**基于Unity的虚拟场景交互与游戏开发DogFight**

**一、项目概述**

**1.目标——总结本次作业总体实现目标**

本次项目的总体目标是开发一个以“**第三人称战斗模拟**”为核心的游戏，通过Unity游戏引擎的功能实现丰富的玩法和系统，构建完整的游戏流程，为玩家提供沉浸式的游戏体验。本项目的核心目标可以归纳为以下几点：

**（1）实现流畅的游戏机制和逻辑**

**玩家角色控制**：实现玩家角色在虚拟世界中的移动、攻击和交互功能，提供流畅的操作反馈。

**敌人AI**：设计具备不同行为模式的敌人，包括idle、追击、攻击等逻辑，通过状态机实现敌人行为动态变化。

**计时与胜负判定**：构建一个倒计时机制，设定游戏时间的限制，同时判断玩家的胜负条件，例如击败一定数量敌人或是否保持生存到游戏结束。

**游戏UI管理**：提供直观的用户界面，包含主菜单、暂停界面、玩家状态（血条、倒计时）、分数统计等，方便玩家理解游戏状态。

**（2）丰富的敌人和奖励机制**

实现多种敌人模型（红、绿、蓝三种），并为其分配不同的生成概率。

为敌人死亡后掉落不同功能的苹果奖励（如回血、增加攻击力、加速），并设计奖励效果的上限及叠加逻辑，提供策略性玩法。

**2.背景——游戏设计背景、为什么设计该游戏**

**（1）游戏设计背景**

游戏的设计背景基于以下几点：

**战斗与策略性游戏的流行**：现代玩家偏爱包含策略与战斗元素的游戏。通过设计一个小型竞技场环境，允许玩家通过有限资源管理与敌人进行对抗，是符合市场需求且易于实现的选择。

**游戏开发学习的需求：**本次游戏的设计不仅是一个作品，更是学习Unity开发技术的载体。我们通过设计游戏场景、玩家与敌人交互逻辑、状态控制、奖励机制等，全面熟悉了Unity的开发环境和组件。

**教学要求与个人兴趣结合：**结合课程内容要求和个人对游戏开发的兴趣，我们选择了一个具备一定挑战性但又不失趣味的战斗模拟游戏作为开发目标。

（2）为什么设计该游戏

本游戏的设计初衷来源于以下几方面的考虑：

【1】**提供创新与趣味性**

我们希望设计一个简单但富有趣味的游戏，既能够让玩家感受到成就感，也能够通过简单的机制带来策略性的选择。例如：

\* 玩家需要合理分配拾取奖励的优先级（如回血vs增强攻击力）。

\* 敌人的种类和生成概率增加了不可预见性，让每次游戏体验都独特。

\* 通过视觉反馈（如动态血条、倒计时动画）提升游戏的互动性和紧张感。

【2】**结合团队能力与时间限制**

考虑到团队的技术水平和项目开发的时间限制，选择开发一个小型但功能完整的战斗模拟游戏是合理的：

\* 游戏内容足够简单，可以在有限时间内完成。

\* 技术实现集中于核心功能，避免不必要的复杂性。

\* 项目规模适中，但包含的功能模块较为全面，有助于锻炼开发能力。

**二、项目开发环境与工具**

**2.1 开发环境**

操作系统：Windows 11

开发引擎：Unity 2022.3.34f1c1

编程语言：C#

开发工具：Visual Studio 2022

**2.2 开发工具与插件**

NavMesh：用于敌人寻路功能。

TextMeshPro：增强UI文本显示效果。

Animator：实现角色动画切换。

Cinemachine: 实现第三人称控制

**2.3 软件资源**

【1】**模型:**

己方角色模型：

[Dog Knight]

(https://assetstore.unity.com/packages/3d/characters/animals/dog-knight-pbr-polyart-135227)

敌方角色模型：

[Toon RTS Units]

(https://assetstore.unity.com/packages/3d/characters/toon-rts-units-demo-69687)

[Dungeon Skeletons Demo]

<https://assetstore.unity.com/packages/3d/characters/creatures/dungeon-skeletons-demo-71087>

**注：虽然项目要求敌人模型为这两个，但是考虑到这两个敌人模型没有受击动画和死亡动画，也考虑到人物与场景画风的一致性（这里我们选择的是像素风模型Dog Knight），最后我们选择将己方角色模型作为敌方，但是这里就有敌我不分的问题，解决方法写在后面。**

场景模型：

[Free Skybox Extended Shader]

<https://assetstore.unity.com/packages/vfx/shaders/free-skybox-extended-shader-107400>

[Prefab Painter 2]

<https://assetstore.unity.com/packages/tools/painting/prefab-painter-2-61331>

掉落物模型：

[Casual Vegetable Pack-Created With Fastmesh Asset]

<https://assetstore.unity.com/packages/3d/props/food/casual-vegetable-pack-created-with-fastmesh-asset-293783>

汉字资源：

<https://timothyqiu.itch.io/vonwaon-bitmap>

音乐资源：

Morning.mp3

Two Steps From Hell.mp3

**三、项目开发流程**

**3.1 项目需求分析**

**1. 用户交互：**

\* 玩家可以通过键盘控制角色移动。

\* 提供攻击功能，与敌人交战。

\* 支持游戏暂停与继续功能。

**2. 敌人行为：**

\* 敌人具有不同类型（红、绿、蓝），每种类型具有不同的生成概率。

\* 敌人在死亡后可以掉落不同颜色的苹果，提供回血或增强玩家能力的效果。

\* 初始竞技场有五个红色类型敌人，敌人死后随机生成另一类型敌人。

**3. 用户界面：**

\* 玩家UI显示血条和时间。

\* 敌人头顶显示血条。

\* 游戏结束时显示得分界面。

1. **背景音乐**

\* 玩家人物在不同的场景切换不同的背景音乐

1. **场景构建**

\* 利用免费资源搭建合理且美观的场景。

\* 为场景模型增加碰撞模型，增加真实感。

**3.2 主要功能模块设计**

**1.玩家控制模块：**

玩家通过`PlayerController`脚本控制移动和攻击等动作。

实现第三人称视角摄像头追踪玩家并控制行走方向。

**2.敌人行为模块：**

使用`MonsterAI`实现敌人的状态机，包括空闲、追击和攻击状态。

通过`EnemyManager`管理敌人复活和类型分配。

**3.UI交互模块：**

通过Canvas实现时间显示、血条更新、游戏暂停和结束界面切换。

使用TextMeshPro优化文本显示。

**4.苹果交互模块：**

红苹果：回复玩家生命值。

绿苹果：增加玩家攻击力，支持叠加效果，效果有限时长。

蓝苹果：提高玩家移动速度，支持叠加效果，效果有限时长。

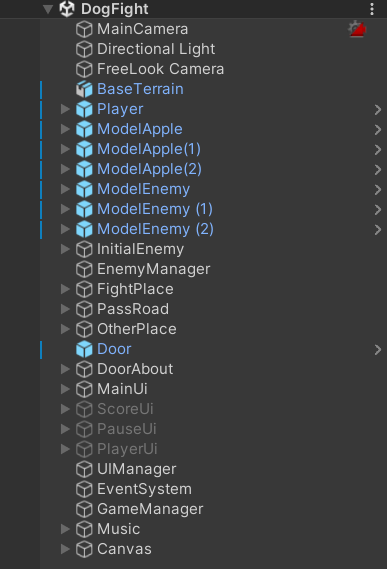
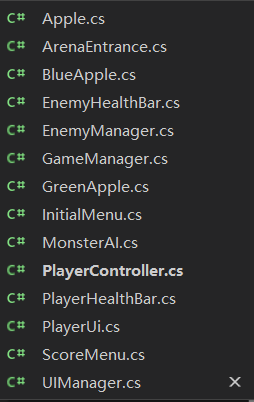
**5.场景模块**

