Part01-Cel Shading/Toon Shading

Jet Set Radio/ The Legend of Zelda: The Wind Waker/ Gravity Rush



1)创建并使用后处理材质
2)创建 Cel Shader
3)仅对指定 Mesh 使用 Cel Shader
4)使用查找表来控制色带

图下的例子-三个 band：阴影、中间色调、高光
错误的概念：描边的着色不一定是 Cel Shading

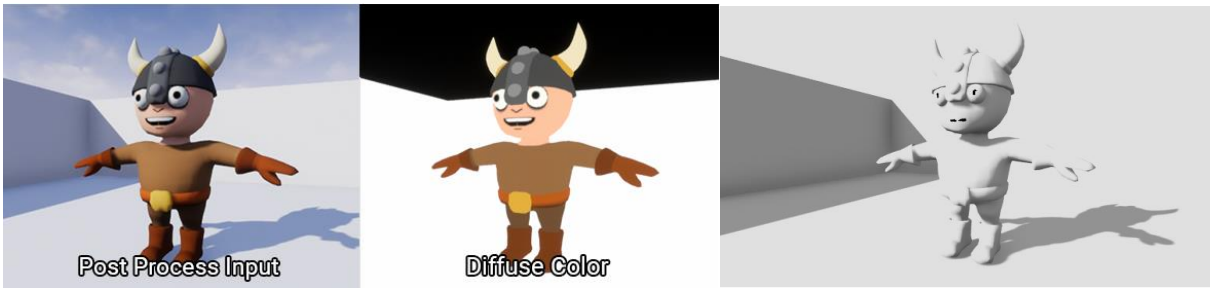

Cel Shading 最常用的实现方法就是比较法线与光线的方向。通过计算二者的点乘，你会获得一个[-1，1]的值。-1 就是表面与光线相反、0 是互相垂直、-1 是方向相同。通过把点乘的结果阈值化，就可以得到多条色块。
比如你设定点乘结果高于-0.8 的为暗色，高于-0.8 的为亮色，就会创建出两个色块。但它的局限性就是，其他的光无法影响着色物体，同时其他物体也不能对着色物体投影。为了修复这个问题，你需要换种方式。不要计算点乘，你应该**计算表面是如何被照亮的**，随后再去阈值化。


2 Band
Cel Shading


这个教程里用到的 Cel Shading 是后处理特效。后处理可以让你在引擎已经完成图像渲染的情况下对其修改，常见的使用领域有景深、运动模糊和辉光效果。

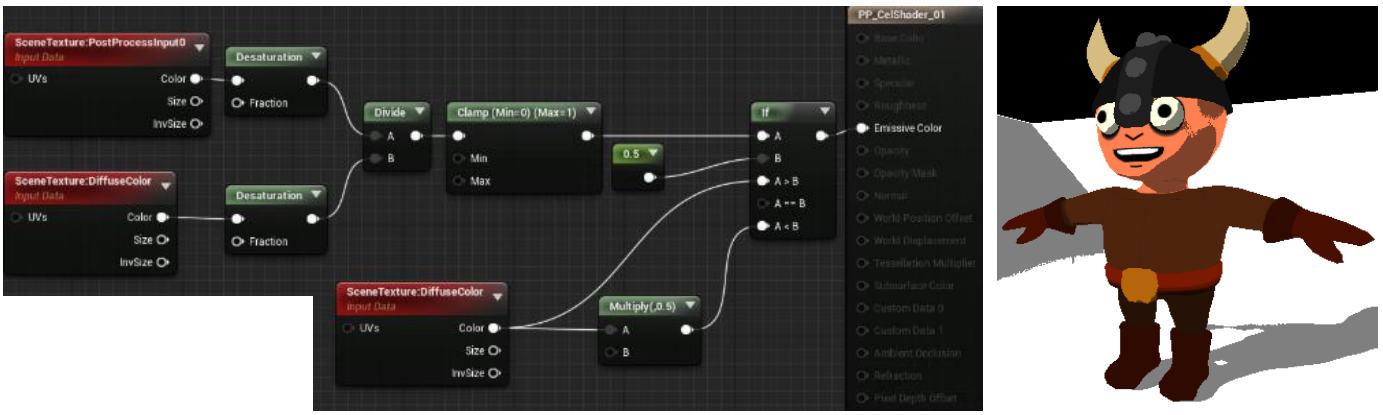
第一步就是计算每个像素是如何被点亮的，我们称之为光照缓冲（Lighting Buffer）
当虚幻引擎把一张图渲染到屏幕上时，它会把 pass 存储到缓冲区。你需要访问以下两个缓冲区来计算光照缓冲：

1） 后处理输入：一旦 Unreal 开始执行光照与后处理，它会把图像存储到这个 Buffer 中。也就是当你不执行任何后处理操作时，Unreal 显示给你的画面。
2） 漫反射颜色：即场景中没有任何光照信息与后处理时的情况。它会包括屏幕上所有物体的漫反射颜色。



你需要用 SceneTexture 节点来访问这两个缓冲区。（即 Scene Texture Id 改成 PostProcessInput0 和 DiffuseColor）
光照缓冲应该仅包含灰度值（就是描述一个东西有多亮）。这就意味着你不需要这两个缓冲区中的颜色信息。为了舍弃它，所以把两个 SceneTexture 的 Color 输出到 Desturation（降低饱和度）节点（其实直接 Mask(R)就可以？）然后用 Divide 节点除一下。随后 Clamp 限定在[0,1]，这是为了让执行阈值化更简单。
（这里的 Divide 其实就是因为引擎不能直接访问光照的一种 Trick）

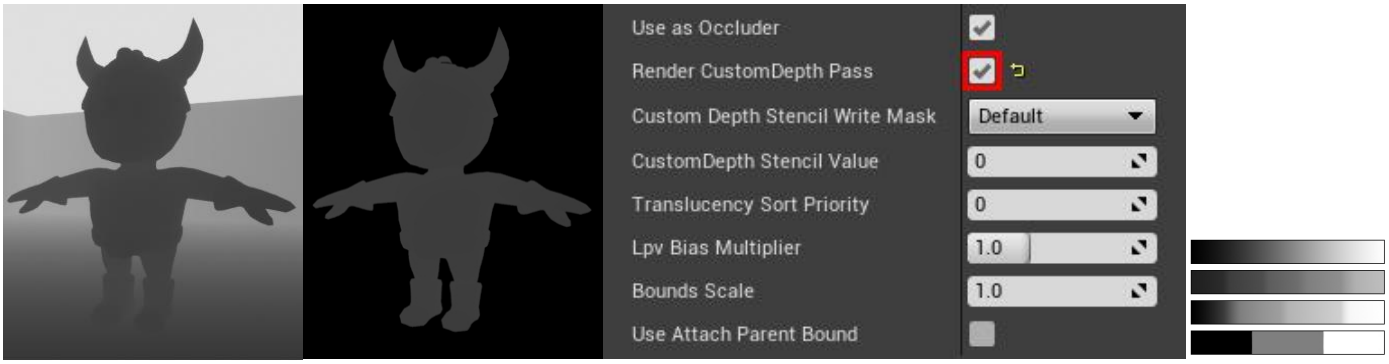
第二步是利用光照缓冲来创建阈值化。
对于这个 Cel Shading 来说，任何大于 0.5 的像素使用正常的漫反射颜色；低于 0.5 的使用漫反射颜色亮度的一半。

