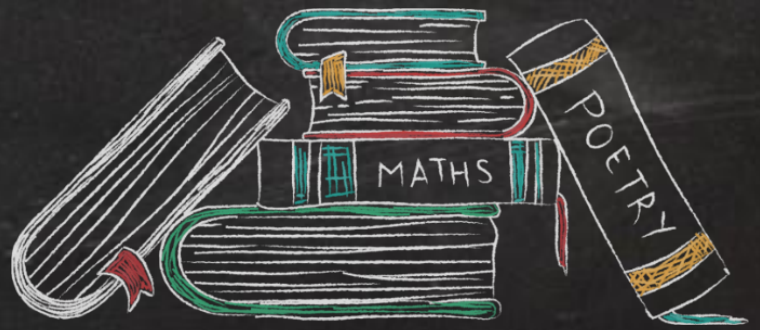
今日内容

- 化神·ShadowPass
- 大乘·烘焙器

1 化神·ShadowPass



01 相关信息

1. 过去课程中，为产生阴影，我们通过Fallback到VertexLit实现；因为，自定义Shader中没有定义ShadowPass时，会去找Fallback的对应Pass；
2. 当需要修改ShadowPass时，就需要自定义ShadowPass；
3. 此例中因为资产模组不是封闭模型，默认ShadowPass中Cull Back会导致严重漏光，故修改为Cull Off；
4. 因本例主要展示烘培光照，实时光照仅辅助打光，故不着重解决实时投影的质量问题，仅快速解决3中的大面积漏光；任存在的质量问题：
    1. 渲染背面时Bias未按正确的法线方向计算；
    2. 模型光滑组有打硬边，按顶点法线偏移渲染深度，导致了这些地方存在漏光缝隙；

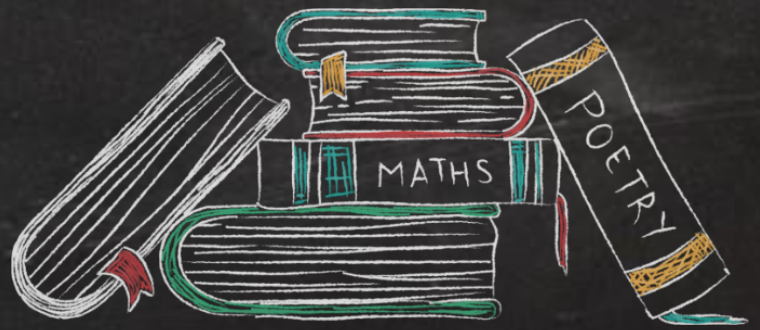
02 极简ShadowPass

1. Tags中注明：“LightMode”=“ShadowCaster”；
2. 因为要做深度比较，故ZWrite为On；
3. 需绘制现有图形前方或距离相同的图形，故ZTest为LEqual；官方文档
4. 修改的地方：Cull Off；
5. 声明多重编译：#pragma multi_compile_shadowcaster；
6. 顶点Shader Input结构用内置的appdata_base；
7. 顶点Shader Output结构成员采用内置宏 V2F_SHADOW_CASTER 声明；
8. 成员变量的赋值通过内置宏 TRANSFER_SHADOW_CASTER_NORMALOFFSET(...) 完成；
9. 像素Shader的内容通过内置宏 SHADOW_CASTER_FRAGMENT(...)完成；

```
Pass
{
    Name "ShadowCaster"
    Tags { "LightMode" = "ShadowCaster" }

    ZWrite On ZTest LEqual Cull Off

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #pragma multi_compile_shadowcaster
    #include "UnityCG.cginc"

    struct v2f {
        V2F_SHADOW_CASTER;
    };

    v2f vert( appdata_base v )
    {
        v2f o;
        TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)
        return o;
    }

    float4 frag( v2f i ) : SV_Target
    {
        SHADOW_CASTER_FRAGMENT(i)
    }
    ENDCG
}
```

