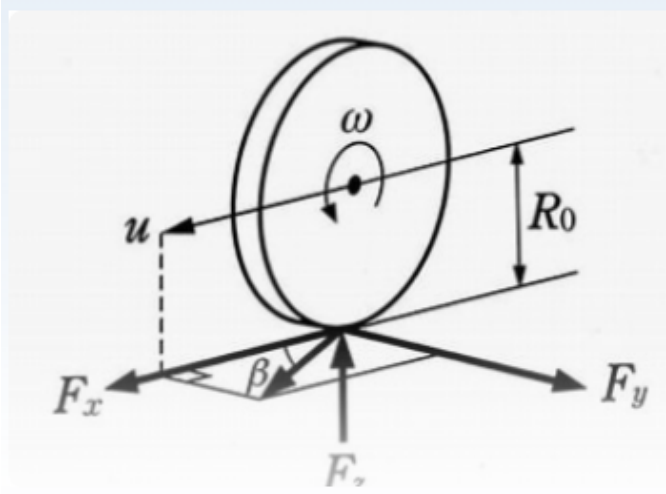


1. Center:Contains the relative center position of the wheel position.
2. The mass and friction coefficient of the wheel can determine the friction from the ground, including the side friction coefficient and the front friction coefficient.
3. Layer Override determines the different effects that occur when the tire is rubbed against different layers.The friction of the wheel on different roads can be changed by setting different layers.

11.1.2 About braking, drive, steering and drift

When setting braking, drive, and steering, changes are made mainly by setting the force values built in the colider.Our code about the design of the car is about mimicking the real physics

of the car.Different from the general velocity vector drive method, we aim to realize the physical simulation of the vehicle through the properties of friction, torque, gravity and so on.



Driving method of the car:

```
private void ApplyDrive()
{
    currentSpeed = moveInput * accleerationSpeed * Time.deltaTime;

    float motorTorqueValue = motorForce * currentSpeed;

    foreach (var wheel in wheels)
    {
        if (driveType == DriveType.FrontWheelDrive)
        {
            if (wheel.axelType == AxelType.Front)
                wheel.cd.motorTorque = motorTorqueValue;
        }
        else if (driveType == DriveType.RearWheelDrive)
        {
            if (wheel.axelType == AxelType.Back)
                wheel.cd.motorTorque = motorTorqueValue;
        }
        else
        {
            wheel.cd.motorTorque = motorTorqueValue;
        }
    }
}
```

1. The currentSpeed is calculated with moveInput (move input), accleerationSpeed (acceleration), and time.deltatime (Time between frames).
2. Calculate the motor torque value motorTorqueValue using motorForce and the currentSpeed calculated earlier.
3. Go through all the wheels and apply the drive.AxiType offers the option of all-wheel, front-wheel, or rear-wheel drive.

In summary, this code calculates the motor torque based on the current input and drive type, and applies it to the corresponding wheel to simulate the driving effect of the vehicle.

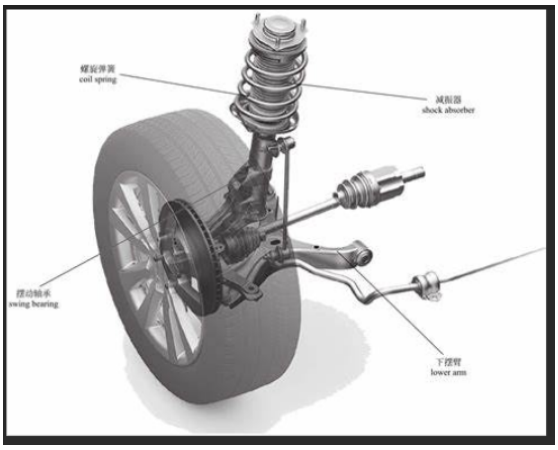
Braking method of the car:

```
private void ApplyBrakes()
{

    foreach (var wheel in wheels)
    {
        bool frontBrakes = wheel.axelType == AxelType.Front;
        float brakeSensetivity = frontBrakes ? frontBrakesSensetivity : backBrakesSensetivity;

        float newBrakeTorque = brakePower * brakeSensetivity * Time.deltaTime;
        float currentBrakeTorque = isBraking ? newBrakeTorque : 0;

        wheel.cd.brakeTorque = currentBrakeTorque;
    }
}
```

1. Go through all wheel objects wheels, processing each wheel.
2. Check that the current wheel is on the front axle and store the result in the frontBrakes variable.
3. The brake sensitivity is determined according to the frontBrakes value. frontBrakesSensetivity is used if it is a front axle wheel, otherwise backBrakesSensetivity is used.
4. Calculate the newBrakeTorque. brakePower (braking force), brakeSensetivity (braking sensitivity), and time.deltatime (Time between frames) are used.
5. The current braking torque is determined according to the isBraking variable. If you are braking (isBraking is true), newBrakeTorque is used, otherwise it is 0.
6. Apply currentBrakeTorque to the brakes of the current wheel (wheel.cd.braketorque).
7. In summary, this code traverses each wheel, calculates and applies braking torque based on its axle type (front or rear axle) and corresponding braking sensitivity to achieve braking effect. The entire process takes into account the combined influence of isBraking as well as braking force and sensitivity.

```
private void ApplySteering()
{
    foreach (var wheel in wheels)
    {
        if (wheel.axelType == AxelType.Front)
        {
            float targetSteerAngle = steerInput * turnSensetivity;
            wheel.cd.steerAngle = Mathf.Lerp(wheel.cd.steerAngle, targetSteerAngle, .5f);
        }
    }
}
```

Also the steering is realised in the same way. Overall, the role of Colider can be described as the wheel and axle parts.



