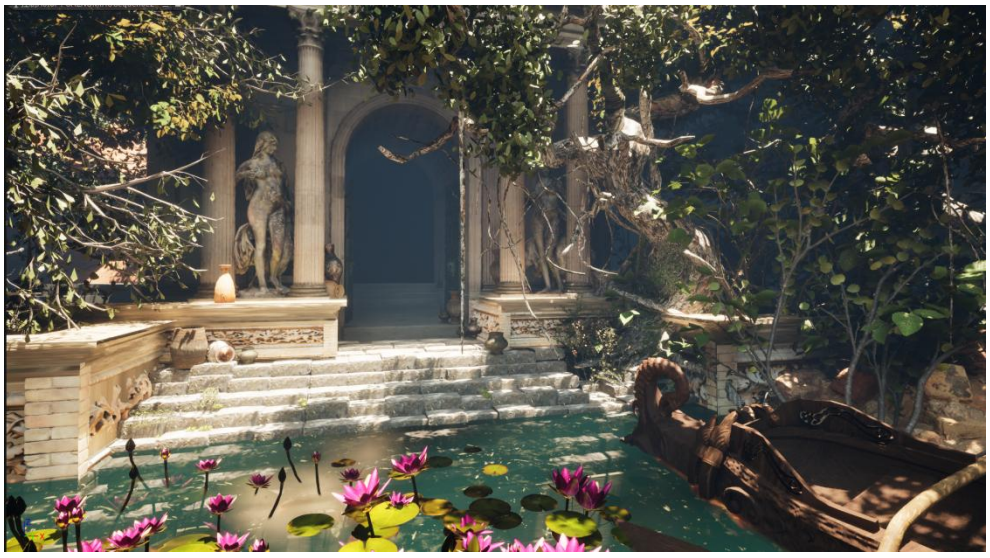
作品集技术总结文档

1. 神庙场景
2. 城市夜景（远景）
3. 水墨风格场景
4. 一键生成不同类型交互草地的 HDA
5. UNITY Shader
6. 第一人称交互 UI 系统
7. 天气管理系统（适用于 URP）

神庙场景



湖面水材质

城市夜景（远景）

在 Blender 里程序化生成城市建筑，结合灯光快速搭建

水墨风格场景

1 找到搭建场景的参考图，提供灵感和概念设计来源

2 Maya 制作白模(制作场景里的物体时,放置适用于场景的人物模型在旁边防止建造比例失误）然后导出到 UE5。

3 简单利用现有粗模搭建概念画面，然后逐步替换模型和细化。（地形图安置，从 landscape 到大石头和小的散布体搭配出合理形态，从大到小）

4 建造出基本场景以后回到 maya 继续雕刻细化物体——中模。

5 制作材质

6 进入 zrubush 精雕物体 和物体纹路

7 场景重新布景，替换高精度高细节模型，以及制作 Niagara 特效。

8 调整光影 添加摄像机 保存关卡。

水墨材质

基础材质

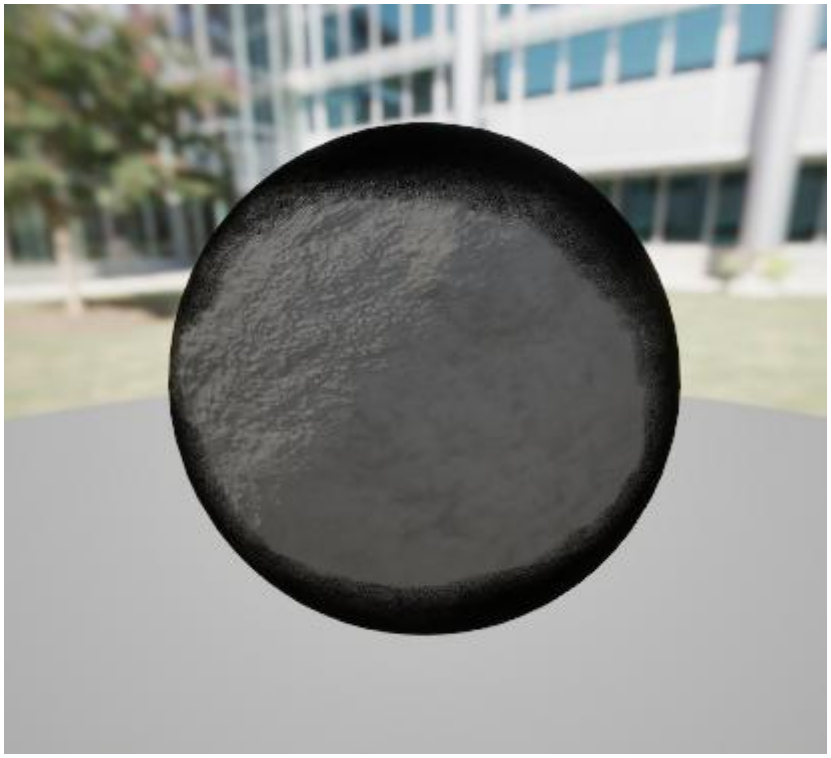
首先利用菲涅尔效应计算边缘，混入笔触纹理扰乱边缘边界规则，用 MPC 和 power 节点控制数值，并用 saturate 控制数值在[0,1]。

利用 Blend_Oerlay 节点加入 noise 虚化和扰动边缘。

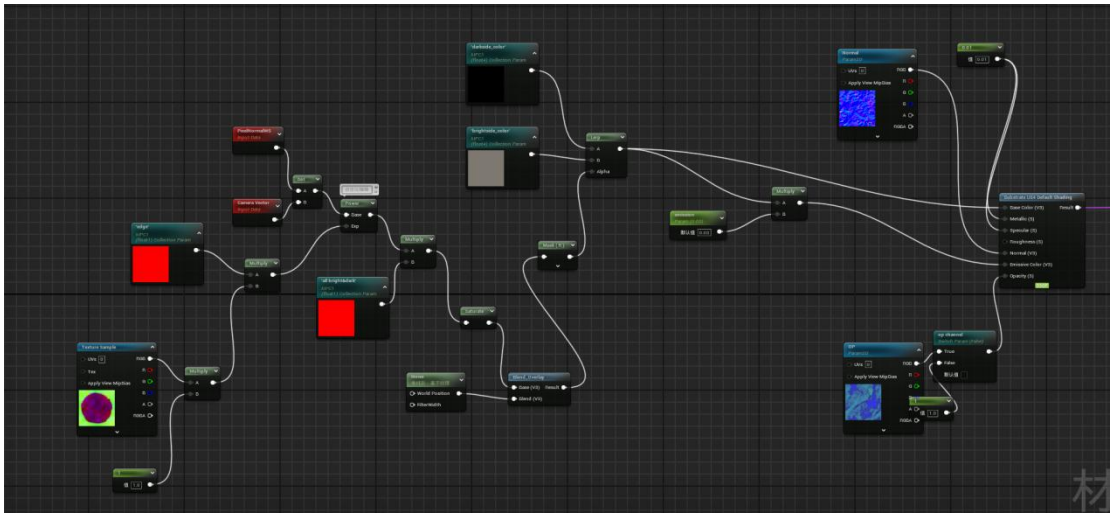
然后 mask 节点分离 R 值作为颜色的 alpha 的数值。

利用 lerp 节点混合明暗颜色，得到最终结果。

笔触纹理、法线贴图，透明度贴图提升为参数方便切换。

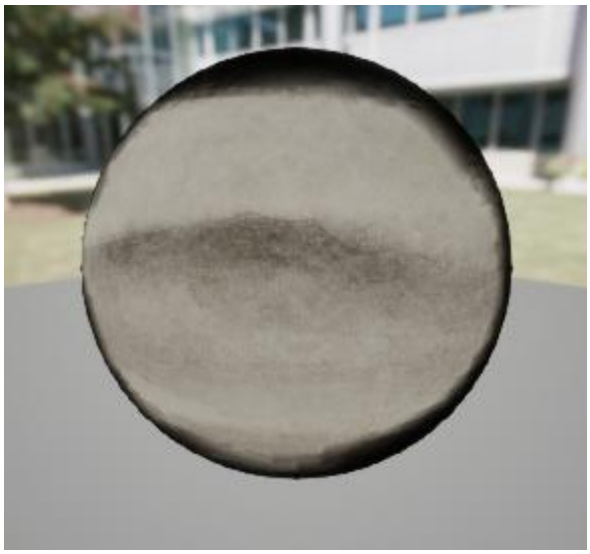


（效果图)



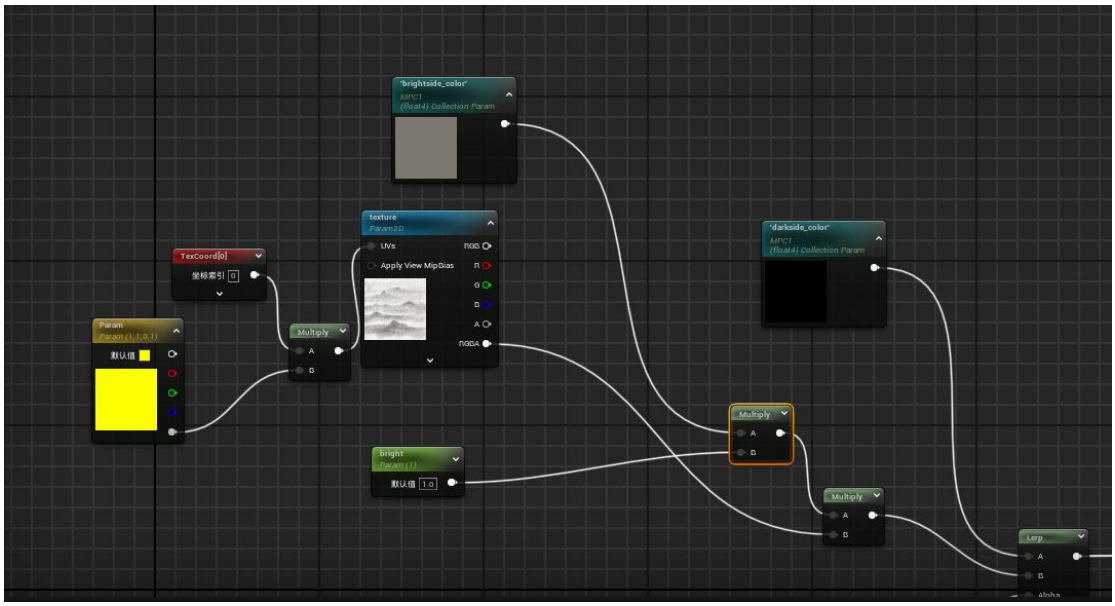
变体

皴法水墨材质

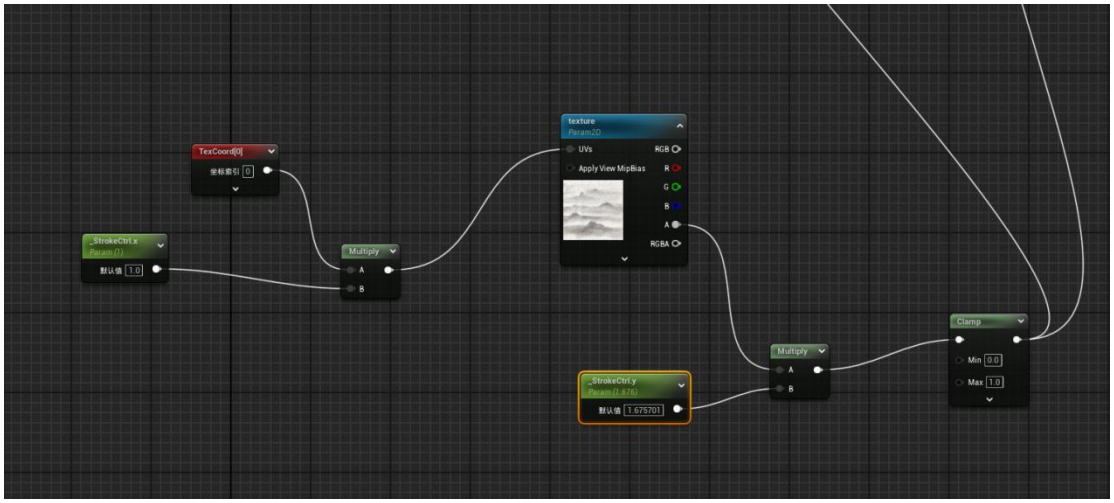


（效果图)

