VR 游戏制作流程

陈玮钰

使用引擎：Unity 2021.3.34

使用设备：PICO 4ultra

一、太空场景

要求：制作与《无限 The Infinite》XR 太空沉浸展中相似的效果，场景中存在一个光球，当玩家靠近时光球放大，同时屏幕渐暗，播放全景视频，当玩家离开时停止播放视频，回到场景中。

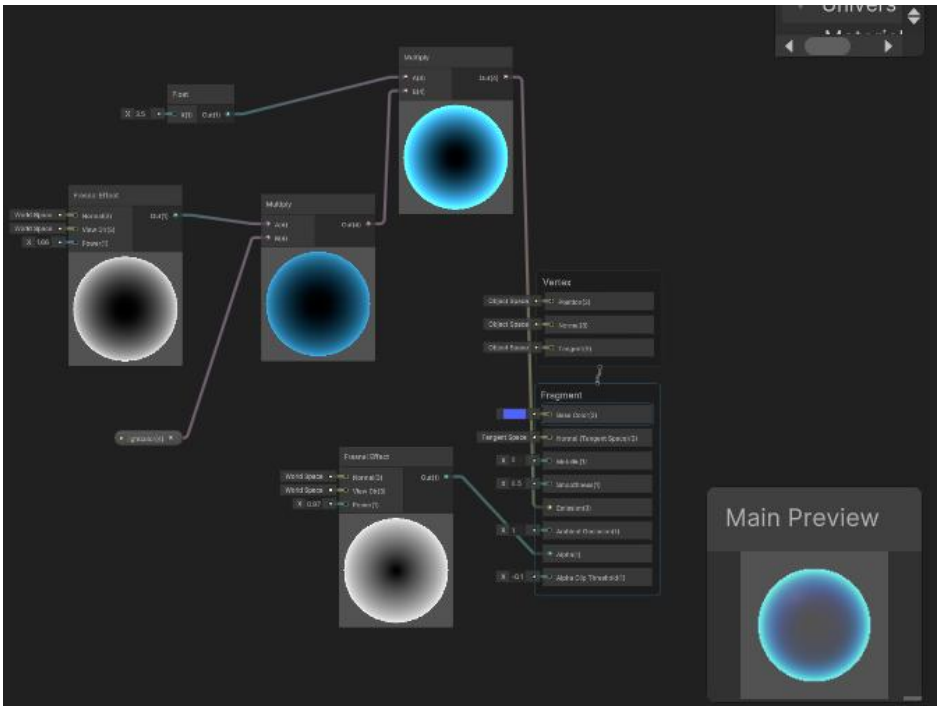
基本逻辑：玩家进入范围触发-->光球开始放大-->屏幕渐暗-->渐暗完成后显示视频-->屏幕渐亮-->光球恢复初始大小（玩家看不到）-->离开触发区域-->屏幕渐暗-->关闭视频显示-->屏幕渐亮

1.XR 配置

在 Package Manager 中搜索 XR，下载 Unity 自带的 XR 相关插件。在 PICO 平台中下载 PICO 的 SDK，添加 PXR Manager、调整最低 API 等级、注册账号获取 APP ID 等，以便后续打包在 PICO 设备上运行测试。如果需要使用手势识别，需要额外设置 PICO 的 input 相关。

2.光球

使用 Shader Graphs1 自定义。创建 shader，再创建一个使用该 shader 的材质，将材质赋予球体。



Surface Type 选择 Transparent，主相机开启 post pocessing 后处理，添加 Global Volume-->Add Override-->Post Pocessing,勾选 Bloom，Threshold 在 0~1，Intensity 调高，使溢出的辉光能被看到，达到球体发光效果。

3. 范围检测

在主相机下新建空物体，将其 tag 设置为“Player”，添加 Collider。为球体添加 Rigidbody 和 Collider，勾选 Is Trigger，创建新脚本挂载在球体上。

玩家进入范围时触发。

```
private void OnTriggerEnter(Collider other)
{
    if (other.name == "Player")
    {
        if (isPlayerInRange) return;
        isPlayerInRange = true;


        StopAllRelatedCoroutines();


        showVideoAndFadeCoroutine = StartCoroutine(ShowVideoFadeAndScale());
    }
}
```