**6.游戏逻辑控制模块：**

实现战斗开始的触发机制，带有动画显示战斗开始

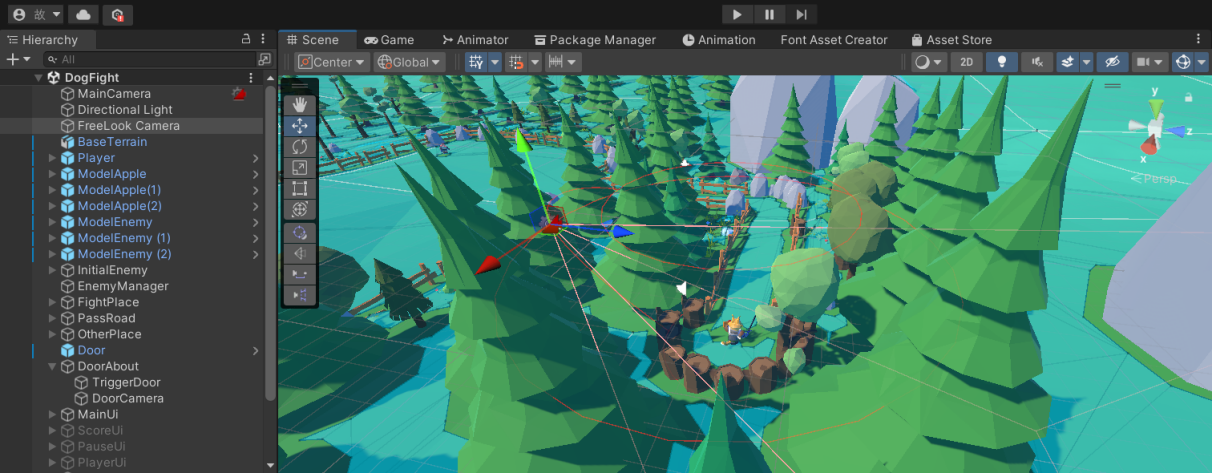
实现倒计时效果和计分逻辑

1. **具体实现**



**4.1 玩家控制模块**

【1】第三人称摄像头（Cinemachine实现）



【2】移动逻辑实现

**PlayerController.cs**

// 移动逻辑

float horizontal = Input.GetAxisRaw("Horizontal");

float vertical = Input.GetAxisRaw("Vertical");

Vector3 direction = new Vector3(horizontal, 0f, vertical).normalized;

if (direction.magnitude >= 0.1f)

{

float targetAngle = Mathf.Atan2(direction.x, direction.z) \* Mathf.Rad2Deg + cam.eulerAngles.y;

float angle = Mathf.SmoothDampAngle(transform.eulerAngles.y, targetAngle, ref turnSmoothVelocity, turnSmoothTime);

transform.rotation = Quaternion.Euler(0f, angle, 0f);

Vector3 moveDir = Quaternion.Euler(0f, targetAngle, 0f) \* Vector3.forward;

bool isSprinting = Input.GetKey(KeyCode.LeftShift);

bool isWalkingSlowly = Input.GetKey(KeyCode.LeftAlt);

if (isSprinting)

{

controller.Move(moveDir.normalized \* speed \* sprintMultiplier \* Time.deltaTime);

ChangeAnimationState(RUNFORWARDBATTLE);

}

else if (isWalkingSlowly)

{

controller.Move(moveDir.normalized \* speed \* slowWalkMultiplier \* Time.deltaTime); // 使用慢速行走倍数

ChangeAnimationState(WALKFORWARDBATTLE);

}

else

{

controller.Move(moveDir.normalized \* speed \* Time.deltaTime);

ChangeAnimationState(RUNFORWARDBATTLE);

}

}

else

{

ChangeAnimationState(IDLE\_BATTLE);

}实现鼠标控制转向和wasd控制方向和行走方式（alt - shift）

【3】其他动画状态转变

动画逻辑具体在**PlayerController.cs**里，这里就不具体展示。

注意到，这里我没有用到animator的状态转换，由于这里动画比较多所以我选择直接在代码里实现动画的播放控制

const string IDLE\_BATTLE = "Idle\_Battle";

const string RUNFORWARDBATTLE = "RunForwardBattle";

const string WALKFORWARDBATTLE = "WalkForwardBattle";

const string ATTACK01 = "Attack01"; // 攻击动画1

const string ATTACK02 = "Attack02"; // 攻击动画2

const string DEFEND = "Defend"; // 防御动画

const string HIT = "GetHit"; // 受击动画名称

const string DIE = "Die"; // 死亡动画名称

void ChangeAnimationState(string newState)

{

// 阻止同一个动画状态再次打断自己

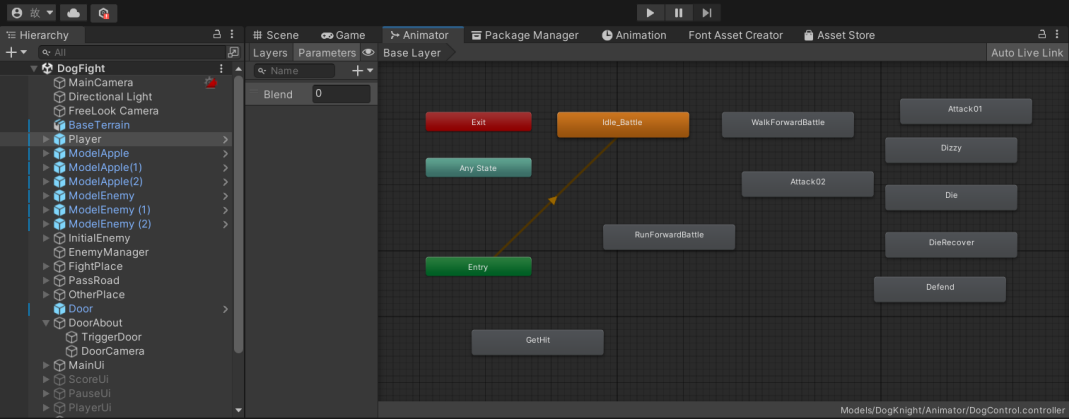
if (currentState == newState) return;