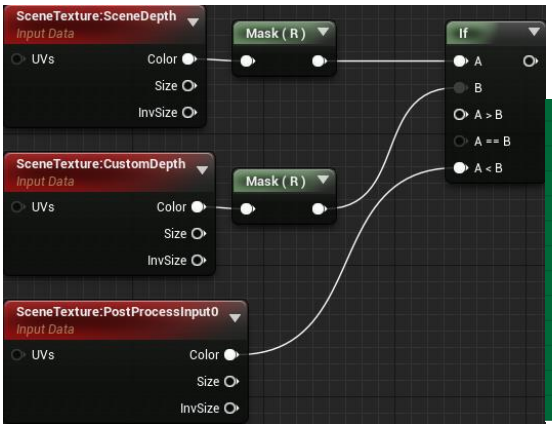


但是右边这个什么鬼？？把 After ToneMapping 改成 Before ToneMapping 就好了！

来介绍下 ToneMapping：
在向玩家展示一幅图像的时候，虚幻引擎会执行一个叫做色调映射（ToneMapping）的工作。色调映射的原因之一是让图像看起来更加自然。从下面两张 Post Process Input buffer 的图像中可以看出，色调映射前的高光太亮了，处理后会变得略加柔和。一旦你使用后处理输入缓冲图像来计算什么，你必须使用色调映射前的原始值才行。

第三步是如何把 Cel Shading 仅用在指定的模型上。
为了把后处理特效独立出来，你需要一个叫做自定义深度（Custom Depth）的东西。跟 PostProcessInput 和 DiffuseColor 一样，这是一个可以用在后处理材质中的缓冲（缓冲就是中间生成）。
但首先你需要知道场景深度缓冲是啥。场景深度存储了每个像素距离摄像机平面的距离。自定义深度存储的是同样的信息，不过只是针对你指定的 Mesh。对 Mesh 如图设定 CustomDepth 后在 Material 中增加一个深度比较（**远处深度值反而小？**）就可以了。



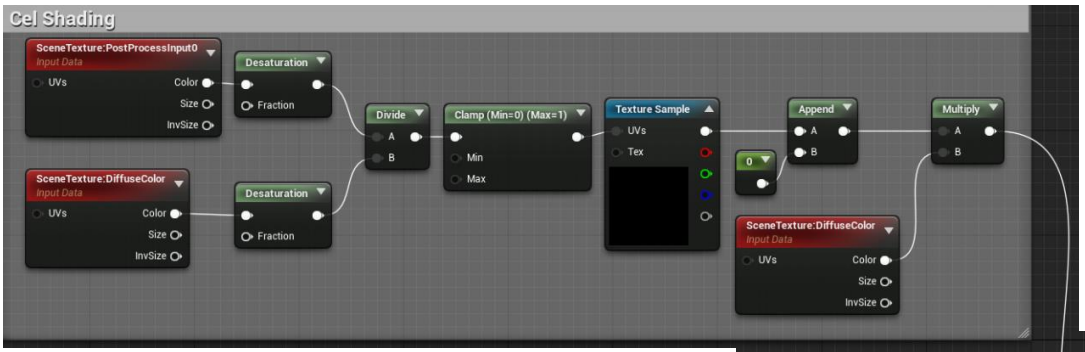


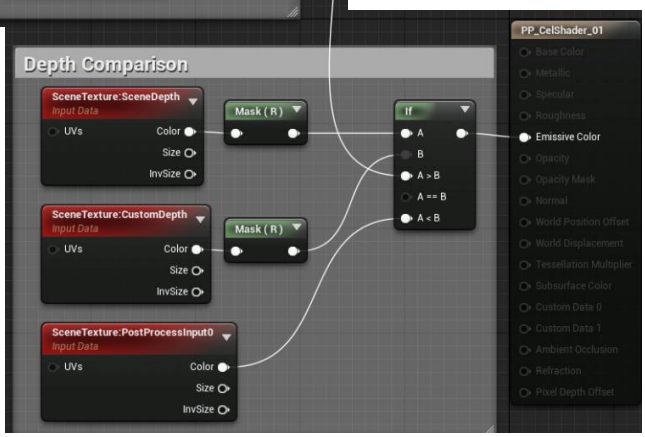
	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	14	16	18	20	22	24
3	3	6	9	12	15	18	21	24	27	30	33	36
4	4	8	12	16	20	24	28	32	36	40	44	48
5	5	10	15	20	25	30	35	40	45	50	55	60
6	6	12	18	24	30	36	42	48	54	60	66	72
7	7	14	21	28	35	42	49	56	63	70	77	84
8	8	16	24	32	40	48	56	64	72	80	88	96
9	9	18	27	36	45	54	63	72	81	90	99	108
10	10	20	30	40	50	60	70	80	90	100	110	120
11	11	22	33	44	55	66	77	88	99	110	121	132
12	12	24	36	48	60	72	84	96	108	120	132	144