玩家离开范围时触发。

```
private void OnTriggerExit(Collider other)
{
    if (other.name == "Player")
    {
        if (!isPlayerInRange) return;
        isPlayerInRange = false;


        StopAllRelatedCoroutines();


        hideVideoAndFadeCoroutine = StartCoroutine(HideVideoFadeAndScale());
    }
}
```

4. 屏幕渐暗

使用协程进行控制。最开始预想用 UI 的 Image 覆盖相机来实现渐暗，在 VR 设备中实测发现由于设备渲染时分左右相机，Image 只能覆盖其中一个相机，因此改为用 XR 自带的屏幕渐暗程序，以相机为中心生成一个球体。

```
public void SetCurrentAlpha(float alpha)
{
    currentAlpha = alpha;
    SetAlpha();
}

IEnumerator ScreenFadeOut()
{
    float nowTime = 0.0f;
    while (nowTime < gradientTime)
    {
        nowTime += Time.deltaTime;
        nowFadeAlpha = Mathf.Lerp(0.0f, 1.0f, Mathf.Clamp01(nowTime / gradientTime));
        Debug.Log("ScreenFadeOut nowFadeAlpha = " + nowFadeAlpha);
        Debug.Log("正在执行淡出");
        SetAlpha();

        yield return null;
    }
}
IEnumerator ScreenFadeIn()
{
    float nowTime = 0.0f;
    while (nowTime < gradientTime)
    {
        nowTime += Time.deltaTime;
        nowFadeAlpha = Mathf.Lerp(1.0f, 0.0f, Mathf.Clamp01(nowTime / gradientTime));
        Debug.Log("ScreenFadeOut nowFadeAlpha = " + nowFadeAlpha);
        Debug.Log("正在执行淡入");
        SetAlpha();
        yield return null;
    }
```

5. 球体放大

同样使用协程，公开定义放大时间和倍数。

```
private IEnumerator ScaleUpThenDown()
{
    Vector3 targetScale = initialScale * scaleMultiplier;
    float timer = 0f;
    Vector3 startScale = transform.localScale;

    while (timer < scaleDuration)
    {
        timer += Time.deltaTime;
        transform.localScale = Vector3.Lerp(startScale, targetScale, timer / scaleDuration);
        yield return null;
    }

    transform.localScale = targetScale;
    Debug.Log("放大完成");


    yield return new WaitForSeconds(1.0f);

    timer = 0f;
    startScale = transform.localScale;
    targetScale = initialScale;

    while (timer < scaleDuration)
    {
        timer += Time.deltaTime;
        transform.localScale = Vector3.Lerp(startScale, targetScale, timer / scaleDuration);
        yield return null;
    }

    transform.localScale = targetScale;
    Debug.Log("已恢复初始大小");
}
```

6. 全景视频

插件使用 AVPro，分为一个面朝里的球体和一个播放器。将球体中心调整至主相机，并设为主相机的子级，跟随相机移动，初始 Active=false。

范围检测触发后启动球体。

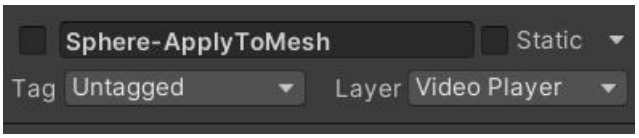
```
if (videoSphere != null)
{
    videoSphere.SetActive(true);
    Debug.Log("显示视频球体");
}
```

进行视频播放。

```
if (mediaPlayer != null && !mediaPlayer.Control.IsPlaying())
{
    mediaPlayer.Control.Play();
    Debug.Log("开始播放全景视频");
}
else if (mediaPlayer != null)
{
    Debug.Log("视频已在播放");
}
```

控制相机只渲染球体所在的那一层。

```
if (targetCamera != null)
{
    targetCamera.cullingMask = LayerMask.GetMask(layer1);
}
```



