CGINCLUDE和CGPROGRAM的使用：

\*在CGINCLUDE和cgprogram中都需要用endcg来标记结束

\*这两个块中基本都能写类似的代码块

\*在cginclude中实现的代码，可以在其他pass中的cgrogram中调用：例如:下面这种就可以在不同的pass中复用不同的vert和frag

Shader

{

    CGINCLUDE

        struct appdata

        {

            float4 pos:POSITION;

            fixed4 color:COLOR;

        };

        struct v2f

        {

            float4 position:SV\_POSITION;

            fixed4 color:COLOR;

        };

        v2f vert11(appdata i)

        {

            v2f o;

            return o;

        }

        v2f vert22(appdata i)

        {

            v2f o;

            return o;

        }

        fixed4 frag11(v2f i):SV\_TARGET

        {

            return i.color;

        }

        fixed4 frag22(v2f i):SV\_TARGET

        {

            return i.color;

        }

    ENDCG

    SubShader

    {

        pass

        {

            CGPROGRAM

                #pragma vertex vert11

                #pragma fragment frag11

            ENDCG

        }

        pass

        {

            CGPROGRAM

                #pragma vertex vert22

                #pragma fragment frag22

            ENDCG

        }

    }

}

预编译指令：

**ARB\_precision\_hint\_fastest** **最快的**，意思就是会用低精度（一般是指fp16），以提升fragment着色器的运行速度，减少时间。会把fixed和float变为half。

**ARB\_precision\_hint\_nicest** **最佳的**，意思就是会用高精度（一般是指fp32），可能会降低运行速度，增加时间。会把fixed和half变成float。

UNITY\_UV\_STATRS\_AT\_TOP宏：

来判断图形api是否为opengl。

Opengl坐标原点在左下角，向右与上递增

d3d坐标原点在坐上叫，向右与下递增

如果判断的不是ogl，那么就需要手动把纹理翻转一下:

#if UNITY\_UV\_STARTS\_AT\_TOP

if (\_MainTex\_TexelSize.y < 0) //其xyzw分别对应（1/width,1/height,width,height）

uv.y = 1-uv.y;

#endif

在采样深度图时，直接tex2D(\_CameraDepthTexture,i.uv)和SAMPLE\_DEPTH\_TEXTURE一样，只不过SAMPLE\_DEPTH\_TEXTURE返回的是单独的一个值，且对vr相机做了兼容。

RenderTexture相关知识详解：（例子按照opengles的实现来讲解）

本质：

Rendertexture的本质是将一个fbo链接到server-side的texture对象。

什么是server-side的texture：

渲染过程中，贴图最开始存在cpu中，这时候我们叫client-side的texture。最终要到gpu中，到了之后叫server-side的texture。这个texture在cpu到gpu之间的拷贝要考虑到带宽的瓶颈。

什么是fbo：

Fbo是gpu的里的渲染结果目的地，绘制的所有结果（color，depth，stencil）都存在这里，有一个默认的fbo链接我们显示器窗口区域。现代gpu可以不连接窗口区域，创建多个fbo，存放在gpu中，需要用的时候用它。

当fbo渲染完之后，有以下几种可以得到texture的形式

\*1\* 将fbo结果传回cpu这边的贴图，用readPixels。

\*2\* 将fbo拷贝到gpu上的texture，一般是copyTexImage2D，可以让gpu立即使用

\*3\* 将这个fbo直接关联在gpu上的texture对象，等于直接绘制在texture上，省去了拷贝，一般用framebufferTexture2D。（这是Unity中所使用的方式）

渲染到RenderTexture的几种方式：

\*1\* 创建个RT的asset，赋给摄像机，相机的实时渲染结果都在rt上了

\*2\* 把相机disable调，然后手动调一个render？？？？不知道

\*3\* 想用特殊shader去渲染RT时，调用camera的RenderWithShader这个函数，用指定的shader去渲染场景

\*4\* 用Graphic.Blit（src，dest，mat）

其他注意问题：

\*1\* RT的支持格式很多，最基本的ARGB32是支持的，在使用前，查询一下当前gpu支持哪些格式

\*2\* 想把RT拷贝回内存，拷贝回来的的rt和texutre的格式要一样，而且必须为ARGB32或ARGB24这种基本类型

\*3\* 不要频繁new RT出来，使用GetTemporary和ReleaseTemporary，他们内部会复用一些大小格式一样的rt资源，而且还会调用DiscardContents，很高效。

