

# ForwardRenderer

## 构造函数

初始化各种Pass和Texture的过程。

贴图初始化的名称看[数据类型及过程](#)

## SetUp

设置各个Pass和贴图的过程。

## 如果相机设置了贴图且格式为Depth

```
bool isOffscreenDepthTexture = cameraData.targetTexture != null && cameraData.targetTexture.format == Render1
    if (isOffscreenDepthTexture)
    {
        ConfigureCameraTarget(BuiltinRenderTextureType.CameraTarget, BuiltinRenderTextureType.CameraTarget);
        AddRenderPasses(ref renderingData);
        EnqueuePass(m_RenderOpaqueForwardPass);
        EnqueuePass(m_DrawSkyboxPass);
#ifdef ADAPTIVE_PERFORMANCE_2_1_0_OR_NEWER
        if (!needTransparencyPass)
            return;
#endif
        EnqueuePass(m_RenderTransparentForwardPass);
    }
    return;
```

上述代码只渲染物体的深度。

## 配置Color贴图

createColorTexture很重要的一个bool变量，控制是否创建一张相机贴图。

```

createColorTexture = (rendererFeatures.Count != 0 && !isPreviewCamera) |= RequiresIntermediateColorTexture(ref
RequiresIntermediateColorTexture(ref cameraData)
{
    if (cameraData.renderType == CameraRenderType.Base && !cameraData.resolveFinalTarget)
        return true;
    if (this.actualRenderingMode == RenderingMode.Deferred)
        return true;
    bool isSceneViewCamera = cameraData.isSceneViewCamera;
    var cameraTargetDescriptor = cameraData.cameraTargetDescriptor;
    int msaaSamples = cameraTargetDescriptor.msaaSamples;
    bool isScaledRender = !Mathf.Approximately(cameraData.renderScale, 1.0f);
    bool isCompatibleBackbufferTextureDimension = cameraTargetDescriptor.dimension == TextureDimension.Tex2D;
    bool requiresExplicitMsaaResolve = msaaSamples > 1 && PlatformRequiresExplicitMsaaResolve();
    bool isOffscreenRender = cameraData.targetTexture != null && !isSceneViewCamera;
    bool isCapturing = cameraData.captureActions != null;
    bool requiresBlitForOffscreenCamera = cameraData.postProcessEnabled || cameraData.requiresOpaqueTexture |
    if (isOffscreenRender)
        return requiresBlitForOffscreenCamera;

    return requiresBlitForOffscreenCamera || isSceneViewCamera || isScaledRender || cameraData.isHdrEnabled |
        !isCompatibleBackbufferTextureDimension || isCapturing || cameraData.requireSrgbConversion;
}

```

可以看到，createColorTexture为True时的条件很多，大部分情况都会为True。

BaseCamera时当createColorTexture为True时，m\_ActiveCameraColorAttachment = m\_CameraColorAttachment即进行离屏渲染，且目标纹理为m\_CameraColorAttachment。

而Overlay则直接m\_ActiveCameraColorAttachment = m\_CameraColorAttachment。

## 配置深度图

createDepthTexture为True时，m\_ActiveCameraDepthAttachment = m\_CameraDepthAttachment

```

bool createDepthTexture = cameraData.requiresDepthTexture && !requiresDepthPrepass;
createDepthTexture |= (cameraData.renderType == CameraRenderType.Base && !cameraData.resolveFinalTarget);
createDepthTexture |= this.actualRenderingMode == RenderingMode.Deferred;

```

勾选需要深度图并且需要提前深度测试便为True或者，相机为Base且有CameraStack。

requiresDepthPrepass条件如下

```

bool requiresDepthPrepass = requiresDepthTexture && !CanCopyDepth(ref renderingData.cameraData);
requiresDepthPrepass |= isSceneViewCamera;
requiresDepthPrepass |= isPreviewCamera;
requiresDepthPrepass |= renderPassInputs.requiresDepthPrepass;
requiresDepthPrepass |= renderPassInputs.requiresNormalsTexture;

```

也是基本上只要RendererFeature有需要depth的便会为True，且拥有CameraStack也会为True。

## 配置相机的Target

1. 若createColorTexture或者createDepthTexture中有一个为True，则调用CreateCameraRenderTarget。
2. 必然会调用**ConfigureCameraTarget(activeColorRenderTargetId, activeDepthRenderTargetId)**而激活的Attachment要么是m\_CamerasAttachment要么是RenderTargetHandle cameraTargetHandle = RenderTargetHandle.GetCameraTarget(cameraData.xr);而该方法返回的RenderTargetIdentifier = -1。