7. 问题修复

运行时发现存在一个协程触发后上一个协程还没进行完的问题，因此在协程开始前先停止其他协程。

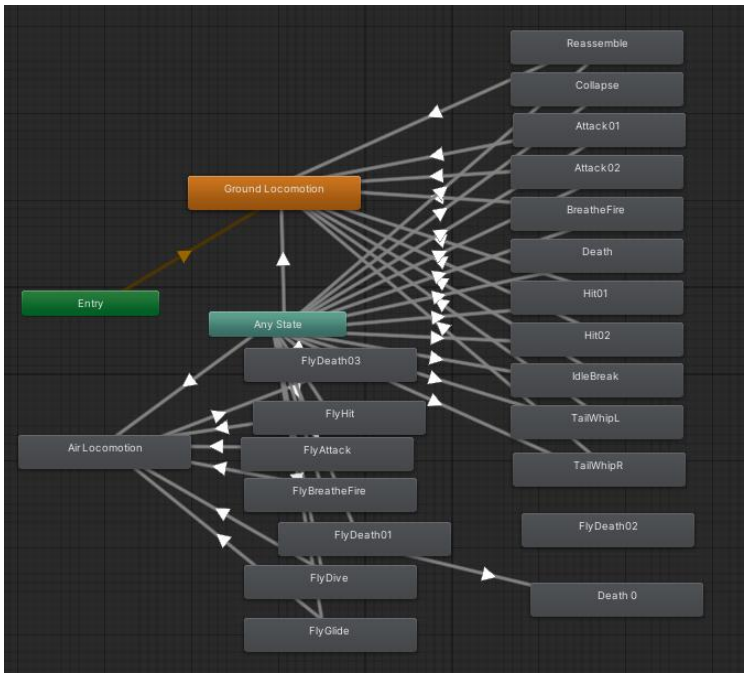
```
private void StopAllRelatedCoroutines()
{
    if (showVideoAndFadeCoroutine != null)
    {
        StopCoroutine(showVideoAndFadeCoroutine);
        showVideoAndFadeCoroutine = null;
    }
    if (hideVideoAndFadeCoroutine != null)
    {
        StopCoroutine(hideVideoAndFadeCoroutine);
        hideVideoAndFadeCoroutine = null;
    }
    if (scaleUpCoroutine != null)
    {
        StopCoroutine(scaleUpCoroutine);
        scaleUpCoroutine = null;
    }
}
```

二、古堡场景

要求：对初始的城堡场景进行处理，在其中添加敌人，能进行攻击交互，且整体的风格要统一。

预想流程：屏幕渐亮-->播放 bgm-->怪物（龙）从远处滑翔并进行吐息-->动画结束后滞留在空中-->玩家面前出现法杖-->玩家通过手势或控制器拾取法杖-->通过扳机控制法杖发射火球-->火球碰到龙后计算伤害-->满足一定伤害时播放坠落动画-->龙为死亡状态，躺在前方地面上-->玩家靠近时开始进行溶解，最终消失