// 播放新的动画状态

animator.Play(newState);

currentState = newState;

}



【4】玩家加生命，受伤害，加攻击，加移速逻辑

public void Heal(int amount)

{

healthBar.Heal(amount);

currentHealth += amount;

currentHealth = Mathf.Min(currentHealth, maxHealth); // 确保生命值不会超过最大值

Debug.Log($"玩家生命值恢复：当前生命值为 {currentHealth}");

}

public void TakeDamage(int damage)

{

// 如果已经死亡、正在受击或处于防御状态，不处理伤害

if (isDead || isGettingHit || isDefending)

{

if (isDefending)

{

Debug.Log("玩家正在防御，未受到伤害！");

}

return;

}

healthBar.TakeDamage(damage);

currentHealth -= damage;

Debug.Log($"玩家受到伤害：{damage}，剩余血量：{currentHealth}");

if (currentHealth > 0)

{

StartCoroutine(PlayHitAnimation());

}

else

{

Die();

}

}

public void AddAttackBoost(int boostAmount, float duration, int maxBoost)

{

int totalBoost = Mathf.Clamp(attackDamage + boostAmount, 0, maxBoost); // 防止超过上限

attackDamage = totalBoost;

// 更新 PlayerStatusEffects 的 UI

int currentStages = Mathf.CeilToInt((attackDamage - 25) / 25) + 1; // 根据提升计算UI阶段

playerStatusEffects.UpdateGreenBar(currentStages);

Debug.Log($"当前攻击力提升：{attackDamage}");

StartCoroutine(RemoveAttackBoostAfterTime(boostAmount, duration));

}

private IEnumerator RemoveAttackBoostAfterTime(int boostAmount, float duration)

{

yield return new WaitForSeconds(duration);

attackDamage -= boostAmount;

attackDamage = Mathf.Max(0, attackDamage); // 防止小于0

int currentStages = Mathf.CeilToInt((attackDamage - 25) / 25) + 1; // 根据提升计算UI阶段

playerStatusEffects.UpdateGreenBar(currentStages);

Debug.Log($"攻击力增强效果结束，当前攻击力提升：{attackDamage}");

}

public void AddSpeedBoost(float boostAmount, float duration, float maxBoost)

{

float totalBoost = Mathf.Clamp(speed + boostAmount, 0, maxBoost); // 防止超过上限

speed = totalBoost;

int currentStages = Mathf.CeilToInt((speed - 6) / 2) + 1; // 计算当前增益阶段

playerStatusEffects.UpdateBlueBar(currentStages);

Debug.Log($"当前速度提升：{speed}");

StartCoroutine(RemoveSpeedBoostAfterTime(boostAmount, duration));

}

private IEnumerator RemoveSpeedBoostAfterTime(float boostAmount, float duration)

{

yield return new WaitForSeconds(duration);

speed -= boostAmount;

speed = Mathf.Max(0, speed); // 防止小于0

int currentStages = Mathf.CeilToInt((speed - 6) / 2) + 1; // 计算当前增益阶段

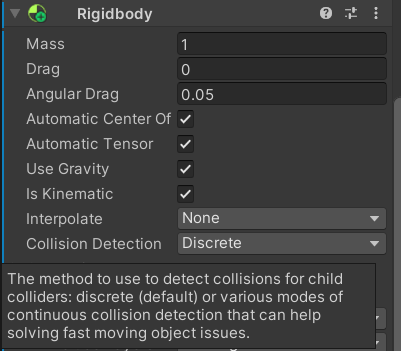
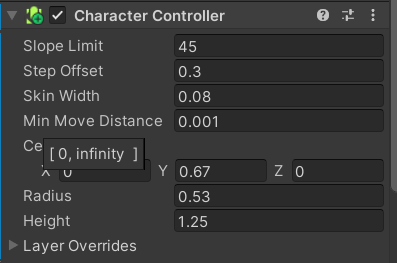
playerStatusEffects.UpdateBlueBar(currentStages);

Debug.Log($"速度增强效果结束，当前速度提升：{speed}");

}

**这里的逻辑，生命叠加不能超过最大生命，由于数值上限设置，攻击力和移速的叠加最多只有三层，且buff有效时长为10秒**

【5】碰撞体积设置和重力设置



// 应用重力

if (controller.isGrounded && velocity.y < 0)

{

velocity.y = -2f; // 确保角色保持在地面

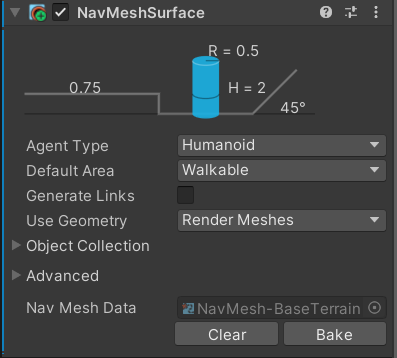
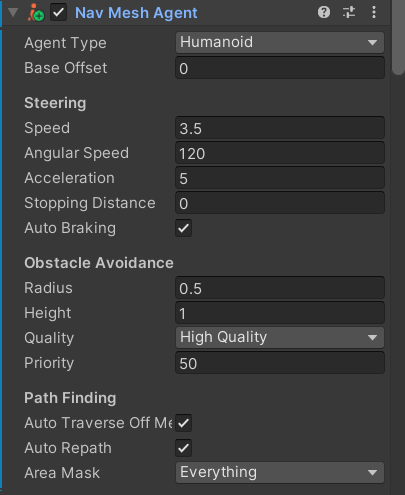
}

velocity.y += gravity \* Time.deltaTime;

controller.Move(velocity \* Time.deltaTime);

**4.2 敌人行为模块**

【1】使用Nav相关组件实现导航



左图挂在敌人上，右图挂在地形上

【2】使用类状态机逻辑根据距离实现敌人状态切换

**MonsterAI.cs**

private enum State

{

Idle,

Chase,

Attack

}

// 状态机处理不同状态

switch (currentStateAI)

{

case State.Idle:

HandleIdleState(distanceToTarget);

break;

case State.Chase:

HandleChaseState(distanceToTarget);

break;

case State.Attack:

HandleAttackState(distanceToTarget);

break;

}



**图中演示了几种不同状态**

【3】实现死亡逻辑生成新的敌人

// 死亡逻辑

private void Die()

{

isDead = true;

Debug.Log("Boss 死亡！");

gameManager.OnEnemyKilled();

// 停止 NavMeshAgent

nav.isStopped = true;

manager.NotifyEnemyDeath(initialPosition); // 通知管理器

// 播放死亡动画

ChangeAnimationState(dieAnimation);

// 记录死亡时的当前位置

Vector3 deathPosition = transform.position;

// 延迟生成苹果

SpawnApple(deathPosition);

// 延迟销毁

Destroy(gameObject, 2f);

}

**EnemyManager.cs**

// 敌人生成概率分布，按顺序对应 red, green, blue

private float[] probabilities = { 0.4f, 0.3f, 0.3f };

// 通知敌人死亡并生成新的敌人

public void NotifyEnemyDeath(Vector3 position)

