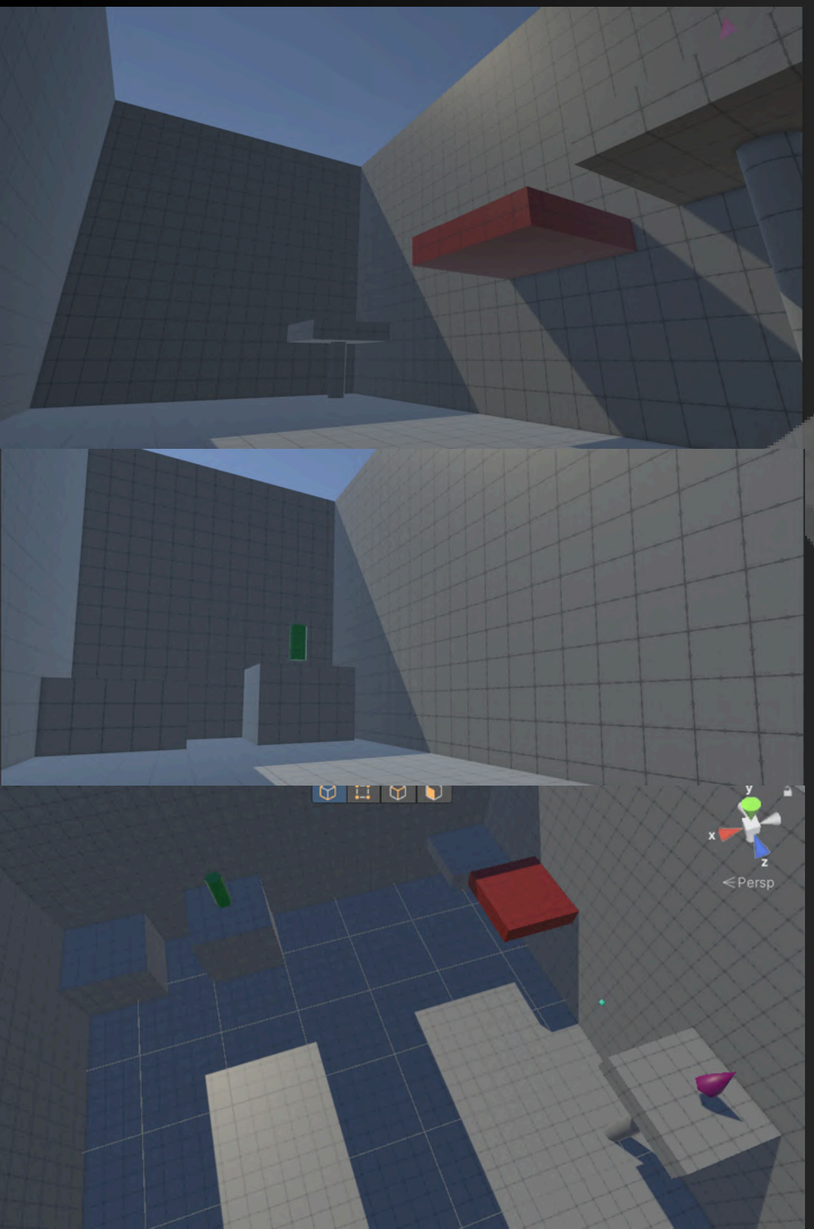
# Game Development with C# and Unity

## 3D FPS Game


The game has been uploaded to junjieg.itch.io/3d-fps. You can play the web version on it, or download the installation package for Windows/MacOS (for a better experience).

# Game Scene Construction




- Use the ProBuilder 3D modeling tool to build a scene graybox, and create the level's spatial layout (including room dimensions, corridor width, and platform height) with low-poly models

- In this testing environment, simultaneously debug basic scene configurations and core character attributes:
  Verify if scene settings meet basic functional requirements;
  Confirm if the character's movement speed matches the scene exploration rhythm and if the jump height is compatible with terrain interactions;

- Validate the rationality of the game scene's size proportions to avoid increased gameplay/art iteration costs due to proportion imbalances in subsequent stages

# Character Setup

## Design Goals

The core goals of character setup are threefold: balancing game difficulty, improving collaboration efficiency, and enhancing player immersion. By aligning character abilities with level difficulty and lowering the threshold for ability adjustments, it ultimately guide players to smoothly engage with the game, avoiding frustration or boredom.

## Implementation Methods

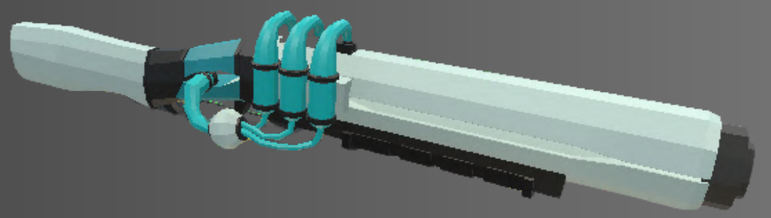
Deeply connect the player's core abilities (movement speed, jump height) to level progression. Set low-threshold challenge for early tutorial levels, such as moderate movement speed and easy-to-control jump height, to help players quickly get familiar with operations. For subsequent levels, fine-tune parameters based on terrain complexity (e.g., narrow paths, high platforms)—for example, increasing movement speed to match exploration needs or expanding jump height tolerance—allowing difficulty to rise naturally as players master their abilities.

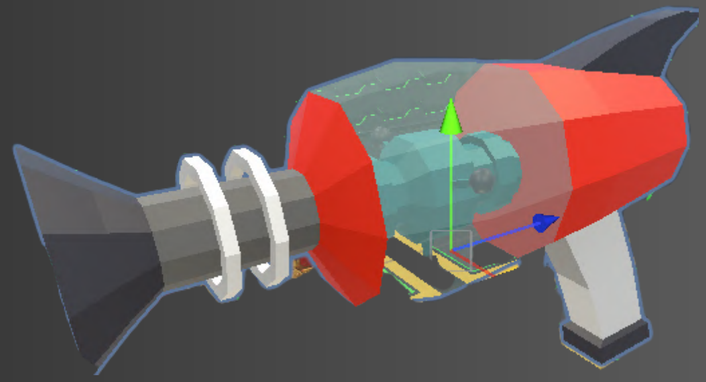
# Character Setup


Set up external adjustable interfaces for character abilities in Unity, syncing key parameters (movement speed, jump height) to the Inspector panel. For teamwork, non-programmers (e.g., game designers, artists) can modify parameters by dragging sliders in the editor without writing code, testing the impact of different values on gameplay in real time, and significantly shortening the parameter iteration cycle.

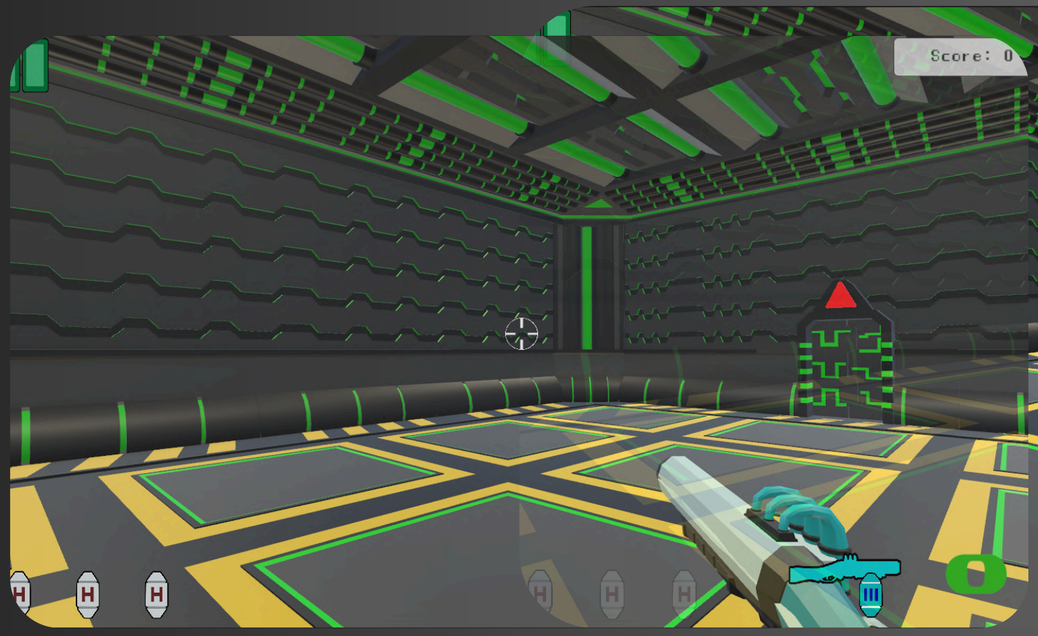
```
/// <summary>
/// This processes the movement on the player based on the player inputs as well as gravity.
/// </summary>
Vector3 moveDirection; // store the character's 3D movement direction
float timeToStopBeingLenient = 0f; // track when to stop being lenient based on the jumpTimeLenience
bool doubleJumpAvailable = false; // keep track if the player can double jump - they cannot until they are grounded and jump once
void ProcessMovement()
{
    // Get the input from the player
    float leftRightInput = moveInput.ReadValue<Vector2>().x;
    float forwardBackwardInput = moveInput.ReadValue<Vector2>().y;
    bool jumpPressed = jumpInput.triggered;

    // Handle the control of the player while it is on the ground
    if (controller.isGrounded && moveDirection.y <= 0)
    {
        doubleJumpAvailable = true;
        timeToStopBeingLenient = Time.time + jumpTimeLeniency;

        // Set the movement direction to be the received input, set y to 0 since player is on the ground
        moveDirection = new Vector3(leftRightInput, 0, forwardBackwardInput);
        // Set the move direction in relation to the transform
        moveDirection = transform.TransformDirection(moveDirection);
        moveDirection = moveDirection * moveSpeed;

        if (jumpPressed)
        {
            moveDirection.y = jumpPower;
        }

    }
    else
    {
        moveDirection = new Vector3(leftRightInput * moveSpeed, moveDirection.y, forwardBackwardInput * moveSpeed);
        moveDirection = transform.TransformDirection(moveDirection);

        if (jumpPressed && Time.time < timeToStopBeingLenient)
        {
            moveDirection.y = jumpPower;
        }
        else if (jumpPressed && doubleJumpAvailable)
        {
            moveDirection.y = jumpPower;
            doubleJumpAvailable = false;
        }
    }

    if (controller.isGrounded && moveDirection.y < 0)
    {
        moveDirection.y = -0.3f;
    }

    // add effect of gravity
    if (isFalling())
    {
        // Mario style falling where gravity is more on fall than on jump, to make the player seem less floaty
        moveDirection.y -= gravity * fallingMultiplier * Time.deltaTime;
    }
    else
    {
        // Apply regular gravity
        moveDirection.y -= gravity * Time.deltaTime;
    }


    controller.Move(moveDirection * Time.deltaTime);
}
```

# Weapon System and In-Game UI





For a 3D FPS game, the weapon system and real-time UI display of ammunition are undoubtedly indispensable core components




# Core implementation logic of the code

```
/// <summary>
/// Description:
/// Checks input and responds accordingly
/// Input:
/// none
/// Return:
/// void (no return)
/// </summary>
void CheckInput()
{
    if (!isPlayerControlled)
    {
        return;
    }

    // Do nothing when paused
    if (Time.timeScale == 0)
    {
        return;
    }

    if (guns.Count > 0)
    {
        if (guns[equippedGunIndex].fireType == Gun.FireType.semiAutomatic)
        {
            if (fireInput.triggered)
            {
                FireEquippedGun();
            }
        }
        else if (guns[equippedGunIndex].fireType == Gun.FireType.automatic)
        {
            if (fireInput.triggered || fireInput.ReadValue<float>() >= 1)
            {
                FireEquippedGun();
            }
        }

        if (cycleWeaponInput.ReadValue<Vector2>().y != 0)
        {
            CycleEquippedGun();
        }

        if (nextWeaponInput.triggered)
        {
            GoToNextWeapon();
        }

        if (previousWeaponInput.triggered)
        {
            GoToPreviousWeapon();
        }
    }
}
```

Monitor the player's input and make the weapon system respond

```
/// <summary>
/// Description:
/// Goes to the next weapon of the available weapons
/// </summary>
public void GoToNextWeapon()
{
    List<Gun> availableGuns = guns.Where(item => item.available == true).ToList();
    int maximumAvailableGunIndex = availableGuns.Count - 1;
    int equippedAvailableGunIndex = availableGuns.IndexOf(guns[equippedGunIndex]);

    equippedAvailableGunIndex += 1;
    if (equippedAvailableGunIndex > maximumAvailableGunIndex)
    {
        equippedAvailableGunIndex = 0;
    }

    EquipGun(guns.IndexOf(availableGuns[equippedAvailableGunIndex]));
}

/// <summary>
/// Description:
/// Goes to the previous weapon of the available weapons
/// </summary>
public void GoToPreviousWeapon()
{
    List<Gun> availableGuns = guns.Where(item => item.available == true).ToList();
    int maximumAvailableGunIndex = availableGuns.Count - 1;
    int equippedAvailableGunIndex = availableGuns.IndexOf(guns[equippedGunIndex]);

    equippedAvailableGunIndex -= 1;
    if (equippedAvailableGunIndex < 0)
    {
        equippedAvailableGunIndex = maximumAvailableGunIndex;
    }

    EquipGun(guns.IndexOf(availableGuns[equippedAvailableGunIndex]));
}
```