第四步是使用查找表（Lookup Tables，LUT）增加更多的 Color Band 并让其边缘柔和过渡。

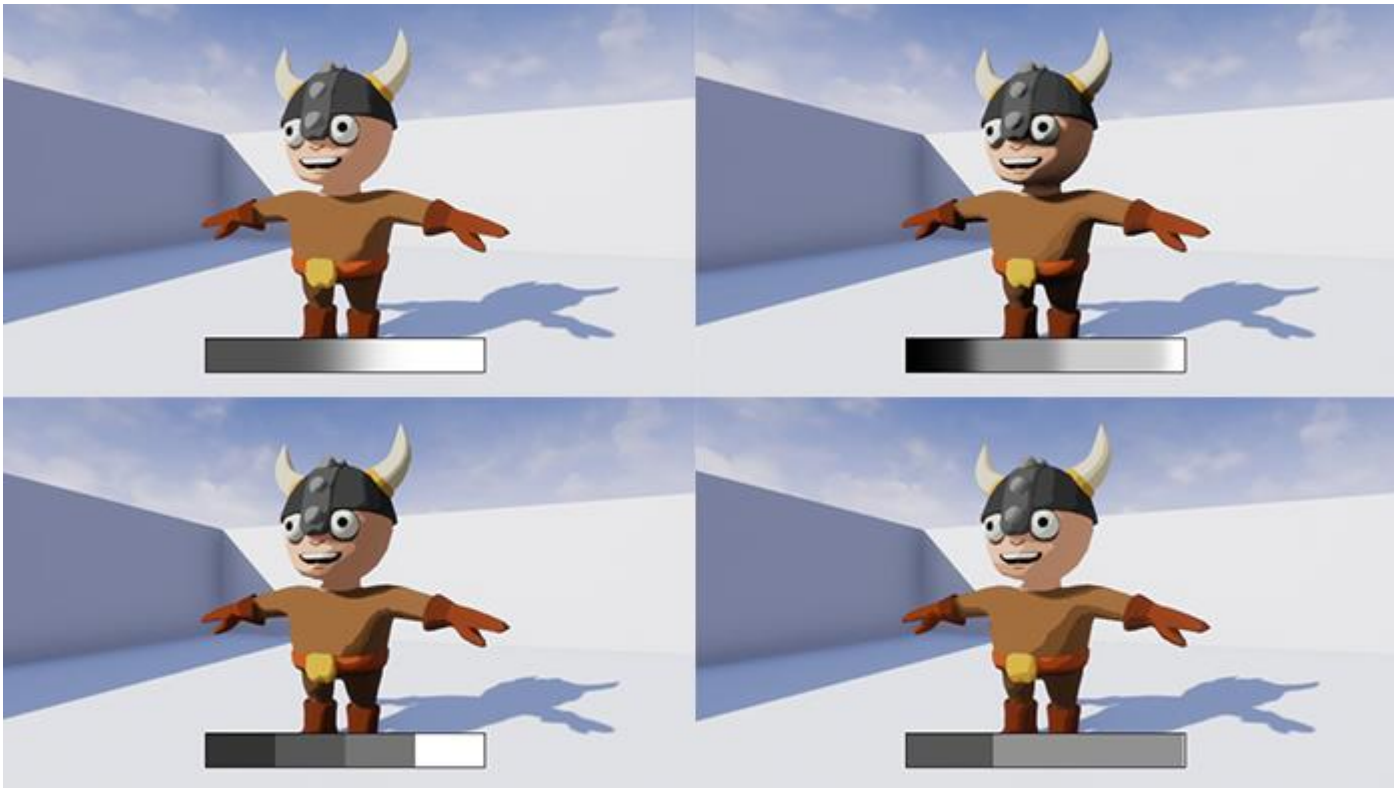
LUT 是什么？一个可以通过输入来访问的预先计算好的数组。上图展示了一个乘法表（multiplication table），输入就是乘数（multiplier）和被乘数（multiplier）。在 Cel Shading 中，LUT 就是一个包含了梯度的纹理。现在为止，你计算阴影的方式还只是简单取漫反射颜色的一半。那么现在你不再是使用常数 0.5 来做乘法了，你要采用 LUT 中的值。这样做的话你可以控制色带的数量以及他们之间的过渡。你可以从 LUT 的样子来直观地判断着色后会是啥样。

获得一个 LUT 后你需要做一些设置。Unreal 在渲染的时候会把所有 **sRGB** 的纹理转换成**线性纹理**。这可以让虚幻引擎更容易地执行渲染计算。sRGB 设置对于描述外观地纹理而言是十分优秀的，但像是法线贴图和 LUTs 的值是用来进行数学计算的（描述外观的纹理就不是进行数学计算的了吗- -？）。所以这个值必须非常非常准确才行。另一个要改的设置就是 Tiling Mode 要变成 Clamp。





为什么要加 Append 呢？因为要把 LUT 的输出变成四通道才能和 DiffuseColor 做乘法**（为啥 LUT 输出是 3？）**



Part02-Toon Outline

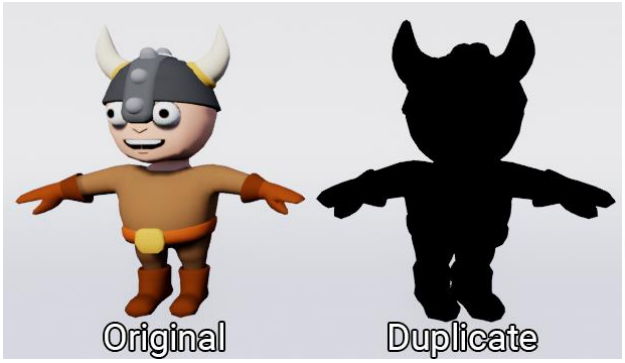
Okami/ Borderlands/ Dragon Ball FighterZ



1)使用翻转（inverted）的 Mesh 来创建边缘
2)使用后处理和卷积来创建边缘
3)创建并使用材质函数
4)邻近像素点采样

第一个方法

核心在于复制你的目标 Mesh，给 Mesh 一个 Solid color（一般是黑色）然后放大一点点来获得轮廓。



但是这张直接叠上去的话会把整个原模型遮挡住。为了修补这个错误，把复制品的法线翻转一下，并把背面剔除打开（backface culling）。这时候你只能看到里面的面，看不到外面的面了。

这样原模型就能透过复制品显示出来；同时因为复制品大一点点，就得到了轮廓线。

其优点在于，因为轮廓线是用 polygon 做的，你总是能得到清晰干净的线条；其外观与厚度可以通过移动顶点来简单调整；轮廓线会根据距离收缩（shirnk），当然这也可能是个缺点。其缺点在于，一般来说，我们不会去勾勒网格内部的细节；因为轮廓是 polygon，有时会发生遮挡裁剪（看左下图的地面鞋子部分）；性能不友好，Mesh 数量翻倍；对于平滑的凸（convetx）网格能够显示得比较好，但是那些硬边缘和凹陷的地方会形成边缘空洞（歪？）。

第一步来创建一个翻转的 Mesh Material。

你必须把面向外部的多边形（outward-facing polygon）遮罩住，留下内侧多边形（inward-facing polygon）。**因为这个步骤需要遮罩，所以会比在三维软件中手动创建 Mesh 要消耗大一些。**（歪？）

先把 Blend Mode 改成 Masked（**就是 Unity 中一般说 Alpha Test?**），这可以让你选定可见或者不可见的区域（比如铁网材质）。你可以通过编辑 Opacity Mask Clip Value 来调节阈值；

把 Shading Model 改成 Unlit，使得光照无法影响 Mesh。

开启 Two Sided，Unreal 引擎是默认剔除背面的，开启双面显示能够关闭背面剔除，这样才能够显示 inward-facing polygon。随后创建一个四向量参数，命名为 OutlineColor 并与 EmissiveColor 连接。

