

# UniversalRenderPipeline

该类为真正去驱动各个Pass做渲染的类，继承**RenderPipeline**，每帧都会调用重写的Render方法。

## 初始化及构造函数

- 初始化的时候会定义一些全局控制参数。如最大阴影Bias，RenderScal，AdditionalLight。
- 构造函数
  1. 会生成一个**UniversalRenderPipelineGlobalSettings**。
  2. SetSupportedRenderingFeatures()生成一个SupportedRenderingFeatures()对象。
  3. 初始化屏幕大小。
  4. 获取是否支持SRPBatcher
  5. 获取MSAA的采样数和设置MSAA
  6. 设置shader的全局名称“UniversalPipeline”，在URP管线下，tag不为空，且不是该值的，则赋予错误材质。
  7. 设置GI
  8. CameraCaptureBridge.enabled = true

## Render ()

1. 通过**GraphicsSettings** 设置一些全局状态
  - 颜色空间
  - 是否使用色温
  - 默认渲染LayerMask
2. 设置shader常量
  - 使用SphericalHarmonicsL2类型的对象存储**RenderSettings.ambientProbe** 中的探针和环境光。

```
Color linearGlossyEnvColor = new Color(ambientSH[0, 0], ambientSH[1, 0], ambientSH[2, 0]) * RenderSettings.ambientProbe.color;
Color glossyEnvColor = CoreUtils.ConvertLinearToActiveColorSpace(linearGlossyEnvColor);
Shader.SetGlobalVector(ShaderPropertyId.glossyEnvironmentColor, glossyEnvColor);
```

随后使用Shader.SetGlobalVector向shader传递一些全局参数

```
glossyEnvironmentCubeMapHDR
glossyEnvironmentCubeMap
ambientSkyColor
```

```
ambientEquatorColor
ambientGroundColor
subtractiveShadowColor
rendererColor
```

以上关于颜色的参数都会通过**CoreUtils.ConvertSRGBToActiveColorSpace** 由SRGB空间转换到当前激活的颜色空间。

3. 判断GlobalSetting 是否为Null, 如果为NULL则调用 **UniversalRenderPipelineGlobalSettings.Ensure()** 方法

4. 按照相机的深度对相机进行排序

5. 针对每个相机进行单独渲染

对于单个的相机, 会判断他是Game相机还是Scene相机。

现在只讨论一下Game相机的情况。

通过调用RenderCameraSack开始渲染单独的一个Game相机

## RenderCameraSack ()

unity将相机分类为baseCamera和overlay。

贴一个Unity官网对此的解释：

URP 中可以使用多个 Camera，去实现一些特殊的效果。与内置渲染管线不同的是，URP 中设置了两类渲染类型的相机：“Base Camera”和“Overlay Camera”。Base Camera 是 URP 中默认的摄像机类型。要以 URP 渲染任何东西，我们的场景中必须至少有一个 Base Camera。Overlay Camera 是将其视图呈现在另一个摄影机的输出之上的摄影机，我们不能单独使用 Overlay Camera，我们只能使用相机堆栈（Camera Stack），将“Overlay Cameras”与一个或多个“Base Camera”结合使用。Overlay Camera 如果不和 Camera Stack 结合使用的话，是不会执行其渲染循环的任何步骤的。

该方法主要做了两件事，一件是驱动相机去做渲染，处理后处理，初始化各种数据，调用 **RenderSingleCamera ()**。另一个便是驱动Overlay相机渲染。

具体过程如下

1. 首先获取相机的**UniversalAdditionalCameraData** 组件

```
if (baseCameraAdditionalData != null && baseCameraAdditionalData.renderType == CameraRenderType.Overlay)
    return;
```