## 添加各个pass

这里只关注几个特殊的pass

1. m\_DepthPrepass和m\_DepthNormalPrepass  
这两个Pass，添加其中一个就不会添加另一个。  
这两个pass干嘛的具体看pass文件
2. m\_CopyDepthPass  
只要有了DepthPrepass，就不会有该pass
3. applyPostProcessing  
这里只关注最后一个相机的后处理的目标输出图  
由resolvePostProcessingToCameraTarget控制，如果为True则为RenderTargetHandle.CameraTarget，直接输出至final，不需要Finalpass。

```
bool resolvePostProcessingToCameraTarget = !hasCaptureActions && !hasPassesAfterPostProcessing && !applyFinalPass;
```

这三个变量都为False时，才会输出至FinalTarget。关注第二个变量，只要有pass的event在后处理后便为True。

如果不是最后一个相机则输出的图为m\_AfterPostProcessColor。

4. m\_FinalPostProcessPass  
开启后处理，最后一个相机，且抗锯齿为FXAA时，加入这个pass。它会在后处理后执行，需要一张source贴图，如果开启了后处理那输入贴图就变为m\_AfterPostProcessColor没开就是m\_ActiveCameraColorAttachment。

5. m\_FinalBlitPass该Pass就是把渲染图输出到默认帧缓冲上。

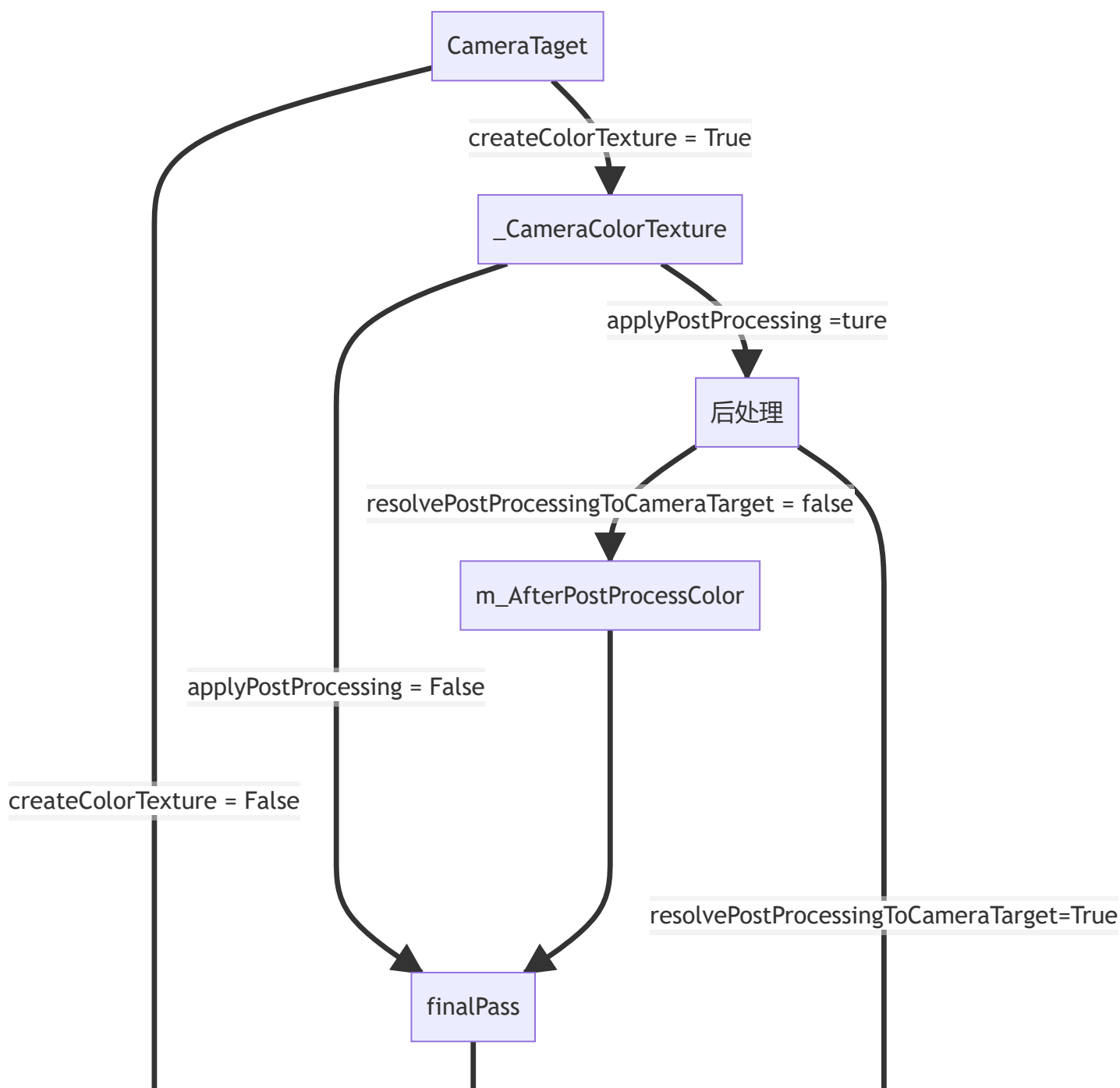
关注触发条件。

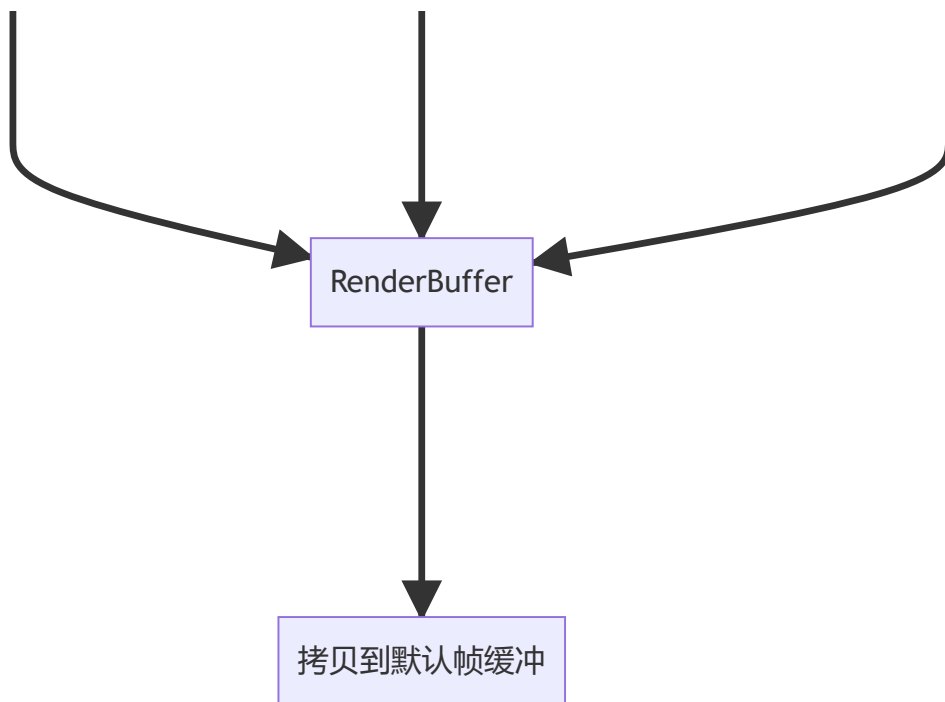
cameraTargetResolved变量为False时，添加这个pass。只要以下条件满足则添加这个pass

- 启动FXAA
- （没有后处理||后处理后没有任何Pass） 满足一个
- createColorTexture为False，意味着没有配置颜色贴图。

## 总结一下贴图流向

Camera





各个Pass还会配置自己贴图。这个会在Excute中通过setRenderTarget来改变渲染目标实现。

# ScriptabRenderer

## Excute

执行渲染的函数，调用每个pass的Excute方法。

1. 调用InternalStartRendering执行每个激活的Pass的**OnCameraSetup**方法。
