游戏引擎作业-光照

引擎光照

相关功能

借助 Unity 可以实现适合各种艺术风格的逼真光照效果。

Unity 中光照的工作方式类似于光在现实世界中的情况。Unity 使用详细的光线工作模型来获得更逼真的结果，并使用简化模型来获得更具风格化的结果。

直接和间接光照

直射光是发出后照射到表面一次再被直接反射到传感器（例如眼睛的视网膜或摄像机）中的光。间接光是最终反射到传感器中的所有其他光线，包括多次照射到表面的光线和天光。为了获得逼真的光照效果，需要模拟直射光和间接光。

Unity 可以计算直接光照和/或间接光照。Unity 使用什么光照技术取决于项目的配置方式。

实时光照和烘焙光照

实时光照是指 Unity 在运行时计算光照。烘焙光照是指 Unity 提前执行光照计算并将结果保存为光照数据，然后在运行时应用。在 Unity 中，项目可以使用实时光照、烘焙光照或两者的混合（称为混合光照）。

有关配置 Light 组件以提供实时、烘焙或混合光照的信息，请参阅光源模式。

全局光照

全局光照是对直接和间接光照进行建模以提供逼真光照效果的一组技术。Unity 有两个全局光照系统，结合了直接光照和间接光照。

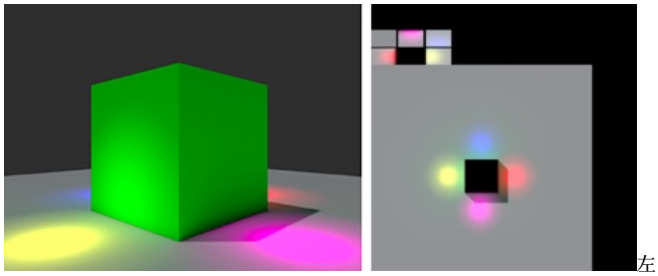
烘焙全局光照系统包括光照贴图、光照探针和反射探针。所有渲染管线均支持烘焙全局光照 (Baked Global Illumination) 系统。每个功能的文档中指出了烘焙全局光照系统中该功能的渲染管线支持情况。

实时全局光照系统包括使用 Enlighten 的实时全局光照，并为光照探针增加了额外功能。内置渲染管线支持实时全局光照。高清渲染管线 (HDRP) 和通用渲染管线 (URP) 不支持实时全局光照系统。请注意，目前已弃用 Enlighten，很快就会从 Unity 中删除实时全局光照系统。想了解更多相关信息，请参阅 Unity 博客。

操作

光照贴图

光照贴图过程将预先计算场景中表面的亮度，并将结果存储在称为“光照贴图”的纹理中供以后使用。

 左：一个简单的光照贴图场景。
右：Unity 生成的光照贴图纹理。请注意捕获阴影和光照信息的方式。

光照贴图可以包含直射光和间接光。该光照纹理可与颜色（反照率）和浮雕（法线）之类的对象表面信息材质相关联的着色器一起使用。

烘焙到光照贴图中的数据无法在运行时更改。实时光源可以重叠并可在光照贴图场景上叠加使用，但不能实时改变光照贴图本身。

通过这种方法，我们可在游戏中移动我们的光照，通过降低实时光计算量潜在提高性能，适应性能较低的硬件，如移动平台。

Unity 提供以下光照贴图程序来生成光照贴图：

- Progressive Lightmapper

渐进光照贴图程序

渐进光照贴图程序 (Progressive Lightmapper) 是一种基于路径追踪的光照贴图系统，提供了能在 Editor 中逐渐刷新的烘焙光照贴图和光照探针。要求不重叠的 UV 具有较小的面积和角度误差，以及棋盘格图表之间有足够的填充。

渐进光照贴图程序采取了一个短暂的准备步骤来处理几何体与实例的更新，同时生成 G-buffer 和图表遮罩。然后，它会立即生成输出，并随着时间的推移逐步细化输出，以实现更完善的交互式照明工作流。此外，烘焙时间更加可预测，因为渐进光照贴图程序在烘焙时提供估计时间。

渐进光照贴图程序还可单独为每个纹素分别以光照贴图分辨率烘焙全局光照 (GI)，无需采用上采样方案或依赖任何辐照度缓存或其他全局数据结构。因此，渐进光照贴图程序具有强大的功能，并允许您烘焙光照贴图的选定部分，从而更快测试和迭代场景。