为了把 outward-facing polygon 遮罩住，创建一个 TwoSidedSign 节点并乘-1，结果输出到 Opacity Mask（歪？）。TwoSidedSign 对正面输出 1，对反面输出-1。这意味着正面是可见的而反面不可见。上面这个步骤就是让它反过来。

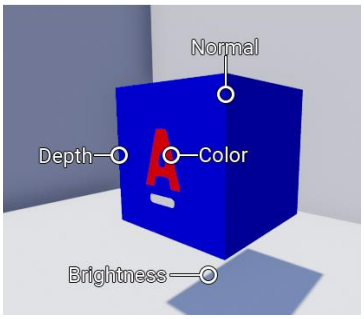
我们还要增加轮廓线厚度。Unreal 引擎中，你可以使用 World Position Offset 来移动每个顶点的位置。我们将 VertexNormalWS 乘上一个数字（命名为 OutlineThickness）并连到 World Position Offset。

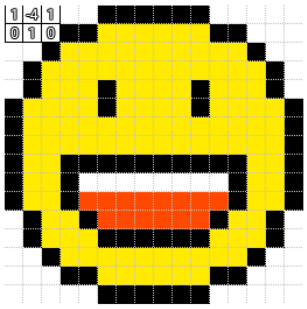
第二步是复制 Mesh。

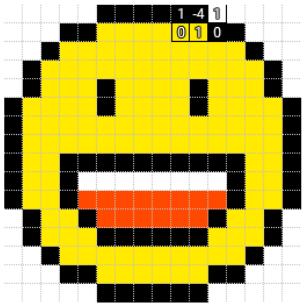
在原蓝图中增加一个 StaticMesh 的 Component 作为 Mesh 的子，并命名为 Outline。随后将其 Static Mesh 和 Material 分别赋值（上面制作材质的实例）即可。

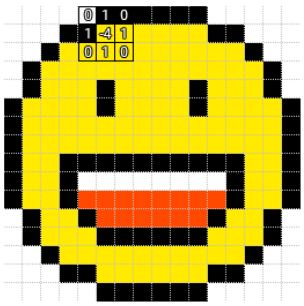
第二个方法

使用边缘检测的方法创建后处理轮廓。这是一项检测图像中区域间不连续的技术。下图展示了一些不连续的种类：







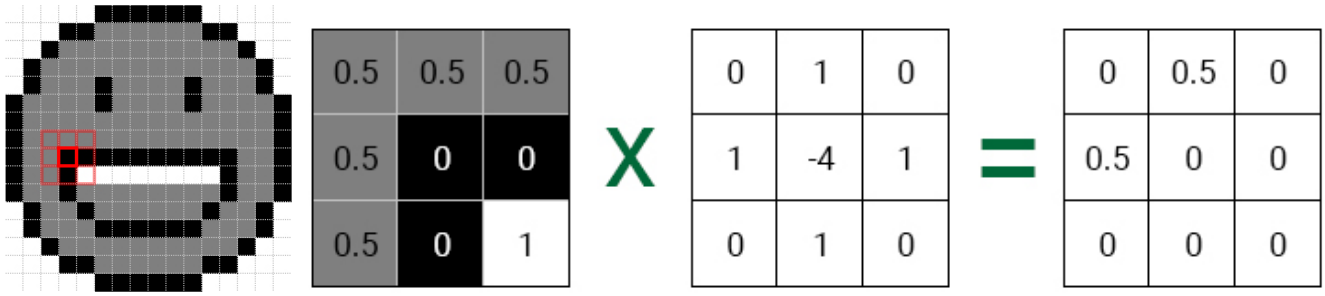


其优点在于可以快速应用于整个场景；因为 shader 总是作用于每个像素，所以能节省性能开销；不同距离下轮廓线粗细相同（虽然有时候也可能是缺点）；因为是后处理效果所以不会因为几何关系被裁剪。

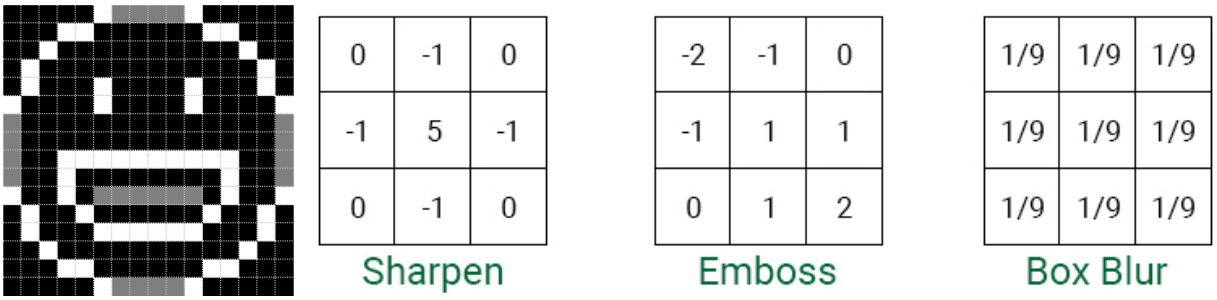
其缺点包括：往往需要多种边缘检测算子来捕获所有的边缘，这对性能其实是有影响的；容易产生噪点，因为这意味着边缘往往出现在变化非常大的区域上。边缘检测的一般方法是对每个像素执行**卷积**操作。

在图像处理中，**卷积是输入两组数字并输出单个数字的运算**。首先你要有一组网格数据（就是我们说的卷积核），用其中心去遍历每个像素。上面是一个 3x3 卷积核沿着图像上面两排运动的例子。

把每个 kernel entry 乘以它对应的像素。我们拿嘴巴左上角的像素举例，为了简化运算我们把图像转换为灰度。我们把卷积核中心对准目标像素，用对应覆盖的像素值去乘卷积核内的数据：