$\frac{24 \times 5}{33}$

1 大乘·烘焙器



01 烘培流程

准备

获取场景烘培信息

1. Lightmap的数量；
2. Lightmap存放的文件夹路径；
3. 每张Lightmap的文件路径及纹理对象；

创建Lightmap缓存容器

1. 存在3次烘培，Lightmap亦被多次覆盖，需缓存3组纹理对象用于最终Lightmap的合成；
2. 还有1组纹理对象存储Lightmap的合成结果；

烘培

Bake主光　写入缓存

Bake天光　写入缓存

Bake自发光GI　写入缓存

合成

从缓存生成Lightmap纹理资产

用新生成的Lightmap覆盖旧的

布置场景

重置场景光照环境

更新全局参数

Bake反射探头

02 烘培信息结构

1. 成员变量：
    1. lightmapsCount：整型数 Lightmap数量；
    2. lightmapsInfo：字典<字符串，纹理2D> Lightmap信息；
    3. assetPath：Lightmap存放文件夹路径；
2. Public方法：
    1. LightmapsInfo(LightmapData[] lightmapsData)：从LightmapSetting.lightmaps构造Info的方法；

```
// 结构: 保存lightmap信息
3 usages
private struct LightmapsInfo
{
    // 数量
    public readonly int lightmapsCount;
    // 信息<路径, 纹理对象>
    public readonly Dictionary<string, Texture2D> lightmapsInfo;
    // 资产路径
    public readonly string assetPath;
    // 构造方法
    Frequently called  1 usage
    public LightmapsInfo(LightmapData[] lightmapsData){...}
}
```

03 Lightmap缓存结构

1. 枚举声明：
    1. BufferType：缓存内容的枚举；
2. 成员变量：
    1. _bufferA：纹理2D数组 存放主光bake生成的Lightmap对象；
    2. _bufferB：纹理2D数组 存放天光bake生成的Lightmap对象；
    3. _bufferC：纹理2D数组 存放自发光GI bake生成的Lightmap对象；
    4. _lightmap：纹理2D数组 存放合成的Lightmap对象；
    - 注：以上自行管理创建和释放，不想被外界访问并修改，故声明为私有；
3. Private方法：
    1. ClearBuffer(BufferType type)：
    清除并释放指定类型的纹理缓存；
    2. Clear()：
    清除并释放所有缓存
    3. WriteBuffer(LightmapsInfo info, BufferType type):
    从给定LightmapsInfo和指定缓存类型 写入缓存；
    4. CreateLightmaps()：
    从_bufferA,B,C创建_lightmap；
    5. OverrideLightmaps(LightmapsInfo info)：
    覆盖当前场景Lightmap；

```
// 结构: lightmaps缓存
4 usages
private struct LightmapsBuffer
{
    // lightmap缓存类型
    14 usages  4 exposing APIs
    public enum BufferType
    {
        MainLight,  // 主光光照: BufferA
        SkyLight,   // 天光光照: BufferB
        EmissionGI, // 自发光GI: BufferC
        Lightmap    // 合成Lightmap
    }
    // lightmap缓存
    private Texture2D[] _bufferA;
    private Texture2D[] _bufferB;
    private Texture2D[] _bufferC;
    private Texture2D[] _lightmap;
    // 清理缓存(纹理对象内存占用较大建议手动释放)
    Frequently called  2 usages
    private void ClearBuffer(BufferType type){...}
    // 清理所有缓存
    Frequently called  1 usage
    public void Clear(){...}
    // 从LightmapInfo写入缓存
    Frequently called  3 usages
    public void WriteBuffer(LightmapsInfo info, BufferType type){...}
    // 从缓存创建lightmap
    Frequently called  1 usage
    public void CreateLightmaps(){...}
    // 覆盖场景lightmap
    Frequently called  1 usage
    public void OverrideLightmaps(LightmapsInfo info){...}
}
```

04 Bake方法

1. 清理之前的Bake数据；
2. 配置烘培环境；
    - ArrangeBakeScene(mode)：按烘培模式布置环境的方法
3. Bake场景；
4. 打印日志

```
// 烘培方法
Frequently called  6 usages
public void Bake(BakeMode mode)
{
    // 清理旧的烘培信息
    Lightmapping.Clear();
    // 准备烘培环境
    ArrangeBakeScene(mode);
    // 执行烘培
    Lightmapping.Bake();
    // 打印日志
    switch (mode)
    {
        case BakeMode.BakeMainLight:
            Debug.Log( message: "LightmapsBaker: 主光已烘培.");
            break;
        case BakeMode.BakeSkyLight:
            Debug.Log( message: "LightmapsBaker: 天光已烘培.");
            break;
        case BakeMode.BakeEmissionGI:
            Debug.Log( message: "LightmapsBaker: 自发光GI已烘培.");
            break;
    }
}
```