2. 设置一些全局shader变量
  - worldSpaceCameraPos
  - screenParams
  - scaledScreenParams
  - zBufferParams
  - orthoParams
3. 设置时间变量
  - time
  - sintime
  - costime
  - deltatime
  - timeParamters
4. 排序渲染队列，按照pass设定的Event进行排序

5. SetUpLights调用ForwardLights的Setup
6. 对排好序的渲染队列进行分块
7. 根据每个分块调用**ExecuteBlock**。
  - 以**BeforeRendering** 为参数调用
  - 以**MainRenderingOpaque**， 为参数调用
  - 以**MainRenderingTransparent**， 为参数调用
  - 以**AfterRendering**， 为参数调用
8. 在BeforeRendering后，URP重新设置了一下时间和全局变量
  - worldToCameraMatrix
  - cameraToWorldMatrix
  - inverseViewMatrix
  - inverseProjectionMatrix
  - inverseViewAndProjectionMatrix
9. DrawWireOverlay,该方法是用来渲染线框的。而且需为ViewCamera。
10. DrawGizmos
11. InternalFinishRendering

## RendererBlocks策略

有三个队列：

- m\_BlockEventLimits, 长度为 $A = \text{Const int } k\_RenderPassBlockCount = 4$ 。
- m\_BlockRanges,长度为  $B = A + 1$ 。
- m\_BlockRangeLengths 长度为  $C = B + 1$ 。

### m\_BlockEventLimits

存储的渲染块的边界，划分了四个渲染块

渲染块名称	int值	对应渲染Event
BeforeRendering	0	BeforeRenderingPrepasses
MainRenderingOpaque	1	AfterRenderingOpaques
MainRenderingTransparent	2	AfterRenderingPostProcessing
AfterRendering	3	(RenderPassEvent)Int32.MaxValue