{

StartCoroutine(RespawnEnemy(position));

}

private IEnumerator RespawnEnemy(Vector3 position)

{

yield return new WaitForSeconds(respawnDelay);

// 根据概率随机生成敌人

GameObject enemyPrefab = GetRandomEnemyPrefab();

// 在原位置生成新的敌人

GameObject newEnemy = Instantiate(enemyPrefab, position, Quaternion.identity);

// 获取 MonsterAI 组件并设置敌人类型

MonsterAI monsterAI = newEnemy.GetComponent<MonsterAI>();

if (monsterAI != null)

{

if (enemyPrefab == redEnemyPrefab)

{

monsterAI.enemyType = MonsterAI.EnemyType.Red;

}

else if (enemyPrefab == greenEnemyPrefab)

{

monsterAI.enemyType = MonsterAI.EnemyType.Green;

}

else if (enemyPrefab == blueEnemyPrefab)

{

monsterAI.enemyType = MonsterAI.EnemyType.Blue;

}

}

newEnemy.GetComponent<MonsterAI>().manager = this; // 设置管理器引用

}



**如图绿狗为随机生成的敌人**

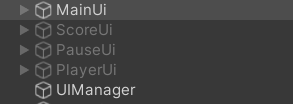
**4.3 用户界面实现**

【1】通过Canvas设置`Canvas Scaler`为`Scale With Screen Size`，并参考分辨率为1920x1080。



**在game里设置好最后要到处的分辨率以便更好设置ui的大小**

【2】基本界面ui的设计



**这里通过UIManager.cs实现各个UI的显示与隐藏，具体逻辑在代码里**

**\*\* MainUi**



**实现开始游戏和退出游戏**

**\*\* PauseUi**



**Esc暂停，并可以结束当前游戏**

**\*\* PlayerUi**

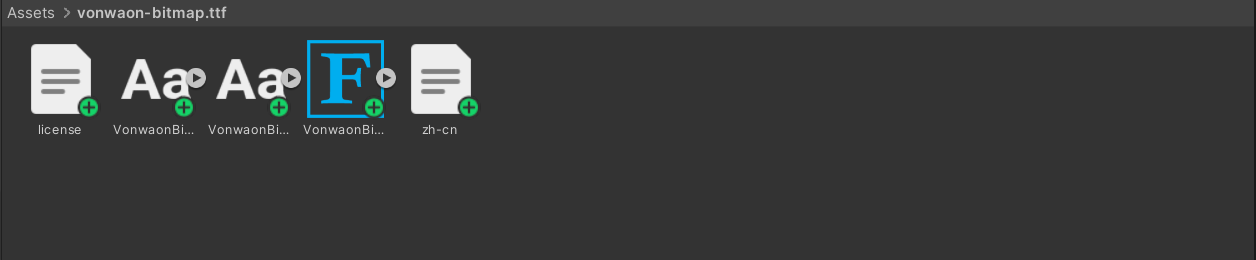
****

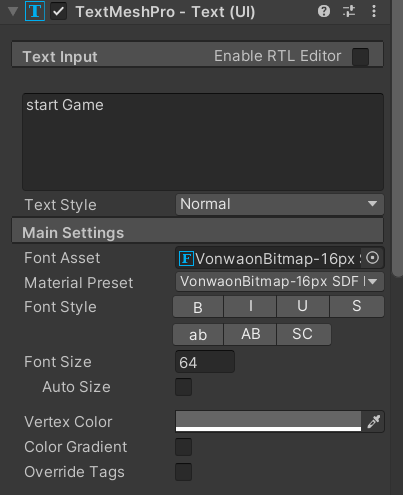
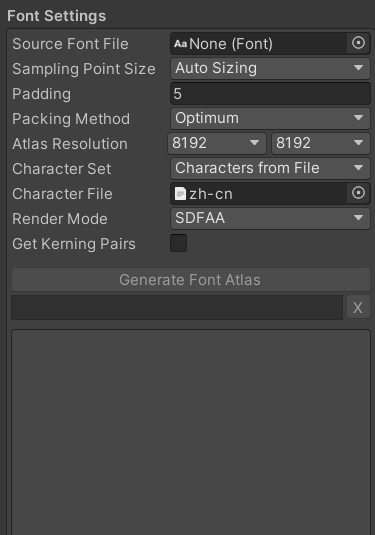
**这里实现了倒计时，玩家血条，和一个buff叠加显示ui**

**\*\* ScoreUi**

****

【3】文本显示

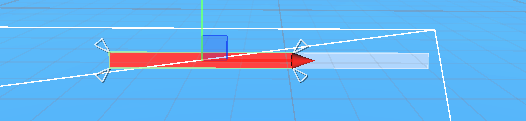


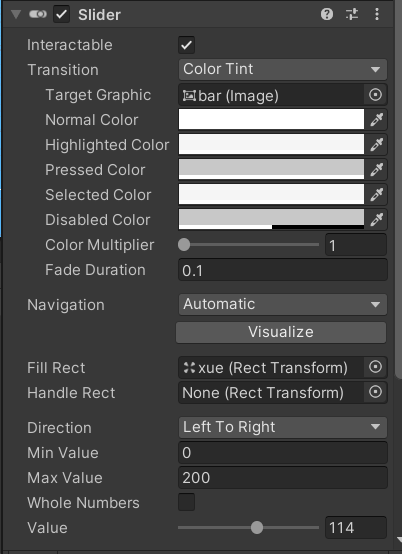


**这里通过开源的字体资源并使用Font Settings工具生成font asset**

【4】血条逻辑实现

玩家血条





**PlayerHealthBar.cs**

// 初始化血条

public void InitializeHealthBar(int maxHealth)

{

this.maxHealth = maxHealth;

currentHealth = maxHealth;

healthSlider.maxValue = maxHealth;

healthSlider.value = maxHealth;

UpdateHealthBarColor(); // 初始化颜色

}

// 玩家受到伤害

public void TakeDamage(int damage)

{

currentHealth -= damage;

currentHealth = Mathf.Clamp(currentHealth, 0, maxHealth); // 防止血量小于0

healthSlider.value = currentHealth;

UpdateHealthBarColor(); // 更新颜色

if (currentHealth <= 0)

{

Die();

}

}

// 玩家回血

public void Heal(int healAmount)

{

currentHealth += healAmount;

currentHealth = Mathf.Clamp(currentHealth, 0, maxHealth); // 防止血量超过最大值

healthSlider.value = currentHealth;

UpdateHealthBarColor(); // 更新颜色

}

// 更新血条颜色

private void UpdateHealthBarColor()

{

if (fillImage != null)

{

if (currentHealth <= maxHealth / 4)