渲染管线支持

内置渲染管线支持渐进光照贴图程序。

- 通用渲染管线 (URP) 支持渐进光照贴图程序。
- 高清渲染管线 (HDRP) 支持渐进光照贴图程序。

使用渐进光照贴图程序

要使用渐进光照贴图程序，请执行以下操作：

1.选择 **Window > Rendering > Lighting Settings** 2.导航到 **Lightmapping Settings** 3.将 **Lightmapper** 设置为 **Progressive CPU** 或 **Progressive GPU**

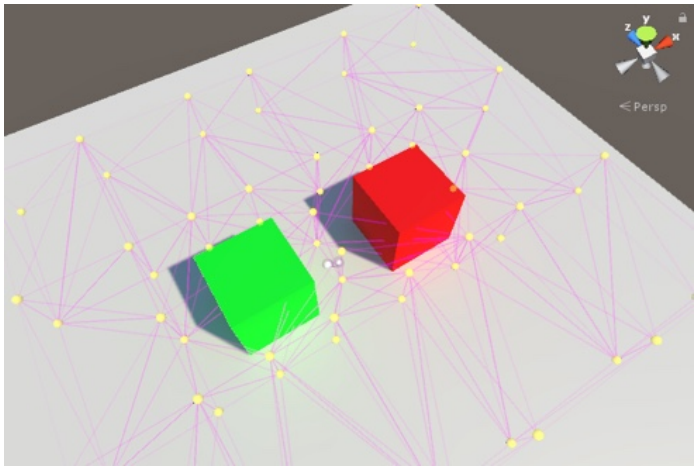
可以使用 LightmapEditorSettings 和 Lightmapping API，通过脚本执行此窗口中提供的许多函数。

光照探针

通过__光照探针__可以捕获并使用穿过场景空白空间的光线的相关信息。

与光照贴图类似，光照探针存储了有关场景中的光照的“烘焙”信息。不同之处在于，光照贴图存储的是有关光线照射到场景中的*表面*的光照信息，而光照探针存储的是有关光线穿过场景中的*空白空间*的信息。

光照探针是在烘焙期间测量（探测）光照的场景位置。在运行时，系统将使用距离动态游戏对象最近的探针的值来估算照射到这些对象的间接光。



一个非常简单的场景显示了放置在两个立方体周围的光照探针

光照探针有两个主要用途:

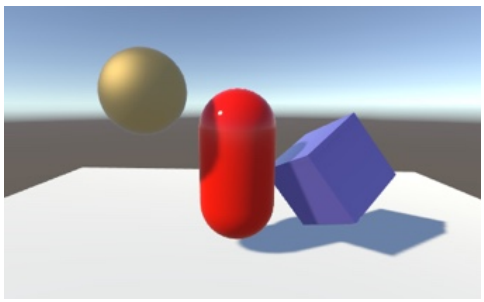
光照探针的主要用途是为场景中的移动对象提供高质量的光照（包括间接反射光）。

光照探针的次要用途是在静态景物使用 Unity 的 __LOD（细节级别）系统__时提供该景物的光照信息。

将光照探针用于这两个不同目的时，需要使用的许多技术都是相同的。了解光照探针的工作原理非常重要，这样才能正确选择将探针放置在场景中的位置

阴影

阴影为场景添加了一定程度的深度和真实感，因为它们可以显示对象的比例和位置，否则对象看起来显得扁平。在 Unity 中，光源可将游戏对象的阴影投射到其自身的其他部分或附近的游戏对象上。



场景中的对象投射了阴影

阴影贴图

Unity 使用一种称为阴影贴图的技术来渲染实时阴影。

阴影贴图的工作原理

阴影贴图使用称为“阴影贴图”的纹理。阴影贴图类似于深度纹理。光源生成阴影贴图的方式与摄像机生成深度纹理的方式类似。假设摄像机与光源位于相同的位置，则摄像机无法看到的场景区域与光源的光线无法到达的场景区域相同；因此，这些区域处于阴影中。