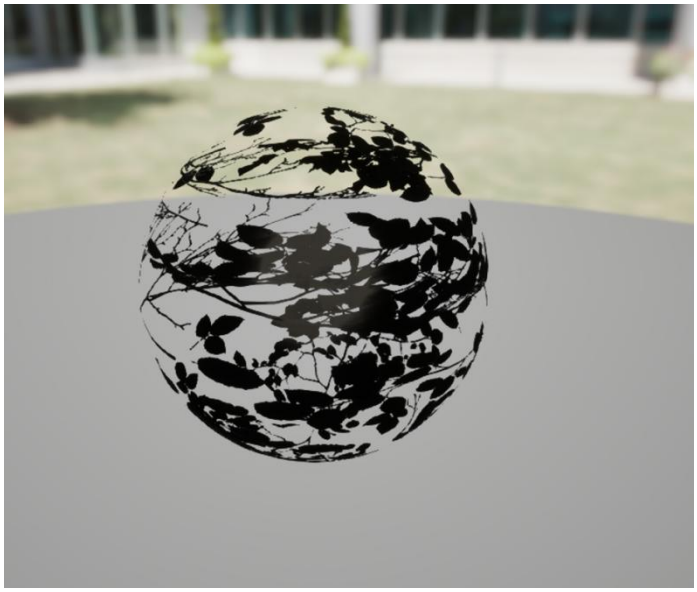
改动 1 在亮部颜色里混入了纹理，并且暴露亮部纹理坐标参数修改。



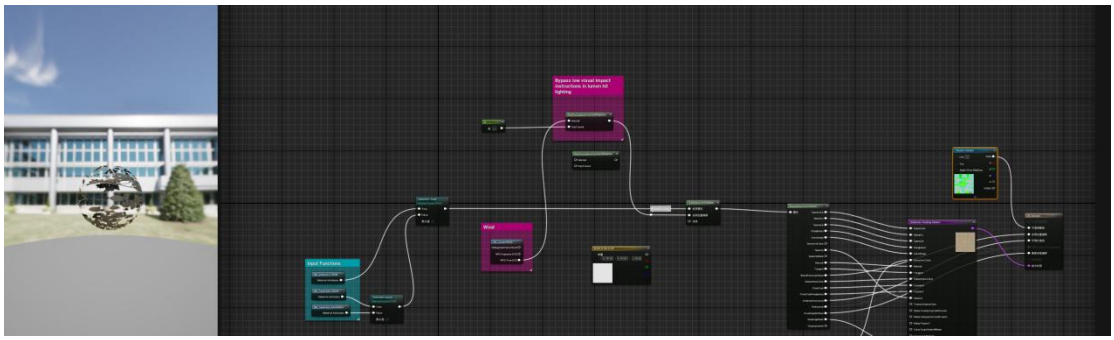
改动 2 将单纯 noise 改为纹理笔触贴图并且利用纹理坐标控制笔触 X、Y 轴的大小。