{

fillImage.color = lowHealthColor; // 低血量时变为红色

}

else

{

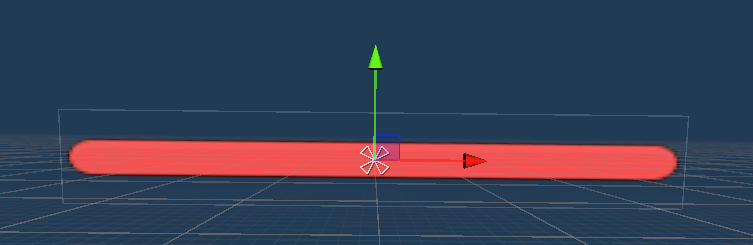
fillImage.color = defaultColor; // 正常血量时为绿色

}

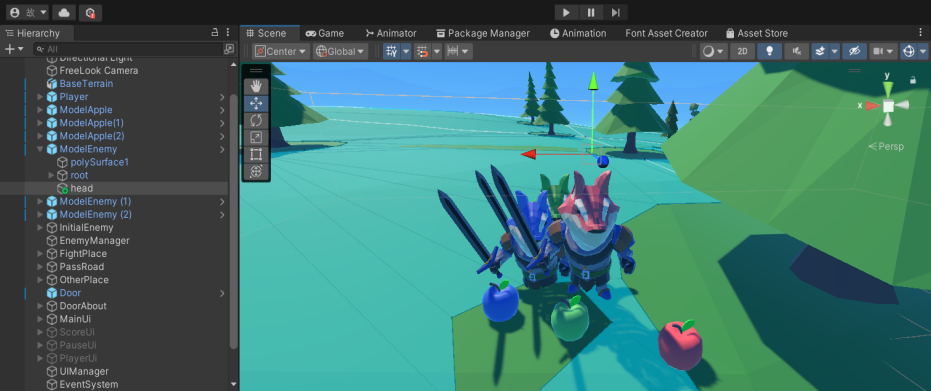
}

}

敌人血条



**每个敌人头上设置一个空位置为血条显示位置**



**EnemyHealthBar.cs**

private void Start()

{

currentHealth = maxHealth;

// 实例化血条

GameObject healthBarObject = Instantiate(healthBarPrefab, healthBarPosition.position, Quaternion.identity);

healthBarObject.transform.SetParent(GameObject.Find("Canvas").transform, false); // 设置为 UI 的子对象

// 获取 Slider 组件

healthBar = healthBarObject.GetComponent<Slider>();

healthBar.maxValue = maxHealth;

healthBar.value = maxHealth;

player = GameObject.FindWithTag("Player")?.transform;

if (player == null)

{

Debug.LogError("Player not found. Make sure the player has the correct tag.");

}

}

private void Update()

{

if (healthBar == null || targetCamera == null || healthBarPosition == null || player == null)

{

Debug.LogWarning("Missing components for health bar functionality.");

return;

}

// 检查敌人与玩家之间的距离

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

// 如果距离超过最大显示距离，则直接隐藏血条

if (distanceToPlayer > maxDisplayDistance)

{

healthBar.gameObject.SetActive(false);

return;

}

// 将世界坐标转换为屏幕坐标

Vector3 screenPosition = targetCamera.WorldToScreenPoint(healthBarPosition.position);

// 检查屏幕坐标是否在屏幕范围内

bool isVisible = screenPosition.z > 0 && screenPosition.x > 0 && screenPosition.x < Screen.width && screenPosition.y > 0 && screenPosition.y < Screen.height;

// 设置血条的可见性

healthBar.gameObject.SetActive(isVisible);

// 如果可见，则更新血条位置

if (isVisible)

{

healthBar.transform.position = screenPosition;

}

}

// 受到伤害时更新血条

public void TakeDamage(int damage)

{

currentHealth -= damage;

currentHealth = Mathf.Clamp(currentHealth, 0, maxHealth);

healthBar.value = currentHealth;

if (currentHealth <= 0)

{

Destroy(healthBar.gameObject); // 同时销毁血条

}

}

}



【5】buff ui实现（蓝-绿）

**PlayerUi.cs**

private void Start()

{

UpdateBar(greenBarParent, attackMultiplier, activeGreenColor);

UpdateBar(blueBarParent, speedMultiplier, activeBlueColor);

}

// 更新UI的方法

public void UpdateGreenBar(float multiplier)

{

attackMultiplier = Mathf.Clamp(multiplier, 1f, maxMultiplier); // 限制倍数

UpdateBar(greenBarParent, attackMultiplier, activeGreenColor);

}

public void UpdateBlueBar(float multiplier)

{

speedMultiplier = Mathf.Clamp(multiplier, 1f, maxMultiplier); // 限制倍数

UpdateBar(blueBarParent, speedMultiplier, activeBlueColor);

}

private void UpdateBar(Transform barParent, float multiplier, Color activeColor)

{

for (int i = 0; i < maxStages; i++)

{

Image stageImage = barParent.GetChild(i).GetComponent<Image>();

if (stageImage != null)

{

if (i < multiplier - 1)

{

stageImage.color = activeColor; // 激活颜色

}

else

{

stageImage.color = defaultColor; // 恢复灰色

}

}

}

}

**根据效果层数设置ui格子数**



如图速度和伤害均叠加了一层效果

**4.4 苹果交互模块**

【1】敌人死后根据类别生成不同颜色苹果

private void SpawnApple(Vector3 spawnPosition)

{

GameObject applePrefabToSpawn = null;

switch (enemyType)

{

case EnemyType.Red:

applePrefabToSpawn = redApplePrefab;

break;

case EnemyType.Green:

applePrefabToSpawn = greenApplePrefab;

break;

case EnemyType.Blue:

applePrefabToSpawn = blueApplePrefab;

break;

}

if (applePrefabToSpawn != null)

{

GameObject apple = Instantiate(applePrefabToSpawn, spawnPosition, Quaternion.identity);

Destroy(apple, appleDestroyDelay);

}

} **苹果生成时间为20秒，一定时间后会自动销毁**

【2】不同苹果叠加不同buff

**Apple.cs**

**BlueApple.cs**

**GreenApple.cs**

**这三个脚本分别实现了不同苹果对玩家人物的作用**

**这里示例其中一个：（苹果不设置碰撞体积）**

public class Apple : MonoBehaviour

{

public int healAmount = 50; // 回复的生命值

private void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Player")) // 检查是否是玩家触碰

{

PlayerController player = other.GetComponent<PlayerController>();

if (player != null)

{

player.Heal(healAmount); // 调用玩家的回血方法

Debug.Log($"玩家触碰苹果，恢复了 {healAmount} 点生命值！");