So we can even say that the WheelColider component is actually a part of the car that removes almost all of the outside of the car. In fact, when we remove the body of the car from the game, the wheels and the base drive forward without being affected.

The motorTorce and braking Torce mentioned above actually act on the central position of the wheel. In fact, Colider's motortorque is calculated by:

```
float motorTorqueValue = motorForce * currentSpeed;
```

The motorForce on the right is the power of the engine that we set.

```
private void ApplyDrift()
{
    foreach (var wheel in wheels)
    {
        bool frontWheel = wheel.axelType == AxelType.Front;
        float driftFactor = frontWheel ? frontDriftFactor : backDriftFactor;

        WheelFrictionCurve sidewaysFriction = wheel.cd.sidewaysFriction;

        sidewaysFriction.stiffness *= (1 - driftFactor);
        wheel.cd.sidewaysFriction = sidewaysFriction;
    }
}
```

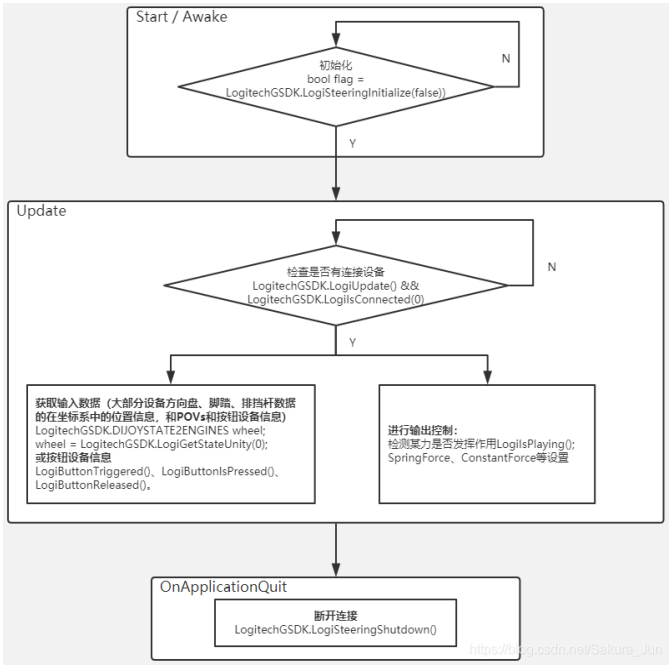
```
driftTimer -= Time.deltaTime;

if (driftTimer < 0)
    isDrifting = false;
```

The drift method of the car is essentially to extend the braking distance of the car by changing the friction of the wheel within a certain time, so as to achieve the drift effect. The timing is calculated by subtracting the time per frame.

11.2 Logitech steering wheel external

11.2.1.API

Logitech's open source offers a variety of methods, using templates roughly as follows:



LogiUpdate Main window device update

Bool LogiUpdate()

(1)The LogiUpdate () finds the main window handler if the force has been found and the controller keeps the connection up to date. It is called every frame of the application.

(2) LogiIsConnected Whether the device is connected

Bool LogiIsConnected(const int index)

The LogiIsConnected () function checks whether the game controller is connected at the specified index

Index: Index of the game controller. 0 corresponds to the first game controller connected. 1 corresponds to the second game controller.

(3) LogiGetStateUnity obtains the device status.

LogiGetStateUnity () returns the state of the struct DIJOYSTATE2 controller.

Index: Index of the game controller. 0 corresponds to the first game controller connected. 1 corresponds to the second game controller.

Here are some important value changes:

Steering wheel：lX

No touch (initial) : 0, after touch, left twist to the end: -32768, right twist to the end: 32767

Throttle: lY

Do not touch (initial) : 0, do not step on: 32767, step out: -32768

Brake: lRz

Do not touch (initial) : 0, do not step on: 32767, step out: -32768

As you can see, these values can be normalized, that is, their value is converted to [0,1], into a coefficient. So we can multiply these values as a coefficient in the original method.As this:

```
private void ApplyLogiSteering(float steering)
{
    float logiSteering = steering;
    foreach (var wheel in wheels)
    {
        if (wheel.axelType == AxelType.Front)
        {
            float targetSteerAngle = 1f * logiSteering * turnSensetivity;
            wheel.cd.steerAngle = Mathf.Lerp(wheel.cd.steerAngle, targetSteerAngle, .5f);
        }
    }
}
```

