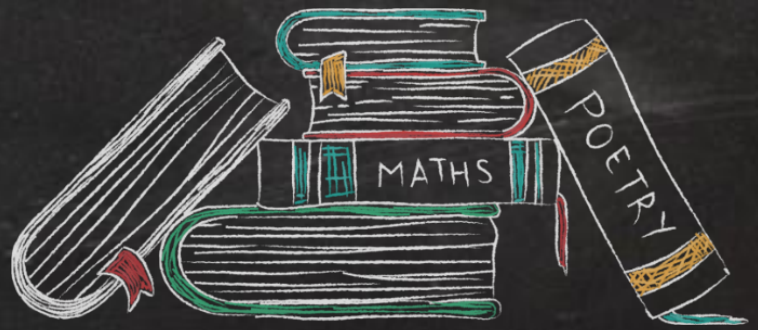
今日内容

- 化神·Shader全局变量
- 化神·Shader分支
- 化神·MetaPass
- 化神·材质GUI

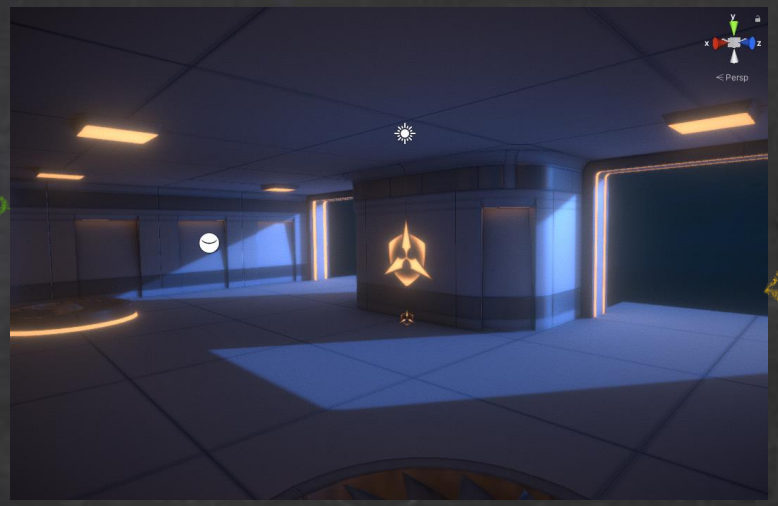
$\frac{24 \times 5}{33}$

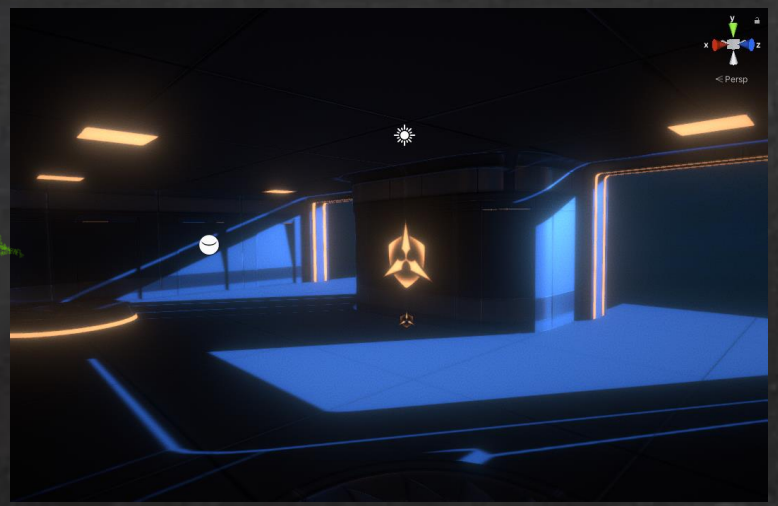
0 下三节内容介绍



01 功能概览

烘培器





功能1：提供实时光照预览 用于打光；

功能2：单独烘培各种光照成分并预览；

· 主光；

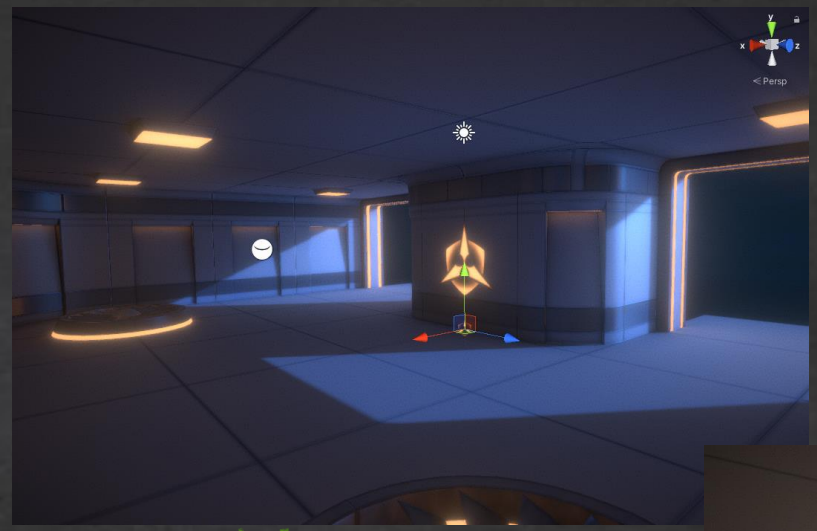
· 天光；

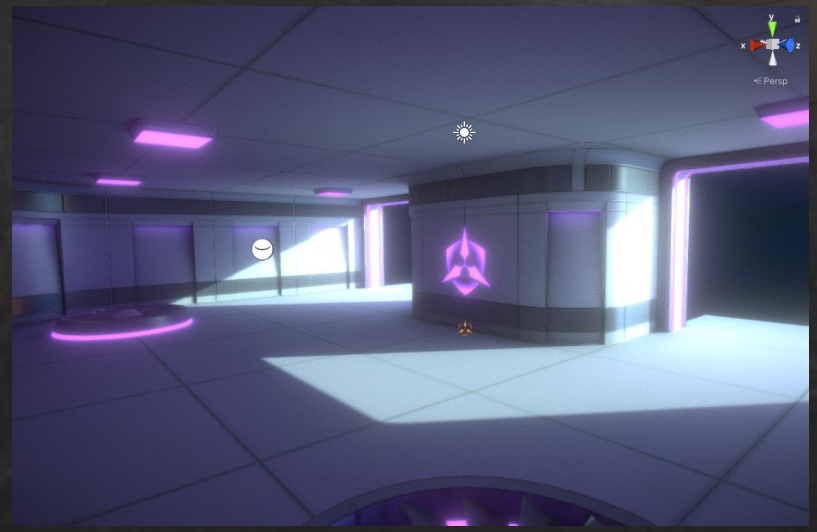
· 自发光GI；

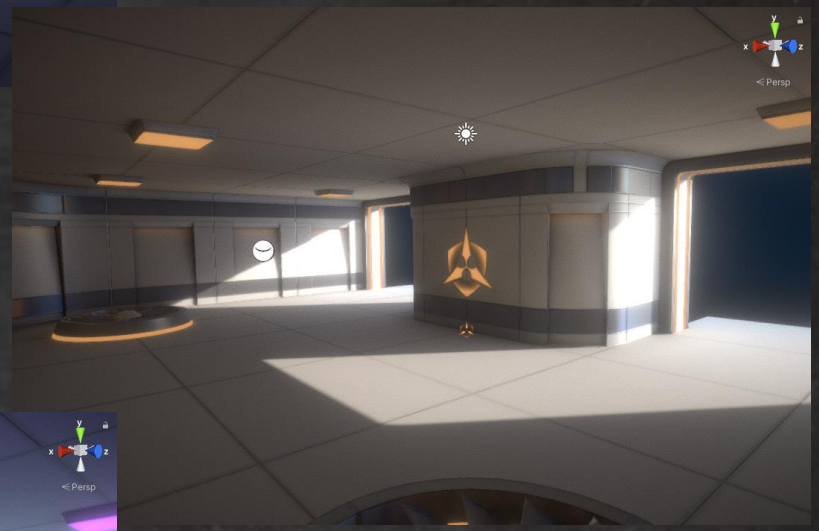
功能3：一键烘培复合光照贴图并预览；

02 功能概览

光照配置面板

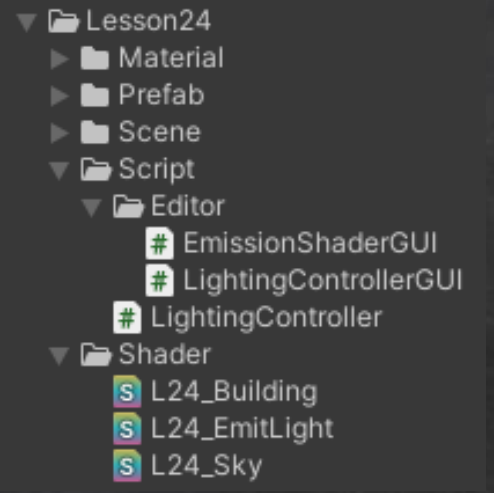






功能4：基于烘培的复合光照贴图 全局配置场景光照；

03 代码清单



Shader:

- L24_Sky : 天空材质
- L24_EmitLight 自发光材质
- L24_Building 建筑材质

C#:

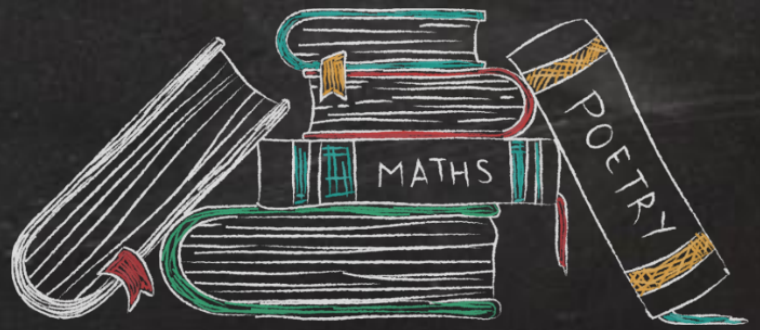
- LightingController 光照控制器