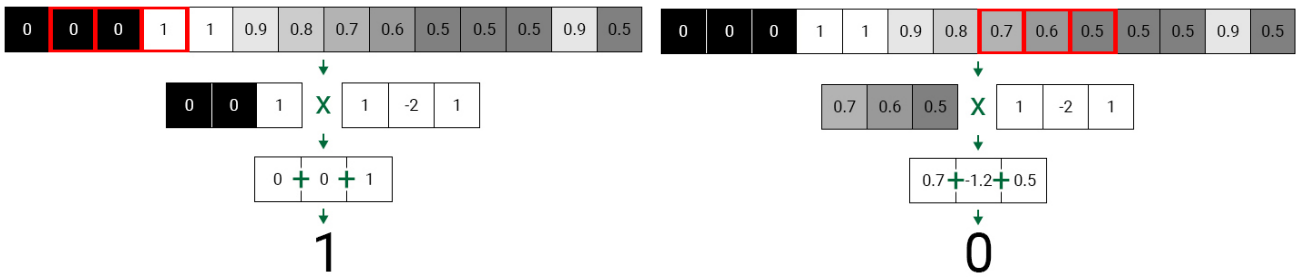


最后把所有的数据加起来，就是新的中心像素值。在这个例子中，新的中间值是 1，下图展示了卷积完成后的图像：



你想实现的效果取决于你用了哪种卷积核。例子中的卷积核是用来做边缘检测的，上图还有些别的例子。

这就是图像编辑程序中的滤镜效果。为了检测一个图像的边缘，你可以使用拉普拉斯边缘检测算法。因为拉普拉斯算子测量的是斜率（slope）的变化。变化较大的区域会偏离 0，表明其为边缘。为了方便理解，我们来考虑拉普拉斯算子为一维时候的情况，左边计算结果表明原像素很可能是一个边缘。右边是一个变化不大的区域：尽管每个像素的值是不同的，但是梯度是线性的。这意味着它在**斜率**上没有变化，即目标像素不是边缘。

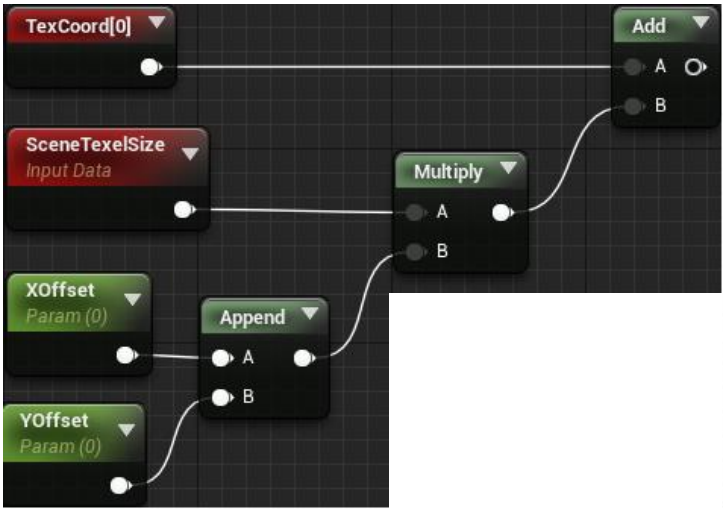


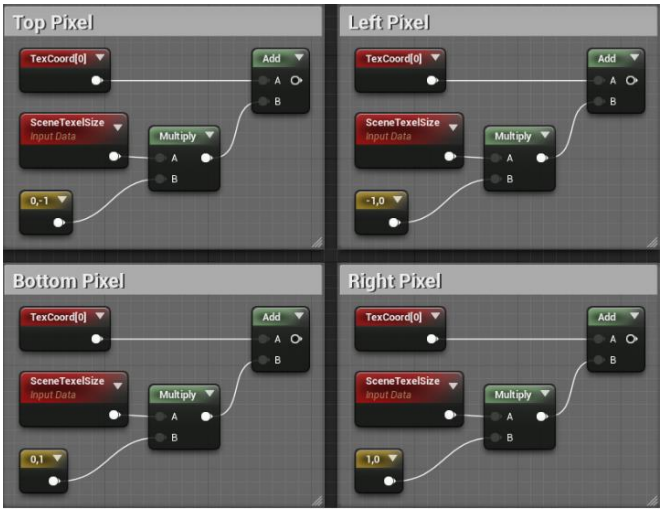
那么如何在 Unreal 引擎中构建拉普拉斯边缘检测呢？

第一步是采样到邻近点。

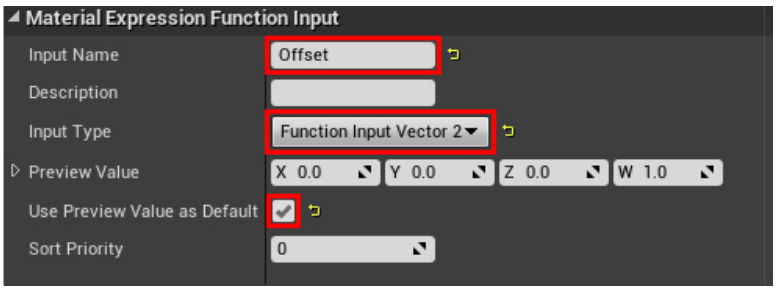
你可以用 TextureCoordinate 来获取当前像素的坐标，也就是我们说的 UV。进一步使用 TextureCoordinate 的 Offset

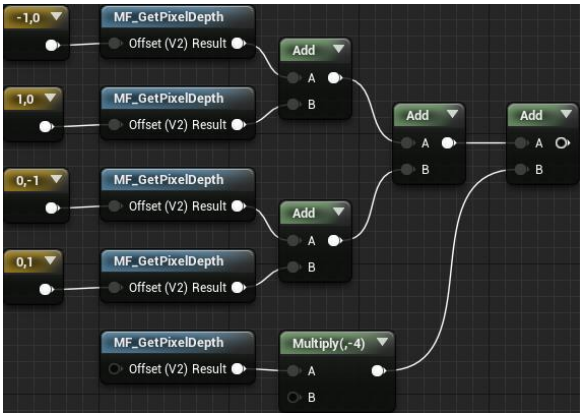
就可以得到另一个像素。在一个 100x100 的图像中，每个像素在 UV 空间中占 0.01 的大小，为了采样右边的像素，你可以在 x 轴上加 0.01。但这里有个问题，如果图像分辨率变了，那像素尺寸也会变。比如用相同 0.01 步长采样，就会跨越两个像素。**SceneTexelSize** 节点可以解决这个问题，那么获取上下左右偏移就如右图所示**（为什么是纹素）**。

