渲染代理，UWidgetCompnent里有CreateSceneProxy方法，重载可以自己创建对应的渲染代理，UWidgetComponent继承自UPrimitiveComponent

但是书上说，所有可以被渲染的组件都需要创建一个对应的渲染代理，然而ACTOR中并没有。。

！Actor中的渲染是由SceneComponent进行的，SceneComponet比UPrimitiveComponent还要高一级！！该函数为UPrimitiveComponent中声明的

Procedural Mesh Component in C++:Getting Started

<https://wiki.unrealengine.com/Procedural_Mesh_Component_in_C%2B%2B:Getting_Started>

应该是类似于图形接口画图创建自己的Actor 空了看看。

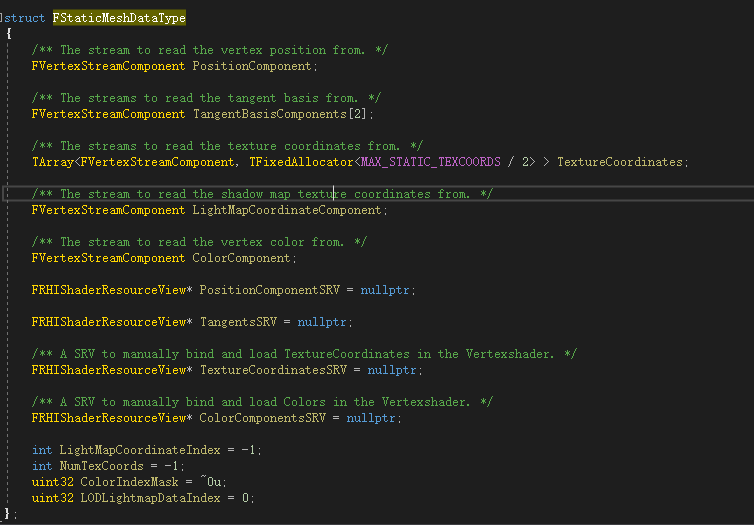
FPrimitiveSceneProxy这个才是渲染用的类，GetDynamicMeshElements在渲染线程中调用

FPrimitiveDrawInterface 用于查询其动态元素的基本接口。

RHI 模块是渲染 API 的接口，是图形编程的另一个关键组件。

自定义工厂

FStaticMeshDataType 模型数据格式？自定义顶点工厂的时候，继承自FLocalVertexFactory，有使用到，里面点开可以看到包含了



不就基本上和U3D Shader的基础数据类似么！坐标，（TangentBasis）法线，切线，UV，光照，顶点颜色，贴图UV等等等！！！！！

然后在工厂中生成一个DataType，向里面填充这些数据，扔给GPU去计算生成吧，整个流程就很清晰了~！！

FVertexStreamComponent(VerTexBuffer,STRUCT\_OFFSET( FDynamicMeshVertex,TextureCoordinate ),

sizeof(FDynamicMeshVertex),EVertexElementType::VET\_Float2)

这些TextureCoordinate以及Position等都是来自于FDynamicMeshVertex中的字段！

VET\_Float2 VET\_Float3 更是与Shader中的一致！！！！

静态这一坨，确实应该有问题，改成研究 Procedural Mesh吧

# Shader与渲染

enum EShaderParameterFlags

{

// no shader error if the parameter is not used

SPF\_Optional,

// shader error if the parameter is not used

SPF\_Mandatory

};

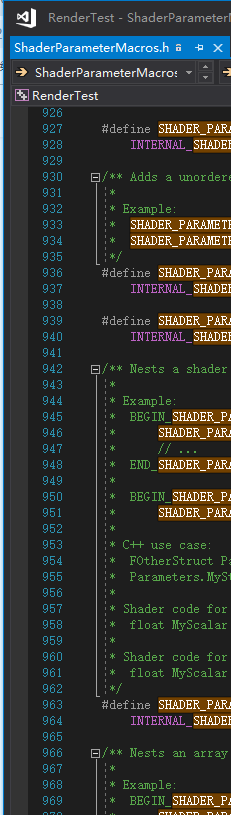
自定义类的传入参数相关

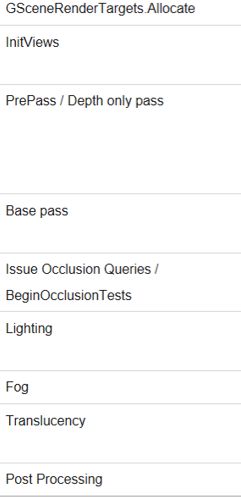
<https://www.unrealengine.com/zh-CN/tech-blog/how-to-add-global-shaders-to-ue4>

大概能知道一个大概

UNIFORMBUFFER作者的用法好像不行了！！！

<https://zhuanlan.zhihu.com/p/66288908> 新的

声明TM在中！！！



LoadingPhase插件加载阶段

ERHIFeatureLevel 这个特么的是ES2 ,ES3\_1这些!不是level

SetRenderTarget 这个接口已经TM的废除了。。

// Set the graphic pipeline state.