Because the braking force feedback is too high, the maximum braking value is set directly beyond a certain strength.

```
private void ApplyLogiBrake(float brakeIntensity)
{
    foreach (var wheel in wheels)
    {
        bool isFrontAxle = wheel.axelType == AxelType.Front;
        float brakeSensitivity = isFrontAxle ? frontBrakesSensetivity : backBrakesSensetivity;
        float newBrakeTorque = brakePower * brakeSensitivity * Time.deltaTime;
        float currentBrakeTorque = 0;
        if (brakeIntensity > 0.1f)
        {
            currentBrakeTorque = newBrakeTorque;
            isDrifting = true;
            driftTimer = driftDuration;
        }
        else
        {
            isDrifting = false;
            currentBrakeTorque = 0;
        }
        // Apply the adjusted brake torque to each wheel
        wheel.cd.brakeTorque = currentBrakeTorque;
    }
}
```

11.2.2. Other applications of the steering wheel

CarAirborne simulates the effect of a car flying off the road

(1) bool LogiPlayCarAirborne(int index); Simulate the effect of the car in the air on the road

(2) bool LogiStopCarAirborne(int index); The effect of stopping the car from flying on the road surface.

11.3 The interaction between the car and the player

1. Change the material of the car part to show the interaction range.

2. Switch controller scripts to change the current control object: from the player to the vehicle.

```
1 个引用
private void GetIntoTheCar()
{
    //先把Virtual Camera设置为true

    SwitchCamera.instance.SetCarCameraActive();
    ControlsManager.instance.SwitchToCarControls();
    carHealthController.UpdateCarHealthUI();
    carController.ActivateCar(true);

    defaultPlayerScale = player.localScale.x;

    player.localScale = new Vector3(0.01f, 0.01f, 0.01f);
    player.transform.parent = transform;
    player.transform.localPosition = Vector3.up / 2;

    CameraManager.instance.ChangeCameraTarget(transform, 12, .5f);

}
```

The way we enter the car is:1. Switch cameras.2. Pause the Player. Give control to CarController.3. Reduce the size of the Player to invisible, set the Player as a child of Car and change the position of the Player.

3. Change the camera Angle to make it more conducive to driving.

4. Use four balls to set the player's position when exiting the vehicle. When there is a volume collision or a specified object (such as an enemy) within the radius of the center of the ball, the player does not choose to get off at that location.

11.4 Other

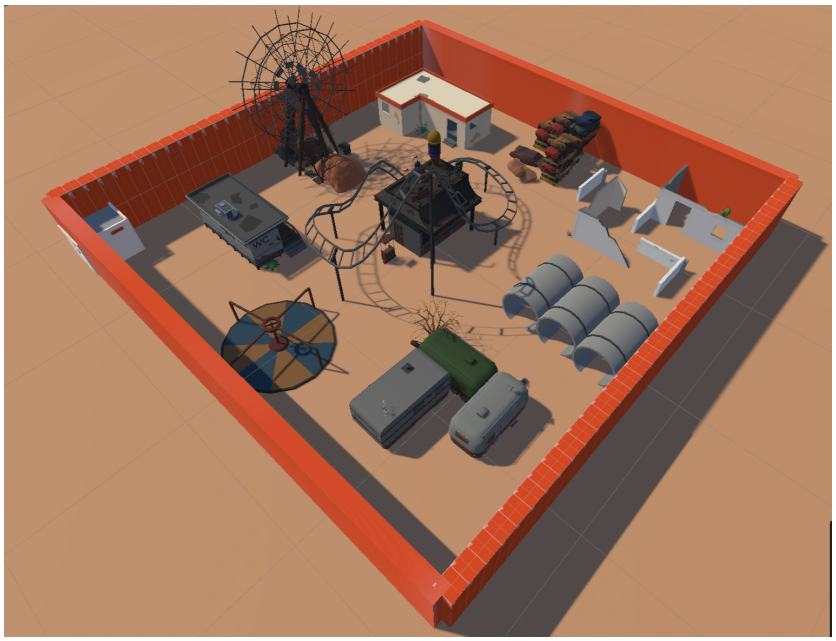
1. Use the Unity engine's own methods to achieve explosive power.

```
hit.GetComponentInChildren<Rigidbody>().AddExplosionForce(explosionForce,
    explosionPoint.position, explosionRadius, explosionUpwardsModifier, ForceMode.VelocityChange);
```

2. Use the HitColider range to detect objects with destructible properties within the explosion range and cause damage and impetus to them during explosion.