如果相机有该组件且为OverLay相机，则跳过。

## 2. 获取Renderer

renderer保存在**UniversalAdditionalCameraData** 组件中

## 3. 判断是否支持CameraStacking

获取相机的overlayCamera队列，存储在cameraStack中

## 4. 判断是否支持任何的后处理

## 5. 获取最后一个激活的OverlayCamera

当有Overalay相机为null时，便会更新整个cameraStack。

这里便是判断相机是否是真正的Overlay相机，如果有不是的则会报警。

```

for (int i = 0; i < cameraStack.Count; ++i)
{
    Camera currCamera = cameraStack[i];
    if (currCamera == null)
    {
        shouldUpdateCameraStack = true;
        continue;
    }

    if (currCamera.isActiveAndEnabled)
    {
        currCamera.TryGetComponent<UniversalAdditionalCameraData>(out var data);

        // Checking if the base and the overlay camera is of the same renderer type.
        var currCameraRendererType = data?.scriptableRenderer.GetType();
        if (currCameraRendererType != baseCameraRendererType)
        {
            Debug.LogWarning("Only cameras with compatible renderer types can be stacked. " +
                $"The camera: {currCamera.name} are using the renderer {currCameraRendererType} " +
                $"but the base camera: {baseCamera.name} are using {baseCameraRendererType}");
            continue;
        }

        var overlayRenderer = data.scriptableRenderer;
        // Checking if they are the same renderer type but just not supporting Overlay
        if ((overlayRenderer.SupportedCameraStackingTypes() & 1 << (int)CameraRenderType.Overlay) == 0)
        {
            Debug.LogWarning($"The camera: {currCamera.name} is using a renderer of type {overlayRenderer.GetType()} " +
                $"which does not support Overlay rendering");
            continue;
        }

        if (data == null || data.renderType != CameraRenderType.Overlay)
        {
            Debug.LogWarning($"Stack can only contain Overlay cameras. The camera: {currCamera.name} " +
                $"has a type {data.renderType} that is not supported. Will skip rendering");
            continue;
        }

        anyPostProcessingEnabled |= data.renderPostProcessing;
        lastActiveOverlayCameraIndex = i;
    }
}

```

## 6. UpdateVolumeFramework ()

Unity对后处理使用的是Volume框架

## 7. 初始化相机数据InitializeCameraData ()

## 8. 获取相机渲染纹理的格式

## 9. RenderSingleCamera () 渲染这个相机。

10. 最后渲染所有的Overlay相机。

这个过程和渲染单个的差不多，只不过初始化的相机数据的方法变成了

**InitializeAdditionalCameraData**

## UpdateVolumeFramework ()

该方法需要两个参数，一个是camera，一个是UniversalAdditionalCameraData。

主要是调用VolumeManager来更新volumeStack；

## InitializeCameraData ()

这个方法主要初始化了CameraData这个结构，调用了**InitializeStackedCameraData** 和 **InitializeAdditionalCameraData** 这两个方法。

并且判断是否开启Msaa，只有相机，管线，renderer同时支持，才会开启msaa，且优先使用相机的msaa设置。

最后调用**CreateRenderTextureDescriptor** 方法创建一个RenderTextureDescriptor并传给CameraData。RenderTextureDescriptor是一个纹理描述结构，用来申请纹理时，需要该结构。

## InitializeStackedCameraData ()

参数：Camera，UniversalAdditionalCameraData，ref CameraData

初始化CameraData的一些参数。

- 纹理  
相机的渲染纹理，一般为null
- 相机类型
- 是否为场景相机  
根据是否场景相机初始化一些volumeLayerMask数据
- VolumeLayerMask
- volumeTrigger
- isStopNaNEnabled
- isDitheringEnabled
- antialiasing
- antialiasingQuality
- 是否支持HDR

- 设置相机的视口Rect

- RenderScale

这个决定了由相机视图向最后的纹理视图怎么转换。

在[0.95,1.05]范围内， scale = 1。

- 设置OpaqueSortFlags
- 设置captureActions

## InitializeAdditionalCameraData ()

参数： Camera,UniversalAdditionalCameraData,bool resolveFinalTarget,ref CameraData cameraData