FGraphicsPipelineStateInitializer GraphicsPSOInit;

RHICmdList.ApplyCachedRenderTargets(GraphicsPSOInit);

GraphicsPSOInit.DepthStencilState = TStaticDepthStencilState<false, CF\_Always>::GetRHI();

GraphicsPSOInit.BlendState = TStaticBlendState<>::GetRHI();

GraphicsPSOInit.RasterizerState = TStaticRasterizerState<>::GetRHI();

GraphicsPSOInit.PrimitiveType = PT\_TriangleList;

GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI = GetVertexDeclarationFVector4();

GraphicsPSOInit.BoundShaderState.VertexShaderRHI = GETSAFERHISHADER\_VERTEX(\*VertexShader);

GraphicsPSOInit.BoundShaderState.PixelShaderRHI = GETSAFERHISHADER\_PIXEL(\*PixelShader);

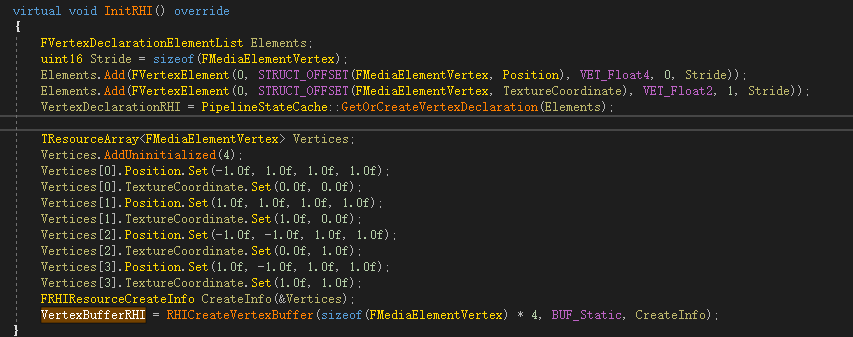
SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit);

针对这一段设置渲染管线参数的几个类做一下查询以及备注：

**FRenderResource** 提供基础渲染资源接口、以及初始化和释放的挂钩。从 FRenderResource（**FVertexBuffer**、**FIndexBuffer** 等）派生出的资源在用于渲染前需要被初始化、在被删除前需要被释放。**FRenderResource::InitResource** 只能从渲染线程调用，因此游戏线程上可调用一个辅助函数（**BeginInitResource**），使渲染命令入列，以便调用 FRenderResource::InitResource。RHI 函数只能从渲染线程调用（创建设备、视口等除外）。

RHI: <https://www.cnblogs.com/zhouxin/p/6418301.html>

关于CommandList以及Draw绘制的问题，在WebMVidoDecoder中（随便点的）看到，



设置一个quad四边形并且绘制！

嘿嘿嘿。而且跟PrimitiveType有关，PT\_TriangleStrip设置成这个后，能够正确绘制

## 传输、设置TEXTURE相关



根据图片设置，一个r g b a通道是uint8大小 （注，目前的图片操作，暂时没有什么用，所以先了解一下暂时搁置）

## ComputeShader

用GPU线程组进行计算。 numthreads[32,32,1]一个子线程组的数量,32 X 32并且只有一页。

UAV ： Unordered Access Views

ID3D11UnorderedAccessView interface:

A view interface specifies the parts of a resource the pipeline can access during rendering

视图接口指定管道在呈现期间可以访问的资源部分

[**ID3D11UnorderedAccessView**](http://msdn.microsoft.com/en-us/library/ff476639(v=VS.85).aspx) 使用像素着色器或者计算着色器访问一个无序资源.

computeShader拿出来单独计算？用RHICmdList计算

/\*

之所以调用\*星号，是因为TShaderMapRef重载了！

FORCEINLINE ShaderType\* operator\*() const

{

return Shader;

}

\*/

DispatchComputeShader(RHICmdList, \*ComputeShader, 32, 32, 1);

uint32 EncodedPixel = \*PixelPtr;

uint8 r = (EncodedPixel & 0x000000FF);

uint8 g = (EncodedPixel & 0x0000FF00) >> 8;

uint8 b = (EncodedPixel & 0x00FF0000) >> 16;

uint8 a = (EncodedPixel & 0xFF000000) >> 24;

Bitmap.Add(FColor(r, g, b, a));

0x000000FF 即

+ --------------- + --------------- + --------------- +- -------------- +

| 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 1 1 1 1 1 1 1 1 |

+ --------------- + --------------- + --------------- +- -------------- +

取了原始颜色指针所指地址的后8位，为（因为选择的格式为PF\_A32B32G32R32F看出应该在地址中的排列顺序是ABGR）

>>8右移八位即把最右边的R颜色舍弃，G变为最右边8位。

/\*\*

\* Encapsulates the render targets used for scene rendering.