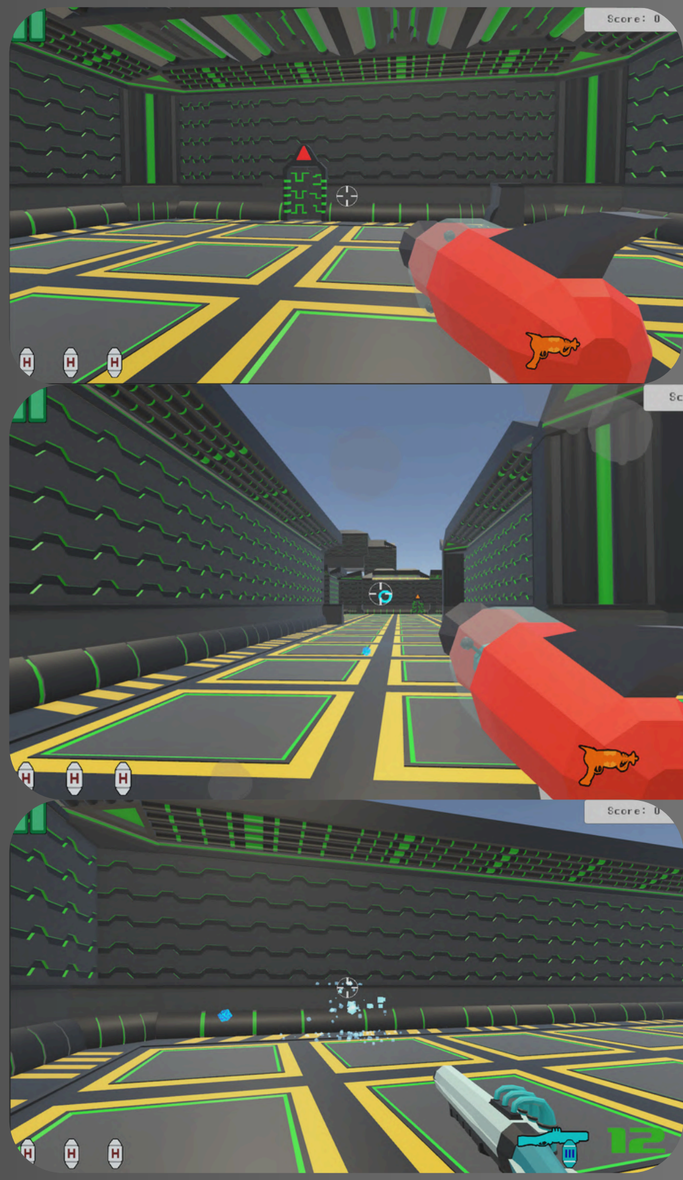
```
/// <summary>
/// Description:
/// Cycles through the available guns starting from the currently equipped gun
/// and moving in the direction of the mouse scroll input.
/// </summary>
void CycleEquippedGun()
{
    float cycleInput = cycleWeaponInput.ReadValue<Vector2>().y;
    List<Gun> availableGuns = guns.Where(item => item.available == true).ToList();
    int maximumAvailableGunIndex = availableGuns.Count - 1;
    int equippedAvailableGunIndex = availableGuns.IndexOf(guns[equippedGunIndex]);
    if (cycleInput < 0)
    {
        equippedAvailableGunIndex += 1;
        if (equippedAvailableGunIndex > maximumAvailableGunIndex)
        {
            equippedAvailableGunIndex = 0;
        }
    }
    else if (cycleInput > 0)
    {
        equippedAvailableGunIndex -= 1;
        if (equippedAvailableGunIndex < 0)
        {
            equippedAvailableGunIndex = maximumAvailableGunIndex;
        }
    }

    EquipGun(guns.IndexOf(availableGuns[equippedAvailableGunIndex]));
}
```

Players can switch between different weapons conveniently, using methods such as the mouse wheel or keyboard shortcuts (can configurable by players).

# Core implementation logic of the code

```
/// <summary>
/// Description:
/// Fires the gun, creating both the projectile and fire effect
/// </summary>
public void Fire()
{
    bool canFire = false;

    if (gunAnimator != null)
    {
        canFire = gunAnimator.GetCurrentAnimatorStateInfo(0).IsName(idleAnimationName);
    }
    else
    {
        canFire = ableToFireAgainTime <= Time.time;
    }

    if (canFire && HasAmmo())
    {
        if (projectileGameObject != null)
        {
            for (int i = 0; i < maximumToFire; i++)
            {
                float fireDegreeX = Random.Range(-maximumSpreadDegree, maximumSpreadDegree);
                float fireDegreeY = Random.Range(-maximumSpreadDegree, maximumSpreadDegree);
                Vector3 fireRotationInEular = fireLocationTransform.rotation.eulerAngles + new Vector3(fireDegreeX, fireDegreeY, 0);
                GameObject projectile = Instantiate(projectileGameObject, fireLocationTransform.position,
                    Quaternion.Euler(fireRotationInEular), null);
                if (childProjectileToFireLocation)
                {
                    projectile.transform.SetParent(fireLocationTransform);
                }
            }
        }

        if (fireEffect != null)
        {
            Instantiate(fireEffect, fireLocationTransform.position, fireLocationTransform.rotation, fireLocationTransform);
        }

        ableToFireAgainTime = Time.time + fireDelay;
        PlayShootAnimation();

        GunSmokeHandler.OnGunFire(this);

        if (useAmmo)
        {
            AmmoTracker.OnFire(this);
            roundsLoaded = Mathf.Clamp(roundsLoaded - 1, 0, magazineSize);
        }
    }
    else if (useAmmo && mustReload && roundsLoaded == 0)
    {
        StartCoroutine(Reload());
    }
    GameManager.UpdateUIElements();
}
```

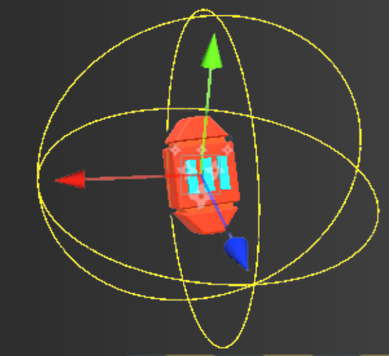
This is the core method for weapon firing, with its logic unfolding around four steps: "Check if firing is allowed → Execute firing → Post-firing processing → No-ammo handling". Additionally, it synchronizes the update of in-game UI displays such as ammo count and weapon status through method calls, ensuring players see weapon data in real time.

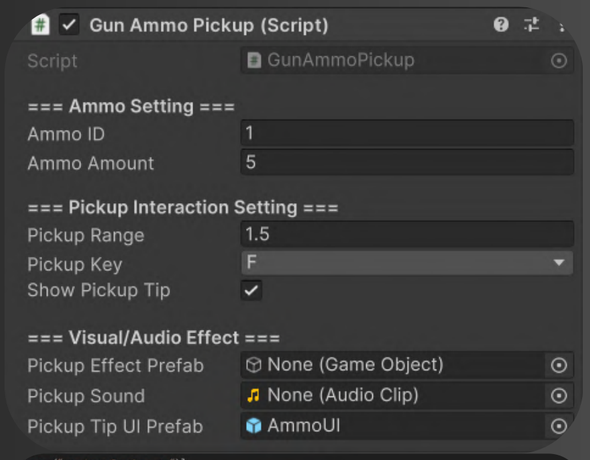



In-game actual demonstration screen

# Ammunition Mechanic

In level design, the ammo capacity of the player's primary weapon and some powerful weapons can be limited based on enemy difficulty tiers (e.g., regular mobs, elite bosses) or wave counts. This design balances resource supply and combat pressure, prompting players to find a strategic balance between ammo management and breaking through enemy encirclements, thereby infusing the game with moderate challenge — and this "strategic sense of challenge" is one of the core traits of high-quality games.



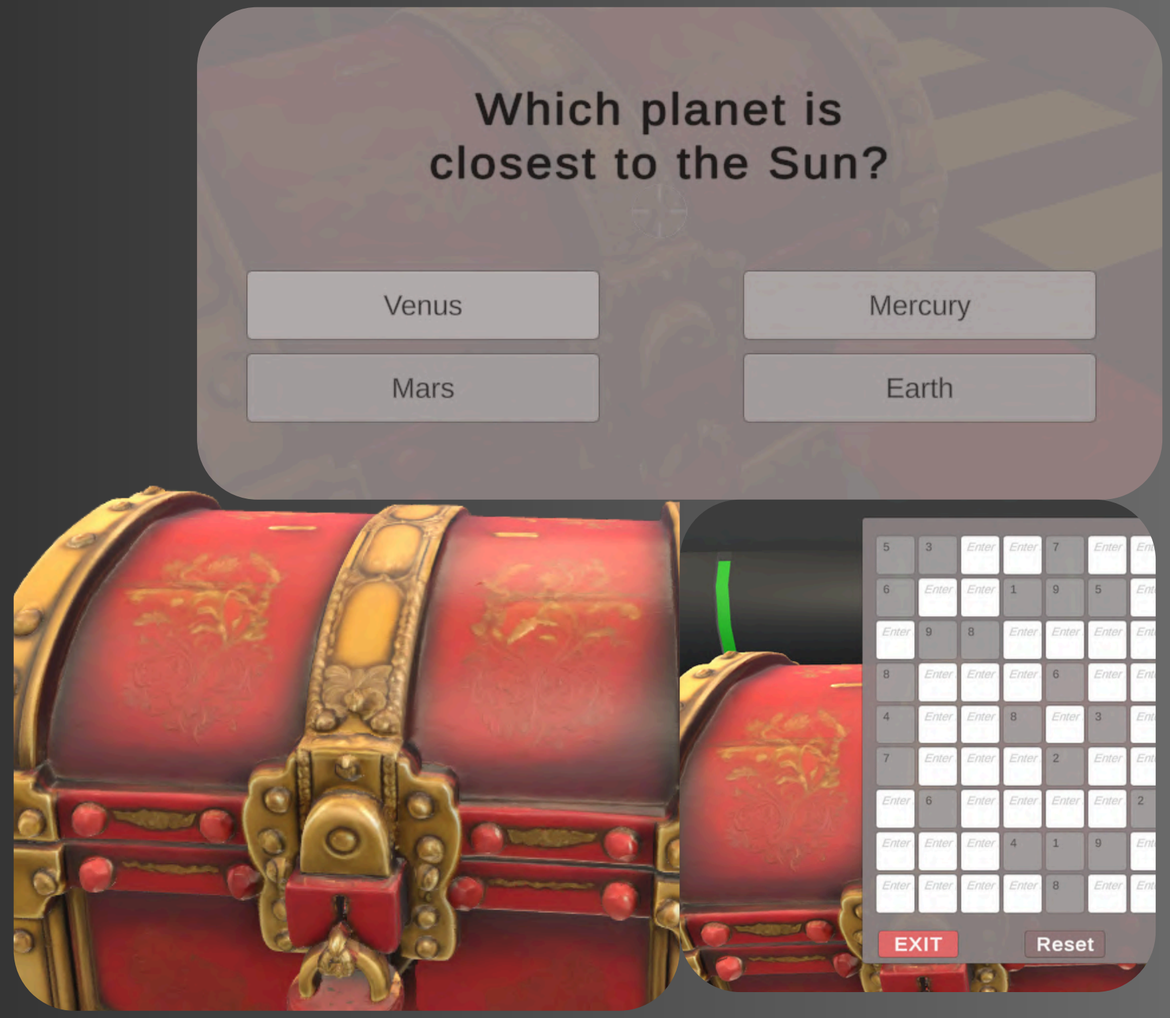

```
[Header("=== Ammo Setting ===")]
[Tooltip("AmmoID (ammunitionID need to align with Gun ID) ")]
public int ammoID = 0;

[Tooltip("The amount of ammunition replenished after pickup")]
public int ammoAmount = 20;

[Header("=== Pickup Interaction Setting ===")]
[Tooltip("Maximum pickup distance between the player and collectibles (Unit: meters, recommended 1.5-2)")]
public float pickupRange = 1.8f;

[Tooltip("The key to trigger pickup (default: F key, customizable based on operation habits)")]
public KeyCode pickupKey = KeyCode.F;
```