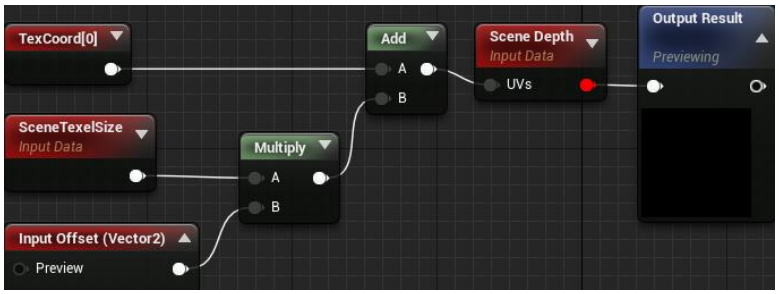


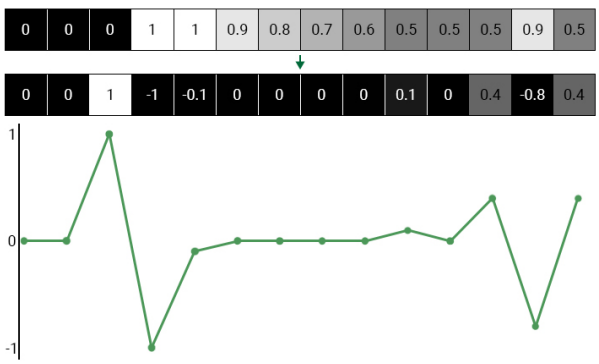


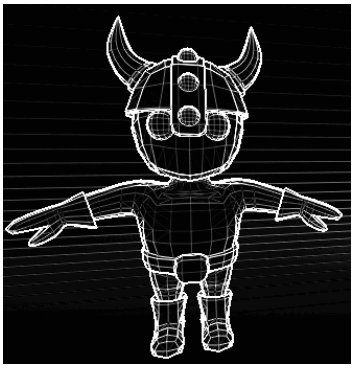
当然如果你按右图那样去计算很快就会变得一团糟。我们使用材质函数（Material Function）来整理图表。
一个材质函数跟你在蓝图中或者 C++中用到的那些函数一样。
创建一个材质函数，命名为 MF_GetPixelDepth。首先创建一个接受偏移的输入结点（FunctionInput），并对其执行以下设置：其次用像素大小乘以偏移，并将结果与 TexCoordinate 相加。
最后将结果输入 **SceneDepth 进行深度缓冲采样**。（你也可以用 SceneTexture，把 id 改成 SceneDepth）
总结来说就是这个函数实现了输入像素坐标，输出像素深度的功能。
第二步是用这个函数在深度缓冲中进行卷积。
创建四个二维向量，分别表示上下左右四个偏移，用 MaterialFunctionCall 节点引用到之前创建的函数并依次连接。右图实现了拉普拉斯算法的输出。为了解决一些像素值输出负值，所以还要加个 Abs 再连到自发光颜色上。









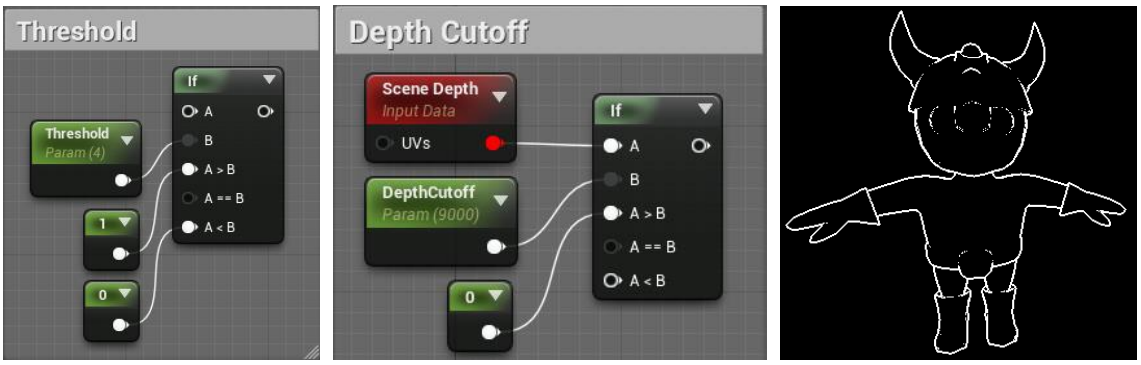


但是轻微的深度变换都会被检测出来，同时因为背景是个 sphere，会被检测出同心圆形状。

第三步是用增强阈值化来修复这些问题。

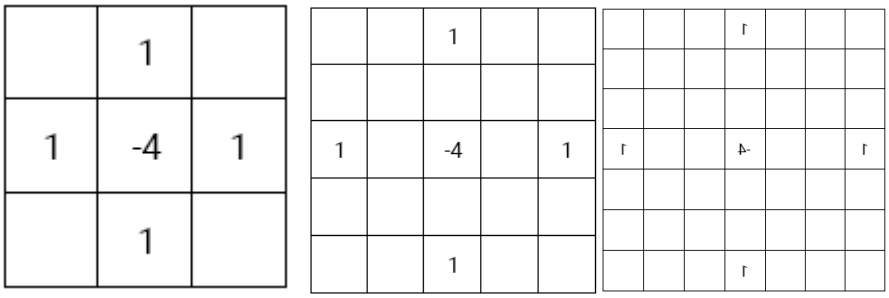
首先解决轻微深度变换的问题。在材质编辑器中做如下设置，并与前面的输出节点依次连接。

这样像素值在 4 以上、深度小于 9000 才会被显示出来。



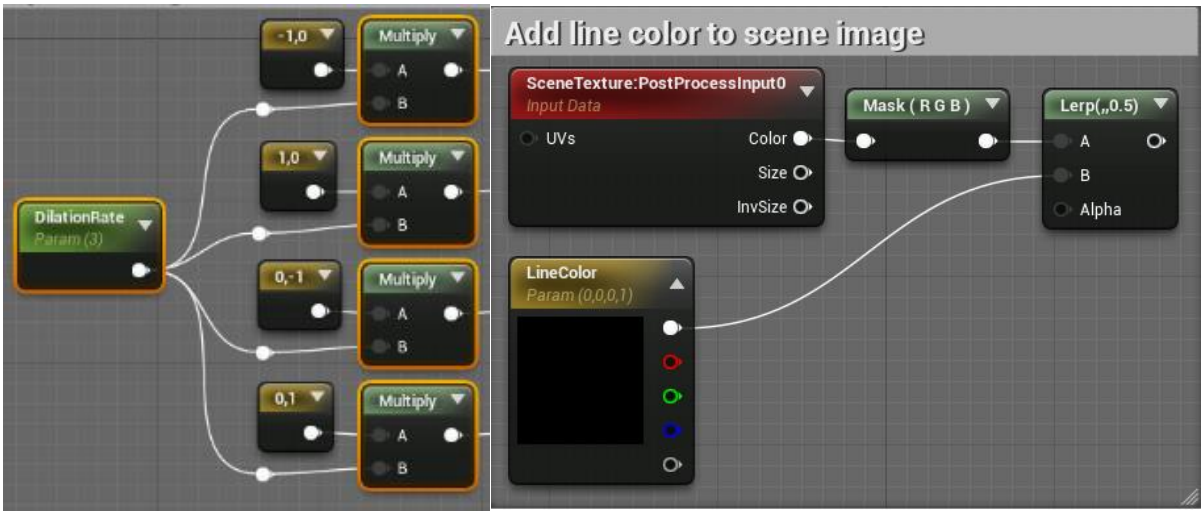
第四步是希望能够改变描边的大小。

我们需要一个更大的卷积核。一般来说，更大的卷积核会对性能造成更大的影响。**因为这意味着你采样的点也会更多**（不都是每个像素采样一次吗：应该是指从 4 点采样变成 8 点采样？）。有没有一种办法能够增大卷积核的同时其性能保持跟我们前面采用的 3x3 一样呢？这就引入了扩张卷积/空洞卷积（dilated convolution）你需要把偏移量向外扩张，我们这里引入了扩张率，它定义了卷积核因子彼此间的距离。（从左至右分别为扩张率为 1、2、3 的情况）这种情况即允许在采样相同像素数量的情况下扩大卷积核。在材质编辑器中，我们增加一个 ScalarParameter 并命名为 DilationRate，设置其值为 3。即偏离中心像素 3 个像素单位采样。