C# (Editor):

- EmissionShaderGUI 自发光材质GUI
- LightingControllerGUI 光照控制器GUI

注：代码量较之前课程大了不少，从此PPT不再做逐行讲解注释，仅截取核心功能相关代码展示；请结合课程录播视频及Git源码注释学习或复习；参与直播的同学可积极提问。

1 化神·Shader全局变量



01 声明全局变量

1. 材质属性段未声明，但Uniform变量段声明的变量，Shader会去试图获取全局变量；
2. 全局变量与一般变量在命名上要做区分，Unity建议前缀g_，亦可按自己习惯追加前缀；

```
Shader "AP01/L24/Building" {
    Properties {
        _MainTex ("颜色纹理", 2D) = "white" {}
        _MaskTex ("遮罩纹理", 2D) = "gray" {}
    }
    SubShader {
        Tags {
            "RenderType"="Opaque"
        }
        Cull Off
        Pass {
            Name "FORWARD"
            Tags {
                "LightMode"="ForwardBase"
            }
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"
            #include "AutoLight.cginc"
            #pragma multi_compile_fwdbase_fullshadows
            // 声明分支
            #pragma multi_compile __ _BAKE_MAINLIGHT _BAKE_SKYLIGHT _BAKE_EMISSIONGI
            // 输入纹理
            uniform sampler2D   _MainTex;
            uniform sampler2D   _MaskTex;
            // 全局变量
            uniform float       _G_MetalDarken;
            uniform float3      _G_MainLightCol;
            uniform float4      _G_SpecParams;
            uniform float       _G_SkyLightInt;
            uniform float4      _G_ReflectParams;
            uniform float       _G_FresnelPow;
            uniform float3      _G_EmissionCol;
            // 输入结构
```