1. 动画控制

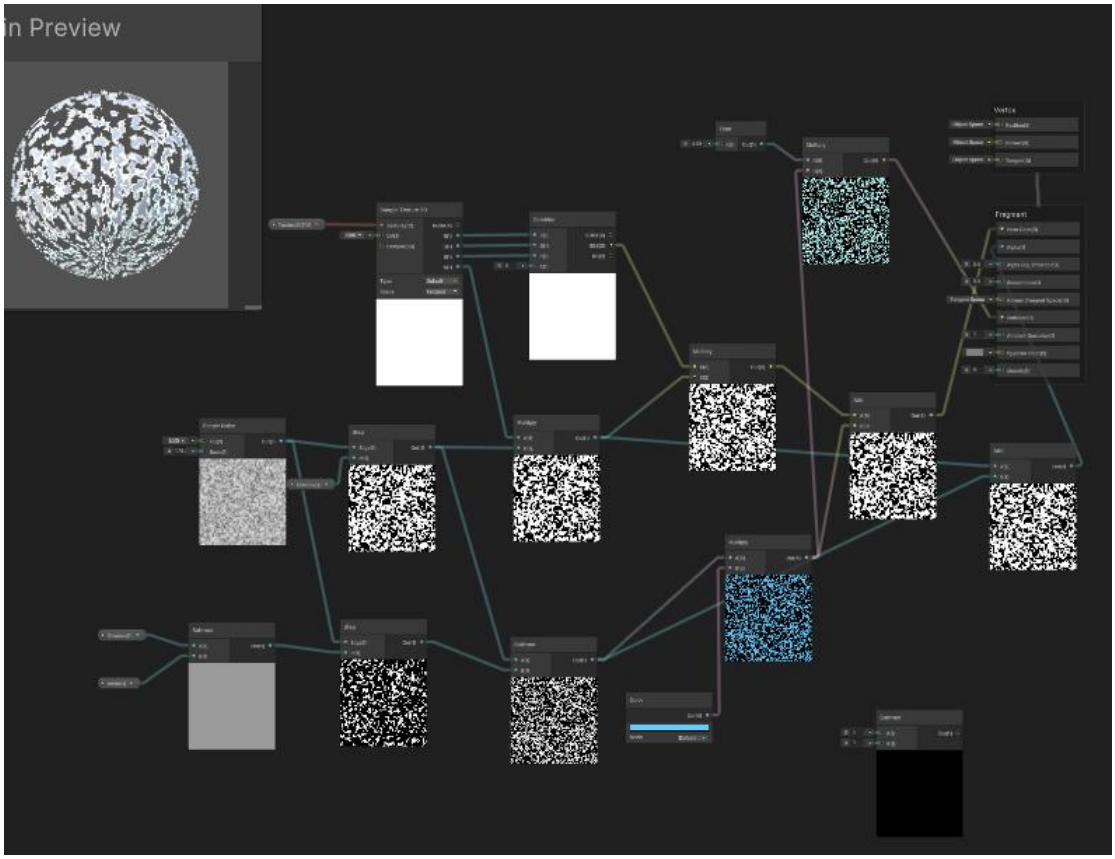


主要用到 FlyBreathFire、FlyHit、FlyDeath 等。创建 bool 变量，连接需要过渡的动画，初始设为 false，当满足条件时改为 true。

2. 溶解

使用 Shader Graphs，新建暴露的 float 参数 Dissolve 来控制溶解的程度。Alpha 随噪声图逐渐变化的同时，在边缘减出发光的部分，

Width 控制发光的宽度，新建 Color，利用 Multiply 节点可以自由选择发光的颜色。



龙身上有两种材质，因此用数组储存。通过协程，控制龙身上的两种材质的 Disslove 参数从 1.1f~0f 变化（由不透明到透明）。

```
private IEnumerator DissolveRoutine()
{
    Material[] mats = targetRenderer.materials;


    float start = mats[0].GetFloat(dissolveProp);
    float end = 0f;
    float t = 0f;

    while (t < 1f)
    {
        t += Time.deltaTime / duration;
        float v = Mathf.Lerp(start, end, t);
        foreach (Material m in mats)
            m.SetFloat(dissolveProp, v);
        yield return null;
    }

    foreach (Material m in mats)
        m.SetFloat(dissolveProp, end);
}
```

3. 发射系统

先在 Start 中获取 XR 设备，再在 Update 中监听扳机指令，若检测到扳机输入，则触发发射指令。

```
void Start()
{
    rightHandDevice = UnityEngine.XR.InputDevices.GetDeviceAtXRNode(UnityEngine.XR.XRNode.RightHand);


}

private void OnGunGrabbed(SelectEnterEventArgs args)
{
    Debug.Log("已被抓取");
}


private void OnGunReleased(SelectExitEventArgs args)
{
    Debug.Log("已被释放");
}

void Update()
{
    if (rightHandDevice.isValid)
    {
        bool triggerValue;
        if (rightHandDevice.TryGetFeatureValue(UnityEngine.XR.CommonUsages.triggerButton, out triggerValue) && triggerValue)
        {
            Shoot();
            Debug.Log("Trigger button is pressed.");
        }

    }
```

创建一个空物体在法杖前端，拖入 firepoint 字段。发射时会在 firepoint 处生成临时的预制体。

```
void Shoot()
{

    GameObject bullet = Instantiate(bulletPrefab,
                                    firePoint.position,
                                    firePoint.rotation);


}
```

4. 怪物脚本

检测到碰撞时触发受伤，武器配置完成前先用监听键盘或鼠标输入来进行测试。

```
void Update()
{
//   if (Input.GetKeyDown(KeyCode.Space))
//   {
//       Debug.Log("按下空格");
//       TakeDamage();
//   }
}
void OnCollisionEnter(Collision collision)
{

    Debug.Log("撞到了：" + collision.collider.name);
    TakeDamage();


}
```