Unity 会在阴影贴图中填充与光线在射到表面之前传播的距离有关的信息，然后对阴影贴图进行采样，以便计算光线射中的游戏对象的实时阴影。

有关阴影贴图的更多信息，请参阅关于阴影贴图 (shadow mapping) 的 Wikipedia 页面。

阴影贴图分辨率

为了计算阴影贴图的分辨率，Unity 会进行以下工作:

1.确定光源可以照亮的屏幕视图区域。方向光可以照亮整个屏幕。对于聚光灯和点光源，该区域是光源边界形状（点光源为球体，聚光灯为锥体）在屏幕上的投影。投影形状在屏幕上具有宽度和高度（以像素为单位）；然后采用这两个值中的较大者。这个值就称为光源的像素大小。

2.确定阴影质量乘数。为此，Unity 采用 **Shadow Resolution** 设置（在 Quality Settings 窗口中进行设置）。质量设置对应于以下值： - 非常高：1.0 - 高：0.5 - 中：0.25 - 低：0.125 3.执行以下计算，然后将结果确定为最大大小：

光源类型：	公式：	最大分辨率（以像素为单位）：
Directional	NextPowerOfTwo (像素大小 * 阴影质量乘数 * 3.0)	当阴影分辨率的质量为非常高时并且/或者如果 GPU 的 RAM 为 512MB 或更高，最 4096，否则为 2048 x 2048。
聚光灯	NextPowerOfTwo (像素大小 * 阴影质量乘数 * 2.0)	如果 GPU 的 RAM 为 512MB 或更高，最高分辨率为 2048 x 2048，否则为 1024 x 1
点光源	NextPowerOfTwo (像素大小 * 阴影质量乘数 * 1.0)	如果 GPU 的 RAM 为 512MB 或更高，最高分辨率为 1024 x 1024，否则为 512 x 51

点光源的大小下限比其他类型的大，因为它们对阴影使用立方体贴图。这意味着此分辨率下的六个立方体贴图面必须同时保存在视频内存中。渲染它们的成本也非常高，因为潜在的阴影投射物可能需要渲染到所有六个立方体贴图面中。

覆盖阴影贴图分辨率

在内置渲染管线中，可以通过将 Light.shadowCustomResolution 属性设置为大于 0 的值来设置光源阴影贴图的分辨率。当该值大于 0 时，Unity 对所有光源类型都会执行以下计算：

NextPowerOfTwo (shadowCustomResolution)

然后会根据光源类型和硬件来确定最大分辨率，如上表所示。

注意，仅在内置渲染管线中支持 `shadowCustomResolution` 属性。高清渲染管线 (HDRP) 或通用渲染管线 (URP) 不支持它。

配置阴影

可以使用 Inspector 为每个 Light 组件配置实时阴影和烘焙阴影设置。

场景中的每个网格渲染器 (Mesh Renderer) 还具有 **Cast Shadows** 和 **Receive Shadows** 属性，必须根据需要启用它们。

通过从下拉菜单中选择 **On** 启用 **Cast Shadows__，即可启用或禁用网格的阴影投射。或者，选择** Two Sided__ 以允许表面的任一面投射阴影（因此在进行阴影投射时会忽略背面剔除），或选择 **Shadows Only** 以允许不可见的游戏对象投射阴影。

阴影贴图

阴影距离

使用 **Shadow Distance** 属性可以确定 Unity 渲染实时阴影的最大距离（与摄像机之间的距离）。

游戏对象距摄像机的距离越远，产生的阴影就越不明显。这有两个原因：一个原因是阴影在屏幕上看起来更小，另一个原因是远处的游戏对象通常不是人们关注的焦点。可以通过为远处的游戏对象禁用实时阴影渲染来利用此原理。这样可以节省浪费的渲染操作，并可以提高运行时性能。此外，没了远处阴影后，场景通常看起来更好。

如果当前摄像机远平面小于阴影距离，Unity 将使用摄像机远平面而不是阴影距离。

要掩盖超出阴影距离的缺失阴影，可以使用诸如雾效之类的视觉效果。

设置阴影距离

在内置渲染管线中，请在项目的质量设置 (Quality Settings) 中设置 Shadow Distance 属性。

在通用渲染管线 (URP) 中，请在通用渲染管线资源中设置 Shadow Distance 属性。

在高清渲染管线 (HDRP) 中，请为每个体积 (Volume) 设置 Shadow Distance 属性。

Shadow Distance 和 Shadowmask 光照模式

如果场景使用 Shadowmask 光照模式，则 Unity 会使用光照探针或阴影遮罩纹理来渲染从混合光源产生的超出阴影距离的阴影。还可以配置 Unity 如何渲染超出阴影距离的阴影。