3. Use the volume collision between the wheel collider and the ground to change the material and leave ruts.

12. Online part



12.1 Understanding Client-Server Model

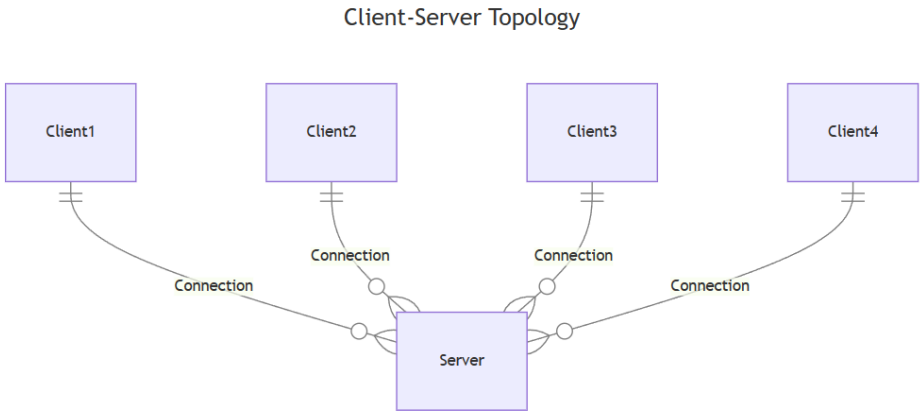
Online gaming has revolutionized interactive entertainment, offering unparalleled opportunities for players to engage in immersive virtual worlds. However, the proliferation of cheating in peer-to-peer gaming environments has underscored the necessity for robust infrastructure to maintain fairness and integrity. The client-server model emerges as a viable solution, centralizing game logic and enforcement mechanisms on a dedicated server. This report elucidates the fundamental principles and practical implementation strategies of the client-server model, augmented by client-side prediction, within the context of Netick.

Traditionally, peer-to-peer gaming architectures suffer from inherent vulnerabilities, as each client possesses unrestricted access to game state and mechanics. Consequently, malicious actors exploit this decentralized structure to manipulate gameplay and gain unfair advantages. In contrast, the client-server model establishes a hierarchical relationship, with clients interfacing exclusively with a central server. By relinquishing client authority over game execution, the server assumes responsibility for processing inputs and disseminating authoritative game state updates. This architectural shift fortifies the integrity of online gaming ecosystems, thwarting cheating attempts and ensuring equitable gameplay experiences for all participants.

While the client-server model offers unparalleled security benefits, it introduces latency issues inherent in network communications. To mitigate perceptible delays and maintain fluid gameplay, Netick employs client-side prediction techniques. By extrapolating future game states based on locally generated inputs, clients preemptively simulate the effects of player actions, ensuring seamless continuity in the absence of immediate server responses. This proactive approach to synchronization not only minimizes perceived latency but also heightens player immersion by fostering a responsive and dynamic gaming environment.

Netick represents a paradigmatic shift in online gaming infrastructure, embodying the

symbiotic fusion of client-server architecture and client-side prediction. Through meticulous input validation and deterministic execution, Netick empowers servers to enforce game rules with unwavering authority, thereby eradicating cheating vectors prevalent in peer-to-peer environments. Moreover, Netick's modular design facilitates seamless integration with diverse gaming frameworks, affording developers unparalleled flexibility in crafting immersive multiplayer experiences.



12.2 Core Concepts

12.2.1 Network Sandbox

NetworkSandbox is what controls the whole network game. It can be thought of as representing an instance of the game. You can have more than one network sandbox in a single Unity game, and that happens when you start both a client and a server on the same project. This can be extremely useful for testing/debugging, because it allows you to run a server and a client (or multiple thereof) in the same project and therefore see what happens at both at the same time, without interference.

- Therefore you can think of a sandbox as representing a server or a client.
- You can show/hide the current sandboxes from the Network Sandboxes panel.

12.2.2 Network Object

Any GameObject which needs to be synced/replicated must have a NetworkObject component added to it. If you want to see something on everyone’s screen, it has to have a NetworkObject component added to it. It’s the component that tells Netick that a GameObject is networked. The NetworkObject component by itself just informs Netick that the object is networked. To add networked gameplay-logic to it, you must do so in a component of a class derived from NetworkBehaviour. Netick comes with a few essential built-in components:

- NetworkTransform: used to sync position and rotation
- NetworkRigidbody: used to sync controllable physical objects
- NetworkAnimator: used to sync Unity’s animator’s state

12.2.3 Network Behaviour

The NetworkBehaviour class is your old friend MonoBehaviour, just the networked version of it. To implement your networked functionality, create a new class and derive it from NetworkBehaviour. You have several methods you can override which correspond to Unity's non-networked equivalents (they must be used instead of Unity's equivalents when doing anything related to the network simulation):

- NetworkStart
- NetworkDestroy
- NetworkFixedUpdate
- NetworkUpdate
- NetworkRender

12.3 Remote Procedure Calls (RPCs)

RPCs are method calls on Network Behaviors that are replicated across the network. They can be used for events or to transfer data.
An important use of RPCs is to set up the game and send configuration messages. Use reliable RPCs for things like that.

12.3.1Static RPCs

Static RPCs must have their first parameter as a type. Which can be used to access the instance if needed NetickEngineNetworkSandbox.

