Dark Allies Technical Summary

Benhou Li

2024

1 Physics System

1.1 Movement

For the player character with control, movement commands are read from the input device, and movement parameters (such as vertical and horizontal movement amounts, movement intensity, etc.) are sent to the server.

For non-controlled characters, movement parameters are received from the server.

The movement direction is calculated using the forward vector of the camera multiplied by the forward input, plus the right vector of the camera multiplied by the lateral input.

Jumping is controlled by a separate free-fall motion equation to achieve the jumping effect.

AI character movement is controlled by the navigation mesh (NavMesh).

1.2 Camera

The horizontal and vertical movement of the camera is controlled by two parent objects, each handling one axis.

When there is no target lock, the camera rotates freely based on player input.

Horizontal and vertical rotation is achieved based on joystick or mouse input, adjusting the angle accordingly.

Collision Avoidance:

A spherical ray (Physics.SphereCast) is emitted from the camera's pivot point (rotation axis) in the current direction to check if there are any obstacles between the camera and the pivot.

If a collision is detected, the distance between the pivot and the collision point is calculated.

The camera's position is adjusted to be at a certain distance from the collision point to ensure it doesn't penetrate the obstacle.

A minimum distance limit is set, and if the adjusted distance is less than the camera's collision radius, the camera is set to a safe minimum distance from the pivot.

The camera's current position is gradually moved to the target position using interpolation (Mathf.Lerp) to ensure smooth adjustment.

2 Locking System

Search for all lockable objects within a sector-shaped area.

Lock onto the nearest target.

Store the nearest targets on the left and right sides for switching between targets in the locked state.

Use quaternion interpolation to smoothly adjust the camera's current rotation to face the target.

Create a coroutine to control the camera's vertical movement to a fixed height slightly above the target.

Finally, save the current left/right and up/down angles to ensure the camera does not jump suddenly when unlocking.

3 Saving and Loading

3.1 Saving

The game's save data is stored as a JSON file in the device's persistent storage path, ensuring data can be saved and loaded correctly across different devices.

When the player chooses to save the game, the system selects the corresponding file name based on the current save slot.

Game data (such as player attributes, status, items, etc.) is extracted and stored in the current character data.

The character data is then written to the JSON file using a file writer to complete the save process.

3.2 Loading

When the player chooses to load the game, the system selects the appropriate save file name based on the current save slot, reads the data from the JSON file, and populates the current character data.

The player's state is then reloaded into the game based on the character data.

During loading or creating a new game, the system invokes a coroutine to asynchronously load the game world scene while restoring player data.

4 Damage and Attack System

The damage and attack systems operate independently. When damage is triggered, sound effects and visual effects are played simultaneously, and the hit animation is determined based on the hit angle.

Attacks are executed by enabling or disabling colliders in the animation system to check whether the attack hits the target.

5 Weapon System

5.1 Weapon Attribute Definition

The system defines a weapon class that describes various weapon attributes, including appearance model, attack power, usage requirements, attack effects, stamina consumption, etc. This part defines the basic characteristics of the weapon and serves as the core data source for the weapon system.

The class also contains information related to weapon actions, such as attack animations and sound effects, which provide feedback when the weapon is used.

5.2 Player Weapon Management

The system includes a player equipment management module that manages the player's currently equipped weapon. The player can switch between different weapons for the left and right hands and manage quick-switch slots for weapons.

This module connects the player and the weapon, maintaining the current weapon state and providing interfaces for other systems to query the player's equipped weapon.

5.3 Weapon Damage Management

The system also includes a weapon management module to handle weapon damage in combat. When the player attacks with a weapon, this module extracts damage information from the weapon definition and applies it to the attack collider (responsible for detecting hit targets).

By associating with the weapon's attributes, this module applies the weapon's damage information, attack power, and effects to the combat system, ensuring the attack's damage calculation matches the weapon's attributes.

Summary

- **Data Layer**: The weapon attribute class provides the core data definition of the weapon, containing all the attribute information related to the weapon.
- **Management Layer**: The player weapon management module maintains and operates the weapon state related to the player's equipment, providing functions such as switching, equipping, and querying.
- **Application Layer**: The weapon damage management module handles the actual application of weapon attributes in combat, combining

the player's attack with the weapon's damage attributes to achieve combat effects.

Overall, the weapon system separates weapon definition, equipment management, and damage handling into different modules, ensuring clear code organization and easy maintenance and extension.

6 Monster AI

State Machine Driven: Uses a Finite State Machine (FSM) to manage AI behaviors, including idle, chasing, combat, etc., switching states based on different situations.

Navigation and Movement: Utilizes Unity's NavMesh to detect and update movement status in real-time.

Combat Management: Adjusts combat behavior based on the target's position and status, intelligently pursuing and attacking.

Animation and Synchronization: Animation management ensures synchronization between movements and combat actions with actual positions, while network synchronization ensures consistent actions in a multiplayer environment.

7 Multiplayer Functionality

Host: Automatically generates an initial character when entering the game. If loading a saved game, the initial character is destroyed, and a new one is generated based on saved data.

Client: When joining a game, the initialized character data is loaded. In the future, the feature to load custom characters will be added.