### resolveFinalTarget 的解释

如果这是堆栈中的最后一个摄像机，并且渲染应该解析为摄像机目标，则为True。

这里初始化一些关于阴影和后处理的东西

- cameraData.maxShadowDistance

```
bool anyShadowsEnabled = settings.supportsMainLightShadows || settings.supportsAdditionalLightShadows;
cameraData.maxShadowDistance = Mathf.Min(settings.shadowDistance, camera.farClipPlane);
cameraData.maxShadowDistance = (anyShadowsEnabled && cameraData.maxShadowDistance >= camera.nearCli
```

如果主光源和任何附加光源投射阴影，则为True。

- renderType
- clearDepth

```
(additionalCameraData.renderType != CameraRenderType.Base) ? additionalCameraData.clearDepth : true;
```

BaseCamera的clearDepth为True，而Overlay的则根据自己的设置来

- requiresDepthTexture

默认管线检测到是OverlayCamera会设置为False

- requiresOpaqueTexture

这一项不仅由相机设置决定，如果是场景相机也会开启，后处理开启MSAA，MotionBlur，DepthOfField等后处理开启也会设置为True。

默认管线检测到是OverlayCamera会设置为False

- renderer
- antialiasing
- antialiasingQuality
- 是否支持后处理

如果是OpenGL ES2则不支持。

- resolveFinalTarget  
相机是basecamera，且没有CameraStack则为True，如果是overlay且是camerastack中的最后一个也为True
- 需要DepthRTexture != isSceneViewCamera;
- 如果是Overalay则禁用 OpaqueTexture 和 DepthTexture

上诉的过程都是在初始化CameraData，操作则是将baseCameraAdditionalData的值赋给CameraData，而BaseCamera和Overlay有一些区别，具体区别看[数据类型及过程](#)。

## RenderSingleCamera ()

参数： ScriptableRenderContext, CameraData, bool anyPostProcessingEnabled

anyPostProcessingEnabled参数

如果堆栈中至少有一个摄像头启用了后处理，则为True，否则为false

- 首先获取相机裁剪数据**TryGetCullingParameters ()**
- 调用ScriptableRenderer.clear () 方法  
该方法如下:

```
internal void Clear(CameraRenderType cameraType)
{
    m_ActiveColorAttachments[0] = BuiltinRenderTextureType.CameraTarget;
    for (int i = 1; i < m_ActiveColorAttachments.Length; ++i)
        m_ActiveColorAttachments[i] = 0;
    m_ActiveDepthAttachment = BuiltinRenderTextureType.CameraTarget;
    m_FirstTimeCameraColorTargetIsBound = cameraType == CameraRenderType.Base;
    m_FirstTimeCameraDepthTargetIsBound = true;
    m_CameraColorTarget = BuiltinRenderTextureType.CameraTarget;
    m_CameraDepthTarget = BuiltinRenderTextureType.CameraTarget;
}
```

主要是将颜色附件，深度附件设置为相机的Target。

- 调用Renderer子类定义的SetupCullingParameters方法  
主要是设置一下可见光数量和从cameraData获得最大阴影距离
- 初始化Renderdata。调用InitializeRenderingData  
包括灯光，阴影和后处理的开关。
- 根据场景还是game相机渲染UI。

```
if UNITY_EDITOR
    // Emit scene view UI
    if (isSceneViewCamera)
        ScriptableRenderContext.EmitWorldGeometryForSceneView(camera);
    else
#endif
    if (cameraData.camera.targetTexture != null && cameraData.cameraType != CameraType.Preview)
        ScriptableRenderContext.EmitGeometryForCamera(camera);
```

- InitializeRenderingData初始化rendererData
- 调用renderer.setup, 该方法由Renderer类自己实现。
- renderer.Execute执行渲染
- 结束。