12.3.2 method constraints

- Must have the return type of void.
- Must not have reference types as parameters.
- Must not have class-based network collections as parameters. Only varaints are allowed as array RPC parameters.NetworkArrayStruct
- Must not have as a parameter type. Instead, you can use NetworkString varaints.string

12.3.3 attribute parameters

- Source: the peer/peers the RPC should be sent from
- Target: the peer/peers the RPC will be executed on
- isReliable: whether the RPC is sent reliably or unreliably
- localInvoke: whether to invoke the RPC locally or not

Source and target can be any of the following:

- Owner (the server)
- Input Source: the client which is providing inputs for this Network Object
- Proxies: everyone except the Owner and the Input Source
- Everyone: the server and every connected client

12.3.4 Source Connection of RPCs

If you need to know which connection (a client, or the server) the current RPC is being executed from, you can use Sandbox.CurrentRpcSource

12.4 Client-Side Prediction Code

12.4.1 Network Input

Network Input describes what the player wants to do, which will be used to simulate the state of objects they want to control. This ensures that the client can't directly change the state – the change happens by executing the input, which, even if tampered with, won't be game-breaking.

12.4.2 Defining Inputs

To define a new input, create a struct that implements INetworkInput interface:

```
public struct MyInput : INetworkInput
{
    public NetworkBool ShootInput;
    public float       MoveDirX, MoveDirY;
}
```

12.4.3 Network Input Constraints

- Must not have class-based network collections as fields. Only NetworkArrayStruct variants are allowed as network input fields.
- Must not have reference types as fields.
- Must not have string as a field. Instead, you can use NetworkString variants.

12.4.4 Setting Inputs

To set the fields of an input, you first need to acquire the input struct of the current tick, using Sandbox.GetInput.

Then, you can set it inside NetworkUpdate on NetworkBehaviour:

```
public override void NetworkUpdate()
{
    var input = Sandbox.GetInput<MyInput>();

    input.MoveDirX = Input.GetAxis("Horizontal");
    input.MoveDirY = Input.GetAxis("Vertical");

    Sandbox.SetInput(input);
}
```

12.4.5 Simulating (Executing) Inputs

To drive the gameplay based on the input struct, you must do that in NetworkFixedUpdate:

```
public override void NetworkFixedUpdate()
{
    if (FetchInput(out MyInput input))
    {
        // movement
        var movement = transform.TransformVector(new Vector3(input.MoveDirX, 0, input.MoveDirY)) * Speed;
        movement.y   = 0;
        _CC.Move(movement * Time.fixedDeltaTime);
        // shooting
        if (input.ShootInput == true && !IsResimulating)
            Shot();
    }
}
```

Fetch Input tries to fetch an input for the state/tick being simulated/resimulated. It only returns true if either:

- We are providing inputs to this object – meaning we are the Input Source of the object.
- We are the owner (the server) of this object – receiving inputs from the client who's the Input Source. And only if we have an input for the current tick being simulated. If not, it would return false. Usually, that happens due to packet loss.

And to avoid the previous issue we talked about, we make sure that we are only shooting if we are simulating a new input, by checking IsResimulating.

12.4.6 Input Source

For a client to be able to provide inputs to be used in an Object's NetworkFixedUpdate, and hence take control of it, that client must be the Input Source of that object. Otherwise, FetchInput will return false. To check if you are the Input Source, use IsInputSource.
The server can also be the Input Source of objects, although it won't do any CSP, since it needs not to, after all, it's the server.

12.4.8 Callbacks

There are two methods you can override to run code when Input Source has changed or left the game:

- OnInputSourceChanged: called on the Input Source and server when the Input Source changes.
- OnInputSourceLeft: called on the owner (server) when the Input Source client has left the game.

12.5 Listening to Network Events

Netick has several useful callbacks we use:

Callbacks	Description	Invoke target
OnStartup(NetworkSandbox sandbox)	Invoked when Netick has been started.	Client/Server
OnShutdown(NetworkSandbox sandbox)	Invoked when Netick has been shut down.	Client/Server
OnInput(NetworkSandbox sandbox)	Invoked to read inputs.	Client/Server
OnConnectRequest(NetworkSandbox sandbox, NetworkConnectionRequest request)	Invoked when a client tries to connect. Use request to decide whether or not to allow this client to connect.	Server
OnConnectFailed(NetworkSandbox sandbox, ConnectionFailedReason reason)	Invoked when the connection to the server was refused, or simply failed.	Client
OnConnectedToServer(NetworkSandbox sandbox, NetworkConnection server)	Invoked when the connection to the server has succeeded.	Client
OnDisconnectedFromServer(NetworkSandbox sandbox, NetworkConnection server, TransportDisconnectReason transportDisconnectReason)	Invoked when connection to the server ended, or when a network error caused the disconnection.	Client
OnPlayerConnected(NetworkSandbox sandbox, Netick.NetworkPlayer client)	Invoked when a specific player has connected. Called for the server, when started as a Host.	Server
OnPlayerDisconnected(NetworkSandbox sandbox, Netick.NetworkConnection client)	Invoked when a specific player has disconnected. Invoked in host mode for the host player when Netick shuts down.	Server
OnClientConnected(NetworkSandbox sandbox, NetworkConnection client)	Invoked when a specific client has connected.	Server
OnClientDisconnected(NetworkSandbox sandbox, NetworkConnection client)	Invoked when a specific client has disconnected.	Server
OnSceneOperationBegan(NetworkSandbox sandbox, NetworkSceneOperation sceneOperation)	Invoked when a scene operation has began.	Client/Server
OnSceneOperationDone(NetworkSandbox sandbox, NetworkSceneOperation sceneOperation)	Invoked when a scene operation has finished.	Client/Server
OnObjectCreated(NetworkObject obj)	Invoked when a network object has been created/initialized.	Client/Server
OnObjectDestroyed(NetworkObject obj)	Invoked when a network object has been destroyed/recycled.	Client/Server