Set public vars in Unity, edit directly in UI, easy for team collaboration

```
/// <summary>
/// Initialize the player reference (found via tag)
/// </summary>
private void InitPlayerReference()
{
    GameObject playerObj = GameObject.FindGameObjectWithTag("Player");
    if (playerObj != null)
    {
        _playerTransform = playerObj.transform;
        Debug.Log($"ItemPickup[{gameObject.name}] find the player successfully: {playerObj.name}", this);
    }
    else
    {
        Debug.LogError($"ItemPickup[{gameObject.name}] can not find 'Player' tag", this);
        enabled = false;
    }
}

/// <summary>
/// Generate the pick-up prompt UI (generate only once and follow the pick-up object's position)
/// </summary>
private void SpawnPickupTipUI()
{
    if (pickupTipUIPrefab == null) return;

    // Generate the UI (ensure the UI follows the movement of the pick-up object)
    _spawnedTipUI = Instantiate(pickupTipUIPrefab, transform.position + Vector3.up, Quaternion.identity, transform);

    _spawnedTipUI.SetActive(false);
}

/// <summary>
/// Detect the distance between the player and the pick-up object, and update the "whether within range" state
/// </summary>
private void CheckPlayerDistance()
{
    // Calculate the straight-line distance between the player and the pick-up object
    float distanceToPlayer = Vector3.Distance(transform.position, _playerTransform.position);

    // When the player enters the range: Show the prompt UI
    if (distanceToPlayer <= pickupRange && !_isPlayerInRange)
    {
        _isPlayerInRange = true;
        ShowPickupTip();
    }
    // When the player enters the range: Close the prompt UI
    else if (distanceToPlayer > pickupRange && _isPlayerInRange)
    {
        _isPlayerInRange = false;
        HidePickupTip();
    }
}
/// <summary>
/// Detect the pick-up key(F) input and trigger the item pick-up
/// </summary>
private void CheckPickupInput()
{
    if (_isPlayerInRange && !_hasBeenPicked && Input.GetKeyDown(pickupKey))
    {
        ExecutePickup();

        if (_isPlayerInRange && !_hasBeenPicked)
        {
            if (Input.GetKeyDown(pickupKey))
            {
                Debug.Log($"[ItemPickup{gameObject.name}] Detected the F key press, ready to execute the pick-up!", this);
                ExecutePickup();
            }
        }
    }
}

/// <summary>
/// Execute the pick-up logic (take ammunition pick-up as an example)
/// </summary>
private void ExecutePickup()
{
    // Mark as picked up to prevent repeated triggering of the pick-up logic
    _hasBeenPicked = true;
    HidePickupTip();

    // Call AmmoTracker(related script) to replenish ammunition (reuse the existing system interface)
    AmmoTracker.AddAmmunition(ammoID, ammoAmount);
    Debug.Log($"ItemPickup[{gameObject.name}]: pick up successfully! ID={ammoID}, number={ammoAmount}", this);

    // Play pick-up effect (sound effects and particle effects if having)
    PlayPickupFeedback();

    Destroy(gameObject, 0.8f);
}
```

# Treasure Chest System

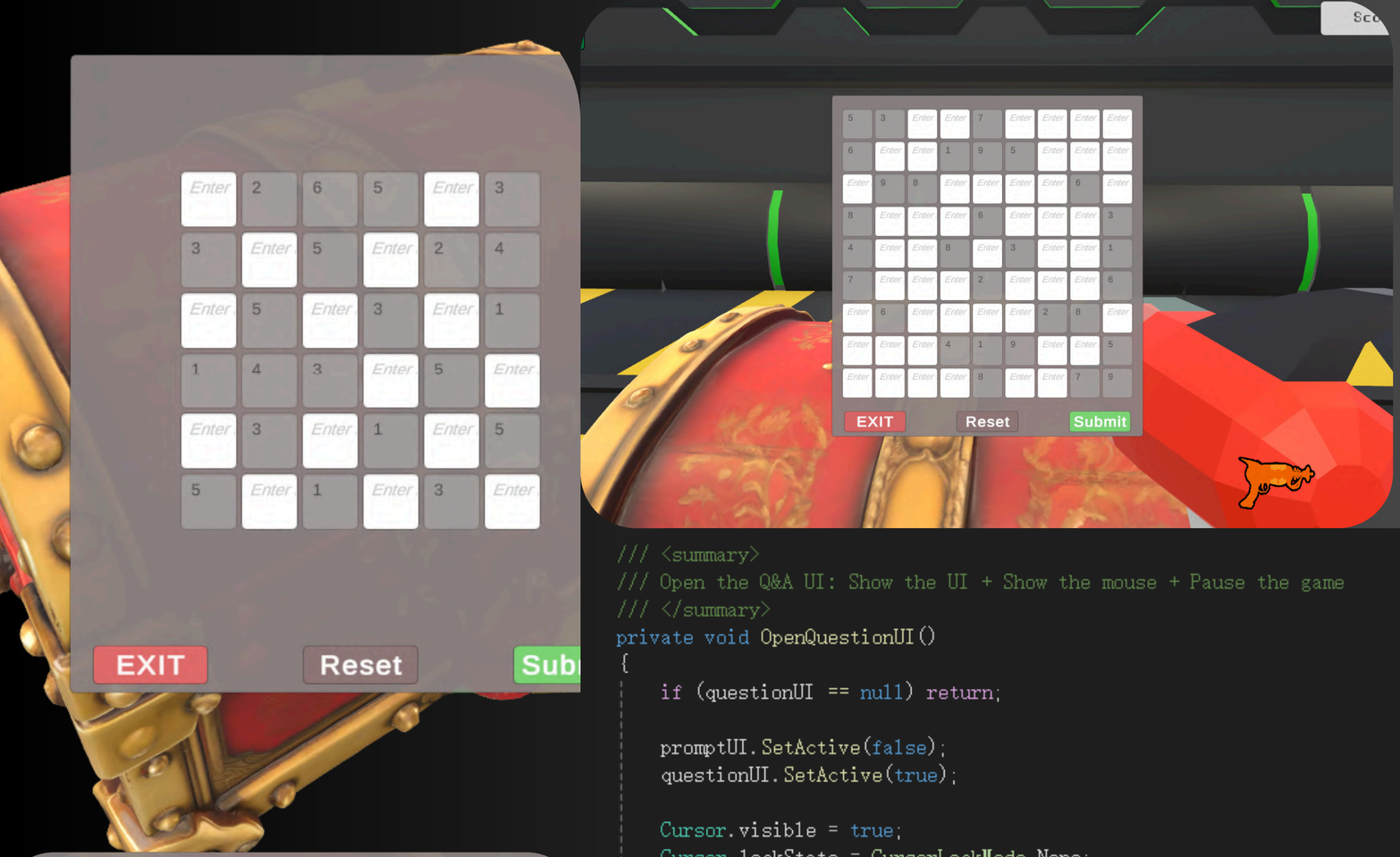
In the game, it has designed a treasure chest mechanism. Players can answer simple trivia questions or complete number-based puzzle mini-games (such as Sudoku) to open the chests and claim the in-game items inside. These items can be used to progress through the main storyline or unlock additional game content like hidden levels.

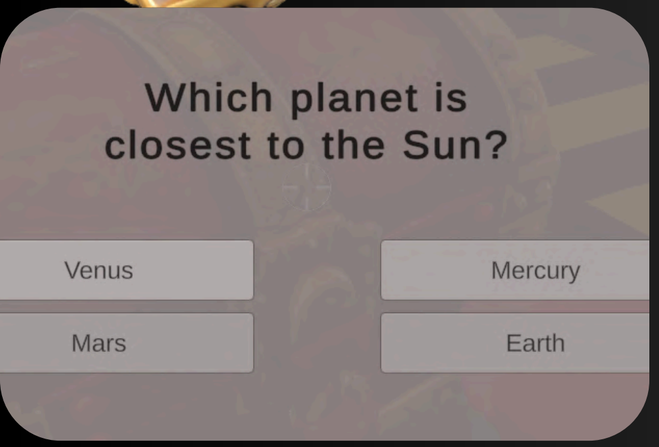


# Treasure Chest Interaction

This is a treasure chest interaction feature. When the player moves the character near the treasure chest and enters the trigger area, a "press F" prompt text will automatically appear on the screen, instructing the player to press the F key to open the treasure chest.If the player presses the F key, the treasure chest will open, and corresponding UI interfaces (such as an item acquisition pop-up, a chest details panel, etc.) will then be displayed.


```
/// <summary>
/// When the player enters the Trigger range, display the prompt "Press F"
/// </summary>
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player") && !isChestOpened)
    {
        isPlayerInRange = true;
        promptUI.SetActive(true);
    }
}

/// <summary>
/// When the player leaves the Trigger range, not display the prompt "Press F"
/// </summary>
private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        isPlayerInRange = false;
        promptUI.SetActive(false);
    }
}
```




```
/// <summary>
/// Open the Q&A UI: Show the UI + Show the mouse + Pause the game
/// </summary>
private void OpenQuestionUI()
{
    if (questionUI == null) return;

    promptUI.SetActive(false);
    questionUI.SetActive(true);

    Cursor.visible = true;
    Cursor.lockState = CursorLockMode.None;
    // Pause the game (prevent the player from moving while answering questions)
    if (pauseGameWhenShowUI) Time.timeScale = 0;

    // Transmit the current treasure chest reference to the Q&A manager (for triggering rewards later)
    QuestionManager questionManager = questionUI.GetComponent<QuestionManager>();
    if (questionManager != null)
    {
        questionManager.SetTargetChest(this.gameObject);
    }
    else
    {
        Debug.LogError($"【Error】{questionUI.name} The QuestionManager script is not attached! ");
    }
```

After the player presses the F key, the treasure chest opens and a question-answering UI interface pops up. The player answers the questions in the interface, and if they answer correctly, they will receive corresponding in-game rewards (such as items, gold coins)

```
void GetAllTMPCells()
{
    TMP_InputField[] allCells = gridParent.GetComponentsInChildren<TMP_InputField>();
    int index = 0;

    for (int row = 0; row < gridSize; row++)
    {
        for (int col = 0; col < gridSize; col++)
        {
            if (index < allCells.Length)
            {
                sudokuCells[row, col] = allCells[index];

                SudokuCellEvent oldEvent = sudokuCells[row, col].GetComponent<SudokuCellEvent>();
                if (oldEvent != null) DestroyImmediate(oldEvent);

                sudokuCells[row, col].contentType = TMP_InputField.ContentType.IntegerNumber;
                sudokuCells[row, col].characterLimit = 1;

                Transform textArea = sudokuCells[row, col].transform.Find("Text Area");
                if (textArea != null)
                {
                    TMP_Text inputText = textArea.GetComponent<TMP_Text>();
                    if (inputText != null)
                    {
                        sudokuCells[row, col].textComponent = inputText;
                        inputText.color = Color.black;
                        inputText.fontSize = 24;
                        inputText.alignment = TextAlignmentOptions.Center;
                    }
                }

                SudokuCellEvent cellEvent = sudokuCells[row, col].gameObject.AddComponent<SudokuCellEvent>();
                cellEvent.Init(row, col, OnCellSelected, OnCellPointerExit);

                sudokuCells[row, col].onEndEdit.AddListener(text =>
                {
                    if (text == "0") sudokuCells[row, col].text = "";
                });

                index++;
            }
        }
    }
}
```

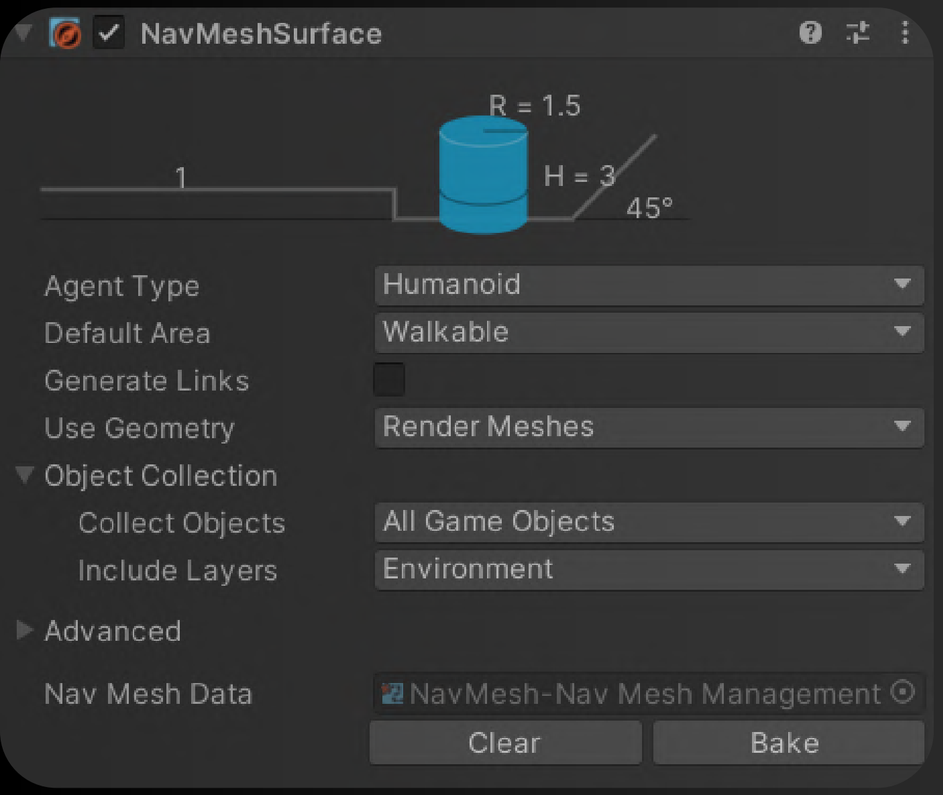
Get all Sudoku input cells From UI canvas and perform initial configuration on the cells, like Initializing the style, interaction logic, and event binding for each input box to ensure compliance with Sudoku input rules.