05 环境配置

```
case BakeMode.Default:
    // 关闭主光
    mainlight.enabled = true;
    // 设置环境
    RenderSettings.ambientMode = AmbientMode.Skybox;
    RenderSettings.ambientIntensity = 1.0f;
    // 设置Shader全局分支
    Shader.DisableKeyword("_BAKE_MAINLIGHT");
    Shader.DisableKeyword("_BAKE_SKYLIGHT");
    Shader.DisableKeyword("_BAKE_EMISSIONGI");
    break;
```

```
case BakeMode.BakeMainLight:
    // 开启主光
    mainlight.enabled = true;
    // 设置主光
    mainlight.color = Color.white;
    mainlight.intensity = 1.0f;
    mainlight.lightmapBakeType = LightmapBakeType.Baked;
    var staticFlags = StaticEditorFlags.ContributeGI | StaticEditorFlags.ReflectionProbeStatic;
    GameObjectUtility.SetStaticEditorFlags(mainlight.gameObject, staticFlags);
    // 设置环境
    RenderSettings.ambientMode = AmbientMode.Flat;
    RenderSettings.ambientSkyColor = Color.black;
    // 设置Shader全局分支
    Shader.EnableKeyword("_BAKE_MAINLIGHT");
    Shader.DisableKeyword("_BAKE_SKYLIGHT");
    Shader.DisableKeyword("_BAKE_EMISSIONGI");
    break;
```

```
case BakeMode.BakeSkyLight:
    // 开启主光
    mainlight.enabled = false;
    // 设置环境
    RenderSettings.ambientMode = AmbientMode.Flat;
    RenderSettings.ambientSkyColor = Color.white;
    // 设置Shader全局分支
    Shader.DisableKeyword("_BAKE_MAINLIGHT");
    Shader.EnableKeyword("_BAKE_SKYLIGHT");
    Shader.DisableKeyword("_BAKE_EMISSIONGI");
    break;
```

```
case BakeMode.BakeEmissionGI:
    // 开启主光
    mainlight.enabled = false;
    // 设置环境
    RenderSettings.ambientMode = AmbientMode.Flat;
    RenderSettings.ambientSkyColor = Color.black;
    // 设置Shader全局分支
    Shader.DisableKeyword("_BAKE_MAINLIGHT");
    Shader.DisableKeyword("_BAKE_SKYLIGHT");
    Shader.EnableKeyword("_BAKE_EMISSIONGI");
    break;
```

06 反射探头Bake方法

```
// 烘培反射探头方法
Frequently called  1 usage
private void BakeReflectProbe()
{
    var allProbe :ReflectionProbe[] = FindObjectsOfType<ReflectionProbe>();
    foreach (var probe in allProbe)
    {
        var path :string = AssetDatabase.GetAssetPath(probe.texture);
        Lightmapping.BakeReflectionProbe(probe, path);
    }
}
```

07 串起来

注：存在获取Info时 场景并未烘培 获取失败的情况；
故 先执行第一次Bake后再获取Info；

```
// 多重烘培方法
Frequently called  1 usage
public void MultiBake()
{
    // 创建lightmap缓存
    var buffer = new LightmapsBuffer();
    // 烘培主光并写入缓存
    Bake(BakeMode.BakeMainLight);
    var info = new LightmapsInfo(LightmapSettings.lightmaps);
    buffer.WriteBuffer(info, LightmapsBuffer.BufferType.MainLight);
    // 烘培天光并写入缓存
    Bake(BakeMode.BakeSkyLight);
    buffer.WriteBuffer(info, LightmapsBuffer.BufferType.SkyLight);
    // 烘培自发光并写入缓存
    Bake(BakeMode.BakeEmissionGI);
    buffer.WriteBuffer(info, LightmapsBuffer.BufferType.EmissionGI);
    // 从缓存创建lightmap
    buffer.CreateLightmaps();
    // 覆盖场景lightmaps
    buffer.OverrideLightmaps(info);
    // 清空lightmap缓存
    buffer.Clear();
    // 恢复场景光照环境
    ArrangeBakeScene(BakeMode.Default);
    // 更新全局参数
    UpdateGlobalProperties();
    // 烘培反射探头
    BakeReflectProbe();
}
```