12.6 Physics Prediction

Predicting physics means resimulating multiple physics steps in one tick. This can be very expensive and so by default physics prediction is turned off. To enable it, go to Netick -> Settings and enable Physics Prediction.

To make a Rigidbody/Rigidbody2D predictable, add NetworkRigidbody/NetworkRigidbody2D to its GameObject.

To enable/disable Physics Prediction in the client at runtime, use Sandbox.PhysicsPrediction.

12.7 Cost of Predicting PhysX (Rigidbody3D)

It's very expensive to predict 3D physics as PhysX and its integration with Unity perform very badly when calling PhysicsScene.Simulate multiple times in one frame, even with small numbers of rigidbodies.

The cost of predicting physics increases with two factors:

- Ping
- Tickrate

As ping increases, you would need to simulate more ticks in one frame for rollback and resimulation. As tickrate increases, the time period between each tick becomes smaller, therefore you will need to simulate more ticks for smaller values of latency.

We did not use 3D physics prediction, as it's almost impractical on some machines on relatively high tickrates (+33). It can take more than 10ms on some machines on 100ms ping just to resimulate a bunch of physics ticks.

13. Task and dialog system

13.1 Overview

An Objective Manager and Quest system for Unity that is provided by third-party library, and easily add them to the Objective Manager.

Our package is very easy to use and supports five different types of objectives:

1. **Time:**
    - **Description:** Perform an action for a certain period.
    - **Examples:** Don't die for 30 seconds, swim for 60 seconds.
2. **Kill:**
    - **Description:** Destroy or kill something a certain number of times.
    - **Examples:** Kill 10 enemy soldiers, hunt 10 animals, destroy 5 enemy tanks.
3. **Collect:**
    - **Description:** Collect a specific amount of something.
    - **Examples:** Collect 30 magical mushrooms, find and collect 5 keys, collect the missing parts of a puzzle.
4. **Go:**
    - **Description:** Go to a specific location or move to a specific position.

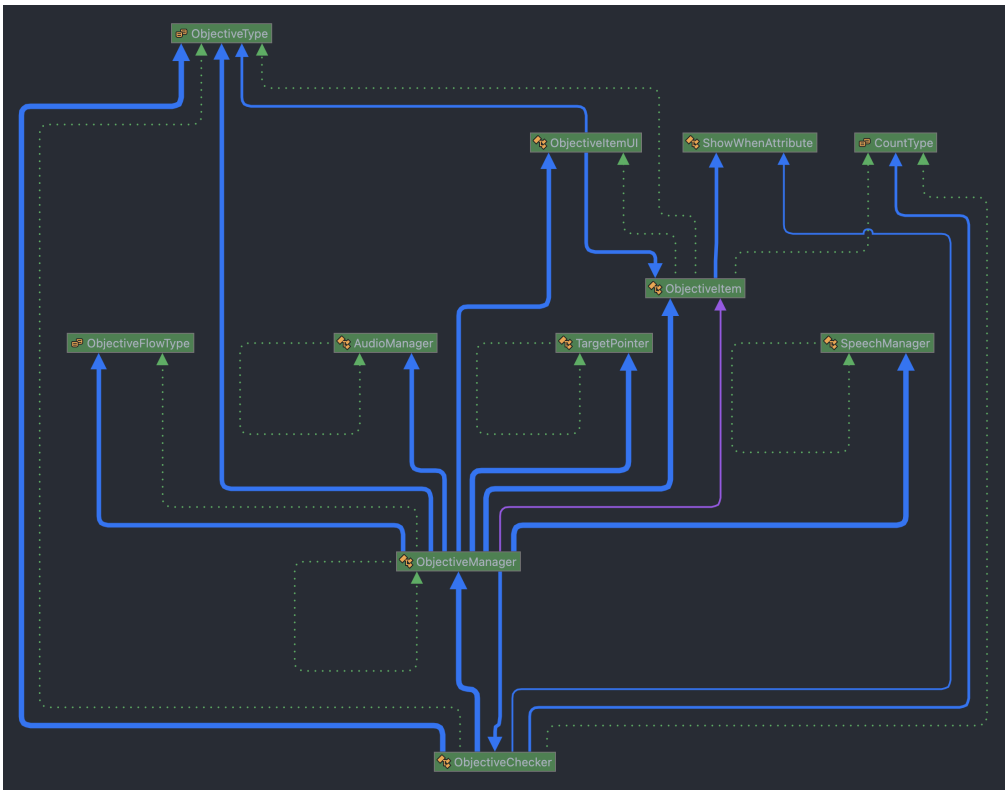
- **Examples:** Go to the chief, find the mystical portal, go to the lobby, find the hostages.