植物水墨材质



改动 1 混合双面植物材质属性和风力算法



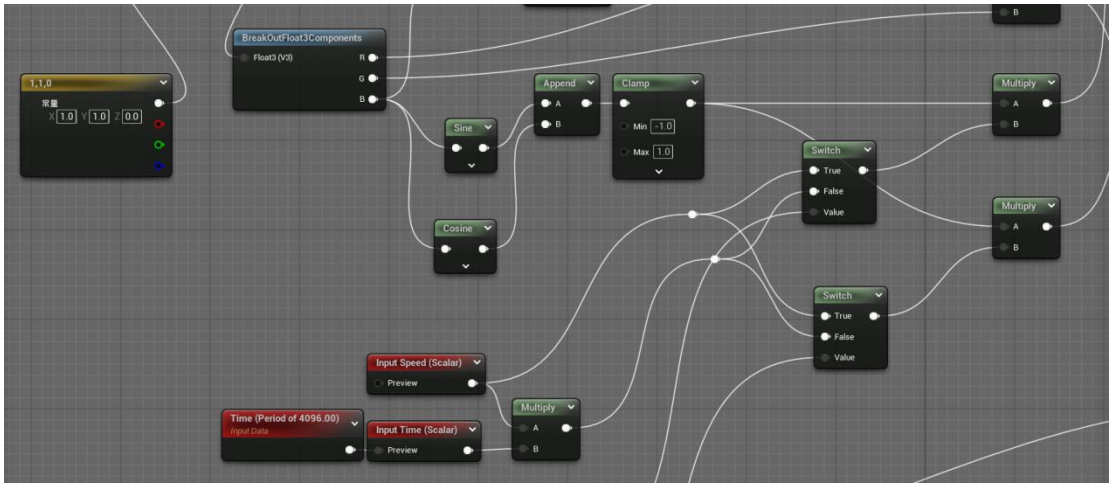
石料水墨材质

人物水墨材质

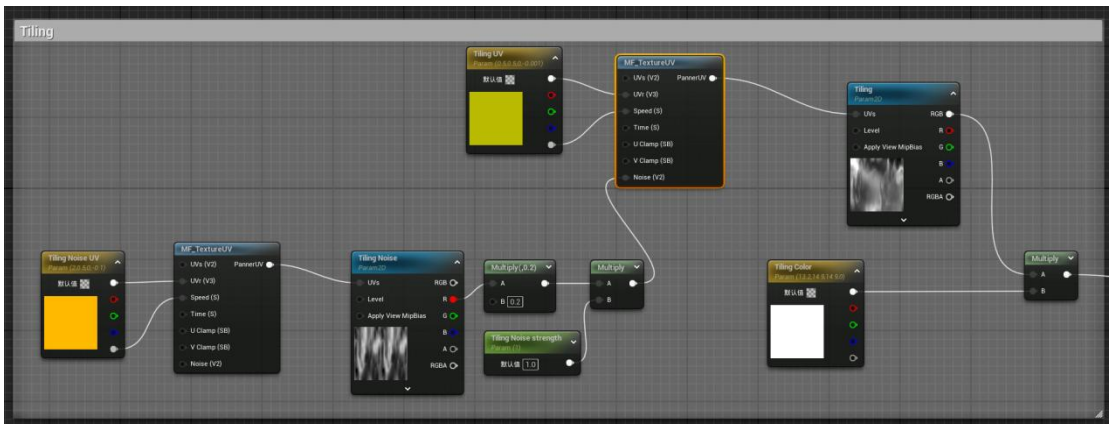
瀑布流动水材质（附着在平面上）

基础材质

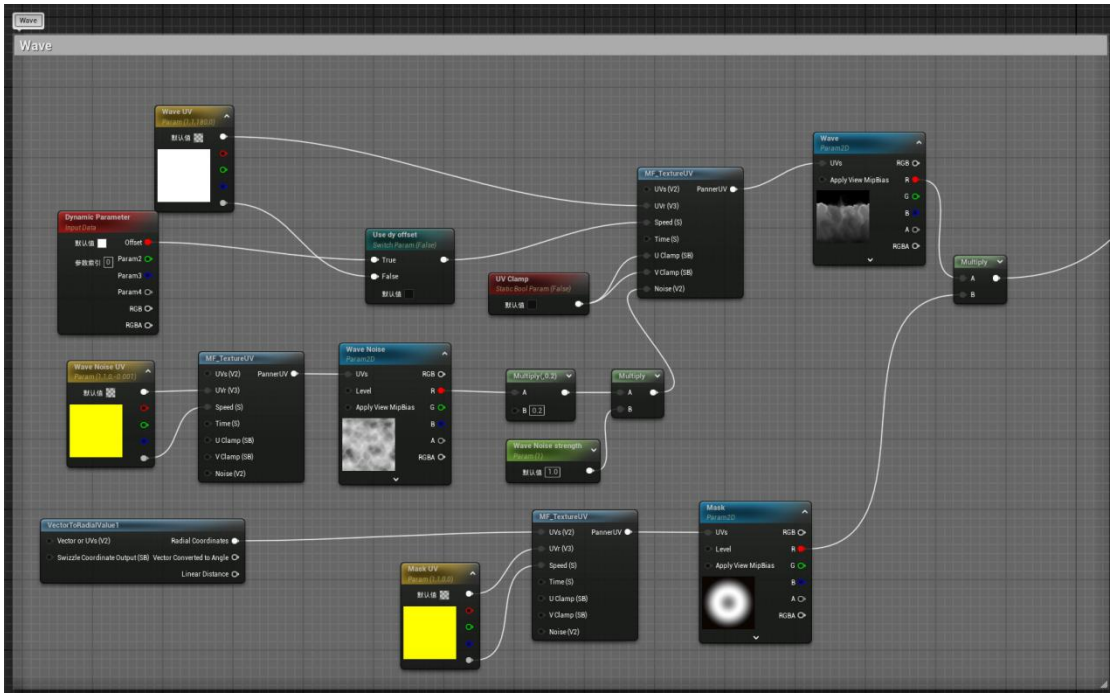
材质算法利用 sine 和 cosine 结合形成随着时间变化的流动幅度



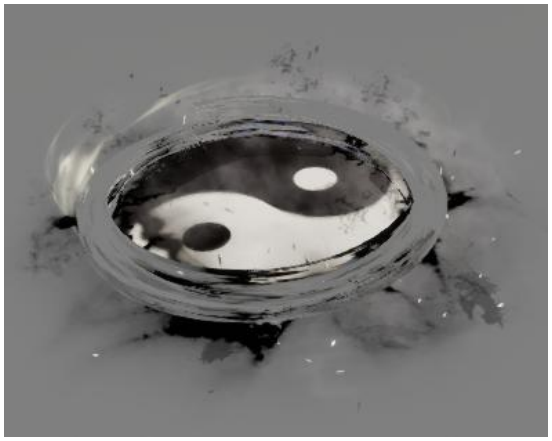
Tiling 平铺的流动形状和幅度

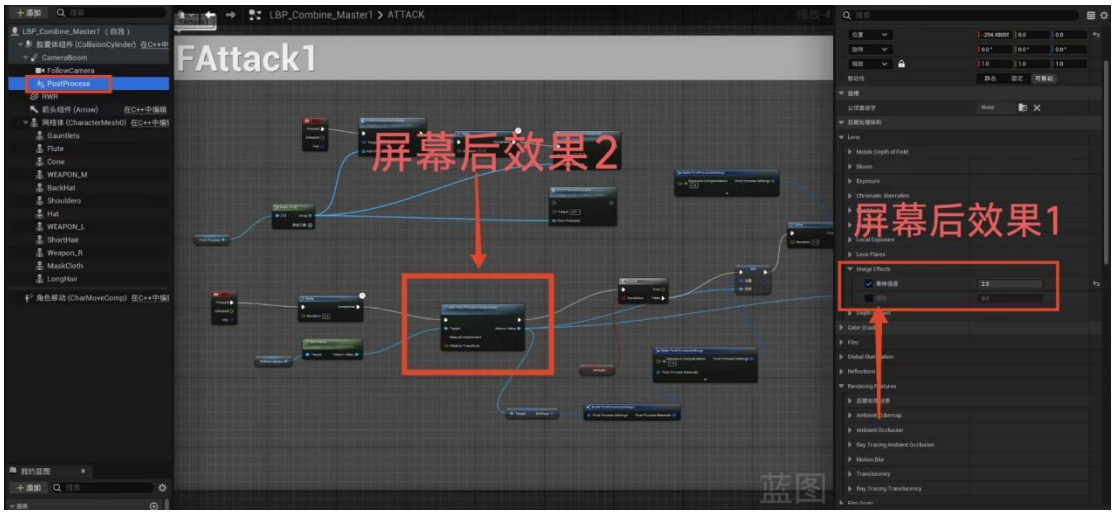


Wave 波浪的形状和速度



Niagara 系统粒子结合屏幕后效果实现技能特效，通过按键触发

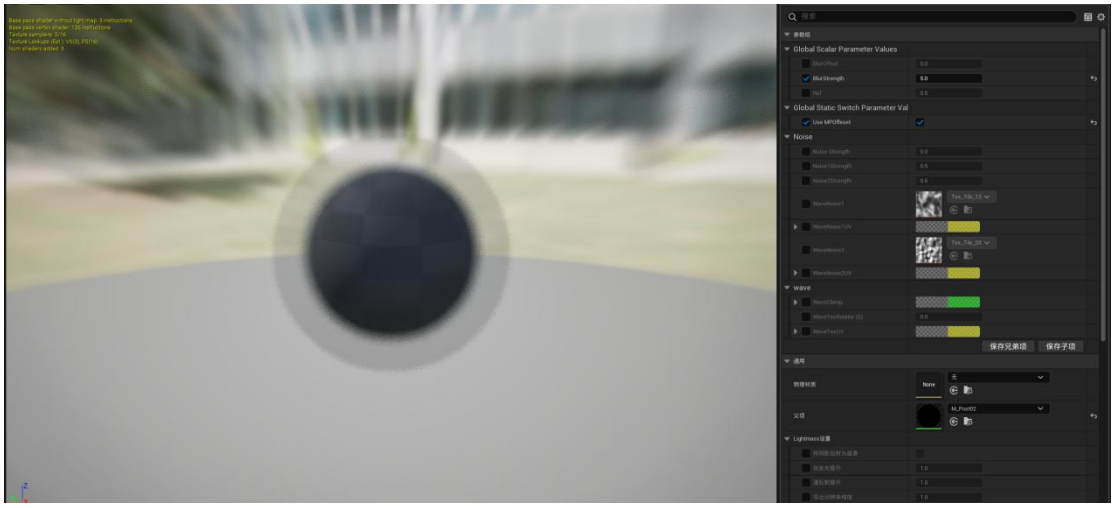




通过两个屏幕后效果与按键触发时间的时间差来配合粒子特效加强氛围感

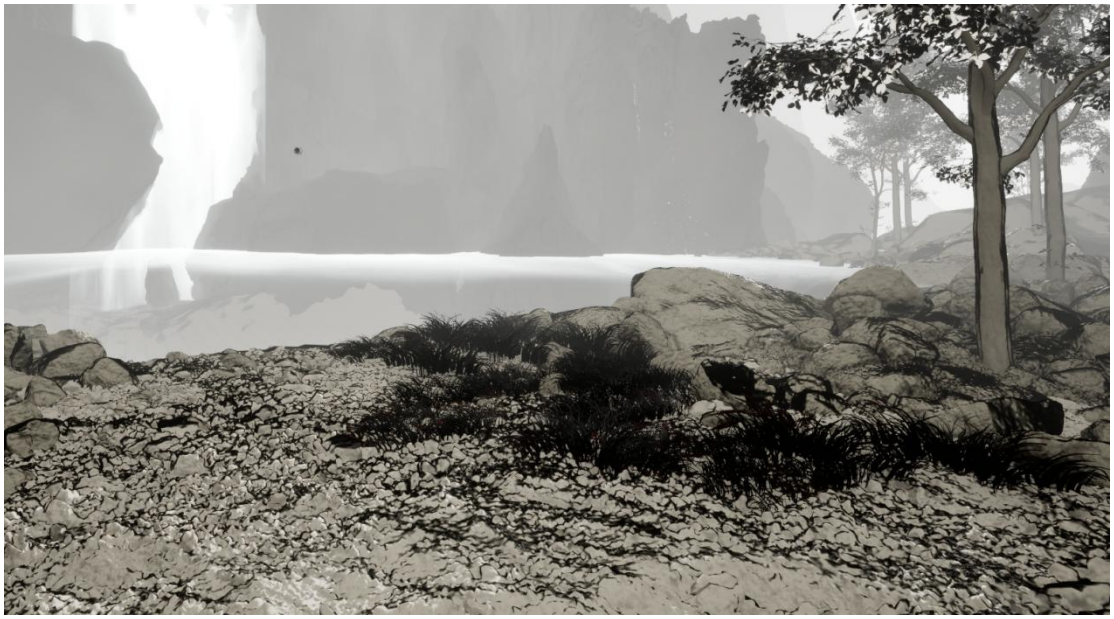
屏幕后效果 1 通过晕映强度来营造暗角

屏幕后效果 2 的材质（模糊折射屏幕）

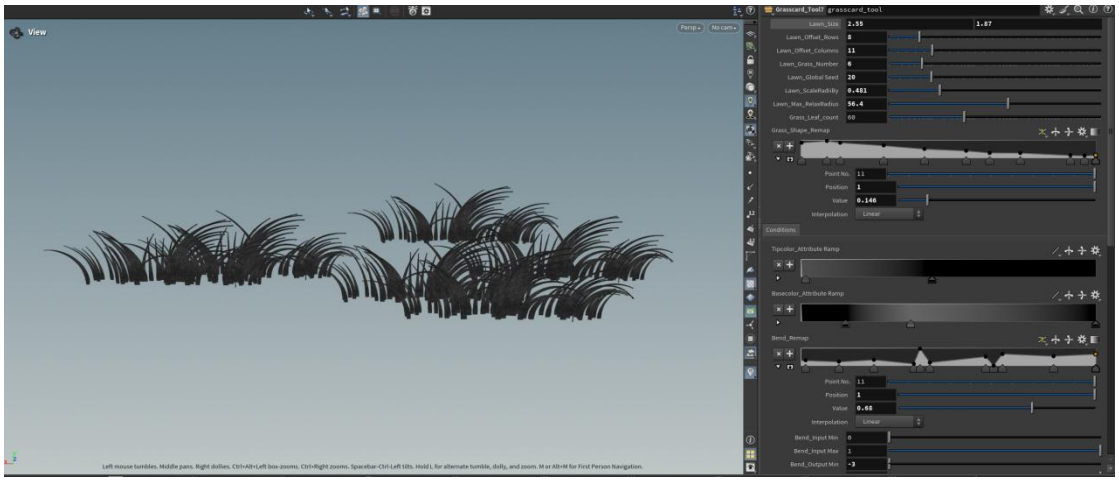


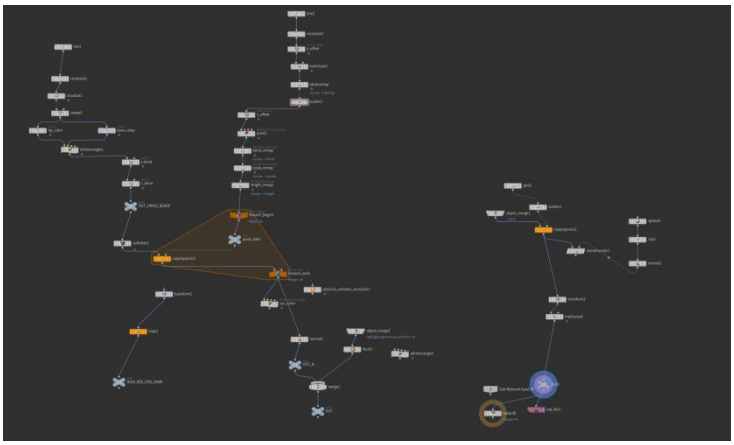
人物绑定 UE5 骨架，自定义动作以及不同动作附带的音效

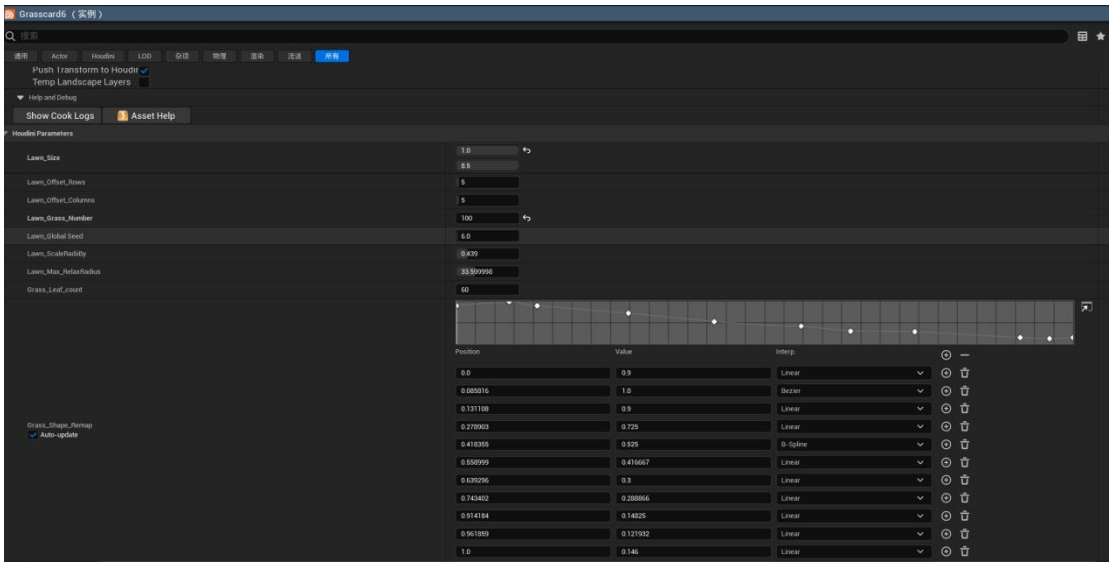
一键生成不同类型交互草地的 HDA

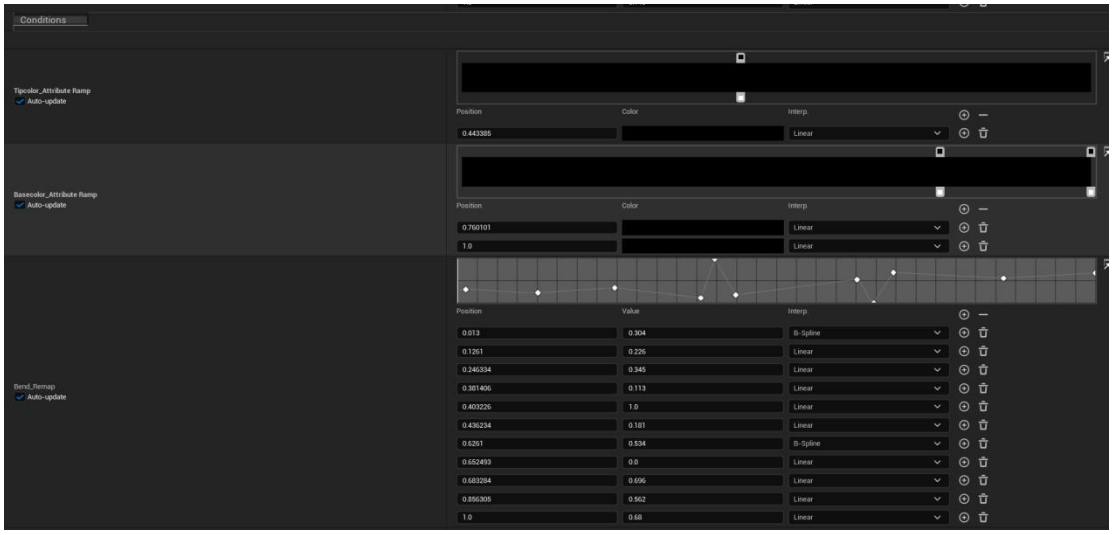


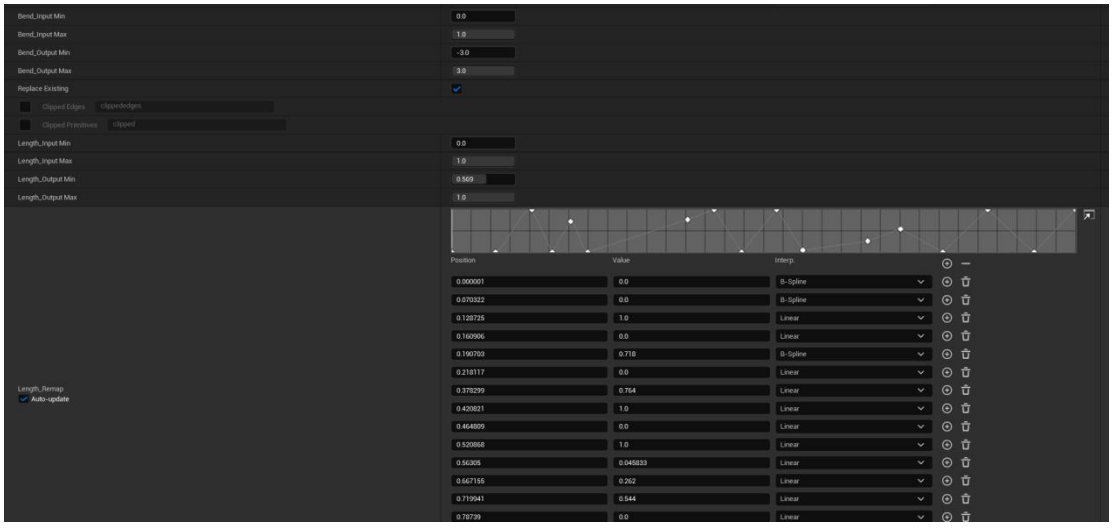
在 houdini 里面先制作单簇草丛，然后添加数量。可以在 houdini 或者 UE5 界面内随意修改草丛的叶片形状和草丛的面积数量等数值，灵活控制草丛的各种参数。

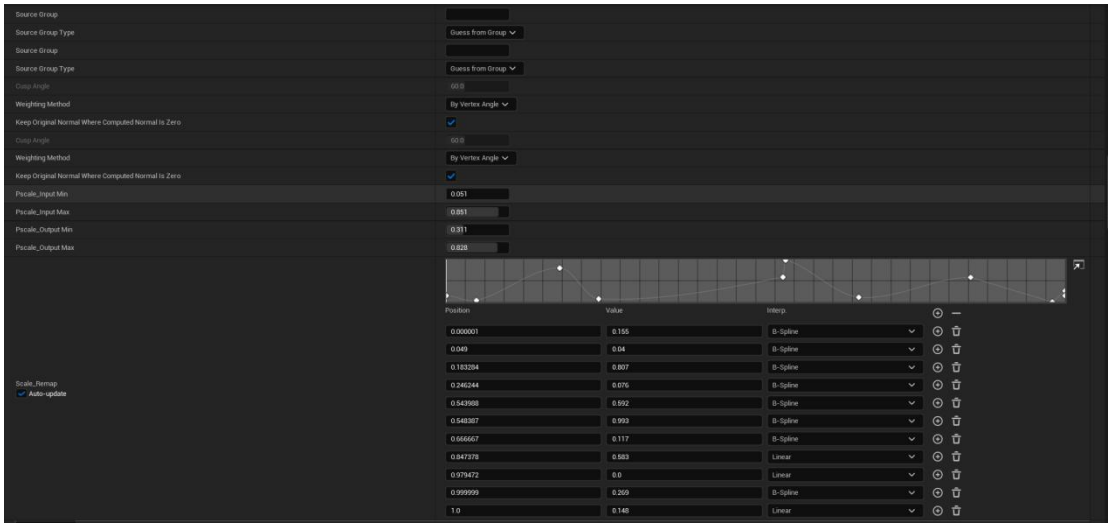





Grasscard6 （实例）
Push Transform to Houdini
Temp Landscape Layers
Help and Debug
Show Cook Logs
Asset Help
Houdini Parameters
Lawn_Size
Lawn_Offset_Rows
Lawn_Offset_Columns
Lawn_Grass_Number
Lawn_Global Seed
Lawn_ScaleRadiiBy