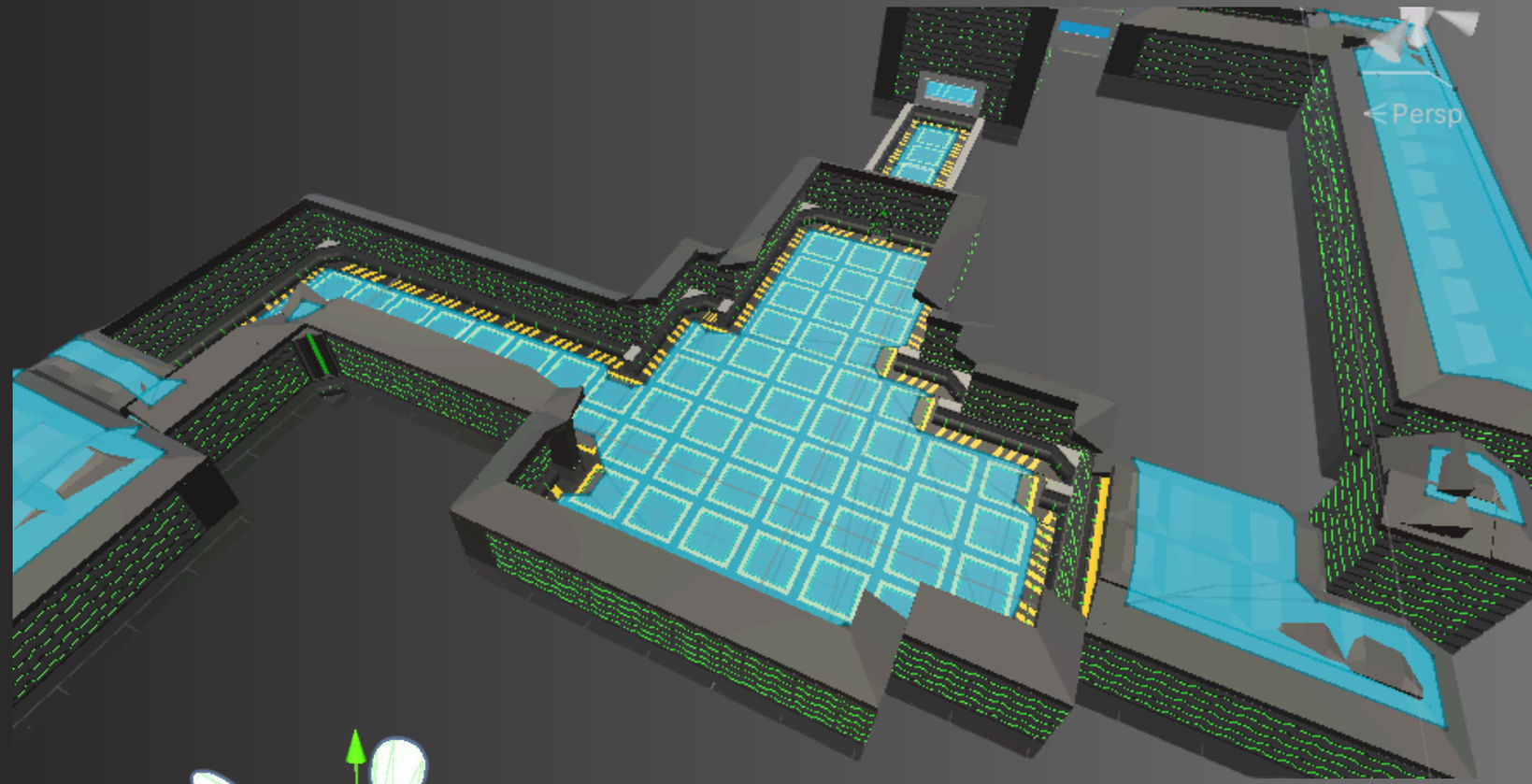
This is the core verification method for Sudoku answers, it checks the validity of the answer by iterating through all cells.

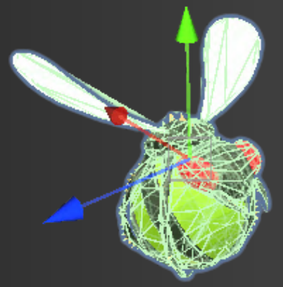
```
// Verify the Sudoku answer entered by the player
public void CheckPlayerAnswer()
{
    bool isAllCorrect = true;

    for (int row = 0; row < gridSize; row++)
    {
        for (int col = 0; col < gridSize; col++)
        {
            if (string.IsNullOrEmpty(sudokuCells[row, col].text))
            {
                isAllCorrect = false;
                sudokuCells[row, col].image.color = emptyHintColor;
            }
            else if (int.TryParse(sudokuCells[row, col].text, out int playerInput))
            {
                // Verify the range of the input number (1-9)
                if (playerInput < 1 || playerInput > gridSize)
                {
                    isAllCorrect = false;
                    sudokuCells[row, col].image.color = wrongColor;
                }
                // Verify if the current cell's value violates the "no duplicates in subgrid" rule
                else if (!IsCellInBoxValid(row, col, playerInput))
                {
                    isAllCorrect = false;
                    sudokuCells[row, col].image.color = wrongColor;
                }
                // Verify if the player's input equals the correct answer
                else if (playerInput != correctAnswer[row, col])
                {
                    isAllCorrect = false;
                    sudokuCells[row, col].image.color = wrongColor;
                }
                else
                {
                    sudokuCells[row, col].image.color = correctColor;
                }
            }
            else
            {
                isAllCorrect = false;
                sudokuCells[row, col].image.color = wrongColor;
            }
        }
    }
```

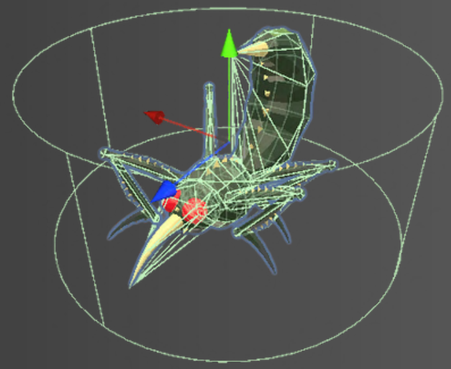
```
    if (isAllCorrect)
    {
        isSudokuSolved = true;
        OnSudokuSolved?.Invoke();
        Debug.Log($"[{gameObject.name}] successfully! ");

        // Call the treasure chest opening method
        if (targetChest != null)
        {
            ChestInteractSu chestInteract = targetChest.GetComponent<ChestInteractSu>();
            if (chestInteract != null)
            {
                chestInteract.OpenChestAndGiveReward();
            }
            else
            {
                Debug.LogError($"[{targetChest.name}] The ChestInteractSu script is not attached! ");
            }
        }
        else
        {
            Debug.LogError($"targetChest is not assigned!");
        }

        // Resume the game time that was paused to prevent the player from moving while answering questions
        Time.timeScale = 1;

        Invoke("ExitSudokuUI", 1.0f);
    }
    else
    {
        Debug.Log($"[{gameObject.name}] The answer is incorrect, please check!");
    }
}
```

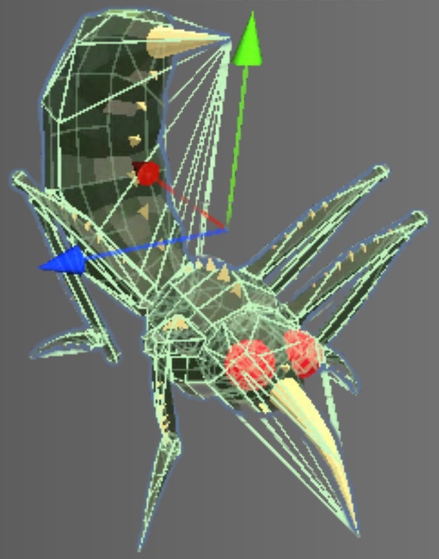
# Enemy



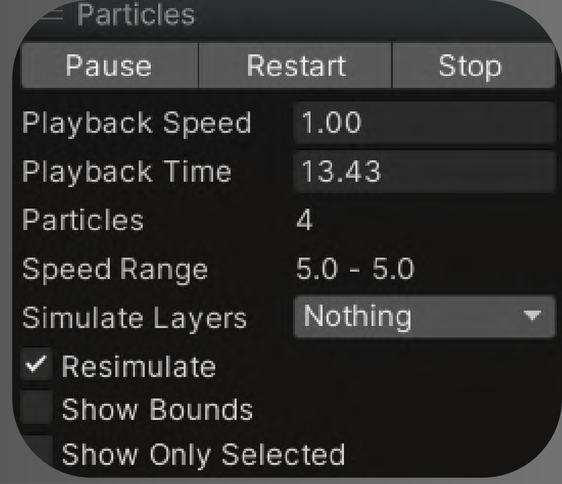




Multiple differentiated enemy types have been designed around level gameplay to adapt to various combat scenarios; meanwhile, a navigation system has been built using Unity's NavMesh AI functionality. A navigation mesh covering the passable areas of the level is baked, and then corresponding navigation parameters are configured for different enemies, enabling them to autonomously bypass obstacles, track targets, and enhance the sense of combat interaction.

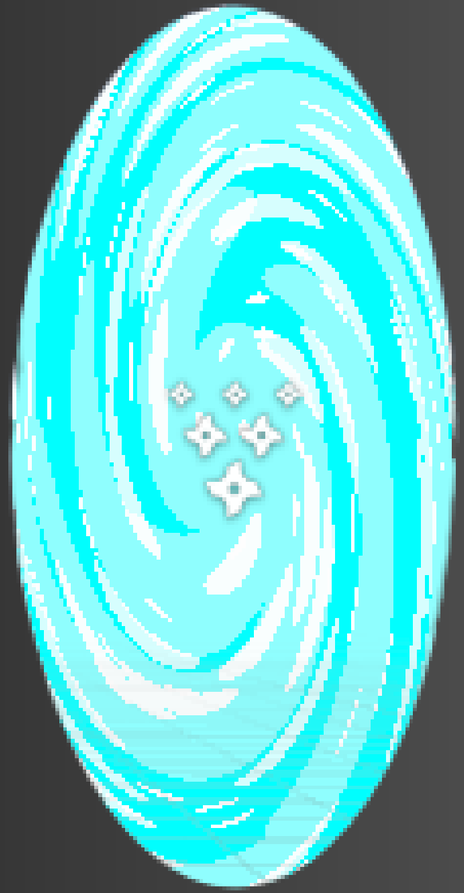
# Other mini prop settings

Particle effects to make props look more realistic




## Teleportal

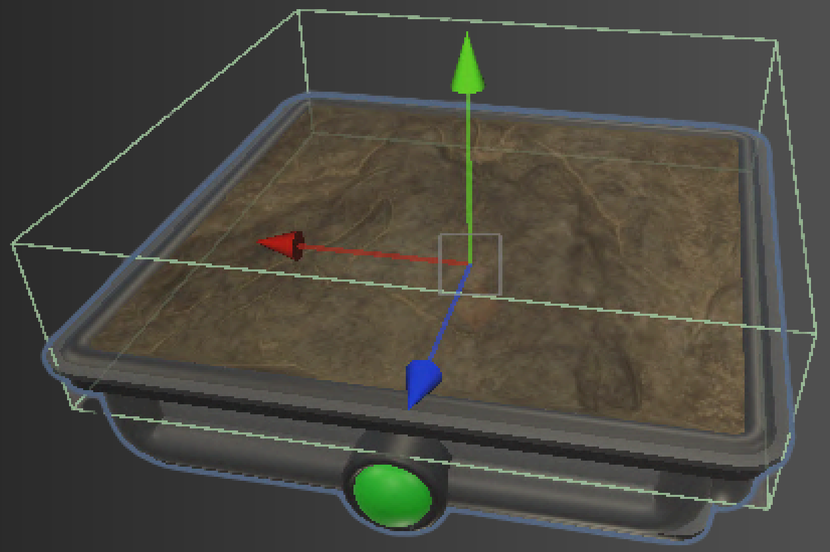
As a core scene interaction prop, the teleportal enables players to quickly traval between different game scenes. It not only eliminates the redundant operation of repeated travel and improves exploration efficiency, but also serves as a "hidden link" for story narration — for example, using "one-way portals" to guide players to unlock areas in the order of the plot. This way, scene switching is no longer a simple map jump, but an important part of driving the story rhythm and deepening the sense of immersion.