5. **Speak:**
   - **Description:** Speak with a target and have a dialogue with them.
   - **Examples:** Go to the seller and talk with him.

In addition to these core features, our package also includes extra functionalities:

- **Show Waypoint Marker:**
  - If you activate this feature for an objective, an arrow will direct you to the target's position on your camera while the player is on this objective.
- **Dialogue System:**
  - The package includes an embedded dialogue system, allowing targets to have multiple dialogues with the player.
- **Events to Trigger When Objective is Done:**
  - You can assign any kind of Unity Event or set a GameObject name and method name for objectives. When a player completes an objective, this specific GameObject will be found and the event will be invoked automatically.

13.2 Code diagram



Above is the code structure, and below is our critique of the code structure.

ObjectiveManager:

- **Central Management:** Acts as the central hub for managing all objectives within the game. It coordinates the flow of data between different components and ensures objectives are tracked and updated correctly.

- **Interactions:** Connects with various other classes such as ObjectiveType, ObjectiveFlowType, ObjectiveChecker, AudioManager, and SpeechManager.

ObjectiveType:

- **Defines Objective Types:** Holds information about different types of objectives (e.g., Time, Kill, Collect, Go, Speak).
- **Usage:** Referenced by the ObjectiveManager to categorize and handle different types of objectives.

ObjectiveFlowType:

- **Flow Control:** Manages the flow or sequence of objectives. Determines how objectives are linked and in what order they should be completed.
- **Interaction:** Works with ObjectiveManager to ensure objectives are presented and handled in a logical sequence.

ObjectiveChecker:

- **Validation:** Checks the status of objectives to determine if they have been completed.
- **Interaction:** Communicates with ObjectiveManager to update the status of objectives and trigger events when objectives are completed.

ObjectiveItem:

- **Individual Objectives:** Represents individual objective items within the system. Each item contains specific details about an objective.
- **Interaction:** Connects with ObjectiveManager, ObjectiveType, and other relevant classes to retrieve and update objective details.

ObjectiveItemUI:

- **User Interface:** Handles the display of objectives in the game's UI. Ensures that players can see their current objectives and track their progress.
- **Interaction:** Communicates with ObjectiveItem to retrieve objective details and display them to the player.

ShowWhenAttribute:

- **Conditional Display:** Controls when certain objectives or UI elements should be shown based on specific conditions.
- **Interaction:** Works with ObjectiveItemUI and ObjectiveManager to manage the visibility of objectives.

C**ountType:**

- **Counting Mechanism:** Tracks the counts related to objectives, such as the number of items collected or enemies defeated.
- **Interaction:** Works with ObjectiveItem and ObjectiveChecker to update counts and validate completion.

AudioManager:

- **Audio Feedback:** Manages audio cues related to objectives, such as notifications for objective completion or updates.
- **Interaction:** Connects with ObjectiveManager to trigger audio events based on objective status.

TargetPointer:

- **Navigation Aid:** Provides directional cues to guide players towards objective

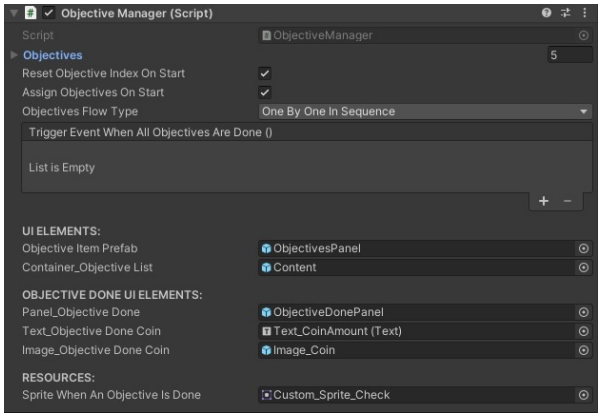
locations.

- **Interaction:** Communicates with ObjectiveManager and ObjectiveItem to determine target locations and provide guidance.

SpeechManager:

- **Dialogue Management:** Handles dialogues related to objectives, allowing for interactions between the player and NPCs.
- **Interaction:** Works with ObjectiveManager and ObjectiveItem to manage and trigger dialogues based on objective progress.