使用Shadow Map技术。把摄像机与光源位置重合，光源的阴影部分就是摄像机看不到的地方。

前向渲染路径中，最重要的平行光如果开启了阴影，Unity就会为光源计算阴影映射纹理（shadowmap），本质就是深度图，记录光源出发到最近表面位置。两种方法：

1. 摄像机放在光源位置，然后按正常渲染流程（调用Base Pass 和 Additional Pass）来更新深度信息，得到阴影映射纹理。
2. 摄像机放在光源位置，调用额外的Pass：LightMode = ShadowCaster，把顶点变换到光源空间，渲染目标不是帧缓存，而是阴影映射纹理。

阴影采样

- 传统方法： 正常渲染Pass，计算顶点的光源空间，用xy分量对纹理采样，如果顶点值大于该深度值，就说明在阴影区域。
- Unity5及以后： 屏幕空间的阴影映射技术（Screenapce Shadow Map）。是在延迟渲染中产生阴影的方法。不过需要显卡支持MRT。根据阴影映射纹理和深度纹理得到屏幕空间的阴影图。阴影图包含了屏幕空间所有阴影区域。

接收其他物体阴影

在Shader中对阴影映射纹理（包括屏幕空间的阴影图）进行采样，结果和最后的光照相乘即可。

向其他物体投射阴影

把物体加入到光源的阴影映射纹理计算中（让其他物体可以得到该物体信息），即执行LightMode 为ShadowCaster的Pass。

操作

使用AutoLight.cginc文件内的三个宏：

- SHADOW_COORDS：声明了一个名为_ShadowCoord的阴影纹理坐标变量。
- TRANSFER_SHADOW：根据不同平台实现。如果使用了屏幕空间的阴影映射技术，会使用内置ComputeScreenPos函数计算_ShadowCoord；否则就直接把顶点转换到光源空间存到_ShadowCoord。
- SHADOW_ATTENUATION：对_ShadowCoord采样，得到阴影信息。

需要注意：内置宏用了一些变量名需要在自己定义的时候要匹配：a2f的顶点坐标为vertex、输出的v2f结构体名为v，v2f中顶点位置名为pos。

后处理

相关功能

Unity的屏幕后期处理效果，使用MonoBehaviour.OnRenderImage来实现，包括调整屏幕的亮度、饱和度、对比度的方法

在屏幕上显示图像之前，__后期处理__将全屏滤镜和效果应用于摄像机的图像缓冲区。此技术可以通过很短的设置时间大幅改善应用程序的视觉效果。后期处理效果可用于模拟物理摄像机和胶片属性。

本手册的这一部分概述了 Unity 中可用的后期处理效果。有关如何使用这些后期处理效果的具体信息，请单击每个页面上的链接。

使用后期处理

下面的图像展示了应用和未应用后期处理的场景。

未应用后期处理场景

应用后期处理后的场景

后期处理和渲染管线

可用的后期处理效果和这些效果的应用方式取决于所使用的渲染管线。一个渲染管线的后期处理解决方案与其他渲染管线不兼容。

渲染管线	后期处理支持
内置渲染管线	默认情况下，内置渲染管线不包含后期处理解决方案。要在内置渲染管线中使用后期处理效果，请下载后期处理 (Post-P 2 包。有关在内置渲染管线中使用后期处理效果的信息。请参阅后期处理 (Post-Processing) 版本 2
通用渲染管线 (URP)	URP 包括自身的后期处理解决方案（由 Unity 在使用 URP 模板创建项目时安装）。有关在 URP 中使用后期处理效果 P 后期处理文档。
高清渲染管线 (HDRP)	HDRP 包括自身的后期处理解决方案（由 Unity 在使用 HDRP 模板创建项目时安装）。有关在 HDRP 中使用后期处理 HDRP 后期处理文档。

操作

屏幕后处理效果（screen post-processing effects）
即在渲染完场景得到屏幕图像后在进行处理的效果，如景深、运动模糊等。

Unity提供接口方便处理渲染后的图像：OnRenderImage。第一个参数渲染得到纹理，第二个参数是输出到屏幕的

纹理。
MonoBehaviour.OnRenderImage (RenderTexture src, RenderTexture dest)