检测到碰撞-->若状态不为死亡，则当前 HP 减去伤害量；若死亡则返回。

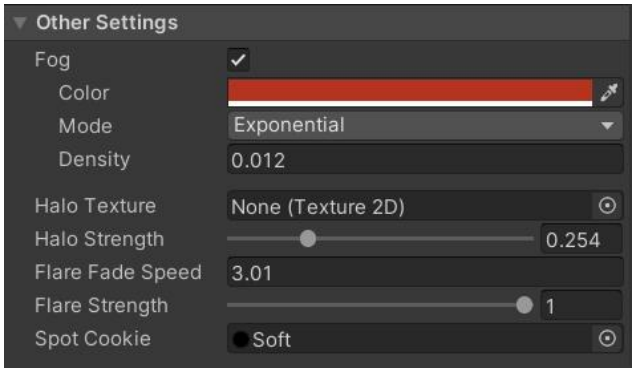
-->若剩余血量>0，则播放受击动画

-->若剩余血量<=0，则死亡状态=true，开始坠落

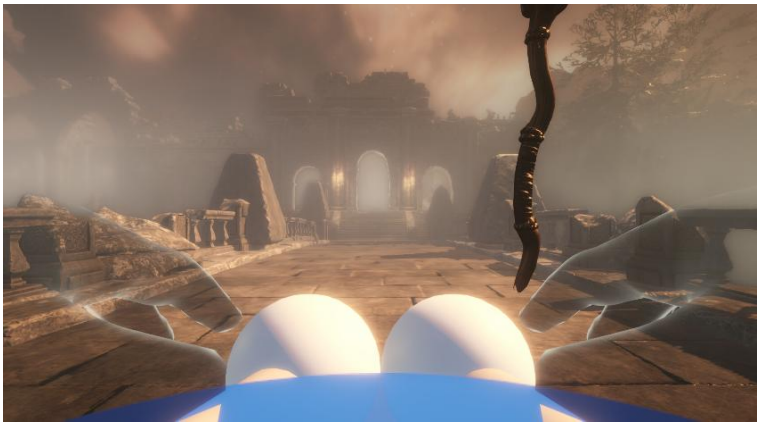
```
public void TakeDamage(int damage=30)
{
    if (isDead) return;

    currentHP -= damage;

    Debug.Log($"【DEBUG】受到伤害：{damage}，当前HP：{currentHP}，是否死亡：{isDead}");



    Debug.Log($"龙受到 {damage} 点伤害，剩余血量：{currentHP}");
    if (currentHP <= 0)
    {
        StartFalling();
    }

    else {
        animator.SetTrigger("flyGotHit");
    }
}
```