Lawn_Max_RelaxRadius
Grass_Leaf_count
Grass_Shape_Remap
Auto-update
Position
Value
Interp.


Conditions
Tipcolor_Attribute Ramp
Auto-update
Basecolor_Attribute Ramp
Auto-update
Bend_Remap
Auto-update
Position
Color
Value
Interp.


Bend_Input Min
Bend_Input Max
Bend_Output Min
Bend_Output Max
Replace Existing
Length_Input Min
Length_Input Max
Length_Output Min
Length_Output Max
Length_Remap
Auto-update
Position
Value
Interp.

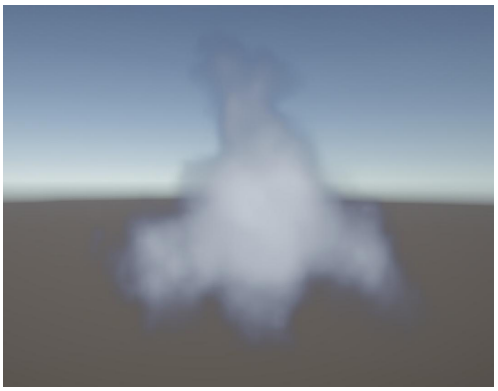


草丛会和人物产生交互反应，并且被风力影响。

UNITY Shader

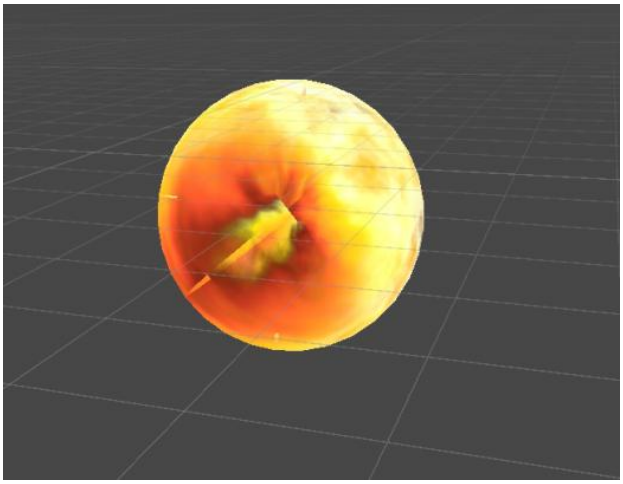
Raymarching shader

通过体积采样、光照累积、透光量得出最终结果。



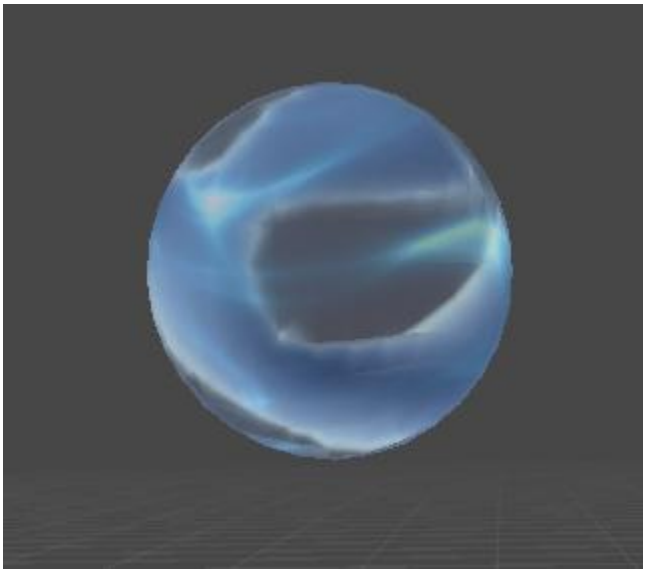
火焰 shader

以底部中心为原点向外以高度为标准扩散，添加噪声模拟火焰，三种颜色（红橘黄）平滑插值过渡到完全透明，添加光线步进，散射效果。



水球 shader

动态 UV 噪声和世界噪声结合扭曲得出最终顶点位置偏移，uv 噪声动态扭曲法线，计算水球颜色和反射然后混合。

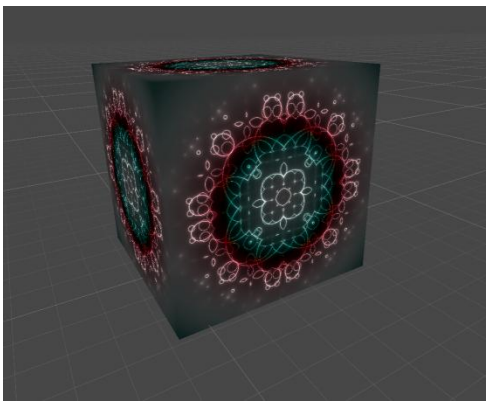


水墨扩散粒子效果 shader

光线追踪每刻更新累计颜色，通过 field、field2 和 outershape 函数确定 shader 扩散的整体形状。

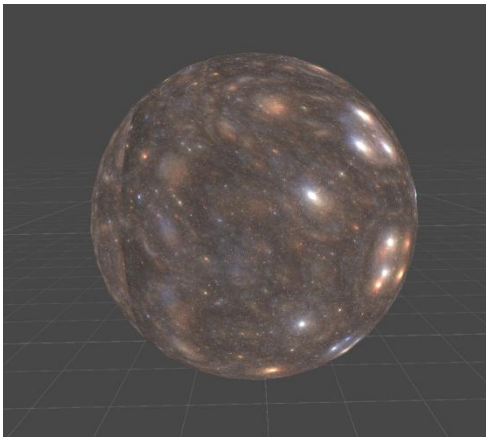
奇幻风格 shader

通过 sin 和 cos 函数生成线条和平滑的颜色过渡，并且不断循环，随着时间产生动态渐变。

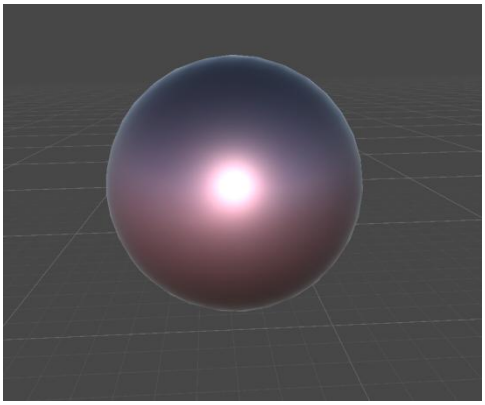


银河风格 shader

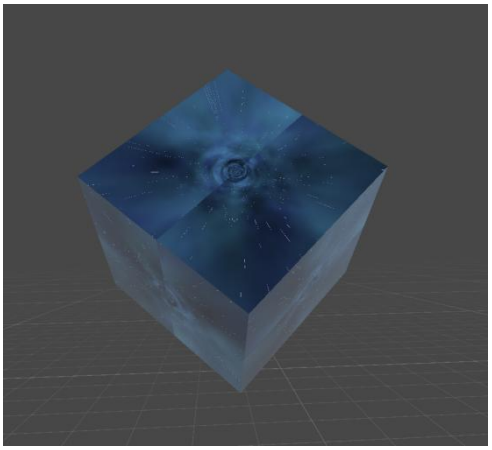
通过体积渲染逐步计算每个点的颜色和透明度，在每个位置叠加来得到最终的颜色。abs(p) / dot(p, p) - formuparam 模拟银河物理分形形态。