在上面函数内，通常使用Graphics.Blit函数处理渲染纹理。

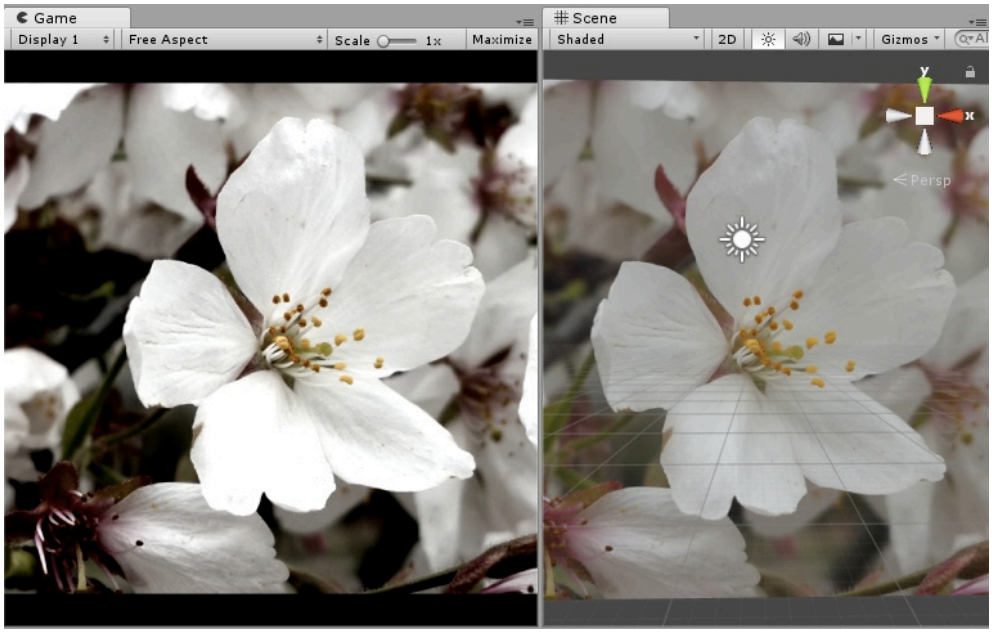
```
public static void Blit(Texture source, RenderTexture dest);
public static void Blit(Texture source, Material mat);
public static void Blit(Texture source, RenderTexture dest, Material mat);
public static void Blit(Texture source, Material mat, [Internal.DefaultValue("-1")] int pass);
public static void Blit(Texture source, RenderTexture dest, Vector2 scale, Vector2 offset);
public static void Blit(Texture source, RenderTexture dest, Material mat, [Internal.DefaultValue("-1")] int pass);
```

- source：当前屏幕的渲染纹理或上一步处理完的渲染纹理。
- dest：目标渲染纹理，为null时就直接把结果显示在屏幕上。
- mat：使用的材质，材质的shader会进行屏幕后处理，source会传入这个shader名为_MainTex的纹理。
- pass：默认-1，即顺序调用Shader内所有Pass。否则只会调用给定索引的Pass。
- sacle：源纹理的缩放。
- offset：源纹理的偏移量。

实现屏幕后处理效果的基类 PostEffectsBase

```
using UnityEngine;
/// <summary>
/// 屏幕后处理效果基类
/// </summary>
[ExecuteInEditMode]
[RequireComponent(typeof(Camera))]
public class PostEffectsBase : MonoBehaviour
 {
    private Camera mainCamera;
    public Camera MainCamera  { get  { return mainCamera = mainCamera == null ? GetComponent<Camera>() : mainCamera; }  }
    public Shader targetShader;
    private Material targetMaterial = null;
    public Material TargetMaterial  { get  { return CheckShaderAndCreateMaterial(targetShader, targetMaterial); }  }
    /// <summary>
    /// 检测资源，如果不支持，关闭脚本活动
    /// </summary>
    protected void Start()
     {
        if (CheckSupport() == false)
            enabled = false;
    }
    /// <summary>
    /// 检测平台是否支持图片渲染
    /// </summary>
    /// <returns></returns>
    protected bool CheckSupport()
     {
        if (SystemInfo.supportsImageEffects == false)
         {
            Debug.LogWarning("This platform does not support image effects or render textures.");
            return false;
        }
        return true;
    }
    /// <summary>
    /// 检测需要渲染的Shader可用性，然后返回使用了该shader的material
    /// </summary>
    /// <param name="shader">指定shader</param>
    /// <param name="material">创建的材质</param>
    /// <returns>得到指定shader的材质</returns>
    protected Material CheckShaderAndCreateMaterial(Shader shader, ref Material material)
     {
        if (shader == null || !shader.isSupported)
            return null;
        if (material && material.shader == shader)
            return material;
        material = new Material(shader);
        material.hideFlags = HideFlags.DontSave;
        return material;
    }
}
```