13.3 The method of using the system.



Let's explore the properties of Objective Manager by checking on Inspector area:

- **Objectives:** All Objectives and Quests will be stored in this Array. If your game has 30 different Objectives, there should be 30 Objective Item in this array.
- **Reset Objective Index on Start:** If this feature is checked, all objectives will be reset and player will start to play the game from beginning when player launches the game. If this feature is not checked, this means that when the game is launched, the player will continue to the objective where he left uncompleted before.
- **Assign Objectives on Start:** If this feature is checked, objective manager will assign the next objective to the player as soon as the scene is started.
- **Trigger Event When All Objectives are Done:** You can add Unity Events and functions as

Objectives Flow Type: This is very important configuration for your game. You determine how your Objective Flow will work by configuring this setting. There are two types:

- One By One In Sequence: All Objectives will be assigned one by one in queue. When player completes an objective, the next objective in the queue will be assigned. Player will see only one objective at a time on the UI.
- Multiple in Parallel: All Objectives will be assigned at the same time as a list.

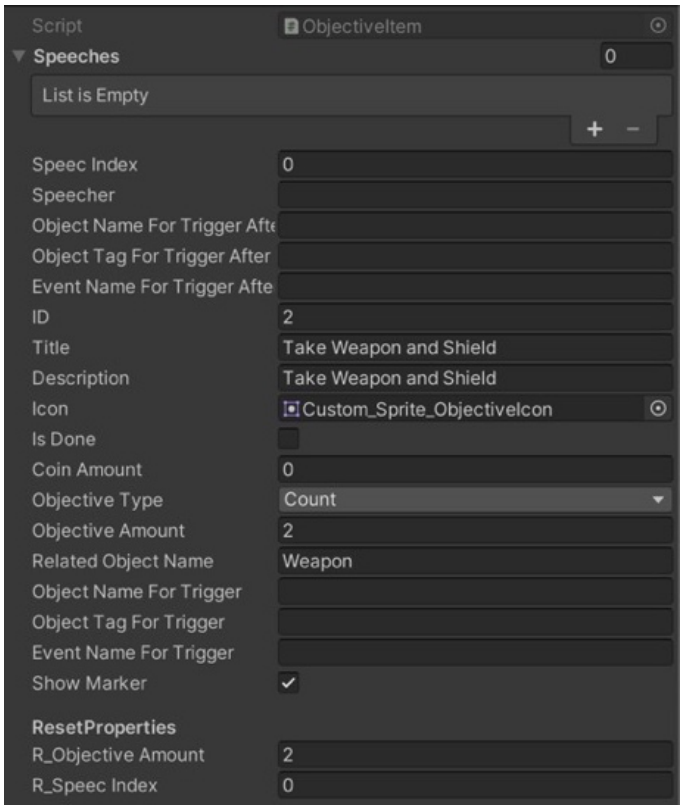
Objective Manager and Creating Objectives

The most of the games have got Objectives in their story and players must complete the objectives. This package has got an easy to use but powerful Objective System. So, you can easily create objectives in your game for different scenarios.

The package supports 4 type of main Objectives types. These are:

- Time: To do something for a certain of period. (Example: Don't die for 30 Seconds, Swim for 60 Seconds, etc...)
- Count: To do something a certain number of times. (Example: Collect 30 Flowers, Kill 10 Enemy Soldiers, etc...)
- Go: To go somewhere. (Example: Go to Chief, Find the Mystical Portal, etc...)
- Speak: To speak with target. (Example: Go to Seller and talk with him, etc...)

You can check Objective Manager component on Game Manager object in Sample Scene. You can create a new Objective by simply right click on Project Panel > Create > Scriptable Object > Objective Item.



Speeches: Add conversations here if you have want to make your character have conversation with someone before taking the objective.
Speecher: Add related Game Object name here (Who will we talk with)
Object Name for Trigger After Speech: Type GameObject name here if you want to trigger any function on a Game Object.
Object Tag for Trigger After Speech: Type GameObject tag here if you want to trigger any function on a Game Object which has this tag.
Event Name for Trigger After Speech: Type function name here that will be triggered right after the conversation.

An example scenario:
Speech 1: Hello!
Speech 2: Take the car. Drive it and find my missing daughter please!
Object Name For Trigger After Speech: GarageDoor

Event Name For Trigger After Speech: Open
As soon as the conversation ends. The Objective Manager will find the game object called "GarageDoor" and trigger a function called as "Open". So the garage door will be opened and your player will be able to drive it.

Objective Type: You can select the objective type from here. Go, Speak, Time or Count.
Objective Amount: The amount of objective. How many times the objective needs the Related Object Name in order to complete the objective. - Object Name For Trigger: When you complete the objective, this object will be searched on the scene with GameObject.Find method.
Object Tag For Trigger: When you complete the objective, this object will be searched on the scene with GameObject.FindByTag method.
Show Waypoint: If you want to point the way and direct the player towards to the objective target, you can use this feature.