\*4\* DiscardContents这个RT的接口很重要，最好每次像一个已有内容的rt上绘制前去调用这个，得到的优化大致是：在一些基于Tile的GPU上，RT和tile存在各种同步，调用了这个方法后，告诉gpu我要重新绘制他，这块rt的内容你先不要管了

一些宏指令：

[loop]：作用于for循环上，表示在编译时不进行循环展开

[unroll]：同样在for上，表示要展开循环

Unity一些渲染标签：

[ImageEffectOpaque]：加了后，渲染后会在opaque之后，transparent之前。常用使用深度渲染不印象透明度的渲染，例如ssao等。

**一些冷知识：**

\*\*贼坑，COMPUTE\_DEPTH\_01不是用在屏幕空间去计算的，他是用在物体上。

\*\*Lighting设置中指定skybox可以让天空盒在当前场景下所有的相机生效，想让指定的相机绘制指定的天空盒，可以在相机上添加skybox组件。

\*\*Untiy的fov指的是竖直方向的视椎体夹角

\*\*NDC是左手坐标时：z越小越靠前。。NDC是右手坐标系时：z越大越靠前

\*\*DX使用左手坐标系，OGL使用右手坐标系

\*\*EarlyZ是个好东西啊，在GPU内部自己实现的。但是如果手动在fragment中修改深度或开启alphatest或discard像素，EarlyZ就会关闭,知道下一次clear z-buffer或pass才会开启,因为上述操作可能会在fragment和late-z阶段导致深度的变化，所以GPU会自动关闭early-z。因为如果还有earlyz的话，比如某个像素通过了深度测试，但是最后却被alphaTest给干掉了，所以就会出现莫名其妙为透明的状态。所以为了保障这一点，earlyz就会关闭，就会变成传统的frag之后的深度测试，造成overdraw。所以一般用alphablend来代替之。 同时，alphatest是最后一次拒绝像素写入的机会。

\*\*glsl中ddx和ddy的计算是以像素左上角为原点的。Unity中用的hlsl的计算是以右下角为原点进行计算的。

混合顺序：

模板测试（Stencil Test）  
深度测试（Depth Test）  
透明测试（Alpha Test）  
混合模式（Blend Mode）  
更新帧缓存中的颜色值

Unity的tiling和offset：

Tiling就相当于缩放，offset相当于位移

TRNASFORM\_TEX就是通过xxx\_ST来实现的，st的xy相当于tiling，zw相当于offset。

\_LightAsQuad：

Unity通过\_LightAsQuad变量告诉我们正在处理那种情况，在处理四边形（直射光时）返回1，否则返回0;

ClipSpace和NDC和透视除法：

VertexShader输出的，只是单纯的ClipSpace，然后GPU会进行剔除与裁剪，视椎体剔除和背面剔除？然后GPU再做透视除法将顶点变换到NDC。

NDC是一个xyz都在[-1,1]的立方体，但是DX的z范围是[0,1]！！！，Unity中可以用UNITY\_NEAR\_CLIP\_VALUE获取z方向的近裁剪面值，ogl是-1，dx是0。

假设ndc中的xywz是(x,y,z,1)，那么反推回去clipPos中的是(wx,wy,wz,w)，其中，这个w=-ViewZ，也就是ViewSpace中的Z值，很显然，ClipSpace和NDC的范围是不一样的。

透明度混合公式：

前景色的alpha = a

前景色的rgb = c

背景色的rgb = d

Res = a\*c+d\*(1-a)