调整亮度（Brightness）、饱和度（Saturation）、对比度（Contrast）
调整后图像与原图对比：



通过mono脚本处理屏幕后效果：

```
public class BrightnessSaturationAndContrast : PostEffectsBase
 {
    [Range(0.0f, 3.0f)]
    public float brightness = 1.0f;
    [Range(0.0f, 3.0f)]
    public float saturation = 1.0f;
    [Range(0.0f, 3.0f)]
    public float contrast = 1.0f;
    void OnRenderImage(RenderTexture src, RenderTexture dest)
     {
        if (TargetMaterial != null)
         {
            TargetMaterial.SetFloat("_Brightness", brightness);
            TargetMaterial.SetFloat("_Saturation", saturation);
            TargetMaterial.SetFloat("_Contrast", contrast);
        }
        Graphics.Blit(src, dest, TargetMaterial);
    }
}
```

材质的shader：

```
Pass  {
    // 关掉深度写入，避免挡住后面物体渲染，（如果OnRenderImage在所有不透明Pass后直接执行（而非所有都渲染完））
    ZTest Always Cull Off ZWrite Off
    ...
    fixed4 frag(v2f i) : SV_Target  {
        fixed4 renderTex = tex2D(_MainTex, i.uv);
        // 亮度
        fixed3 finalColor = renderTex.rgb * _Brightness;
        // 饱和度
        fixed luminance = 0.2125 * renderTex.r + 0.7154 * renderTex.g + 0.0721 * renderTex.b;
        fixed3 luminanceColor = fixed3(luminance, luminance, luminance);    // 饱和度为0的颜色值
        finalColor = lerp(luminanceColor, finalColor, _Saturation);
        // 对比度
        fixed3 avgColor = fixed3(0.5, 0.5, 0.5);        // 对比度为0的颜色值
        finalColor = lerp(avgColor, finalColor, _Contrast);
        return fixed4(finalColor, renderTex.a);
    }
}
```

雾效

相关功能

雾效就是在远离我们视角的方向上，物体看起来像被蒙上了某种颜色。
实现方法：

- Unity内置雾效可以产生基于距离的线性或指数雾效。
- 自行编写雾效需要添加#pragma multi_compile_fog指令，并使用相关宏。缺点是需要为所有物体添加渲染代码，实现效果有限。
- 根据深度纹理重建每个像素世界空间下位置，使用公式模拟雾效。

雾的计算

颜色混合类似透明度混合：f为混合系数。
float3 afterFog = f * fogColor + (1 - f) * origColor;

雾效系数f的计算方法：z为距离（一般是点的高度）。

- 线性：f = (dmax - |z|) / (dmax - dmin)。d表示受雾影响的距离。
- 指数：f = e-d * |z|。d是控制浓度的参数。
- 指数的平方：f = e-(d - |z|)2。d是控制浓度的参数。

操作

1、对于雾效的开启在新版的Unity中通过界面菜单Windows->Lighting窗口中Scene页签打开
2、勾选Fog即开启雾效，雾效三种方式（雾的浓度和距离相机的距离有关，详细请参考Unity中雾效的模拟）

1)、Linear 即线性可配参数Start、End两个距离，雾效从Start开始越接近End越浓，到达End时达到最大浓度，End之后也为最大浓

2)、Exponential 指数性可配参数Density，雾的浓度，浓度越大雾越大

3)、Exponential Squared 可配参数Density，越大表示雾越浓

3、关于程序里雾效的动态实现：

主要有以下参数：

--开启雾效

```
RenderSettings.fog = true;
```

--雾效颜色
RenderSettings.fogColor = Color.red;