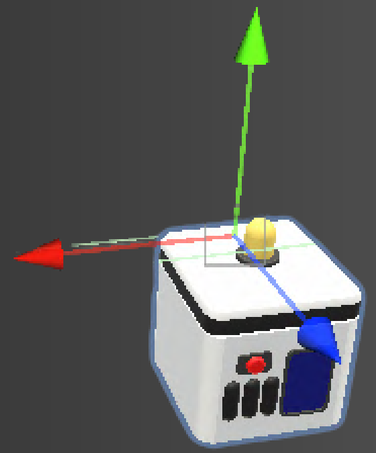


```
public class Teleporter : MonoBehaviour
{
    [Header("Setting")]
    [Tooltip("The destination portal object and can be directly assigned")]
    public Teleporter destinationTeleport;
    [Tooltip("Visual or audio effect")]
    public GameObject teleportEffect;

    // This variance tracks the state of teleporter, and prove player do not be teleported teleport2 immediately.
    private bool teleAvailable = true;

    // If another collider enters the trigger, call OnTriggerEnter
    private void OnTriggerEnter(Collider other)
    {
        // if tag is player, then act
        if (other.tag == "Player" && teleAvailable && destinationTeleport != null)
        {
            Debug.Log(other.name + " collider with " + this.transform.name +
                " transport to " + destinationTeleport.transform.name);

            Instantiate(teleportEffect, transform.position, transform.rotation, null);

            // Turn off the destinationTeleporter and prevent the player being teleported back immediately.
            destinationTeleport.teleAvailable = false;

            // To prevent players controlling character movement during teleportation, player control is disabled.
            CharacterController characterController = other.gameObject.GetComponent<CharacterController>();
            if(characterController != null)
            {
                characterController.enabled = false;
            }

            // reposition the player
            other.transform.position = destinationTeleport.transform.position;

            //
            if(characterController != null)
            {
                characterController.enabled = true;
            }
        }
    }

    // OnTriggerExit is called when another collider stops making contact with the trigger.
    private void OnTriggerExit(Collider other)
    {
        if (other.tag == "Player")
        {
            teleAvailable = true;
        }
    }
```

# Other mini prop settings

## Moving platform
## &
## Respawn Point





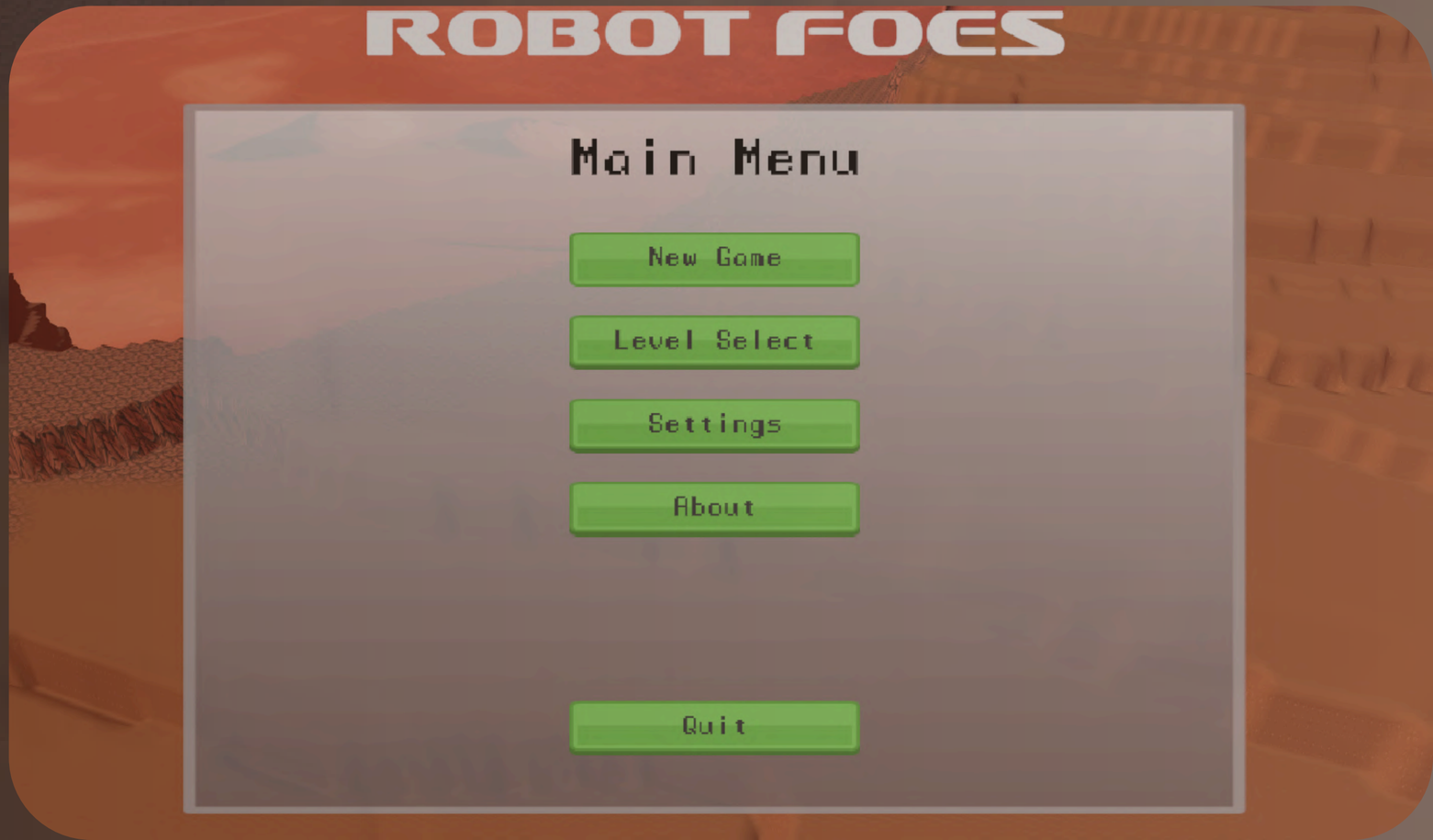
```
/// <summary>
/// When the player comes into contact with it, the respawn point automatically records the player's game progress.
/// </summary>
private void OnTriggerEnter(Collider collision)
{
    if (collision.tag == "Player" && collision.gameObject.GetComponent<Health>() != null)
    {
        Health playerHealth = collision.gameObject.GetComponent<Health>();
        playerHealth.SetRespawnPoint(respawnLocation.position); // public Transform respawnLocation;  public void SetRespawnPoint(Vector3 newRespawnPosition)

        // Reset the last respawn point if it exists
        if (CheckpointTracker.currentCheckpoint != null)
        {
            if (CheckpointTracker.currentCheckpoint.checkpointAnimator != null)
            {
                CheckpointTracker.currentCheckpoint.checkpointAnimator.SetBool(animatorActiveParameter, false);
            }
        }

        if (CheckpointTracker.currentCheckpoint != this && checkpointActivationEffect != null)
        {
            Instantiate(checkpointActivationEffect, transform.position, Quaternion.identity, null);
        }

        // Set current checkpoint to this and set up its animation
        CheckpointTracker.currentCheckpoint = this;
        if (checkpointAnimator != null)
        {
            checkpointAnimator.SetBool(animatorActiveParameter, true);
        }
    }
```

```
private void Waiting()
{
    timeWaiting += Time.deltaTime;
    if (timeWaiting >= waitTime)
    {
        timeWaiting = 0.0f;
        isWaiting = false;
    }
}

private void Moving()
{
    if (moveDirectionB)
    {
        percentMoved += Time.deltaTime * moveSpeed;
        if (percentMoved >= 1)
        {
            isWaiting = true;
            moveDirectionB = false;
        }
    }
    else
    {
        percentMoved -= Time.deltaTime * moveSpeed;
        if (percentMoved <= 0)
        {
            isWaiting = true;
            moveDirectionB = true;
        }
    }
    transform.position = Vector3.Lerp(wayPonitA.position, wayPointB.position, percentMoved);
```

There are additional content that enrich the game. If you wish, you can experience more of the game's content and mechanics directly on [junjieg.itch.io/3d-fps](junjieg.itch.io/3d-fps)

# Game Development with C# and Unity

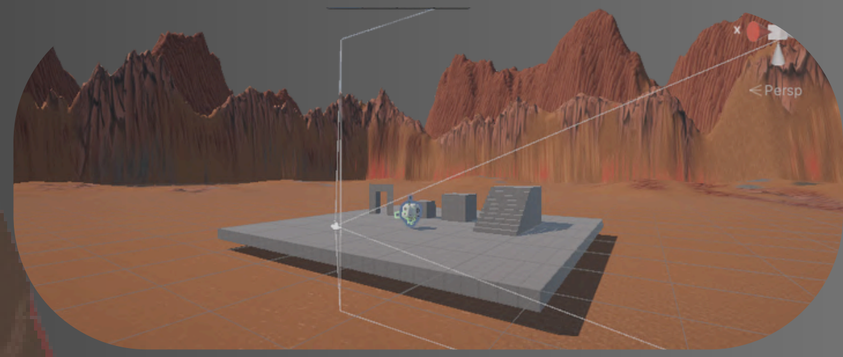
## 3D Platform

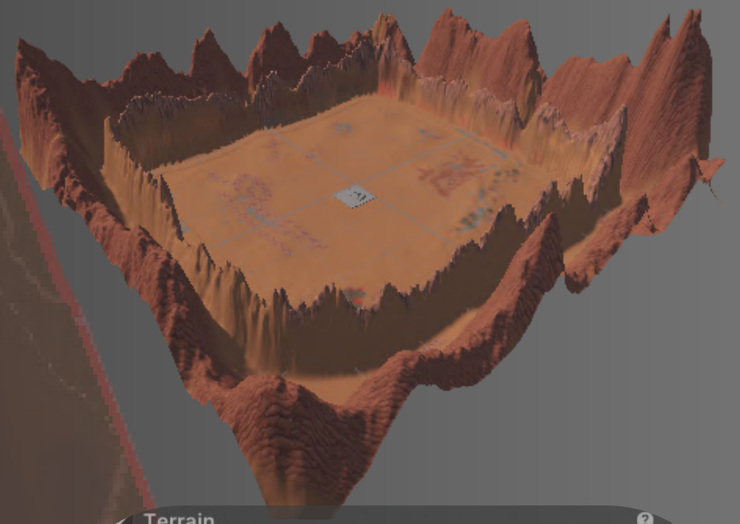


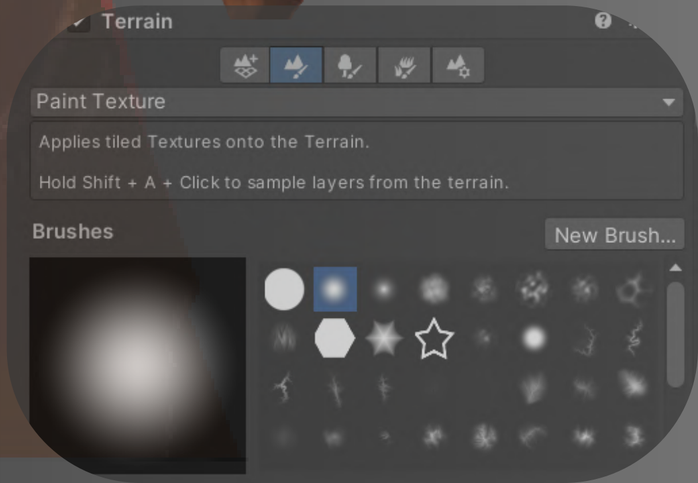

The game has been uploaded to junjieg.itch.io/3d-platform. You can download the installation package for Windows/MacOS (for a better experience)

# Terrain Construction

In the scene construction of 3D platform games, the Terrain tool is used to build the basic terrain framework. Then, style-appropriate materials and textures are added to the terrain, enabling the terrain to have both an interactive physical form and scene-matching visual texture, which further enhances the game's immersive atmosphere.