珍珠质感 shader
利用 hash 函数随机生成噪声然后平滑插值过渡，使用世界位置和时间信息来动态生成表面变化，模拟珍珠表面的不规则性。噪声值被缩放为一定强度后应用到基色上，形成视觉上的扭曲。



星空 shader
通过极坐标（半径和角度）将纹理坐标转化为环形的模式，模拟了隧道的效果。利用噪声函数、FBM（分形布朗运动）以及隧道效果合成最终颜色。



第一人称交互 UI 系统

实现效果

UI 响应站立和蹲下动作切换

按下 c 键蹲下时，脚本会捕捉输入事件然后切换 UI 的可见性。使用虚拟相机的位置锚点来修改镜头的高度

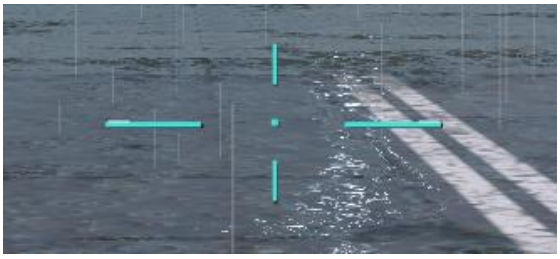
```
// 检查蹲下键是否被按下
if (Input.GetButtonDown("Squat") && !isCrouching)
{
    // 按下蹲下键且当前不是蹲下状态时，开始蹲下
    Debug.Log("Squat button pressed, toggling crouch state.");

    targetPosition.y = originalHeight - lowerHeight;
    isCrouching = true;  // 设置为蹲下状态

    isIconToggled = true;  // 标记为已按下

    // 切换图标状态
    standingIcon.enabled = false;  // 隐藏站立图标
    crouchingIcon.enabled = true;  // 显示蹲下图标
}
else if (Input.GetButtonDown("Squat") && isCrouching)
{
    // 按下蹲下键且当前是蹲下状态时，恢复站立
    Debug.Log("Squat button pressed, toggling crouch state.");
    targetPosition.y = originalHeight;  // 恢复到原始位置
    isCrouching = false;  // 设置为站立状态
    isIconToggled = false;  // 标记为未按下

    // 切换图标状态
    standingIcon.enabled = true;  // 显示站立图标
    crouchingIcon.enabled = false;  // 隐藏蹲下图标
}
Vector3 newPosition = PlayerCapsule.transform.localPosition;
newPosition.y = Mathf.Lerp(newPosition.y, targetPosition.y, smoothSpeed * Time.deltaTime);

// 更新PlayerCapsule的localPosition
PlayerCapsule.transform.localPosition = newPosition;
```

射击瞄准准星

按下鼠标左键瞄准键会有缩放和渐变的效果。如果检查到了调用瞄准键，那么会使用方法 OnAim 来判断是否需要切换动画进入瞄准状态，进入瞄准状态以后准星会缩小，渐渐变透明。

```
public void OnAim(bool aim)
{
    if (!Interactable)
        return;

    isAim = aim;
}
```

方法 OnFire 来限制缩放频率。

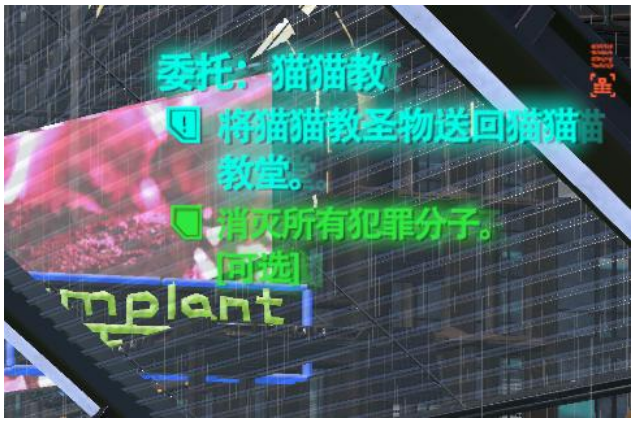


```
public void OnFire()
{
    if (!Interactable)
        return;
    if (Time.time < lastTimeFire)
        return;

    Vector2 size = new Vector2(GetCrosshair.OnFireScaleAmount, GetCrosshair.OnFireScaleAmount);
    RootContent.sizeDelta = size;
    lastTimeFire = Time.time + OnFireScaleRate;
}
```

完成击杀任务以后 UI 会变成完成状态

通过公开暴露变量获取 UI 组件来控制显示和隐藏

```
void FixedUpdate()
{
    if (!isInited) return;

    // 计算下一步位置
    Vector3 nextPos = transform.position + velocity * Time.fixedDeltaTime;
    Vector3 move = nextPos - transform.position;

    // 检测沿途碰撞
    if (Physics.Raycast(transform.position, move.normalized, out RaycastHit hit, move.magnitude, hittableLayers, QueryTriggerInteraction.Ignore))
    {
        // 碰到目标
        transform.position = hit.point;
        if (hit.transform.CompareTag("Target"))
        {
            var anim = hit.transform.GetComponent<Animator>();
            if (anim != null)anim.Play("death");

                if (taskIcon != null)
                {
                    taskIcon.enabled = false;  // 隐藏原始图标
                }

                if (targetIcon != null)
                {
                    targetIcon.enabled = true;  // 显示目标图标
                }
                // 切换UI图标，隐藏站立图标，显示目标图标
                if (taskIcon2 != null)
                {
                    taskIcon2.enabled = false;  // 隐藏原始图标
                }

                if (targetIcon2 != null)
                {
                    targetIcon2.enabled = true;  // 显示目标图标
                }


        }
        Destroy(gameObject,0.1f);  // 击中后销毁子弹
        return;
    }
```

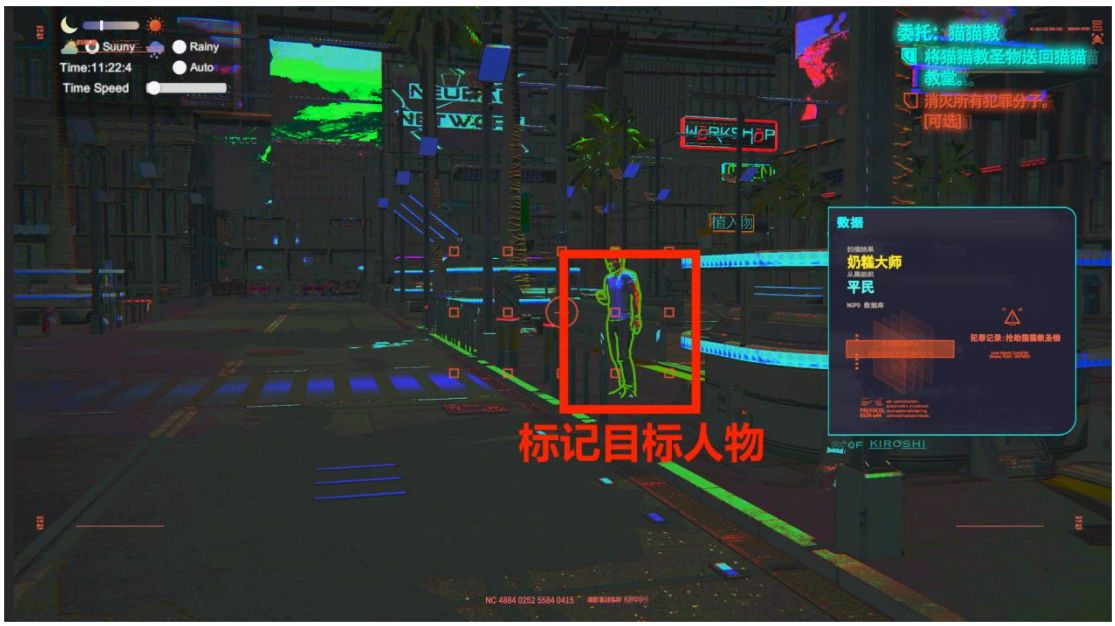


按“tab”键切换至扫描系统，显示人物信息，使用外轮廓线标记目标人物

在方法 update 里面持续更新关于 tab 键的信息，默认处于射击 UI 界面，如果按下 tab 键则进入扫描界面。当进入扫描模式以后，会开始获取镜头里的物体材质，如果发现物体的第[1]个材质插槽属于特定人物目标的材质，那么就会将材质切换为外轮廓材质。

```
public void Update()
{

    // 检测 Tab 键按下事件
    if (Input.GetKeyDown(KeyCode.Tab))  // 如果按下 Tab 键
    {
        if (tabKeyPressedOnce)  // 如果是第二次按下 Tab 键，停止扫描
        {
            isScanning = false;
            materialChangeTime = Time.time + scanDuration;
            RevertMaterial();  // 恢复材质
            tabKeyPressedOnce = false;  // 重置标记

            if (akTexturedObject != null)
            {
                isAKTextured = akTexturedObject.activeInHierarchy;  // 获取物体是否在层级中激活

                akTexturedObject.SetActive(!akTexturedObject.activeSelf);
                Debug.Log("isAKTextured toggled: " + isAKTextured);
            }

            if (globalVolumeObject != null)
            {
                //isGlobalVolume = globalVolumeObject.isActiveAndEnabled;  // 获取 Volume 组件的启用状态
                globalVolumeObject.enabled = !globalVolumeObject.enabled;
                Debug.Log("isGlobalVolume toggled: " + isGlobalVolume);
            }
            if (materialChangeTime >= scanDuration)
            {
                RevertMaterial();
                //materialChangeTime = 0f;
            }
            Debug.Log("第二次按下 Tab 键，停止扫描。");
            Debug.Log("materialChangeTime: " + materialChangeTime);  // 输出 materialChangeTime
            ShootUI.gameObject.SetActive(true);  // 启用 ShootUI
            ScanUI.gameObject.SetActive(false);  // 禁用 ScanUI


        }
        else  // 第一次按下 Tab 键，开始扫描
        {
            isScanning = true;
            tabKeyPressedOnce = true;  // 设置标记为已按下
            Debug.Log("第一次按下 Tab 键，开始扫描。");
            ToggleMaterial();  // 切换材质

            if (akTexturedObject != null)
            {
                isAKTextured = akTexturedObject.activeInHierarchy;  // 获取物体是否在层级中激活

                akTexturedObject.SetActive(!akTexturedObject.activeSelf);
                Debug.Log("isAKTextured toggled: " + isAKTextured);
            }

            if (globalVolumeObject != null)
            {
                //isGlobalVolume = globalVolumeObject.isActiveAndEnabled;  // 获取 Volume 组件的启用状态
                globalVolumeObject.enabled = !globalVolumeObject.enabled;
                Debug.Log("isGlobalVolume toggled: " + isGlobalVolume);
            }
            ShootUI.gameObject.SetActive(false);  // 禁用 ShootUI
            ScanUI.gameObject.SetActive(true);   // 启用 ScanUI
        }
    }

    // 如果扫描已启用，执行扫描逻辑
    if (!isScanning) { currentFrame = 0; return; }

    // 帧数控制逻辑
    currentFrame = (currentFrame + 1) % CheckFrameRate;

    // 扫描时间到后恢复材质

}
```