--Linear有效
RenderSettings.fogStartDistance = 1;
RenderSettings.fogEndDistance = 3;

--Exponential Exponential Squared 有效
RenderSettings.fogDensity = 8;

代码

FogWithDepthTexture类：

```
using UnityEngine;
/// <summary>
/// 雾效
/// </summary>
public class FogWithDepthTexture : PostEffectsBase
 {
    private Camera targetCamera;
    public Camera TargetCamera  { get  { return targetCamera = targetCamera == null ? GetComponent<Camera>() : targetCamera; }  }
    private Transform cameraTransform;
    public Transform CameraTransform  { get  { return cameraTransform = cameraTransform == null ? TargetCamera.transform : cameraTransform; }  }
    [Range(0.0f, 3.0f)]
    public float fogDensity = 1.0f;         // 雾的密度
    public Color fogColor = Color.white;    // 雾的颜色
    public float fogStart = 0.0f;           // 起始高度
    public float fogEnd = 2.0f;             // 终止高度
    void OnEnable()
     {
        TargetCamera.depthTextureMode |= DepthTextureMode.Depth;
    }
    void OnRenderImage(RenderTexture src, RenderTexture dest)
     {
        if (TargetMaterial != null)
         {
            Matrix4x4 frustumCorners = Matrix4x4.identity;  // 存近裁切平面的四个角
```

```
            float fov = TargetCamera.fieldOfView;         // 竖直方向视角范围（角度）
            float near = TargetCamera.nearClipPlane;      // 近平面距离
            float aspect = TargetCamera.aspect;           // 宽高比
            // 计算四个角对应的向量，存到frustumCorners
            float halfHeight = near * Mathf.Tan(fov * 0.5f * Mathf.Deg2Rad);
            Vector3 toRight = CameraTransform.right * halfHeight * aspect;
            Vector3 toTop = CameraTransform.up * halfHeight;
            Vector3 topLeft = CameraTransform.forward * near + toTop - toRight;
            float scale = topLeft.magnitude / near;
            topLeft.Normalize();
            topLeft *= scale;
            ...  // 同样方法计算其他三个角
            frustumCorners.SetRow(0, bottomLeft);
            frustumCorners.SetRow(1, bottomRight);
            frustumCorners.SetRow(2, topRight);
            frustumCorners.SetRow(3, topLeft);
            TargetMaterial.SetMatrix("_FrustumCornersRay", frustumCorners);
            TargetMaterial.SetFloat("_FogDensity", fogDensity);
            TargetMaterial.SetColor("_FogColor", fogColor);
            TargetMaterial.SetFloat("_FogStart", fogStart);
            TargetMaterial.SetFloat("_FogEnd", fogEnd);
        }
        Graphics.Blit(src, dest, TargetMaterial);
    }
}
```

Shader：

```
Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _FogDensity ("Fog Density", Float) = 1.0
    _FogColor ("Fog Color", Color) = (1, 1, 1, 1)
    _FogStart ("Fog Start", Float) = 0.0
    _FogEnd ("Fog End", Float) = 1.0
}
SubShader {
    ...
    struct v2f {
        float4 pos : SV_POSITION;
        half2 uv : TEXCOORD0;
        half2 uv_depth : TEXCOORD1;             // 深度纹理
        float4 interpolatedRay : TEXCOORD2;     // 插值后的像素向量
    } ;
    v2f vert(appdata_img v) {
        ...
        // 靠近哪个角选哪个
        int index = 0;
        if (v.texcoord.x < 0.5 && v.texcoord.y < 0.5) {
            index = 0;
        } else if (v.texcoord.x > 0.5 && v.texcoord.y < 0.5) {
            index = 1;
        } else if (v.texcoord.x > 0.5 && v.texcoord.y > 0.5) {
            index = 2;
        } else {
            index = 3;
        }
        #if UNITY_UV_STARTS_AT_TOP
        if (_MainTex_TexelSize.y < 0)
            index = 3 - index;
        #endif
        o.interpolatedRay = _FrustumCornersRay[index];
        return o;
    }
    fixed4 frag(v2f i) : SV_Target {
        // 采样然后得到视角空间的线性深度值
        float linearDepth = LinearEyeDepth(SAMPLE_DEPTH_TEXTURE(_CameraDepthTexture,
i.uv_depth));
        float3 worldPos = _WorldSpaceCameraPos + linearDepth * i.interpolatedRay.xyz;   //
世界空间相机相加
        float fogDensity = (_FogEnd - worldPos.y) / (_FogEnd - _FogStart);
        fogDensity = saturate(fogDensity * _FogDensity);                    // 限制在0~1
        fixed4 finalColor = tex2D(_MainTex, i.uv);
        finalColor.rgb = lerp(finalColor.rgb, _FogColor.rgb, fogDensity);
        return finalColor;
    }
    ENDCG
    Pass {
        ZTest Always Cull Off ZWrite Off
```