02 设置全局变量

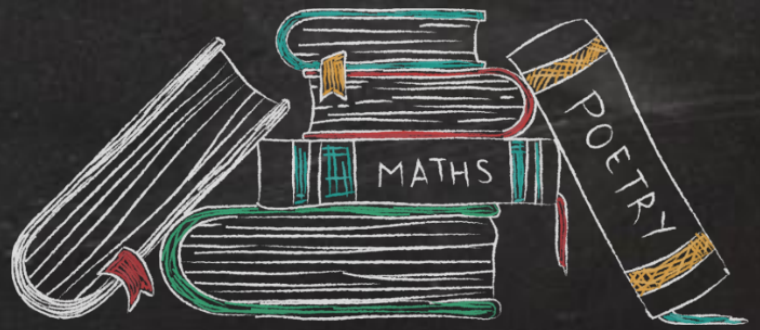
1. 设置全局变量的接口为Shader类的一组静态方法：
   Shader.SetGlobalXXX(String 变量名, XXX 变量值)
   Shader.SetGlobalXXX(Int 变量ID, XXX变量值)
2. 通过变量名设置性能比通过ID设置要差，频繁大量的Runtime执行设置注意使用后者，变量名到ID的转换使用接口：
   Int 变量ID = Shader.PropertyToID(String 变量名)
3. 变量ID在每次运行游戏时都会重新分配，且只分配一次，所以不能预存，需要在Awake，Start等合适的较早时序中获取并缓存；建议统一获取；

```
// 更新Shader全局参数
Frequently called  3 usages
public void UpdateGlobalProperties()
{
    // 注意: Shader的各Set接口都支持按参数名或按参数ID设置 后者性能较好 Runtime代码需先缓存ID 避免按名Set
    Shader.SetGlobalFloat( name: "_G_MetalDarken", metalDarken);
    Shader.SetGlobalColor( name: "_G_MainLightCol", mainLightCol);
    Shader.SetGlobalVector( name: "_G_SpecParams", specParams);
    Shader.SetGlobalFloat( name: "_G_SkyLightInt", skylightInt);
    Shader.SetGlobalVector( name: "_G_ReflectParams", reflectParams);
    Shader.SetGlobalFloat( name: "_G_FresnelPow", metalDarken);
    Shader.SetGlobalColor( name: "_G_EmissionCol", emissionCol);
}
```

03 小演示

Unity演示：全局颜色变量 红绿切

2 化神·Shader分支



01 声明Shader分支

1. 使用宏 #pragma shader_feature __ _A _B ... 声明；
    1. _A _B ... 称Keyword ， 全局Keyword数量有上限；
    2. __ 为默认分支，不占全局Keyword数量，建议使用；
2. 分支有两种声明方式：
    1. multi_compile：多重编译；
    2. shader_feature：特性分支；
    - 前者将编译所有Shader分支，后者只编译用到的分支；
3. 注意：多组不同分支之间为组合关系，多组多重编译分支可能会导致分支爆炸；必要时可使用ShaderVariantsCollection做分支筛选；

```
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
#include "UnityCG.cginc"
#include "AutoLight.cginc"
#pragma multi_compile_fwdbase_fullshadows
// 声明分支
#pragma shader_feature __ _BAKE_MAINLIGHT _BAKE_SKYLIGHT _BAKE_EMISSIONGI
// 输入纹理
uniform sampler2D    MainTex;
```

02 使用Shader分支

1. 使用宏 #if defined (_A) 判断分支是否激活；

```
#if defined(LIGHTMAP_ON)
    #if defined (_BAKE_MAINLIGHT)
                finalRGB = mainLightInt;
    #elif defined (_BAKE_SKYLIGHT) #if defined (_BAKE_MAINLIGHT)
                finalRGB = skyLightInt;
    #elif defined (_BAKE_EMISSIONGI) #elif defined (_BAKE_SKYLIGHT)
                finalRGB = emissionGIInt;
    #endif #elif defined (_BAKE_EMISSIONGI)
#endif #if defined(LIGHTMAP_ON)
                return float4(finalRGB, 1.0);
```

03 设置Shader分支

全局Keyword设置：

- 关闭：Shader.DisableKeyword(String 键名);
- 激活：Shader.EnableKeyword(String 键名);