第五步是把线加到原始图像上，形成勾边效果。

在材质编辑器中添加右边所示节点。



如果你想用边缘检测做更多的事儿，试试看把它应用在法线缓冲中。你可以获得深度边缘检测中不会出现的边缘。卷积应用广泛，包括人工智能、音频处理等等。我鼓励你们通过创造一些诸如锐化、模糊等特效来继续探索它。有些特效仅仅通过改变卷积核的值就能实现。关于卷积的可互动内容可以访问 http://setosa.io/ev/image-kernels/

我还强烈建议你看下 GDC 的这个展示 https://www.youtube.com/watch?v=yhGjCzxJV3E

他们也使用了翻转 Mesh 来实现 outerlines，但是对于 innerline 他们用了个十分简单却天才的技术。

Part03 Custom Shaders Using HLSL

对艺术家来说，基于节点的材质编辑器是创作 shader 的良好工具。但是它也存在一定的局限性。比如你不能实现一些循环或者选择向的功能。幸运的是，你可以通过撰写自己的代码来解决这个局限性——实现一个允许你写 HLSL 代码的自定义节点。

1)实现一个自定义节点并设置其输入

2)把材质节点转换为 HLSL 代码

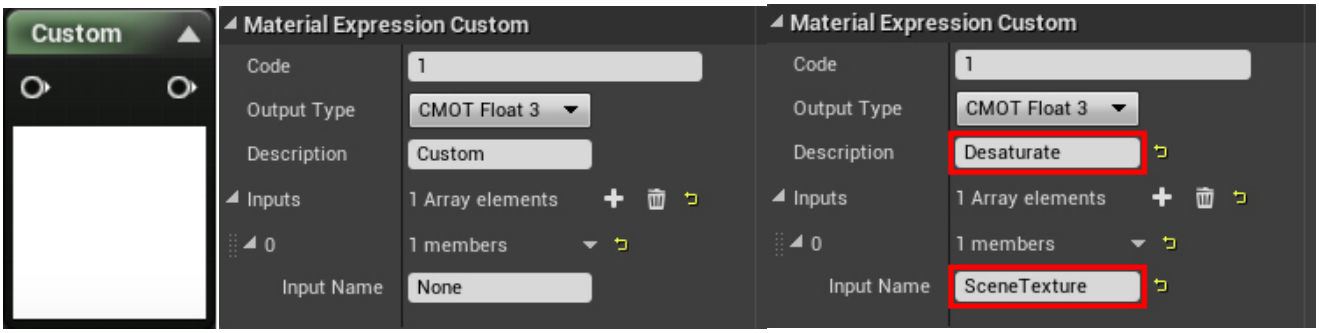
3)用外部文本编辑器来编辑 shader 文件

4)实现 HLSL 函数

为了遍及上面这些内容，你将会使用 HLSL 来降低场景画面饱和度、输出不同的场景纹理并实现高斯模糊。

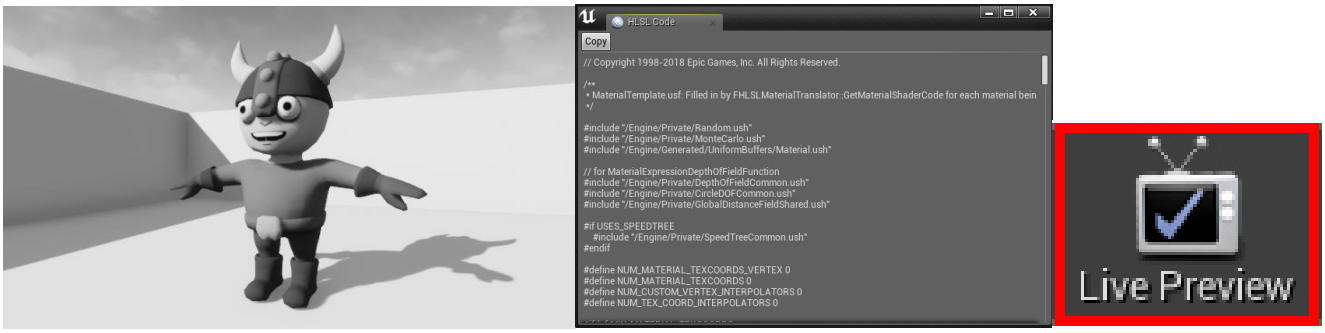
第一步先来实现一个自定义节点

在后处理材质中新建 Custom，其允许有多个输入和一个输出。下图为其细节面板。HLSL 代码放在 Code 处；输出类型可以选择单一值 **CMOT Float 1** 到四通道向量 CMOT Float 4；Description 是显示在节点上方的自定义文本，我们在这里把它改成 Desaturate；你可以在 Input 处添加并命名输入线脚，随后你可以在代码中通过名字来引用它，我们在这里把 Input0 的 InputName 改成 SceneTexture。



在 Code 栏中输入如下，再将 PostProcessInput0 连接到 SceneTexture 线脚**（节点输入输出类型还是不懂）**：

```
return dot(SceneTexture, float3(0.3,0.59,0.11));
```



第二步来把材质节点转换成 HLSL 代码

Unreal 引擎会为所有对最终输出有贡献的节点生成 HLSL 代码。在 PostProcessInput0-EmissiveColor 图表中，引擎会为 SceneTexture 节点生成 HLSL 代码。选择 Windows/Shader Code/HLSL Code 可以看到所有材质的代码。如果 HLSL Code 窗口是黑的，你需要在工具栏上打开 LivePreview。