\*/

class RENDERER\_API FSceneRenderTargets : public FRenderResource

RenderFog雾是用这个RenderTargets进行的

RHIRasterizerState : 光栅化的程序状态

继承自FSceneRenderer 的类，会带有一个Views的成员变量，视口，自行绘制的时候，循环views

检查是否应该为特定平台缓存着色器类型。

\* @param平台-检查的平台。

\* @return True如果这个着色器类型应该被缓存。

ShouldCompilePermutation (const FGlobalShaderPermutationParameters& Parameters)

RHI指的是Render hardware interface

/\* 获取shader TShaderMapRef的构造函数是使用一个shaderMap作为参数 \*/

TShaderMapRef< FTestPipelineVertex> VertexShader(View.ShaderMap);

关于DrawIndexedPrimitive函数的调用

<https://blog.csdn.net/ddupd/article/details/37600337>

## MeshDrawPipline

FPrimitiveComponent的GetDynamicElements这个收集操作通过渲染器的InitViews在可见性剔除之后发起SceneVisibility 中的InitViews

SceneVisibility中的InitViews后，GatherDynamicMeshElements中会调用所有的GetDynamicMeshElements来获取数据，放到收集器FMeshElementCollector& Collector中。

/\*\*

\* Base class of mesh processors, whose job is to transform FMeshBatch draw descriptions received from scene proxy implementations into FMeshDrawCommands ready for the RHI command list

mesh处理器的基类，其工作是将从场景代理实现接收到的FMeshBatch draw描述转换为RHI命令列表中的FMeshDrawCommands

\*/

class FMeshPassProcessor

## 自定义一个简陋的渲染管线

EShadingPath 有Mobile，Deferred和Num,加入一个我们自己的

SceneInterface中，GetShadingPath中用上我们自己的

在SceneRenderTarget.h中加上我们自己对应的RenderTarget格式

在SceneRendering.h中声明我们自己的渲染器，之后再CreateSceneRenderer中NEW，

再创建我们自己的CPP函数来写渲染。

/\*\*

\* This is a utility class for counting the number of cycles during the

\* lifetime of the object. It updates the per thread values for this

\* stat.

这是一个实用程序类，用于计算

对象的生存期。它将更新此属性的每个线程值。

\*/

class FScopeCycleCounter : public FCycleCounter

/\*\*

\* A set of views into a scene which only have different view transforms and owner actors.

将一组视图转换为只有不同视图转换和所有者角色的场景

\*/

class ENGINE\_API FSceneViewFamily

获取RenderTarget代码，其实很简单

FSceneRenderTargets& FSceneRenderTargets::Get(FRHICommandList& RHICmdList)

{

FSceneRenderTargets\* SceneContext = (FSceneRenderTargets\*)RHICmdList.GetRenderThreadContext(FRHICommandListBase::ERenderThreadContext::SceneRenderTargets);

if (!SceneContext)

{

return SceneRenderTargetsSingleton;

}

check(!RHICmdList.IsImmediate());

return \*SceneContext;

}

SetRenderTarget函数被拆分成了

RHICmdList.BeginRenderPass(RPInfo, TEXT("ClearStencilFromBasePass"));

RHICmdList.EndRenderPass();

在DeferredShadingRenderer中，RenderPass里面可以看到进行了

FGraphicsPipelineStateInitializer GraphicsPSOInit;

设置图形渲染管线状态的部分，在里面可以设置SHADER ，顶点等进行一些特殊的绘制！！！他还调用了

DrawRectangle(

RHICmdList,

0, 0,

DownsampledSizeX, DownsampledSizeY,

View.ViewRect.Min.X, View.ViewRect.Min.Y,

View.ViewRect.Width(), View.ViewRect.Height(),

FIntPoint(DownsampledSizeX, DownsampledSizeY),

SceneContext.GetBufferSizeXY(),

\*ScreenVertexShader,

EDRF\_UseTriangleOptimization);

}

去进行了一个矩形的绘制