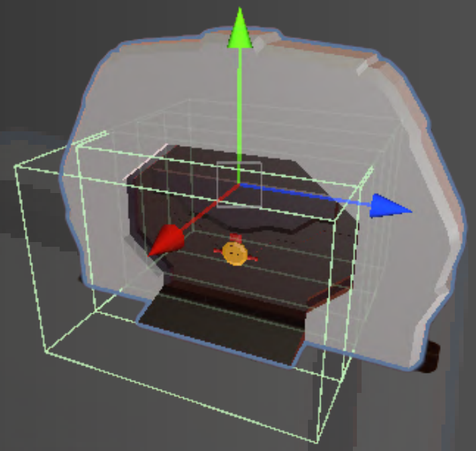


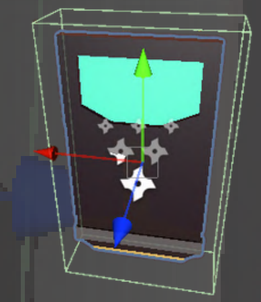



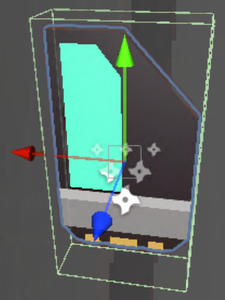


# Door and Key

Doors and their corresponding keys can easily divide spaces: they not only naturally split game scenes into functional zones like safe areas, danger zones, and puzzle zones, but also connect the plot through key acquisition and use. When players search for the key, they may trigger side quests. This makes space division more than just a physical barrier—it becomes a core mechanism for driving gameplay and the story.



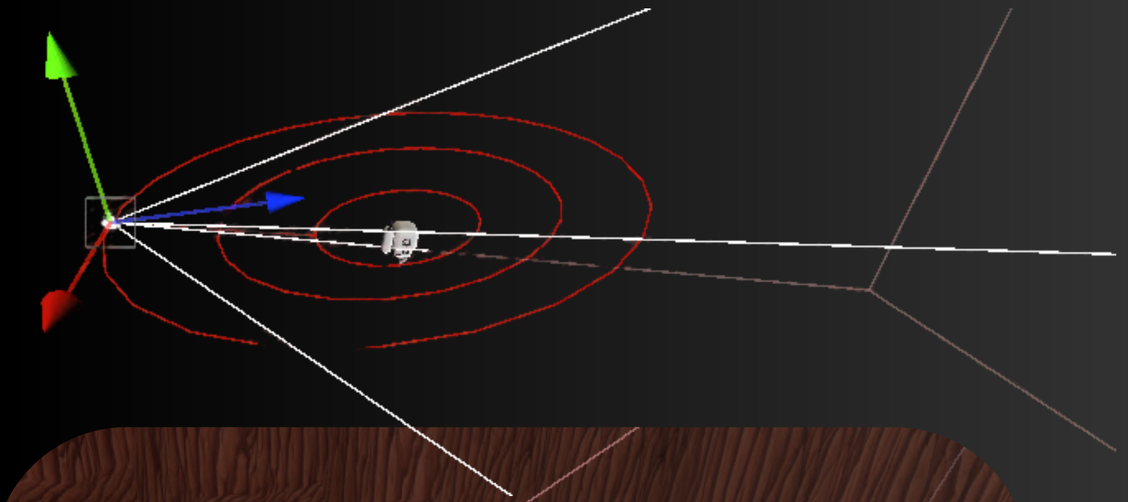


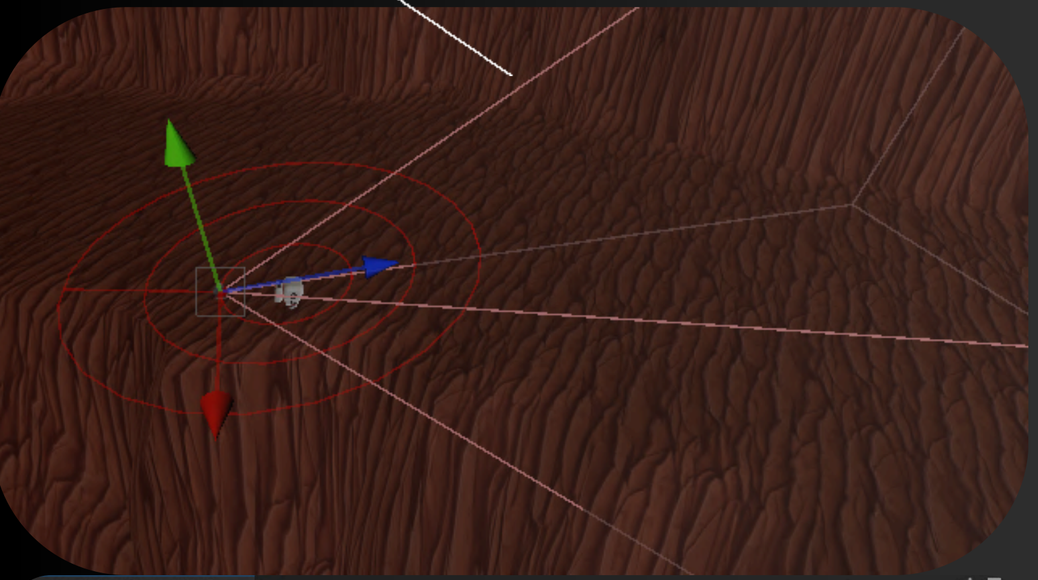


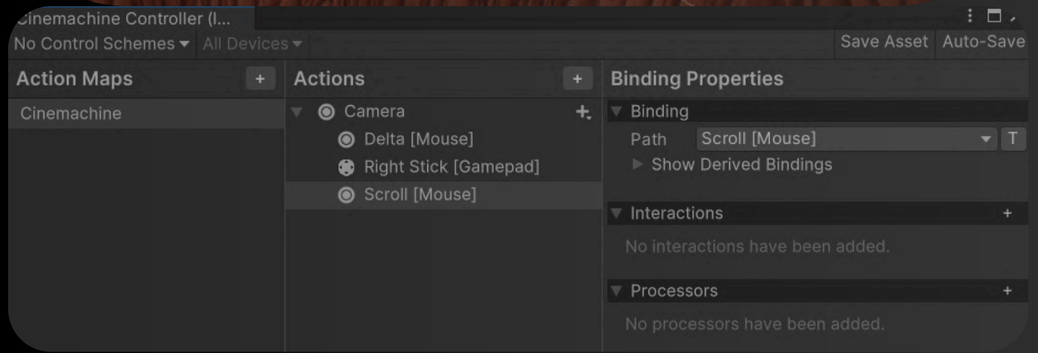
```
// core functional logic (key verification)
/// <summary>
/// first verify the key, then execute the door-opening logic.
/// </summary>
public void TryOpenDoor()
{
    // if the key verification passes and the door is not in an open state → execute door opening
    if (KeyManager.PlayerHasMatchingKey(this) && !isOpen) // call the PlayerHasMatchingKey method in the KeyManager class
    {
        OpenDoor();
    }
    // if key verification fails → play the locked effect to inform the player
    else if (doorLockedEffect != null && !isOpen)
    {
        PlayEffect(doorLockedEffect);
    }
}

/// <summary>
/// door opening logic: update state, trigger animator events, play effects
/// </summary>
protected virtual void OpenDoor()
{
    isOpen = true;
    onOpenEvent.Invoke(); // trigger the door opening animation
    PlayEffect(doorSwitchEffect);
}

/// <summary>
/// door closing logic
/// </summary>
protected virtual void CloseDoor()
{
    if (isOpen)
    {
        isOpen = false;
        onCloseEvent.Invoke(); // trigger the door closing animation
        PlayEffect(doorSwitchEffect);
    }
}
```

```
/// <summary>
/// manage the keys held by the player and provide add/verify/clear functions
/// </summary>
public static class KeyManager
{
    // The IDs of the keys held by the player
    private static HashSet<int> _playerKeys = new HashSet<int>() {};

    public static void AddKeyToPlayer(int keyID)
    {
        if (keyID != 0) // 0 is the id for unlocked doors and no key is required
        {
            _playerKeys.Add(keyID);
        }
    }

    /// <summary>
    /// verify if the player holds the key corresponding to the door
    /// </summary>
    /// <param name="targetDoor">the door to be verified</param>
    public static bool PlayerHasMatchingKey(DoorManager targetDoor)
    {
        if (targetDoor.doorID == 0)
        {
            return true;
        }
        // if the door id is not 0 → check if the player holds the corresponding key
        return _playerKeys.Contains(targetDoor.doorID);
    }

    /// <summary>
    /// clear all keys held by the player (e.g., for scene reset)
    /// </summary>
    public static void ClearAllPlayerKeys()
    {
        _playerKeys.Clear();
    }
}
```

# Camera Setup








In 3D platform games, camera setup is undoubtedly a core element that influences the player experience, and it determines operational smoothness and scene immersion. An ideal camera must not only fully display the full-body movements of the player character — whether it's the stretching of limbs during a jump or the agile turning when dodging obstacles, ensuring the player can clearly judge the positional relationship between the character and the platforms; It also needs to reserve a reasonable view margin. This allows the player to catch a glimpse of upcoming traps, interactive items, or hidden paths in advance, avoiding sudden mistakes caused by limited visibility.

In this project, it used the Cinemachine system throughout to build the camera logic: it implemented 360° surrounding follow for the character via a FreeLook camera, and combined it with the "Composer" function of the Body module to automatically select the character and key scene elements. This not only solves the problem of blind spots in the field of view of traditional fixed lenses but also ensures a smooth transition of the lens without shaking when the character moves at high speed or crosses terrain. At the same time, for special scenarios (such as narrow passages and high-altitude platforms), an additional Virtual Camera preset is configured. The camera automatically adjusts the view width through priority switching — for example, zooming in to focus on the character's landing spot in dense platform areas, and pulling back to show the overall environment in open scenes. This makes the camera more than just a "view window"; it becomes an invisible and useful tool that guides player exploration.

# Enemy Spawner Settings

The enemy spawner with a "Player Range Trigger" mechanism: it continuously detects the distance between the player and itself. When the player enters the preset trigger range, it will spawn enemies within the designated spawn area according to the configured rules (fixed interval, random interval, or controlled call). If no player is detected or the player moves out of the range, it pauses spawning to avoid unnecessary resource consumption.

```
/// <summary>
/// Tests whether the player is in the range and it is time to spawn an enemy
/// </summary>
private void TestSpawn()
{
    if (Time.timeSinceLevelLoad > nextSpawnTime && IsPlayerInDetectRange())
    {
        Spawn();
    }
}

/// <summary>
/// Spawns an enemy if the prefab exists, and updates the next spawn time
/// </summary>
public void Spawn()
{
    if (prefab != null)
    {
        switch (spawnMethod)  // public enum SpawnMethod { Fixed, Random, Controlled }
        {
            case SpawnMethod.Fixed:
                nextSpawnTime = Time.timeSinceLevelLoad + spawnRate;
                break;
            case SpawnMethod.Random:
                nextSpawnTime = Time.timeSinceLevelLoad + spawnRate * UnityEngine.Random.value;
                break;
            case SpawnMethod.Controlled:
                nextSpawnTime = Mathf.Infinity;
                break;
        }
        Vector3 spawnLocation = GetSpawnLocation();
        GameObject instance = GameObject.Instantiate(prefab, spawnLocation, Quaternion.identity, null);
    }
}
/// <summary>
/// Determine and return the location at which to spawn an enemy
/// </summary>
public Vector3 GetSpawnLocation()
{
    Vector3 result = Vector3.zero;
    result.x = transform.position.x + UnityEngine.Random.Range(-spawnAreaSize.x * 0.5f, spawnAreaSize.x * 0.5f);
    result.y = transform.position.y + UnityEngine.Random.Range(-spawnAreaSize.y * 0.5f, spawnAreaSize.y * 0.5f);
    result.z = transform.position.z + UnityEngine.Random.Range(-spawnAreaSize.z * 0.5f, spawnAreaSize.z * 0.5f);
    return result;
}

// Find the player object in the scene (via tag)
private void FindPlayer()
{
    GameObject player = GameObject.FindGameObjectWithTag(playerTag);
    if (player != null)
    {
        playerTransform = player.transform;
    }
    else
    {
        Debug.LogWarning($"Enemy Spawner[{gameObject.name}]: can not find 「{playerTag}」Object! Check the tag setting。", this);
    }
}

// Check if the player is within the detection range
private bool IsPlayerInDetectRange()
{
    // If the "Player Trigger" function is not enable, no spawning restrictions
    if (!enablePlayerTrigger)
    {
        return true;
    }
    // Calculate the distance between the spawner and the player
    float distanceToPlayer = Vector3.Distance(transform.position, playerTransform.position);
    return distanceToPlayer <= playerDetectRadius;
}
```

# Enemy Awareness

The core function of Enemy Awareness is to use two mechanisms, Vision and Hearing, to real-time determine the enemy's perception level of the target player. This then drives the enemy's tracking or return behavior, making the enemy's reactions more in line with realistic logic.