}

Destroy(gameObject); // 摧毁苹果

}

}

}



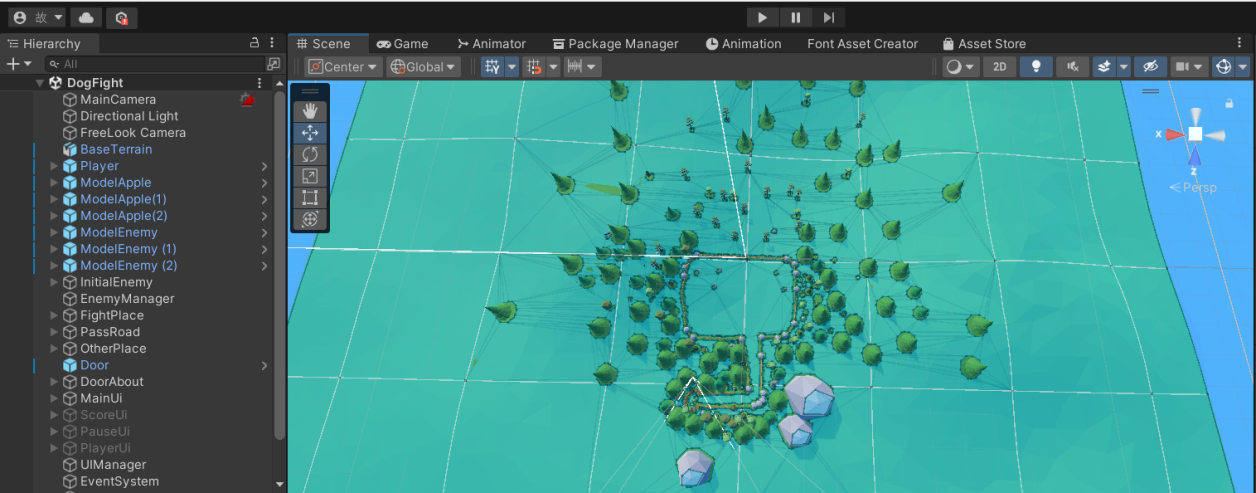
**如图有绿蓝两种苹果**

**4.5 场景模块**

利用已有免费资源搭建场景，为必要物理添加碰撞体积

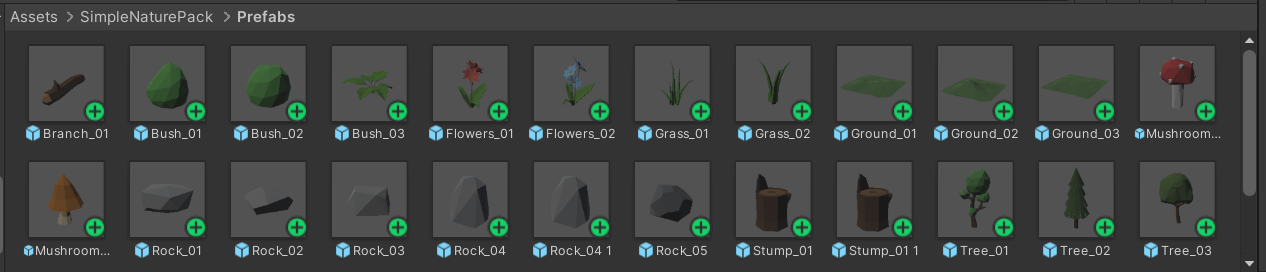
先从免费资源中选中一个基本地形，再在上面搭积木

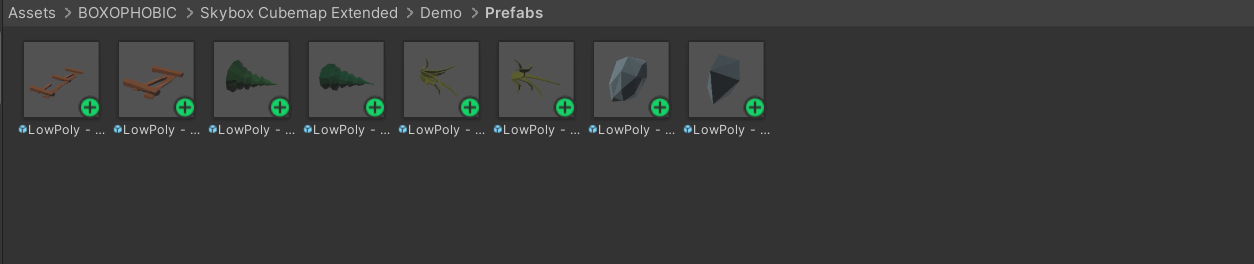
为天空使用免费模型带的天空盒

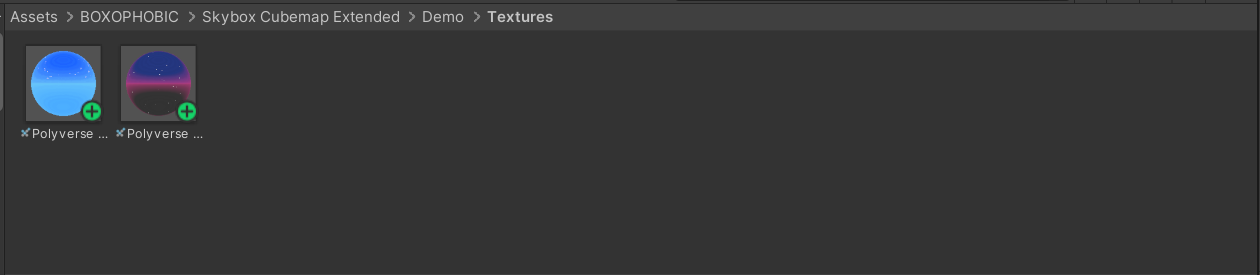


（图为我设计的场景的俯视图，人物可以走一段距离再到竞技场触发战斗，结束战斗可回到起点）

资源：



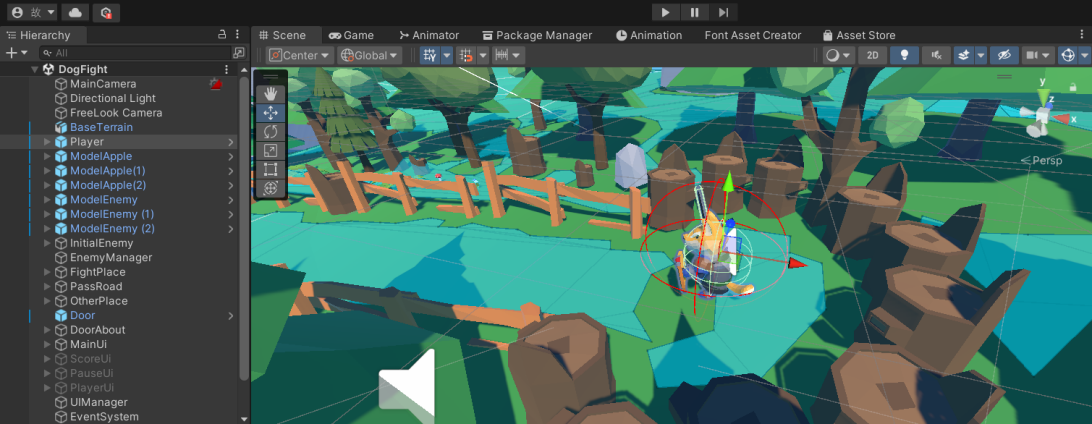




**4.6 游戏逻辑控制模块**

【1】玩家与敌人初始化

玩家初始

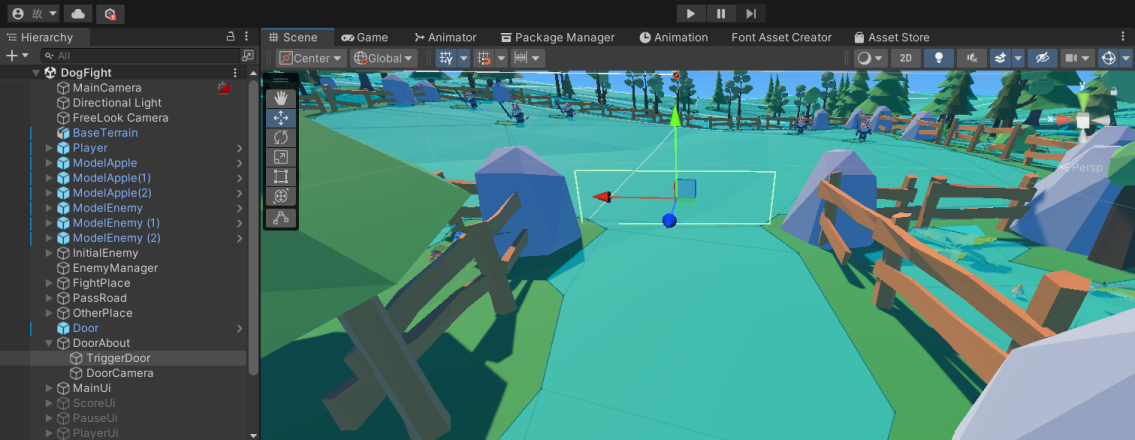


敌人初始

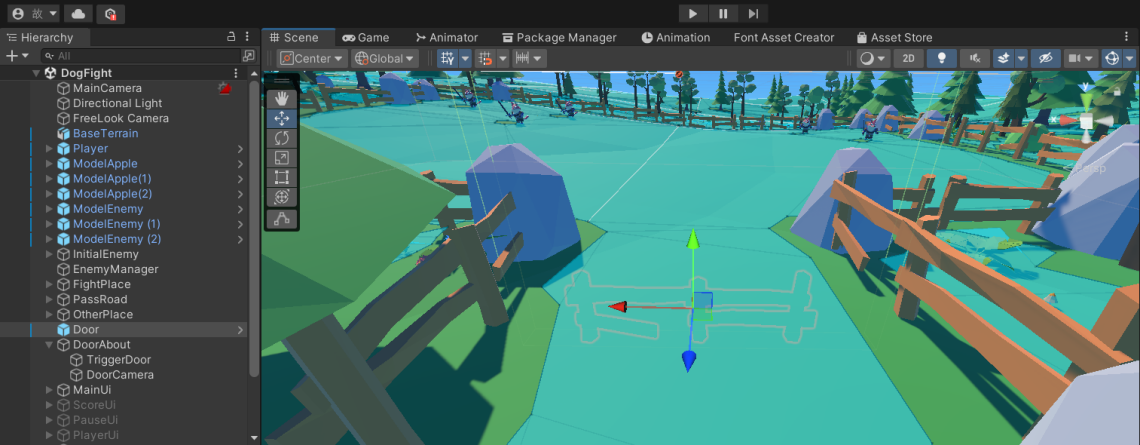


【2】开始战斗触发

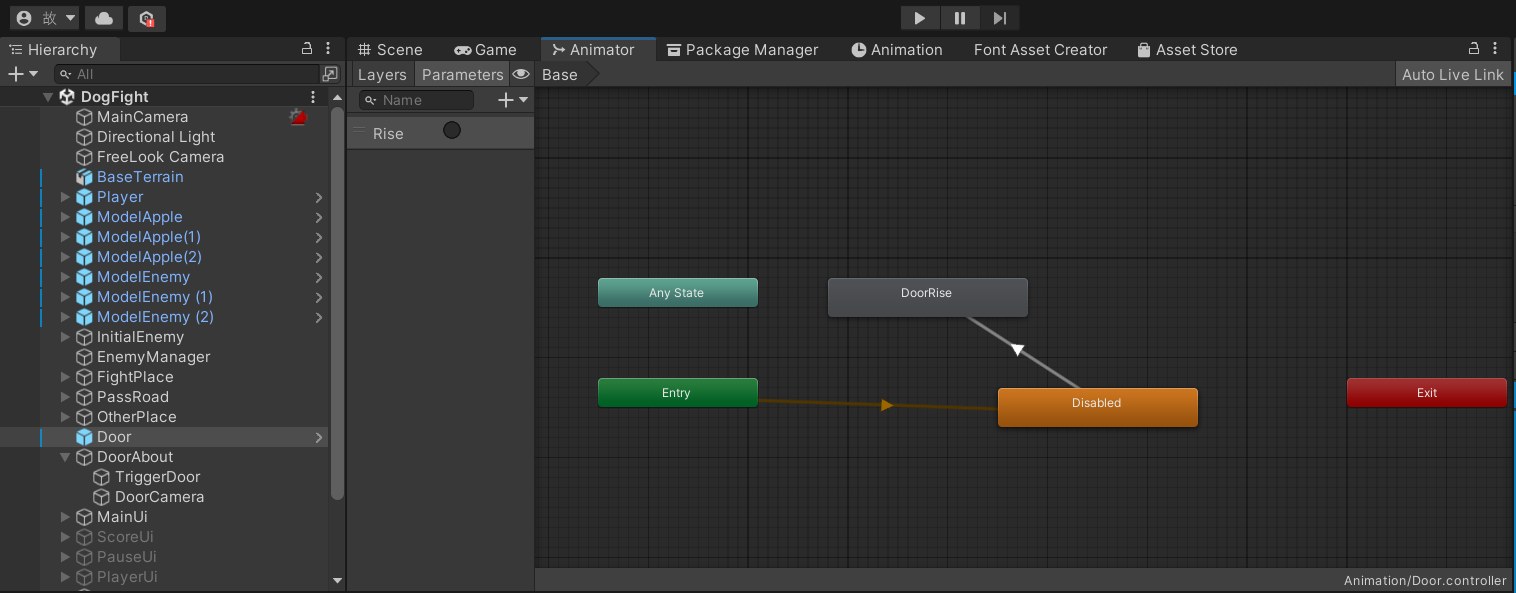
触发墙



触发动画



（门一开始在下面，触发后空气墙生效，并演示上升动画）



（doorrise的动画是我自己实现的，每帧设置相应的Y坐标）

触发逻辑

**ArenaEntrance.cs**

private IEnumerator HandleArenaEntrance()

{

PlayerController playerController = GameObject.FindGameObjectWithTag("Player").GetComponent<PlayerController>();

if (playerController != null)

{

playerController.enabled = false; // 禁用玩家操作

playerController.ForceIdleState();

}

// 激活门的 BoxCollider

BoxCollider doorCollider = doorObject.GetComponent<BoxCollider>();

if (doorCollider != null)

{

doorCollider.enabled = true;

}

// 切换到门摄像头

if (doorCamera != null && mainCamera != null)

{

mainCamera.enabled = false;

doorCamera.enabled = true;