材质Keyword设置：

- 关闭：material.DisableKeyword(String 键名);
- 激活：material.EnableKeyword(String 键名);

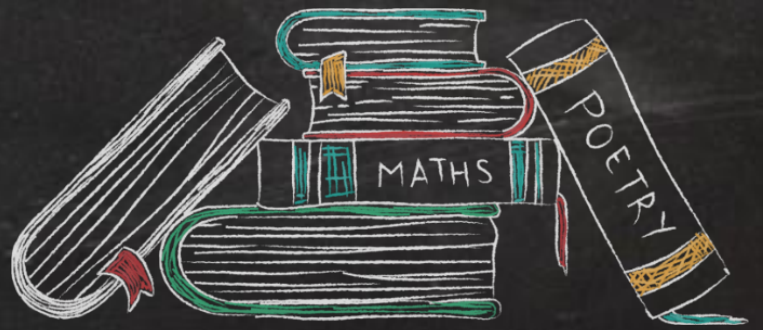
注意：全局设置为Shader类的静态方法，材质设置要对Material类的实例执行；

```
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;
```

04 小演示

Unity演示：全局分支 红绿蓝切

3 化神·MetaPass



01 使自发光参与烘培的条件1

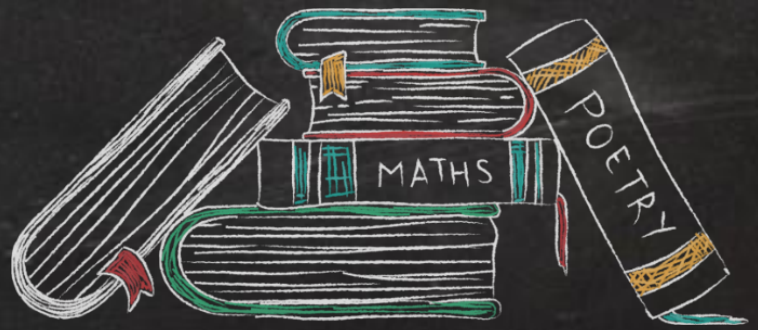
1. 有MetaPass，并有合适的输出；

02 MetaPass写法

1. 与之前写的ForwardBasePass写法一样；
2. Tags需注明“LightMode”=“Meta”；
3. 需包含两个文件：
    1. UnityCG.cginc；
    2. UnityMetaPass.cginc；
4. VertexInput结构需包含：
    1. TEXCOORD1; 光照贴图UV
    2. TEXCOORD2; 动态光照贴图UV
5. 顶点Shader对Pos的转换不再时从本地空间>裁剪空间，而是从本地空间>Lightmap纹理空间；转换使用内置方法即可；
6. 通过UnityMetaInput结构获取必要信息，其成员：
    - Albedo：辐照度；
    - Emission：自发光；
    - SpecularColor：高光颜色；
    - VizUV，LightCoord：编辑器下可视化预览用；
7. 通过UnityMetaFragment(...)方法将UnityMetaInput结构计算为最终输出；
8. 综上，一顿操作需要定制的值也就Albedo，Emission，SpecularColor；

```
Pass {
    Name "META"
    Tags {
        "LightMode" = "Meta"
    }
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #include "UnityCG.cginc"
    #include "UnityMetaPass.cginc"
    // 声明分支
     #pragma shader_feature ___ _BAKE_MAINLIGHT _BAKE_SKYLIGHT _BAKE_EMISSIONGI
    // 输入参数
     uniform sampler2D   _MainTex;
    // 输入结构
     struct VertexInput {
        float4 vertex   : POSITION;      // 顶点位置 总是必要
        float2 uv0      : TEXCOORD0;     // UV信息 采样贴图用
        float2 uv1      : TEXCOORD1;     // 其他UV信息 MetaPass需要
        float2 uv2      : TEXCOORD2;     // 同上
     };
    // 输出结构
     struct VertexOutput {
        float4 pos : SV_POSITION;        // 顶点位置 总是必要
        float2 uv : TEXCOORD0;           // UV信息 采样贴图用
     };
    // 输入结构>>>顶点Shader>>>输出结构
     VertexOutput vert (VertexInput v) {
        VertexOutput o = (VertexOutput)0;
            o.pos = UnityMetaVertexPosition(v.vertex, v.uv1, v.uv2, unity_LightmapST, unity_DynamicLightmapST);
            o.uv = v.uv0;
        return o;
    }
    // 输出结构>>>像素
     float4 frag(VertexOutput i) : COLOR {
        float opacity = tex2D(_MainTex, i.uv).r;
        UnityMetaInput metaIN;
            UNITY_INITIALIZE_OUTPUT(UnityMetaInput, metaIN);
            metaIN.Albedo = 0.0f;
            metaIN.SpecularColor = 0.0f;
#if defined (_BAKE_EMISSIONGI)
            metaIN.Emission = opacity;
#elif defined (_BAKE_MAINLIGHT) || defined (_BAKE_SKYLIGHT)
            metaIN.Emission = 0.0f;
#endif
        return UnityMetaFragment(metaIN);
    }
    ENDCG
}
```