扫描界面添加了 Volume Global 的后处理效果，模拟紫外线识别人物的效果。



人物行走时，自然的晃动效果

分为两个方面，一个是鼠标摆动会影响手部晃动，一个是移动时也会晃动。使用 Perlin 噪声震动效果，使得武器的偏移更自然、随机，模拟真实的步伐。当进入瞄准状态会减少摆动幅度。暴露变量方便调整摆动幅度。

```
void UpdateSway()
{
    float deltaTime = Time.smoothDeltaTime;

    // 实时读取输入（支持持续移动和鼠标sway）
    Vector2 moveInput = inputs.moveRealTime;
    Vector2 lookInput = inputs.lookRealTime;

    // 鼠标视角 sway
    float swayX = -lookInput.x * swayAmount  * amplitudeMultiplier;
    float swayY = -lookInput.y * swayAmount  * amplitudeMultiplier;

    swayX = Mathf.Clamp(swayX, -maxSway, maxSway);
    swayY = Mathf.Clamp(swayY, -maxSway, maxSway);

    // 移动 sway
    float moveZ = -moveInput.y * movePushAmount * amplitudeMultiplier;
    float moveX = -moveInput.x * movePushAmount * 0.5f * amplitudeMultiplier;

    shakeOffsetX = Mathf.PerlinNoise(0, Time.time * shakeFrequency) * 2 - 1; // 生成 -1 到 1 之间的值
    shakeOffsetY = Mathf.PerlinNoise(1, Time.time * shakeFrequency) * 2 - 1; // 生成另一个独立的随机值

    // 应用震动（震动幅度和移动输入幅度相关）
    shakeOffsetX *= shakeAmount * moveInput.magnitude;
    shakeOffsetY *= shakeAmount * moveInput.magnitude;

    // 计算最终目标位置和旋转
    Vector3 targetPosition = initialPos + new Vector3(swayX + moveX + shakeOffsetX, swayY * 0.5f + shakeOffsetY,moveZ);
    Quaternion targetRotation = Quaternion.Euler(swayY * 80f,-swayX * 80f,moveInput.x * -4f);

    // 最终偏移和旋转
    //Vector3 targetPosition = initialPos + new Vector3(swayX + moveX, swayY * 0.5f, moveZ);
    //Quaternion targetRotation = Quaternion.Euler(swayY * 80f, -swayX * 80f, moveInput.x * -4f);

    transform.localPosition = Vector3.Lerp(transform.localPosition, targetPosition, deltaTime * positionSmooth);
    transform.localRotation = Quaternion.Slerp(transform.localRotation, initialRot * targetRotation, deltaTime * rotationSmooth);
}
```

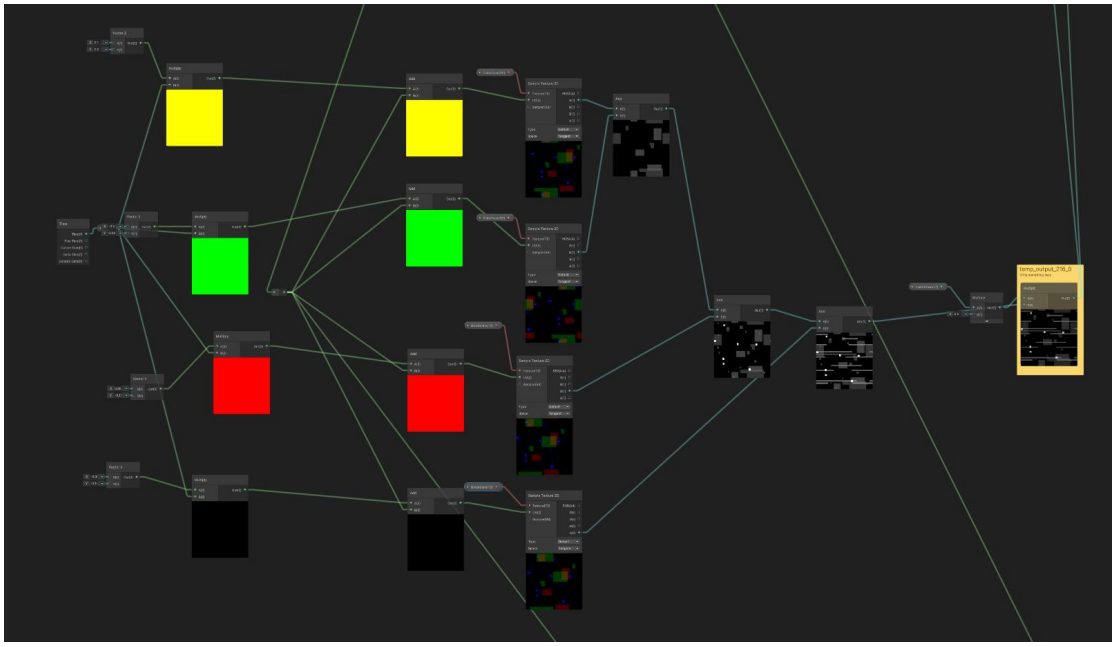
按键射击子弹命中触发人物死亡。

```
// 检测沿途碰撞
if (Physics.Raycast(transform.position, move.normalized, out RaycastHit hit, move.magnitude, hittableLayers, QueryTriggerInteraction.Ignore))
{
    // 碰到目标
    transform.position = hit.point;
    if (hit.transform.CompareTag("Target"))
    {
        var anim = hit.transform.GetComponent<Animator>();
        if (anim != null)anim.Play("death");
```

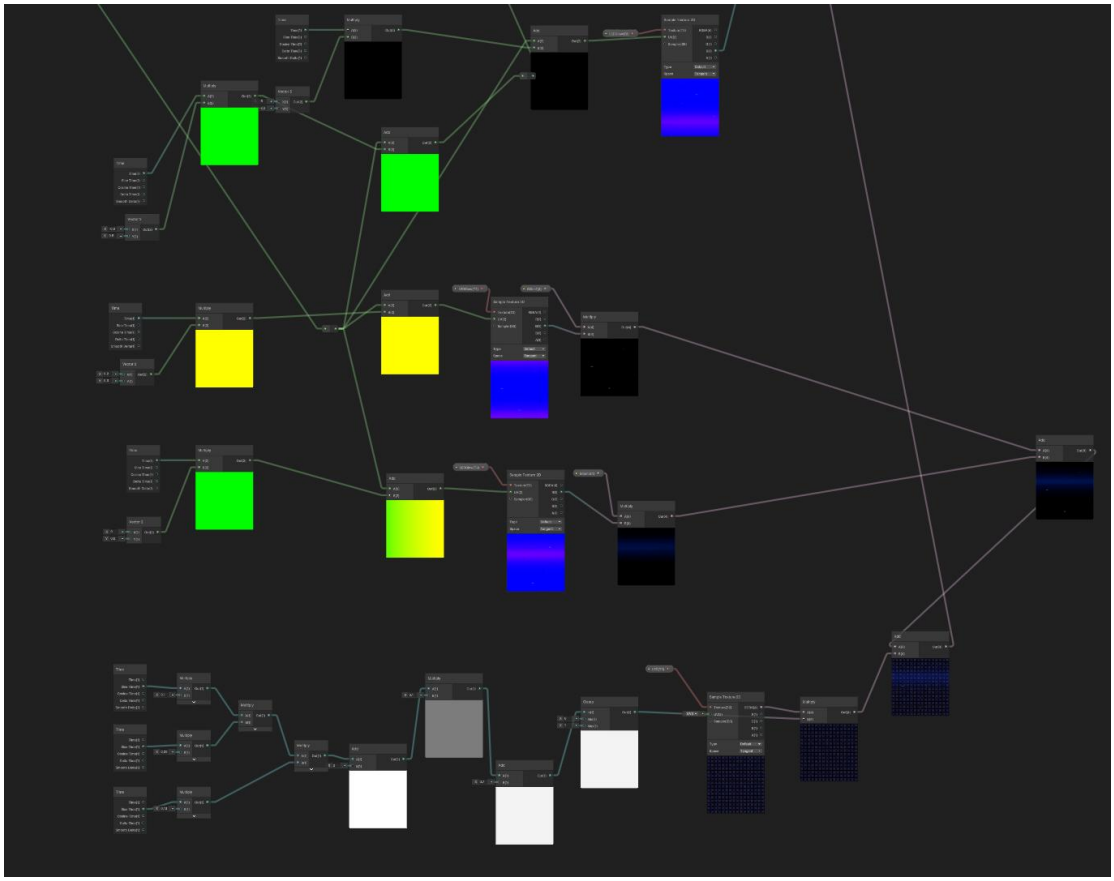
街道广告牌材质着色器制作

（1）Billboard 模拟一个带有动画效果的广告牌，包括闪烁的 LED 效果、背景和文字的失真效果。

失真效果通过 Distortions 纹理和 DistortPower 控制，会影响背景纹理的 UV 坐标，创建类似电子屏幕显示问题的视觉效果。



LED 效果通过_LED 纹理来模拟，结合 LEDGlow 实现动态的发光效果。通过控制 GlitchIntensity，可以模拟 LED 的闪烁或故障效果。

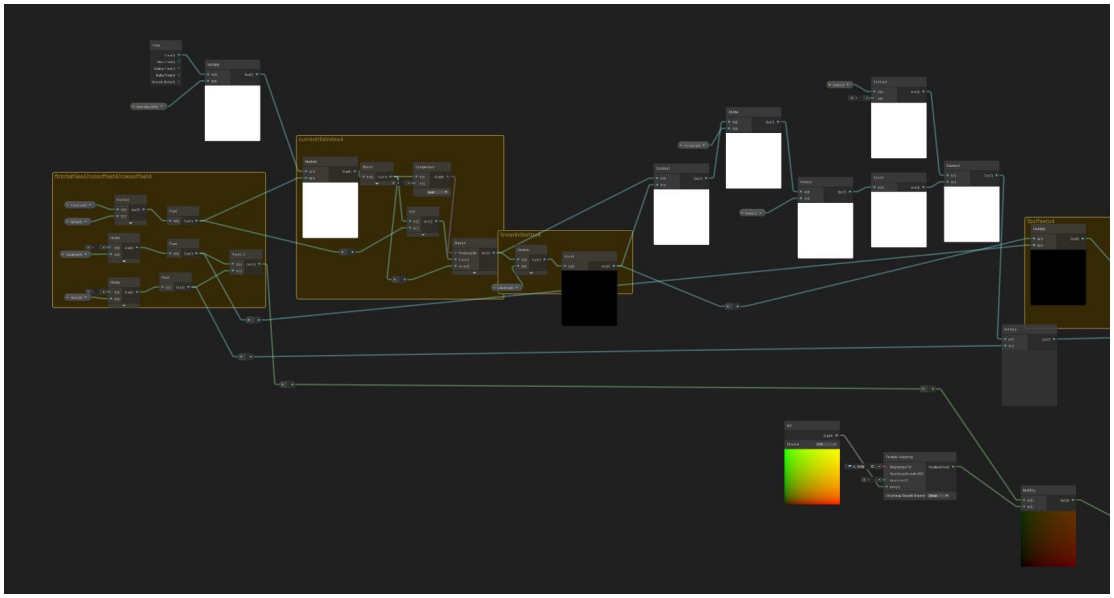


故障（Glitch）效果，通过 Glitch1 和 Glitch2，结合 LEDGlow 的动态变化，模拟广告牌在工作时的故障效应。GlitchIntensity 控制了这些故障效果的强度，带来视觉上的干扰或闪烁效果。