}

// 播放门动画

if (doorAnimator != null)

{

doorAnimator.SetTrigger("Rise");

}

// 平滑切换音乐

if (backgroundMusic != null && fightMusic != null)

{

StartCoroutine(SmoothMusicTransition());

}

// 停留摄像头一段时间（可选）

yield return new WaitForSeconds(cameraFocusDuration);

// 切回主摄像头

if (doorCamera != null && mainCamera != null)

{

doorCamera.enabled = false;

mainCamera.enabled = true;

// 重新启用玩家控制

if (playerController != null)

{

playerController.enabled = true;

}

}

else

{

Debug.LogError("MainCamera 或 doorCamera 未正确分配！");

yield break; // 停止协程

}

}

【3】战斗倒计时和计分逻辑

**GameManager.cs**

**// 玩家击杀敌人时调用的函数**

public void OnEnemyKilled()

{

enemyKillCount++; // 增加击杀计数

Debug.Log($"Enemy killed! Total: {enemyKillCount}");

}

// 开始倒计时

public void StartCountdown()

{

if (!isCountingDown)

{

isCountingDown = true;

StartCoroutine(CountdownTimer());

}

}

private IEnumerator CountdownTimer()

{

int timeRemaining = countdownTime;

// 倒计时逻辑

while (timeRemaining > 0)

{

int minutes = timeRemaining / 60;

int seconds = timeRemaining % 60;

uiManager.UpdatePlayerUiText($"{minutes:00} : {seconds:00}");

yield return new WaitForSeconds(1);

timeRemaining--;

}

// 倒计时结束后逻辑

EndCountdown();

}

public void EndCountdown()

{

isCountingDown = false;

uiManager.UpdateScoreUiText($"{enemyKillCount}");

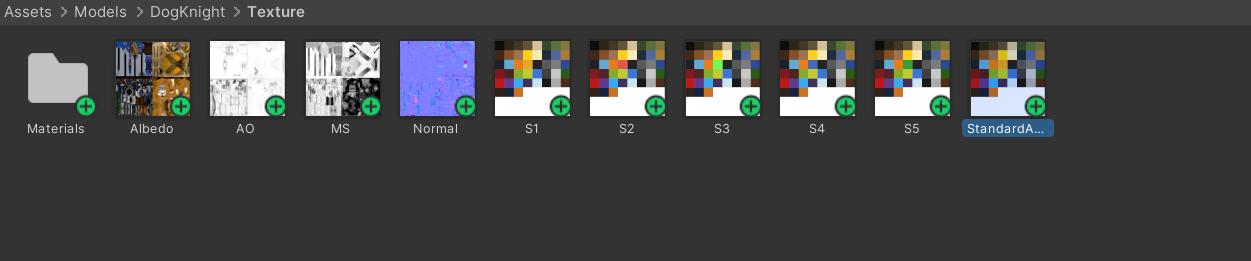
uiManager.ShowScoreUI(); // 显示分数界面

fightMusic.Stop();

}

}

**【敌我区分问题解决】**



**在模型文件中有狗模型的材质文件，我通过改变材质图片上某一色块的颜色来改变狗模型的身体颜色**

**TextureModifier.cs**

public class TextureModifier : MonoBehaviour

{

public Texture2D sourceTexture; // 原始图片

public BlockColor[] blocksToModify; // 存储多个色块及其对应颜色

private const int blockSize = 64; // 每个色块的大小（64x64）

[System.Serializable]

public class BlockColor

{

public int blockX; // 色块的 X 位置

public int blockY; // 色块的 Y 位置

public Color color; // 该色块的目标颜色

}

void Start()

{

if (sourceTexture == null)

{

Debug.LogError("请指定要修改的纹理！");

return;

}

// 创建一个副本以避免修改原始图片

Texture2D editableTexture = Instantiate(sourceTexture);

// 确保纹理是可读写的

if (!editableTexture.isReadable)

{

Debug.LogError("纹理未启用读写，请在Inspector中启用！");

return;

}

// 修改多个色块的颜色

foreach (BlockColor block in blocksToModify)

{

ModifyBlockColor(editableTexture, block.blockX, block.blockY, block.color);

}

// 应用修改

editableTexture.Apply();

// 将修改后的纹理应用到材质上

Renderer renderer = GetComponent<Renderer>();

if (renderer != null)

{

renderer.material.mainTexture = editableTexture;

}

// 保存到文件

SaveTextureToFile(editableTexture, "ModifiedTexture.png");

}

private void ModifyBlockColor(Texture2D texture, int blockX, int blockY, Color color)

{

// 计算色块左下角像素的起始位置

int startX = blockX \* blockSize;

int startY = blockY \* blockSize;

// 检查是否超出纹理范围

if (startX + blockSize > texture.width || startY + blockSize > texture.height)

{

Debug.LogError($"指定的色块({blockX}, {blockY})超出纹理范围！");

return;

}

// 修改整个色块的颜色

for (int x = startX; x < startX + blockSize; x++)

{

for (int y = startY; y < startY + blockSize; y++)

{

texture.SetPixel(x, y, color);

}

}

}

private void SaveTextureToFile(Texture2D texture, string fileName)

{

// 将纹理编码为 PNG 格式

byte[] pngData = texture.EncodeToPNG();

if (pngData != null)

{

// 获取项目文件夹的路径

string path = Path.Combine(Application.dataPath, fileName);

// 将数据写入文件

File.WriteAllBytes(path, pngData);

Debug.Log($"纹理已保存到: {path}");

}

else

{

Debug.LogError("纹理编码失败！");

}

}

}

**最终效果**



**五、项目结果与分析**

**1. 游戏功能实现：**

玩家可以正常移动并攻击敌人。

敌人具有不同的行为模式，并根据类型掉落不同颜色的苹果。

玩家UI实时更新血条和时间，游戏结束后显示得分。

**2. 用户体验：**

通过增强UI效果，提升了游戏的视觉表现力。

红苹果可回血，杀一个敌人积一分。

倒计时结束或玩家死亡显示得分。

绿苹果（可叠三层攻击）和蓝苹果（可叠三层移速）的叠加机制提供了更高的游戏策略性，每层攻击移速buff有时间限制一定时间后自动消失。

1. **结果视频见同压缩包**

**六、总结与展望**

**6.1 总结**

本实验通过Unity引擎成功开发了一款具备多种交互功能的3D虚拟场景游戏。实验的完成不仅提升了我们的团队协作能力，还增强了对Unity引擎及其附加组件的掌握，还提升了我们的C#编程能力，有能力独立做出一个小游戏demo，并发现了做游戏的乐趣和解决各种问题之后的喜悦。

**6.2 展望**

未来可以在以下方面进一步优化：

1. 添加更多的敌人类型与攻击模式；

2. 加入多人联机功能，增加游戏的可玩性；

3. 使用Shader增强场景的视觉效果。

**七、参考文献**

1. Unity官方文档：[Unity Documentation](https://docs.unity3d.com/)

2. 模型资源：[Unity Asset Store](https://assetstore.unity.com/)