5. 雾

开启了 Unity 自带的 Fog，除此之外还添加了 Better Fog 的插件，通过修改 URP 渲染管线和后处理的方式实现雾效。

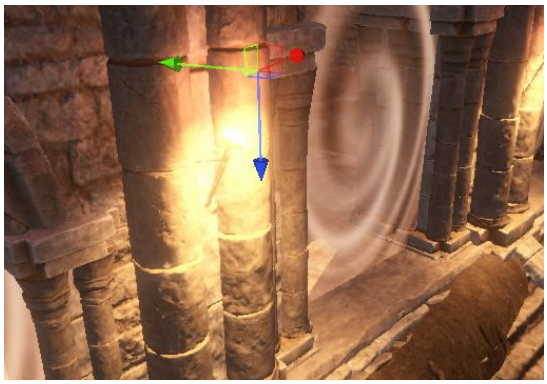


BetterFog 中可实现距离雾和高度雾，雾的动态流动通过噪声实现，但开启后会对性能有一定要求。

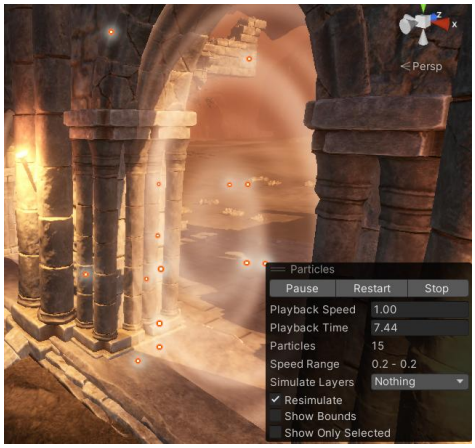


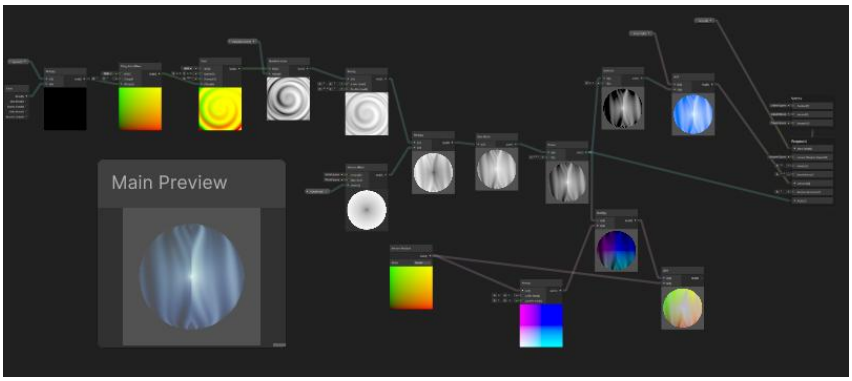
6. 场景美化

在作出加入点光、聚光灯或者区域光，调整主光源，使场景更和谐和丰富。



添加了自制的三个透明传送门，波纹随时间转动，且创建了粒子系统，自定义了发光材质实现粒子发光扩散的效果。





7. 音效系统

新建空物体命名为 AudioManager，搭载 AudioController 脚本，运行时开始播放 Bgm。

```
public class AudioController : MonoBehaviour
{

    [SerializeField] AudioSource BgmAudio;
    [SerializeField] AudioSource SfxAudio;

    public AudioClip bgm;
    public AudioClip Fall;
    private void Start()
    {
        BgmAudio.clip = bgm;
        BgmAudio.Play();
    }
    public void PlaySfx(AudioClip clip)
    {
        SfxAudio.PlayOneShot(clip);
    }
}
```

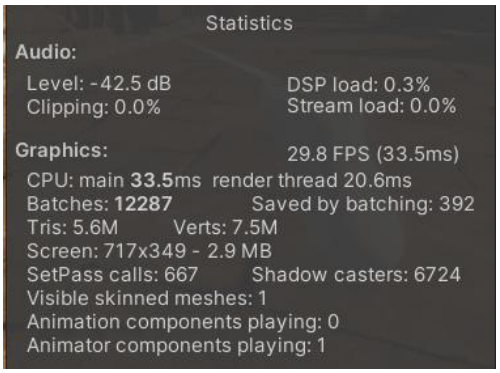
获取 AudioManager 并在特定条件播放音效

```
audioController = GameObject.FindGameObjectWithTag("Audio").GetComponent<AudioController>();
```

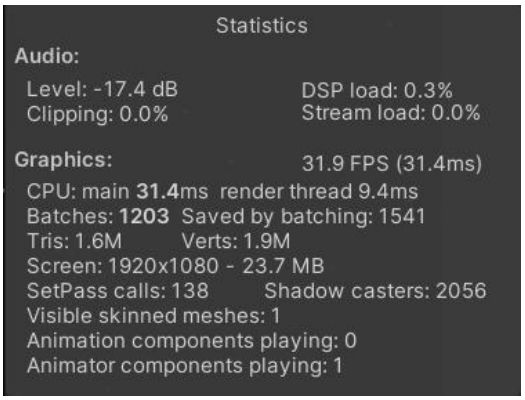
```
audioController.PlaySfx(audioController.Fall);
isDead = true;
```

8. 性能优化

在实机测试中发现帧数过低，因此对整体进行优化。



优化前



优化后

优化过程：

1. 删除玩家视野外的模型

2. 利用建模软件或 unity 插件减面

3. 利用 MeshBaker 等将纹理贴图、网格进行合并

4. 关闭副光，主光模式改为 Mixed，其余光改为 Baked，进行光照烘焙

5. 减少生成的粒子数

6. 删除地形中的一些植被

7. 将较远处的树木等做成单面