（2）Moive 播放材质着色器制作

使用 **FlipBook** 方法，Animation 纹理结合 Columns 和 Rows 将电影内容拆分为多个小块。每个小块表示动画中的一帧，使用 UV 偏移和时间控制帧的切换。动画的播放速度由 MovieSpeed 控制。通过调整时间和 UV 偏移，实现逐帧播放的效果。

失真和发光效果实现原理和 billboard 着色器一致



-

天气管理系统（适用于 URP）

时间控制

24 小时循环的时间系统影响光的旋转，驱动光源旋转，模拟昼夜变化

使用方法 AutoTimeEnabledChanged()当选中“自动”时触发。如果启用自动时间流逝，定向光开始自动旋转旋转。

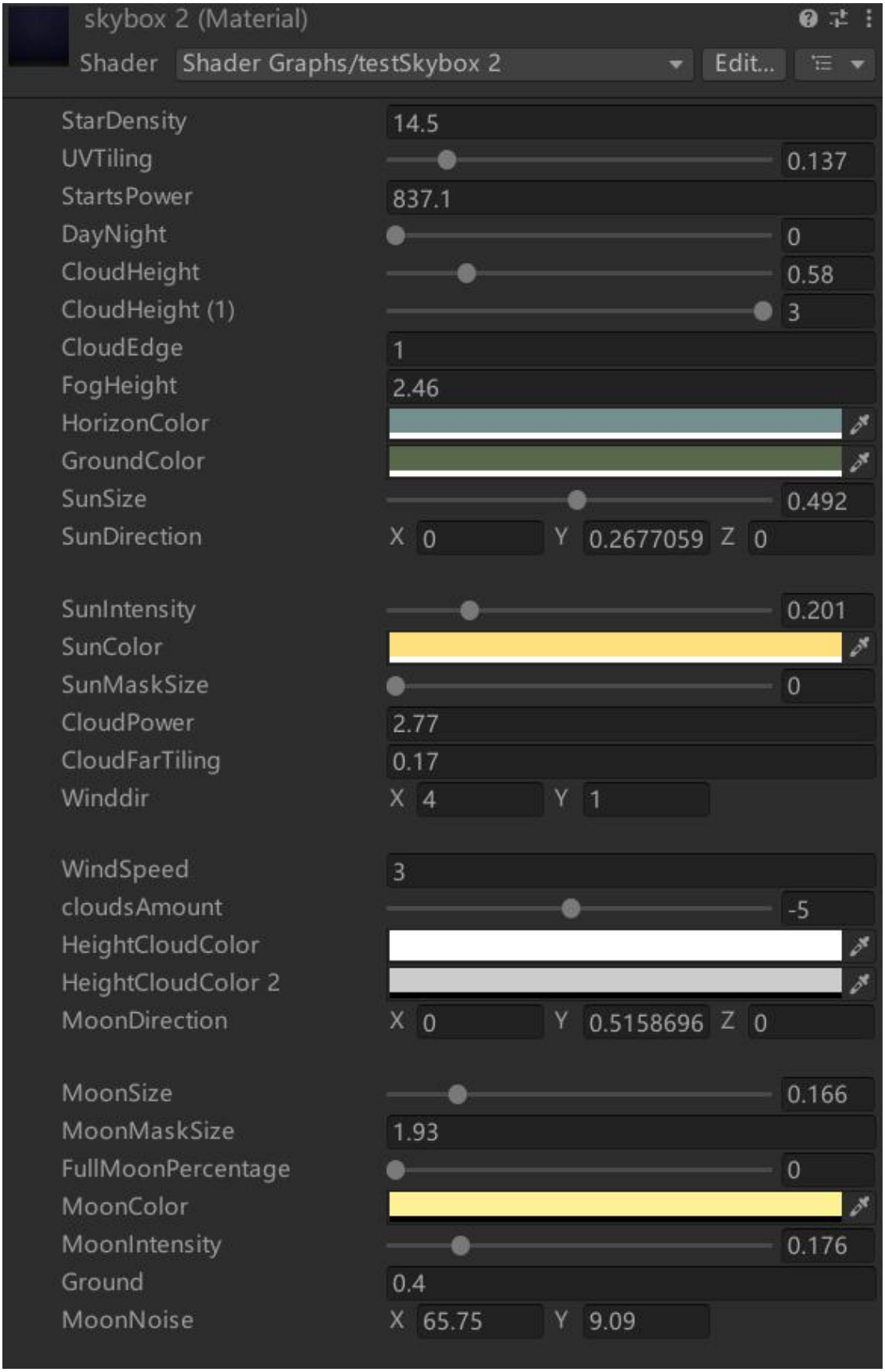
方法 RotateDirectionalLight()这个函数根据时间和设置的速度，模拟昼夜变化，旋转定向光。每秒增加 secondsAngle * TimeSpeed 的角度来更新光源的旋转，并确保角度超过 360 度时重新从 0 度开始。

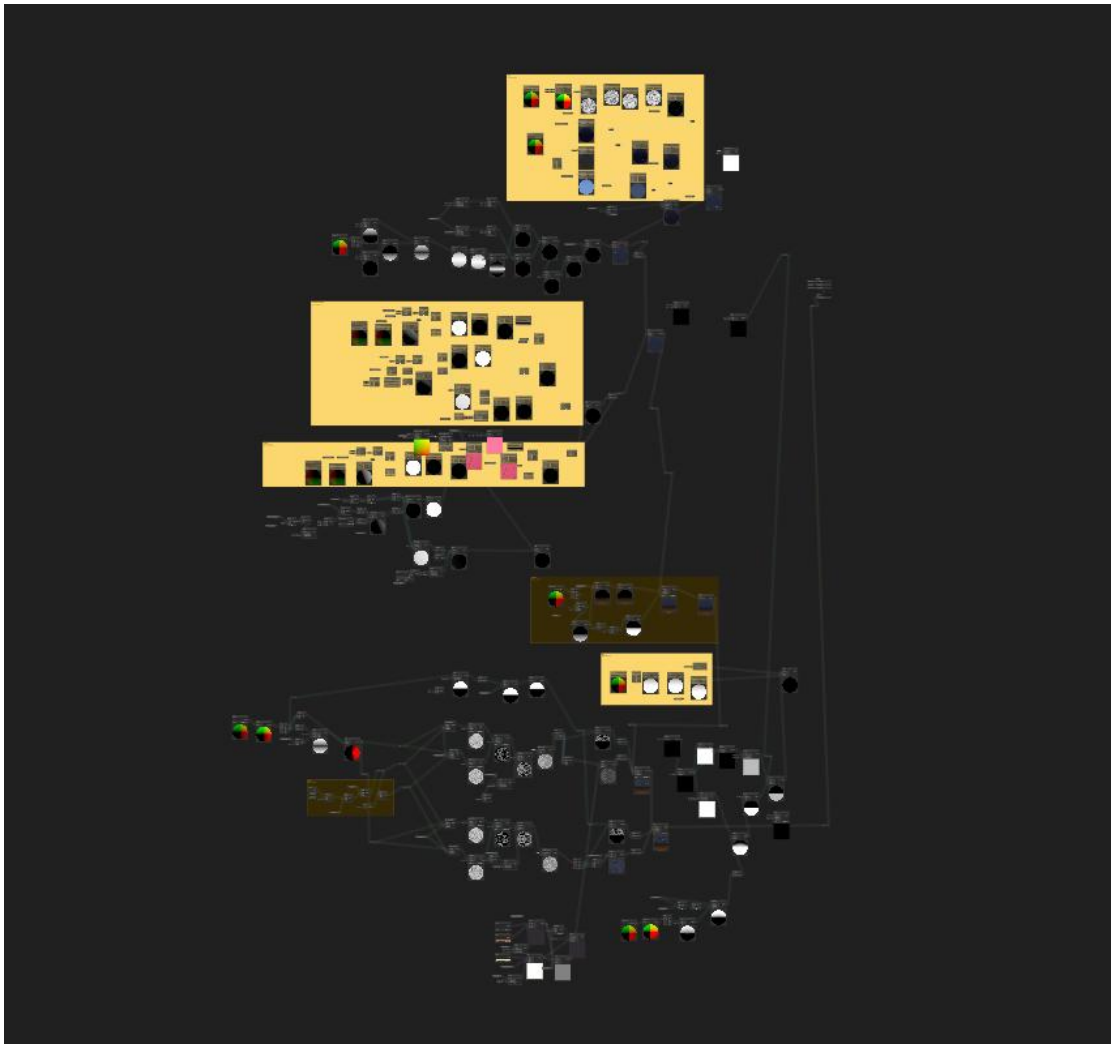
```
public void RotateDirectionalLight()
{
    //TimeSpeed2 = TimeSpeed2 * 0.1f;
    float adjustedAngle = secondsAngle * TimeSpeed * 0.1f;
    // 每帧更新总时间
    totalTime += Time.deltaTime * TimeSpeed;

    // 每秒旋转 0.25f 度
    currentAngle += adjustedAngle;

    if (currentAngle >= 360f)
    {
        currentAngle -= 360f; // 超过 360 度时重新从 0 开始
    }


    // 更新 light 的旋转
    directionalLight.transform.rotation = Quaternion.Euler(currentAngle, 0, 0);

}
```

每当分钟数超过 60 时，增加小时数，并确保小时数在 24 小时制内循环

天空盒

昼夜变化，灵活调整多种变量数据

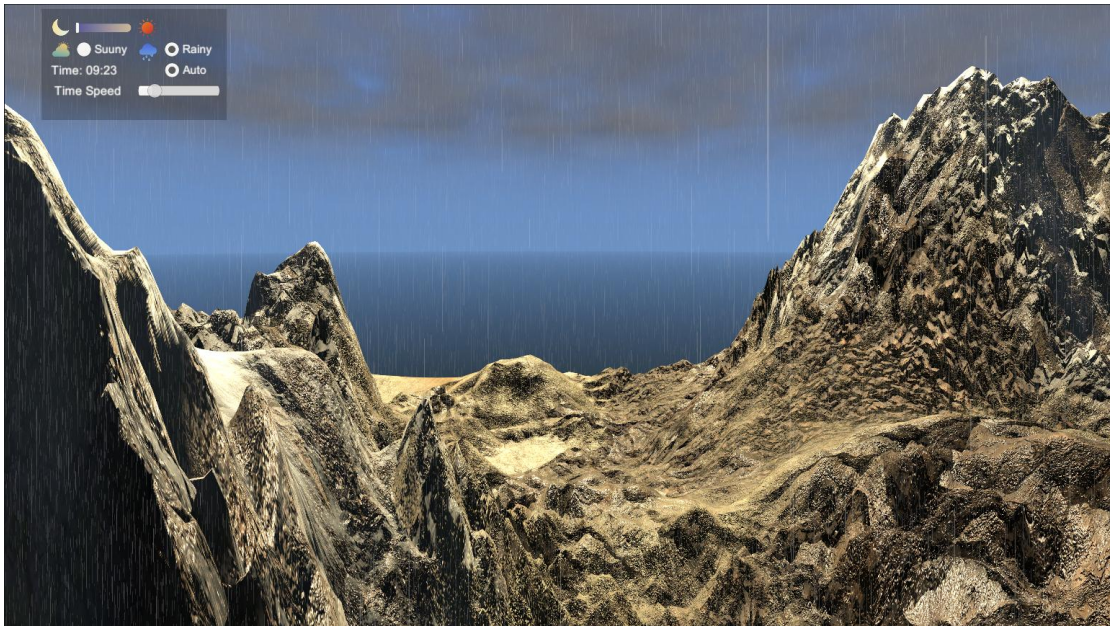

skybox 2 (Material)
Shader
Shader Graphs/testSkybox 2
Edit...
StarDensity
14.5
UVTiling
0.137
StartsPower
837.1
DayNight
0
CloudHeight
0.58
CloudHeight (1)
3
CloudEdge
1
FogHeight
2.46
HorizonColor
GroundColor
SunSize
0.492
SunDirection
X 0
Y 0.2677059
Z 0
SunIntensity
0.201
SunColor
SunMaskSize
0
CloudPower
2.77
CloudFarTiling
0.17
Winddir
X 4
Y 1
WindSpeed
3
cloudsAmount
-5
HeightCloudColor
HeightCloudColor 2
MoonDirection
X 0
Y 0.5158696
Z 0
MoonSize
0.166
MoonMaskSize
1.93
FullMoonPercentage
0
MoonColor
MoonIntensity
0.176
Ground
0.4
MoonNoise
X 65.75
Y 9.09