矩阵的一些运算：

矩阵乘法不满足交换律，但满足结合率

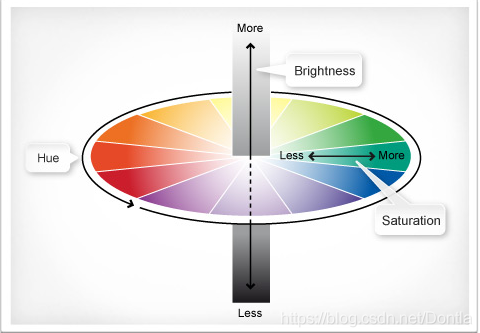
(AB)C = A(BC) 正确

AB = BA 错误

(A+B)C = AC+BC 正确

美术知识：

饱和度和对比度的区别与作用：



饱和度：

指颜色的纯度或强度，简单来说就是颜色中灰色含量的高低

高饱和度高亮度的颜色适合做引导性的按钮

低饱和度低亮度的颜色适合做长时间观看的颜色

对比度：

指颜色最高亮度和最低亮度的比值，比值越大颜色明暗差异越明显

对比度越大让图像越醒目，反之颜色会灰蒙蒙的

锐化：

突出图像边缘的部分，让图像棱角变得清晰

**写Shader的好习惯：**

声明\_CameraDepthTexture时，用

UNITY\_DECLARE\_DEPTH\_TEXTURE(\_CameraDepthTexture);

光线追踪知识：

相机总是放在离屏幕1个单位的距离，而且永不变。按照惯例，相机总是看向z的负方向. RenderMan, Maya, PBRT and OpenGL align the camera along the negative z-axis and we suggest developers to follow the same convention

CommandBuffer进行后处理：

对纹理不能同时进行读和写哦。

void InstanceProperty()

{

if(renderCamera == null)

renderCamera = this.GetComponent<Camera>();

if(filterMat == null)

{

filterMat = new Material(Shader.Find("Unlit/FilterTest"));

}

if(\_cmd == null)

{

\_cmd = new CommandBuffer();

}

\_cmd.name = "FilterTest";

\_cmd.Clear();

int screenCopyID = Shader.PropertyToID("\_ScreenCopyTexture");

\_cmd.GetTemporaryRT(screenCopyID,-1,-1);

RenderTargetIdentifier id = new RenderTargetIdentifier(rt);

\_cmd.Blit(BuiltinRenderTextureType.CurrentActive,screenCopyID);

\_cmd.Blit(screenCopyID,id,filterMat);

\_cmd.ReleaseTemporaryRT(screenCopyID);

//Debug.Log(rt.width);

renderCamera.AddCommandBuffer(CameraEvent.AfterForwardOpaque,\_cmd);

}

void OnRenderImage(RenderTexture source, RenderTexture destination)

{

Graphics.Blit(rt,destination);

}

面试的知识：

Builtin和urp的区别：

详细：<https://www.cnblogs.com/Jaysonhome/p/12900808.html>

内建的是老的渲染管线，很多渲染流程都是固定在底层的没有暴露出来难以控制。

Urp的新的渲染管线，在2019.3以上才有，支持可编程，有shadergraphic等。性能更好，有SRP Batcher，有内置的后处理。主张用hlsl，cg已经老了

SRP学习记录：（主要是记记api怎么用，api太多了我曹！）

renderContext.SetupCameraProperties():把当前上下文设置为当前相机的上下文

renderContext.ExecuteCommandBuffer(\_cb):把当前指令队列提交至当前context处理

CommandBuffer相当于一个指令的记录表，先记录后执行。CommandBuffer可以有多个，可以记录不通的指令，Context只能有一个，且最终所有的commandbuffer都要合并到context中来，在submit时一次性执行。

**CommandBuffer学习：**

什么是：是用来存储渲染指令（例如setrendertarget和drawmesh等等。。）的缓存区。

ScriptableRendercontext：指定自定义管线时的渲染状态和声明绘制指令，负责调度和提交渲染状态以及，提交渲染指令到Gpu。

RenderStateBlock：

**RenderTarget和RenderTexture。**

流水线的输出结果一般都是一张纹理，即RenderTexture。那么这个纹理总不能输出到空气中，所以输出的目标地就是RenderTaget

Buffer.CopyTexture，如果不写mipmaplevel，默认拷贝到mip1，这个好像在quailty里可设置。