4 化神·材质GUI



01 使自发光参与烘培的条件2

2. 要立Flag，设置材质的globalIlluminationFlags；

注意：比较讨厌就是Unity只对Standard Shader的GUI开放了Flag设置，自定义材质就要自己写材质GUI把这个值开放出来；

02 简单的材质GUI

Shader：

1. 关联材质GUI脚本；

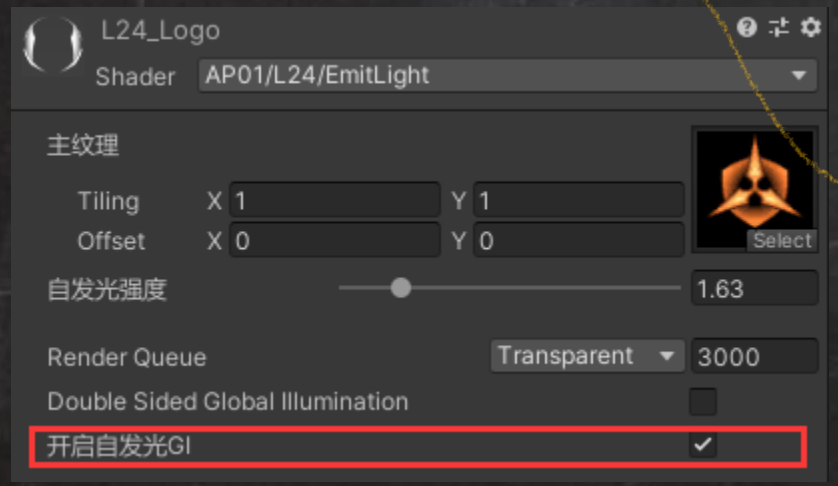
C# (Editor)：

1. 引用命名空间；
2. 声明EmissionShaderGUI类(基类: ShaderGUI)；
3. 重载基类OnGUI方法；
    1. 获取目标材质；
    2. 执行基类方法；
    3. 追加自发光GI开关；

```
            ENDCG
        }
    }
    CustomEditor "EmissionShaderGUI"
}
```

```
using UnityEditor;
using UnityEngine;

// 自发光材质GUI
public class EmissionShaderGUI : ShaderGUI
{
    // GUI绘制方法
    public override void OnGUI(MaterialEditor materialEditor, MaterialProperty[] props)
    {
        // 获取目标Material
        var material = materialEditor.target as Material;
        // 判空
        if (material == null) return;
        // 绘制默认GUI
        base.OnGUI(materialEditor, props);
        // 追加自发光GI开关
        var ifEmissionGIOn = EditorGUILayout.Toggle(
            "开启自发光GI",
            material.globalIlluminationFlags == MaterialGlobalIlluminationFlags.AnyEmissive);
        material.globalIlluminationFlags = ifEmissionGIOn
            ? MaterialGlobalIlluminationFlags.AnyEmissive
            : MaterialGlobalIlluminationFlags.EmissiveIsBlack;
    }
}
```



03 小演示

Unity演示：MetaPass有无 Flag开关 比较