每个渲染块存储是该块结束后的渲染Event

# m\_BlockRanges

存储激活的RenderPass按照m\_BlockEventLimits划分的块的每个块开始的索引  
它的填充代码如下:

```
int currRangeIndex = 0;
int currRenderPass = 0;
m_BlockRanges[currRangeIndex++] = 0;

// For each block, it finds the first render pass index that has an event
// higher than the block limit.
for (int i = 0; i < m_BlockEventLimits.Length - 1; ++i)
{
    while (currRenderPass < activeRenderPassQueue.Count &&
        activeRenderPassQueue[currRenderPass].renderPassEvent < m_BlockEventLimits[i])
        currRenderPass++;
    m_BlockRanges[currRangeIndex++] = currRenderPass;
}
m_BlockRanges[currRangeIndex] = activeRenderPassQueue.Count;
```

BlockEventLimits				
索引	BeforeRendering=0	Opaque=1	Transparent = 2	AfterRender = 3
对应事件	BeferRenderEvent	AfterOpaque	AfterPostProcess	Inter.32.MaxValue

BlockRanges				
0	1	3	8	9

BlockRangeLengths				
1	2	5	1	0

Pass	
Event	PassName
BeforeRenderingShadows	Shadow
BeforeRenderingPrepasses	DepthOnlyPass
BeforeRenderingOpagues	DrawObjectsPass
BeforeRenderingSkybox	DrawSkyBox
AfterRenderingSkybox	CopyColorPass
BeforeRenderingTransparents	TransSetting
BeforeRenderingTransparents	DrawObjectsPass
BeforeRenderingPostProcessing	ObjectCallBackPass
AfterRendering	RenderObjectPass

## BlockRangeLengths

存储当前分块每个块的长度

## ExecuteBlock

参数: int blockIndex, RenderBlocks, ScriptableRenderContext, RenderingData, submit

先根据 blockIndex从m\_BlockRanges中取出一个块。

然后遍历这个块的每个RenderPass，执行**ExecuteRenderPass**。该方法就是按照块来执行渲染。

## ExecuteRenderPass



该方法为执行每个RenderPass的Configure，然后设置PassAttachments,最后执行renderpass的Execute方法。

## SetRenderPassAttachments

1. 获取相机ClearFlag，这个标识符，Overlay相机和SkyBox有不同的要求。
2. 调用RenderingUtils.GetValidColorBufferCount检查该Pass拥有的ColorBuffer。如果为0则退出。
3. 判断pass的colorAttachments是否为MRT，判断方法为，如果该PASS激活的colorAttachments大于1则为MRT。

MRT既是多重渲染，一个shader将不同的结果输出到多个贴图。

## ISMRT

## NotMRT

即没有多重渲染目标的pass。

1. 先判断有没有覆盖相机的Target。  
通过**overrideCameraTarget** 只要这个pass调用了**ConfigureTarget** 方法，该变量就变为True。
2. 如果没覆盖相机的Target，则把Pass的passColorAttachment设置为相机的，depth也是。源码如下：

```
if (!renderPass.overrideCameraTarget)
{
    // Default render pass attachment for passes before main rendering is current active
    // early return so we don't change current render target setup.
    if (renderPass.renderPassEvent < RenderPassEvent.BeforeRenderingOpakes)
        return;

    // Otherwise default is the pipeline camera target.
    passColorAttachment = m_CameraColorTarget;
    passDepthAttachment = m_CameraDepthTarget;
}
```

3. 设置该PassfinalClearFlag

如果passColorAttachment == m\_CameraColorTarget，即没有覆盖相机的target设置，则使用相机的clear。

否则用Pass自定义的设置。

4. 设置该Pass的RenderTarget

只有当以下条件满足时，才会设置。

```
if (passColorAttachment != m_ActiveColorAttachments[0] || passDepthAttachment != m_ActiveDepthAttachment ||
renderPass.colorStoreActions[0] != m_ActiveColorStoreActions[0] || renderPass.depthStoreAction != m_ActiveDepthStoreAction)
```

如果满足则调用**SetRenderTarget**。

## SetRenderTarget

将该Pass的Color, Depth的Attachments设置为传进来的参数, 同时设置纹理load时的动作和StoreAction。

这里可以参考苹果的一篇[文章](#)。

随后调用cmd.SetRenderTarget改变渲染目标。

## renderPass.Execute

最后调用每个pass的Excute方法完成渲染。

## InternalFinishRendering

调用每个Pass的OnFinishCameraStackRendering

调用Renderer的FinishRendering, 就是把相机的目标贴图换成默认的。  
然后清理激活的renderPass。