The enemy's certainty (ranging from 0 to 1) regarding the target's position and movement. A value of 0 means no perception at all, while 1 means complete certainty of the target's position. Trigger thresholds: When the certainty exceeds followThreshold (e.g., 0.2), the enemy starts tracking; when it exceeds detectionThreshold (e.g., 0.5), the target is determined to be "detected".

```
/// <summary>
/// The current position the enemy needs to track
/// If certainty more than follow threshold: Track the player position
/// If certainty less than follow threshold: Return to the initial position or stop moving if no initial position
/// </summary>
public Vector3 FollowPosition
{
    get
    {
        if (playerCertainty > followThreshold)
        {
            return expectedPlayerPosition;
        }
        else
        {
            return returnPosition != null ? returnPosition.position : transform.position;
        }
    }
}

/// <summary>
/// Whether the enemy has detected the player
/// Judgment: Certainty > detection threshold & no line of sight obstruction
/// </summary>
public bool HasFoundPlayer
{
    get
    {
        return (playerCertainty > detectThreshold) && HasLineOfSightToPlayer();
    }
}
/// <summary>
/// Core function: Detect the player's state and update the certainty level and expected position
/// Process: Calculate visual/auditory increments → Update certainty level → Update expected position
/// </summary>
public void CheckForPlayer()
{
    // Calculate the vision and hearing certainty increments
    float visionIncrement = CalculateVisionCertaintyIncrement();
    float hearingIncrement = CalculateHearingCertaintyIncrement();

    // Calculate the total increment: add if there is perception, minus if there is no perception
    float totalIncrement = CalculateTotalCertaintyIncrement(visionIncrement, hearingIncrement);

    // Update the certainty level
    UpdatePlayerCertainty(totalIncrement);

    // Determine whether to lock the player's position based on the certainty level
    UpdateExpectedPlayerPosition();
}
/// <summary>
/// Calculate the vision certainty increment
/// Able to see the player (no obstruction) & the player is within the sight angle & the player is within the sight distance
/// </summary>
private float CalculateVisionCertaintyIncrement()
{
    bool isInSightAngle = IsPlayerInSightAngle();
    bool isInSightRange = IsPlayerInSightRange();
    bool hasLineOfSight = HasLineOfSightToPlayer();

    // If all visual conditions are met, return the increment (the closer the player is, the larger the increment)
    if (isInSightAngle && isInSightRange && hasLineOfSight)
    {
        float distanceRatio = GetDistanceToPlayer() / maxSightDistance;
        return distanceRatio * Time.deltaTime;
    }

    // If the conditions are not met, no visual increment
    return 0;
}
```

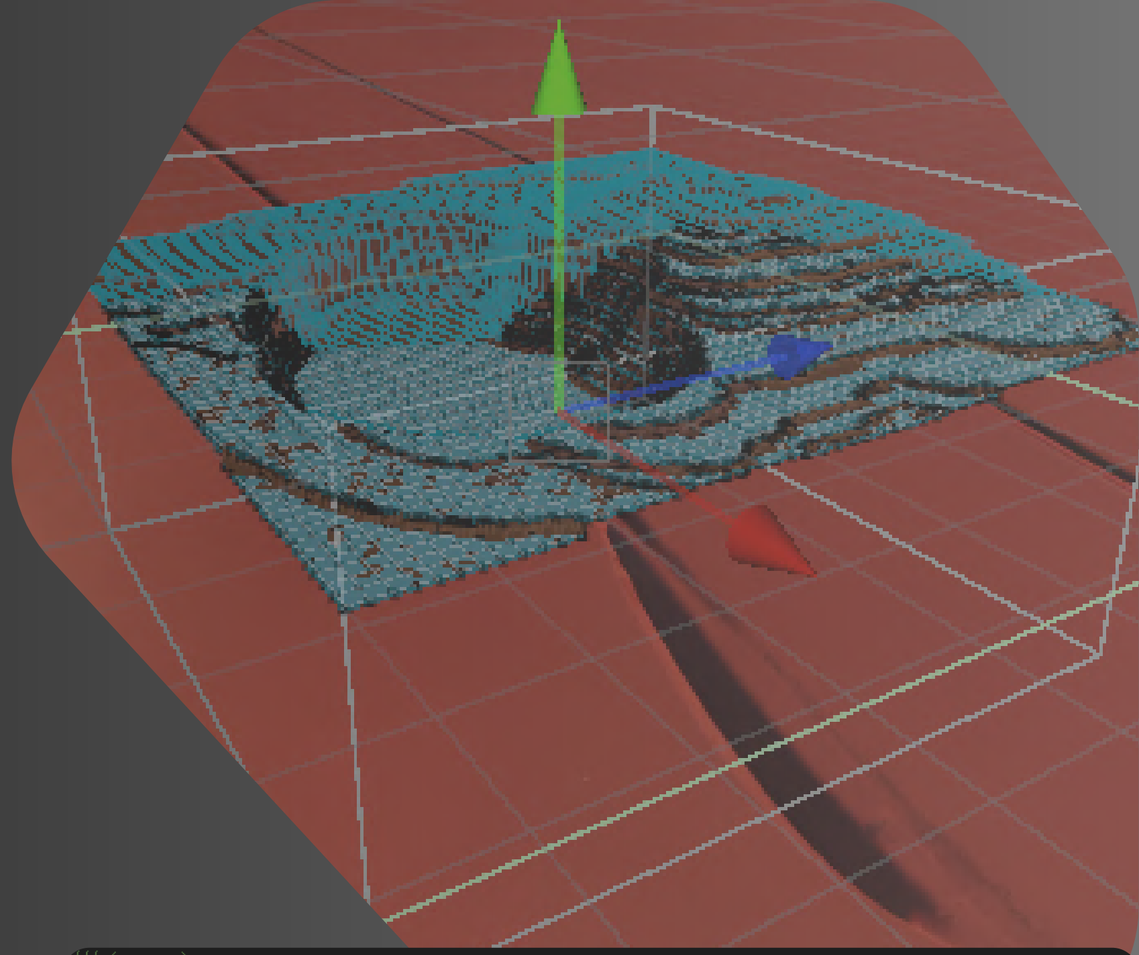
# Enemy Awareness

Moreover, Perception decay rate can detect that If the enemy does not continuously perceive the target, the certainty will decrease over time to prevent the enemy from "permanently remembering" the target's position. After losing the target, the enemy needs to return to its "initial position" (e.g., patrol start point) or they will stay in place. Additionally, the vision detection is set with a collision layer to specify which objects (e.g., walls, obstacles) will block the enemy's line of sight, preventing the enemy from "seeing through walls".

```
/// <summary>
/// Calculate the hearing certainty increment
/// The player is within hearing distance + the player's movement speed more than minimum audible speed
/// </summary>
private float CalculateHearingCertaintyIncrement()
{
    // If the player is outside the hearing distance, return 0
    if (!IsPlayerInHearingRange())
    {
        return 0;
    }

    // Calculate the player's movement speed and avoid time difference of 0
    float timeSinceLastHeard = Time.timeSinceLevelLoad - lastHeardTime;
    timeSinceLastHeard = Mathf.Max(timeSinceLastHeard, 0.01f); // set the minimum time difference to 0.01 seconds
    float playerSpeed = (target.position - lastHeardPosition).magnitude / timeSinceLastHeard;

    // Sound occlusion attenuation: Sound remains normal if there is no line of sight obstruction, it is attenuated by 75% if there is an obstruction
    float soundAttenuation = HasLineOfSightToPlayer() ? 1.0f : 0.25f;

    // Update the last hearing record (prepare for the next speed calculation)
    UpdateLastHeardRecord();

    // If the speed more than min speed to be detect, return the hearing increment
    if (playerSpeed > minSpeedToBeHeard)
    {
        float distanceRatio = maxHearingDistance / GetDistanceToPlayer();
        return playerSpeed * soundAttenuation * distanceRatio * Time.deltaTime;
    }

    return 0;
}
/// <summary>
/// Calculate the total certainty increment (add if there is perception, minus if there is no perception)
/// </summary>
private float CalculateTotalCertaintyIncrement(float visionInc, float hearingInc)
{
    float totalPerceptInc = visionInc + hearingInc;
    return totalPerceptInc > 0 ? totalPerceptInc : -certaintyDecayRate * Time.deltaTime;
}

/// <summary>
/// Update the player certainty (clamp between 0 and 1)
/// </summary>
private void UpdatePlayerCertainty(float increment)
{
    playerCertainty = Mathf.Clamp(playerCertainty + increment, 0.0f, 1.0f);
}

/// <summary>
/// Update the player's expected position (lock the player's current position when the certainty more than threshold)
/// </summary>
private void UpdateExpectedPlayerPosition()
{
    if (playerCertainty > followThreshold && target != null)
    {
        expectedPlayerPosition = target.position;
    }
}

/// <summary>
/// Determine if the enemy can see the player (no line of sight obstruction)
/// </summary>
public bool HasLineOfSightToPlayer()
{
    if (target == null)
    {
        return false;
    }

    // use a line of sight ray
    Vector3 directionToPlayer = target.position - transform.position;
    Ray sightRay = new Ray(transform.position, directionToPlayer);

    if (Physics.Raycast(sightRay, out RaycastHit hitInfo, maxSightDistance, sightBlockingLayers))
    {
        // If the ray hits the player, it is considered to have line of sight
        return hitInfo.transform == target || target.IsChildOf(hitInfo.transform);
    }

    return true;
}
```

# Drop Reset Zone

Considering that players may accidentally fall outside the map boundaries during exploration, it has added a Drop Reset Zone mechanism: this hidden zone will automatically catch players who are falling indefinitely. By immediately triggering drop reset feedback and resetting the scene, it prevents players from being trapped in an unplayable trouble and ensures the smooth flow of the game.



```
/// <summary>
/// This function deals damage to a health component if the collided with gameobject has a health component attached
/// </summary>
private void DealDamage(Collision collision)
{
    Health collidedHealth = collision.gameObject.GetComponentInParent<Health>();
    if (collidedHealth == null && collision.rigidbody != null)
    {
        collidedHealth = collision.rigidbody.GetComponent<Health>();
    }
    if (collidedHealth != null)
    {
        if (collidedHealth.teamId != this.teamId)
        {
            collidedHealth.TakeDamage(damageAmount);
            if (destroyAfterDamage)
            {
                Destroy(this.gameObject);
            }
        }
    }
}
```