环境管理系统

管理天气环境数据，对周围游戏运行环境产生影响。

下雨粒子系统（VFX）的播放和关闭



材质 shader 制作（主要使用 shader graph 完成制作）

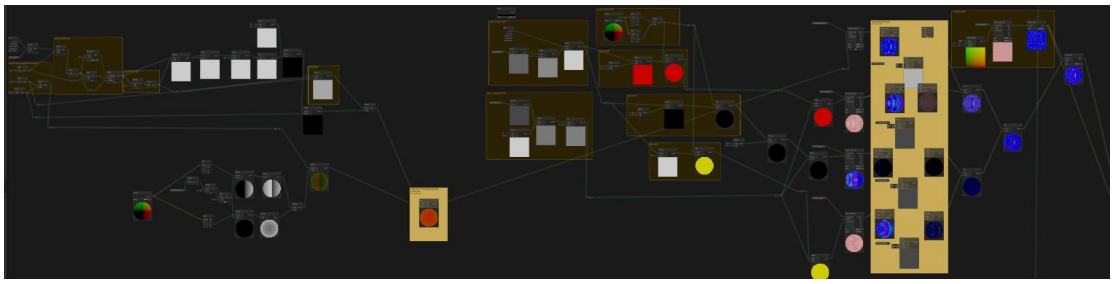
天晴时使用正常光照材质（lit）

下雨时切换成下雨材质

模拟下雨时地面材质逻辑

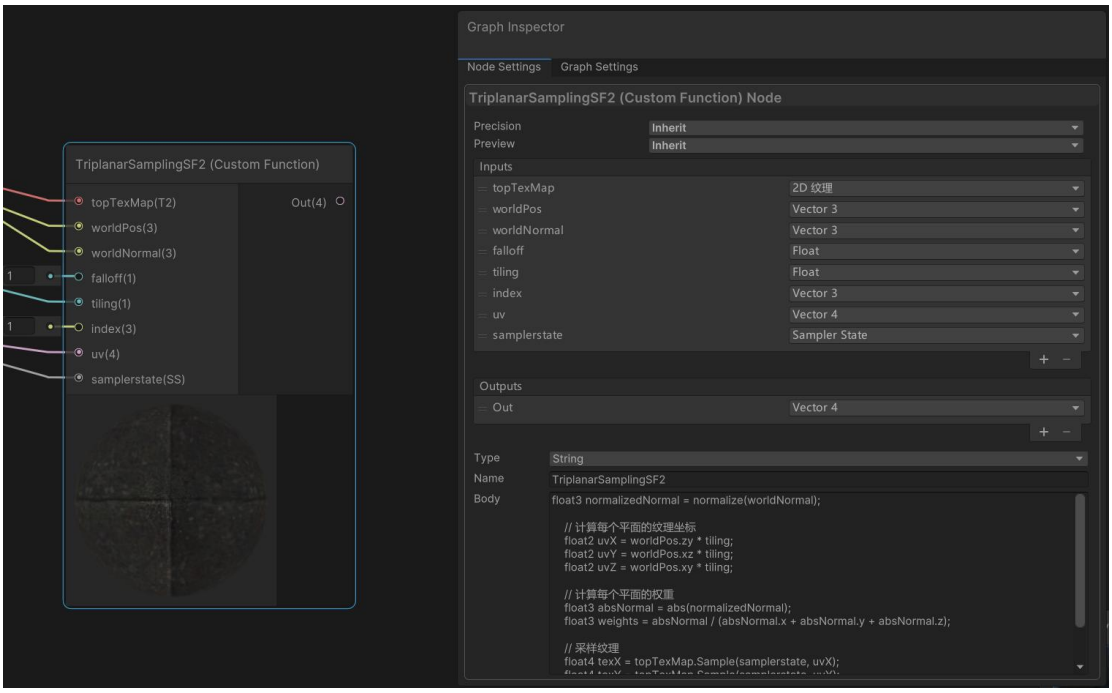
通过不同的法线贴图（雨滴、波浪等）混合计算表面的法线，来模拟表面的动态变化

关键原理 **flipbook**——切割一张纹理图来实现动态纹理帧切换每一帧代表不同雨滴位置，通过物体位置、法线位置和 unity 内置时间实现。

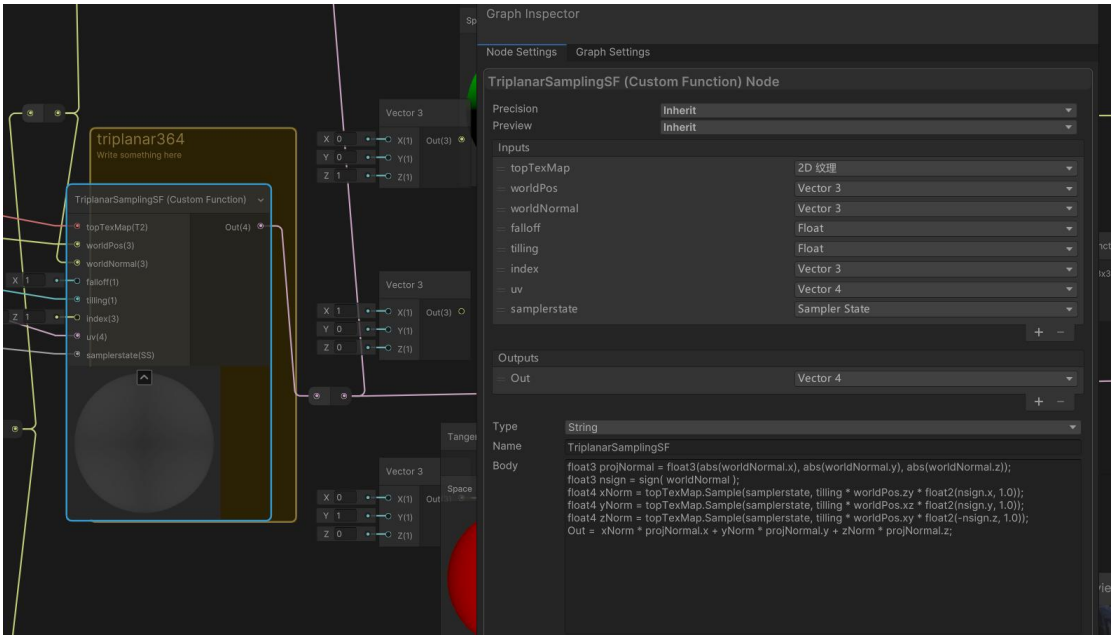


街道材质投射方法（Triplanar Mapping）

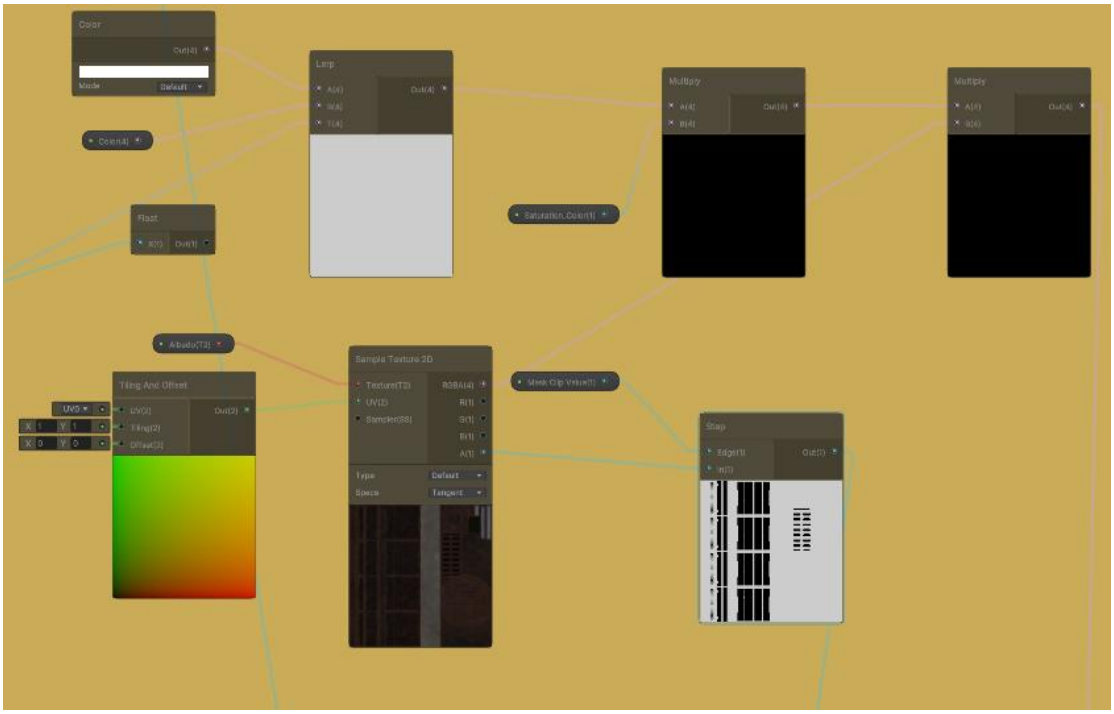
TriplanarSamplingCNF：用于采样法线图，计算纹理的法线和表面细节。该函数结合了表面法线和采样方向，通过多个纹理方向来增强细节表现。



TriplanarSamplingCF：用的三向采样，计算表面颜色等属性。该函数直接通过多方向的纹理值进行混合，提供更好的表面表现。



使用 Mask Clip Value 值在 edge 节点进行透明度裁剪，创建透明区域，使雨水不影响表面的某些区域，并且产生类似镂空的效果。



UI 切换

通过 UI 修改天气。

TimeSpeedSlider：一个 UI 滑块，用于调整时间的速度。

AutoTimeCheckBox：一个复选框，切换时间是否自动流逝。

DayNightSlider：一个滑块，用于手动设置时间，通过调整定向光的角度来实现。

TimeOfDayText：显示当前时间（小时和分钟）的文本框。