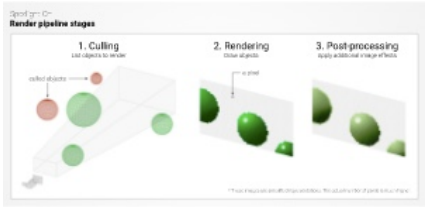
```
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        ENDCG
    }
}
```

渲染

渲染管线确定场景中对象的显示方式，分为三个主要阶段。

- 第一步是剔除；它列出了需要渲染的对象，最好是那些对摄像机可见的对象（视锥体剔除）和其他对象不遮挡的对象（遮挡剔除）。
- 第二个阶段渲染是指将这些对象绘制到基于像素的缓冲区中（通过正确的光照以及它们的一些属性）。
- 最后，可以在这些缓冲区上执行后期处理操作，例如，应用颜色分级、泛光和景深，从而生成发送到显示设备的最终输出帧。

根据帧率，这些操作每秒钟重复很多次。



- 着色器是在图形处理单元 (GPU) 上运行的程序或程序集合的通用名称。例如，在剔除阶段完成后，顶点着色器用于将可见对象的顶点坐标从“对象空间”转换为称为“裁剪空间”的不同空间；然后 GPU 使用这些新的坐标对场景进行光栅化，也就是将场景的矢量表示转换为实际像素。在稍后阶段，这些像素将由像素（或片元）着色器进行着色；像素颜色通常将取决于各自表面的材质属性以及周围的光照。现代硬件上另一种常见的着色器是计算着色器：计算着色器允许程序员利用 GPU 的大量并行处理能力，用于任何类型的数学运算，如光照剔除、粒子物理或体积模拟。
- 直接光照指的是从自发光光源（如灯泡）发出的光照，而不是光从表面反射的结果。根据光源的大小及其与接收者的距离，这种光照通常会产生清晰的不同阴影。
    - 请勿将直接光照与方向光照混淆，后者是指是由无限远的光源（例如计算机模拟的太阳）发出的光。方向光的显著特性是能够用平行光线覆盖整个场景，并且不存在距离衰减（或光衰减）；也就是说，接收到的光照量不会随着与光源距离的增加而衰减。
    - 在现实中，太阳光也会像任何其他光源一样，光照会按照距离的平方反比定律而衰减。简单来说，当接收者与光源之间的距离增加时，收到光照量会迅速降低。例如，水星上的太阳光照度几乎是地球上的 7 倍，火星上的太阳光照度是地球的近一半，而冥王星仅仅为 0.06%。然而，对于海拔高度范围非常有限的大多数实时应用程序来说，太阳光的衰减微不足道。因此，方向光完全足以模拟大多数 Unity 场景（包括以行星为中心的大型空旷空间）中的太阳光。
- 间接光照是由于光从表面反射并通过介质（如大气或半透明物质）传播和散射而形成的结果。在这种状况下，遮挡物通常投射出柔和或难以看清的阴影。
- 全局光照 (GI) 是对直接和间接光照进行建模以提供逼真光照效果的一组技术。GI 有几种方法，如烘焙/动态光照贴图、辐照度体积、光传播体积、烘焙/动态光照探针、基于体素的 GI 和基于距离场的 GI。Unity 支持开箱即用的烘焙/动态光照贴图和光照探针。
- 光照贴图程序是一个基础系统，它通过发射光线、计算光线反弹并将产生的光线应用到纹理来生成光照贴图和光照探针的数据。因此，不同的光照贴图程序通常会产生不同的光照外观，因为它们可能依赖不同的技术来生成光照数据。