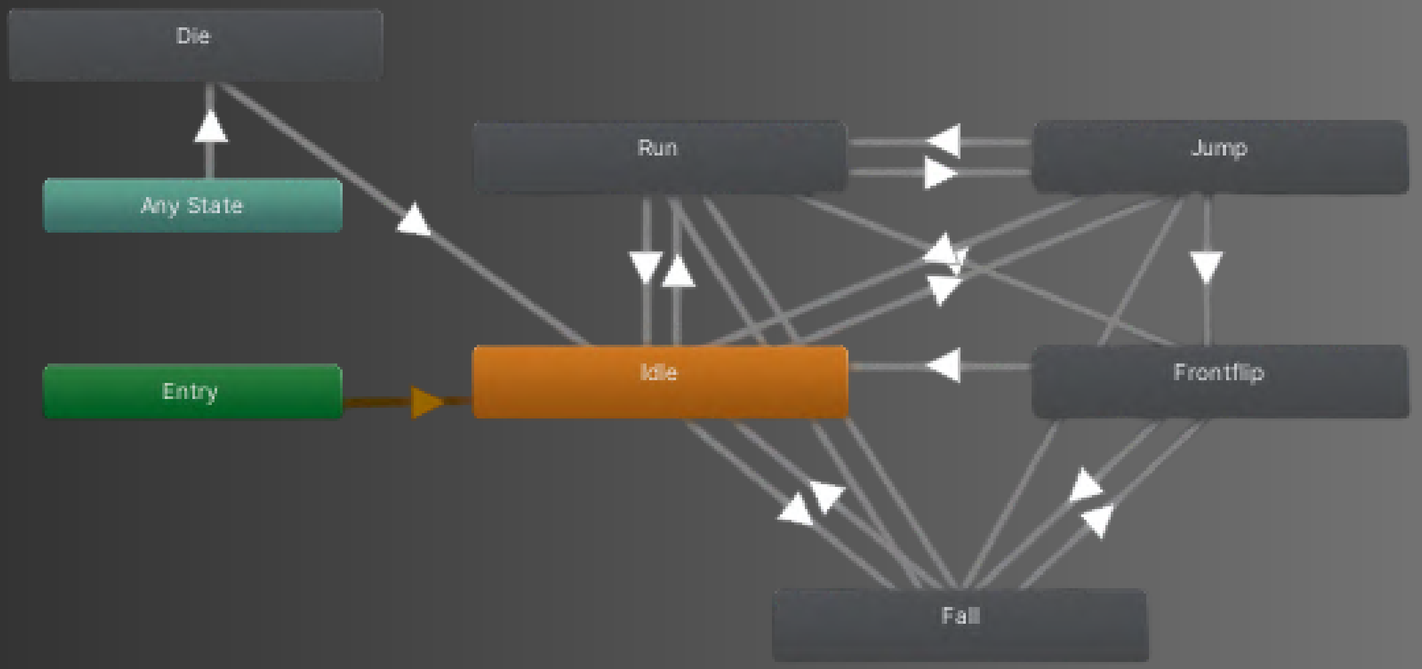
# Random drops

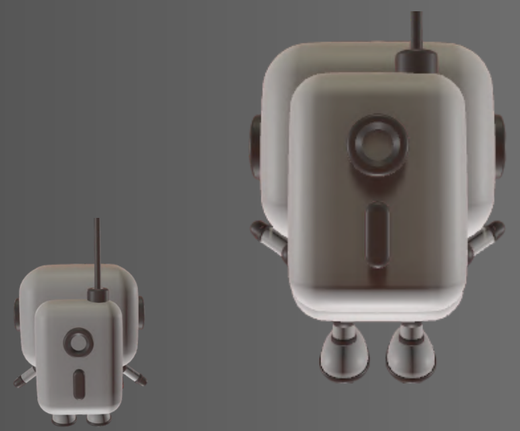
This Random Drops for randomly spawning dropped items when an enemy is destroyed.A variety of dropped items not only meets players' practical needs but also enriches their exploration experience. Rare items obtained by chance bring unexpected surprises, making every action of defeating enemies or destroying objects full of anticipation.

```
/// <summary>
/// Random spawning of drops
/// </summary>
public void SpawnRandomDrops()
{
    // Exit if the configuration is invalid to avoid null reference errors
    if (!ValidateDropConfig() || _dropProbPairList == null) //  private List<(GameObject prefab, float probability)> _dropProbPairList;
    {
        Debug.LogWarning("The drop configuration is invalid, drops cannot be spawned", this);
        return;
    }

    // Loop to spawn the specified number of drops(number of drops default is 1)
    for (int i = 0; i < numberOfDrops; i++)
    {
        GameObject selectedPrefab = null;
        // Iterate through the drops from highest to lowest probability and determine whether to spawn
        foreach (var (prefab, prob) in _dropProbPairList)
        {
            // Generate a random value between 0 and 1, and select the drop if the probability meets the requirement
            if (prob >= Random.value)
            {
                selectedPrefab = prefab;
                break;
            }
        }

        if (selectedPrefab != null)
        {
            Instantiate(selectedPrefab, transform.position, transform.rotation, null);
        }
    }
}
/// <summary>
/// Verify if the drop is valid
/// </summary>
private bool ValidateDropConfig()
{
    // The length of the prefab list and the probability list must be equal
    if (dropPrefabs.Count != dropProbabilities.Count)
    {
        Debug.LogError($"The number of drop prefab ({dropPrefabs.Count}) do not equal probability ({dropProbabilities.Count})", this);
        return false;
    }

    if (dropPrefabs.Count == 0)
    {
        Debug.LogError("no drop prefab! ", this);
        return false;
    }

    // Check if the probability is within the range of 0 to 1
    for (int i = 0; i < dropProbabilities.Count; i++)
    {
        if (dropProbabilities[i] < 0 || dropProbabilities[i] > 1)
        {
            Debug.LogWarning($"The probability of 「{dropPrefabs[i].name}」 ({dropProbabilities[i]}) exceeds the range of 0 to 1, it will be automatically adjusted", this);

            // Automatically adjust probability
            dropProbabilities[i] = Mathf.Clamp(dropProbabilities[i], 0, 1);
        }
    }

    return true;
}
```

# Animation Setup

In 3D platform games, character animations are extremely important. This is because they not only make the character's movements (such as jumping, running, and falling) more in line with physical laws, allowing players to get a realistic sense of feedback during operations, but also endow virtual characters with greater "vitality" and bridge the gap between players and the game world.



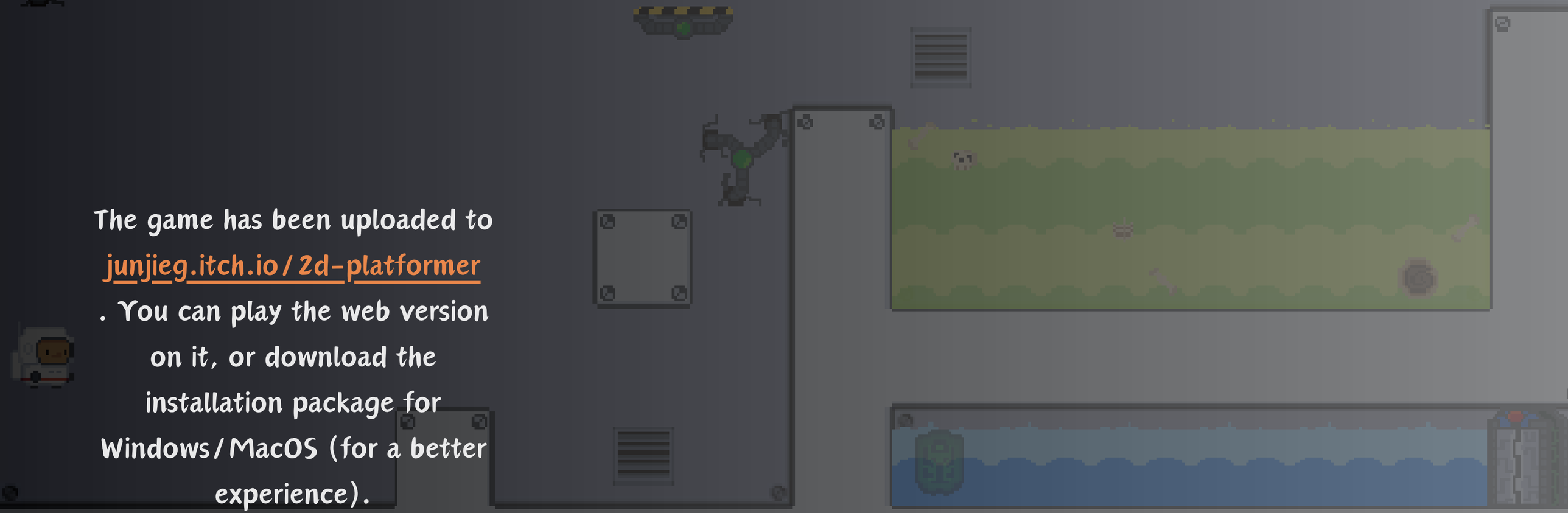



# 3D Platform Game

For a deeper dive, you can find the Windows and MacOS installers for the game directly via this link: junjieg.itch.io/3d-platform

# Game Development with C# and Unity

## 2D Platform Mini Game



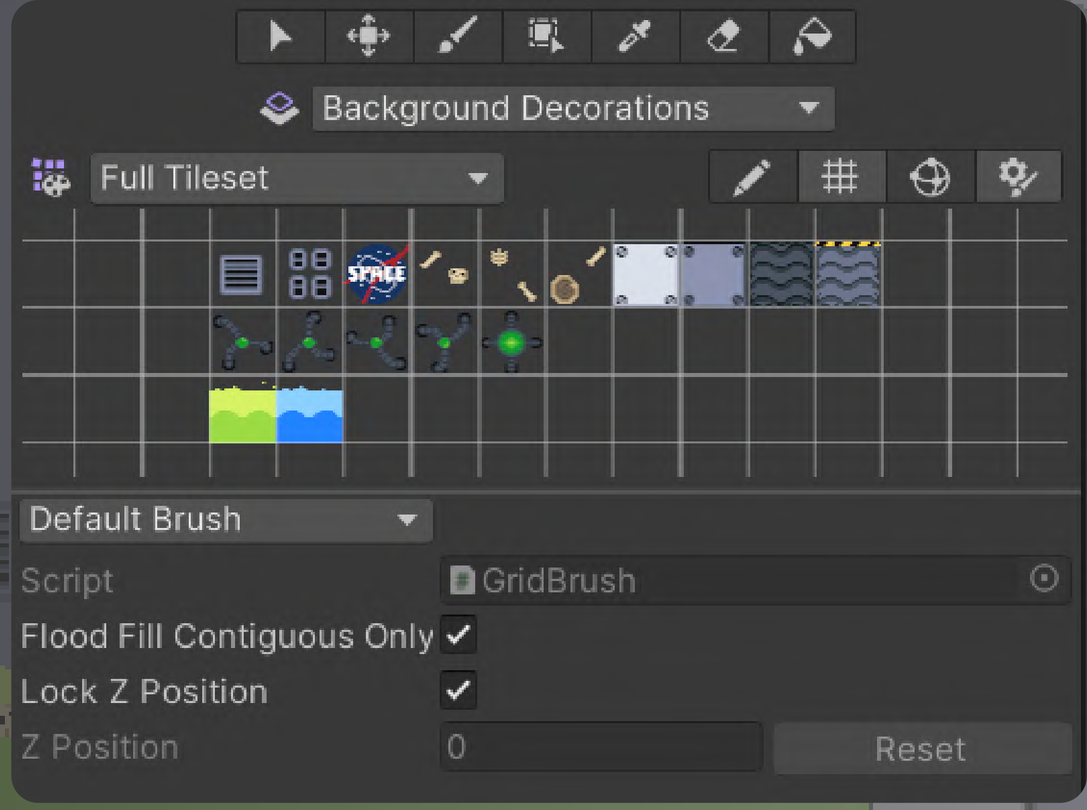
The game has been uploaded to junjieg.itch.io/2d-platformer. You can play the web version on it, or download the installation package for Windows/MacOS (for a better experience).

# Background

This is a 2D platformer: the player is accidentally trapped in an abandoned alien space station, and must overcome the obstacles and challenges of numerous levels to finally reach the end and claim victory.

# Tile Palette

In game level development, using the Tile Palette tool to build scenes, it can split terrain blocks, obstacles, decorative elements, and other components into reusable "tile assets." This eliminates the need for repeated drawing or modeling, significantly reducing the time required to build a single level. Additionally, by unifying the style of assets, it allows for the easy batch creation of multiple level scenes that are consistent in style and rich in details.



# Ground Checking

In 2D platform games that centered around jumping, ground checking is of vital importance as it directly determines the "effectiveness" of the jump: only when the character is detected on the ground can the jump be triggered, which not only prevents "infinite jumps in the air" from disrupting the balance but also makes the jump feedback more in line with physical logic.

```
/// <summary>
/// Checks whether there is a collider overlapping the checking collider which is on a "ground" layer.
/// </summary>
public bool CheckGrounded()
{
    if (groundCheckCollider == null)
    {
        GetCollider();
    }

    // Find the colliders that overlap this one
    Collider2D[] overlaps = new Collider2D[6];
    ContactFilter2D contactFilter = new ContactFilter2D();
    contactFilter.layerMask = groundLayers;
    groundCheckCollider.OverlapCollider(contactFilter, overlaps);

    // Check if one of the overlapping colliders is on the "ground" layer
    foreach (Collider2D overlapCollider in overlaps)
    {
        if (overlapCollider != null)
        {
            // Use bitwise operations to determine whether the layer of the detected collider is within the ground layers
            int match = contactFilter.layerMask.value & (int)Mathf.Pow(2, overlapCollider.gameObject.layer);
            if (match > 0)
            {
                if (landingEffect && !groundedLastCheck)
                {
                    Instantiate(landingEffect, transform.position, Quaternion.identity, null);
                }
                groundedLastCheck = true;
                return true;
            }
        }
    }
    groundedLastCheck = false;
    return false;
}
/// <summary>
/// Attempts to setup the collider to be used with ground checking,
/// </summary>
public void GetCollider()
{
    if (groundCheckCollider == null)
    {
        groundCheckCollider = gameObject.GetComponent<Collider2D>();
    }
}
```

# ShowHide mechanism

This is a small but fun mechanism design in the game. Centered on player triggering, it realizes the switching between visible and hidden plots, enriching the details of level exploration.

```
/// <summary>
/// Controls 2D tile's transparency: Hides by default, reveals when player enters trigger.
/// </summary>
public class FloatingTile : MonoBehaviour
{
    [Header("FloatingTileSetting")]
    [Tooltip("Alpha value (0 = fully transparent, 1 = fully opaque)")]
    public float revealAlpha = 1f;
    public float hiddenAlpha = 0f;

    // Reference to the SpriteRenderer component (control color/transparency)
    private SpriteRenderer _spriteRenderer;

    /// <summary>
    /// Initialize: Get SpriteRenderer and set tile to hidden state on start.
    /// </summary>
    private void Start()
    {
        _spriteRenderer = GetComponent<SpriteRenderer>();
        SetTileAlpha(hiddenAlpha);
    }

    /// <summary>
    /// Triggered when another 2D collider enters this object's trigger.
    /// Reveals tile if the collider belongs to Player.
    /// </summary>
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Player")
        {
            SetTileAlpha(revealAlpha);
        }
    }

    /// <summary>
    /// Hides tile if Player exits.
    /// </summary>
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Player")
        {
            SetTileAlpha(hiddenAlpha);
        }
    }

    /// <summary>
    /// Updates the tile's transparency.
    /// </summary>
    private void SetTileAlpha(float alpha)
    {
        if (_spriteRenderer != null)
        {
            Color newColor = _spriteRenderer.color;
            newColor.a = alpha;
            _spriteRenderer.color = newColor;
        }
    }
}
```

# 2D Platform Game

This game has been released on
junjieg.itch.io/2d-platformer
For more game content, please
visit this website.