生成的代码有上千行，为了便于查找，我们把它复制到外部文本编辑器中，并查找 CalcPixelMaterialInputs 函数。引擎用它来计算所有的材质输出。在函数底部你会找到每个输出的最终值：

```
PixelMaterialInputs.EmissiveColor = Local1;
PixelMaterialInputs.Opacity = 1.00000000;
PixelMaterialInputs.OpacityMask = 1.00000000;
PixelMaterialInputs.BaseColor = MaterialFloat3(0.00000000,0.00000000,0.00000000);
PixelMaterialInputs.Metallic = 0.00000000;
PixelMaterialInputs.Specular = 0.50000000;
PixelMaterialInputs.Roughness = 0.50000000;
PixelMaterialInputs.Subsurface = 0;
PixelMaterialInputs.AmbientOcclusion = 1.00000000;
PixelMaterialInputs.Refraction = 0;
PixelMaterialInputs.PixelDepthOffset = 0.00000000;
```

Scene Texture	Index
Scene Color	0
Scene Depth	1
Diffuse Color	2
Specular Color	3
Subsurface Color	4
Base Color (for lighting)	5
Specular (for lighting)	6
Metallic	7
World Normal	8
Separate Translucency	9
Opacity	10
Roughness	11
Material AO	12
Custom Depth	13

Scene Texture	Index
Post Process Input 0	14
Post Process Input 1	15
Post Process Input 2	16
Post Process Input 3	17
Post Process Input 4	18
Post Process Input 5	19
Post Process Input 6	20
Decal Mask	21
Shading Model	22
Ambient Occlusion	23
Custom Stencil	24
Base Color (as stored in GBuffer)	25
Specular (as stored in GBuffer)	26

因为这个是后处理材质，你只需要关心 EmissionColor 的值，现在的值是 Local1。LocalX 变量是函数用来存储中间值的局部变量。你可以在输出的上方看到引擎是如何处理每个局部变量的。

```
MaterialFloat4 Local0 = SceneTextureLookup(GetDefaultSceneTextureUV(Parameters, 14), 14, false);
MaterialFloat3 Local1 = (Local0.rgba.rgb + Material.VectorExpressions[1].rgb);
```

最后一个局部变量（这里是 Local1）**大多数时候是一个做做样子（dummy）的计算**，所以你可以忽略掉它。这意味着 SceneTextureLookup()是一个针对 SceneTexture 节点的函数。它一共有三个参数。UV 表示采样的 UV 坐标，比如 UV 为(0.5, 0.5)就是采样到中心的像素 ;SceneTextureIndex 决定了采样哪种场景纹理。比如你要 PostProcessInput0，你就要写 14；Filtered 即代表场景纹理是否要使用双线性过滤，一般是 false。

```
float4 SceneTextureLookup(float2 UV, int SceneTextureIndex, bool bFiltered)
```

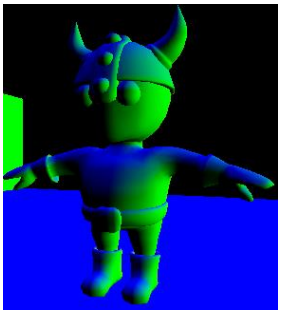
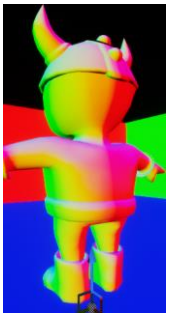
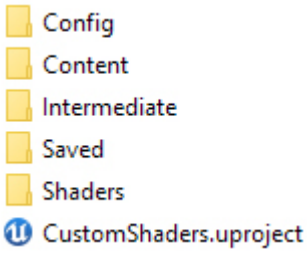
做个测试，你想要输出 WorldNormal（Index=8）。我们回到材质编辑器创建一个叫做 Gaussian Blur 的自定义节点，然后把下面的内容粘贴到 Code 处：

return SceneTextureLookup(GetDefaultSceneTextureUV(Parameters, 8), 8, false);

这可以输出当前像素的 World Normal。GetDefaultSceneTextureUV()可以获得当前像素的 UV。注意，4.19 版本前你可以把 TextureCoordinate 节点当成输入来获取 UVs，4.19 版本后正确的方式是使用 GetDefaultSceneTextureUV()并指明相应的 Index。把 Gaussian Blur 与 Emissive Color 相连，Apply。这个时候会有如下报错（4.20 没看到报错）：

```
[SM5] /Engine/Generated/Material.ush(1410,8-76):  error X3004: undeclared identifier 'SceneTextureLookup'
```

这是告诉你 SceneTextureLookup()在你的函数中并未被定义。为什么在使用 SceneTexture 节点中能用在自定义节点中就不能用呢？当你使用 SceneTexture 节点时，编译器会自动包含 SceneTextureLookup()的定义。你并没使用定义，所以无法使用这个函数。幸运的是要解决这个问题很简单，把 SceneTexture 节点设置成你想采样的纹理就可以了，我们把它设置成 WorldNormal。然后把它与 Gaussian Blur 相连，并把后者的输入线脚改成除 None 之外的其他值，这里我们设置为 SceneTexture。主界面中显示了每个像素的 WorldNormal（世界朝向）。

现在，在自定义节点中编辑代码不是太难，那是因为你只涉及到其中的一小片。一旦你的代码变得越来越长，要维护起来就比较困难了。为了改善工作流，虚幻允许你导入外部 shader 文件。你可以在自己的文本编辑器中写代码，并返回引擎中进行编译。

第三步我们来使用外部 shader 文件

我们在项目目录中创建一个 Shaders 文件夹，然后在其中创建一个名为 Gaussian.usf 的新文件。Shader 文件必须有.usf 或者.ush 后缀。用文本编辑器打开，并如下输入：

return SceneTextureLookup(GetDefaultSceneTextureUV(Parameters, 2), 2, false);

为了让引擎检测到新文件和 shader，你需要重启编辑器。重新打开后处理材质，并在原本自定义节点（命名为 Gaussian.usf）的代码处输入如下：

#include "/Project/Gaussian.usf"

return 1;

这样当你编译的时候，编译器会自动用外部文件代替第一行。注意，你无需用实际项目名字来代替 Project 字样。（但是这个自定义节点的效果不就跟用设置为 DiffuseColor 的 SceneColor 节点直接连到 Emission Color 上一样吗 !）

第四步来生成高斯模糊效果

了解更多关于高斯模糊的内容可以查看：

https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm

https://softwarebydefault.com/2013/06/08/calculating-gaussian-kernels/

就像边缘检测一样，这个特效也需要用到卷积。最终输出的是卷积核中所有像素的平均值。在经典的 box blur 中，每个像素都有着相同的权重，这会造成边缘的扩大。高斯模糊通过减少远离中心的像素权重可以避免这个问题。

因为所需采样数量的原因，使用材质节点的卷积方法其实不太理想。**比如说在一个 5x5 的卷积核中，你可能要采样 25 次**；把维度扩展到 10x10 则采样次数高达 100 次。这样的话，你的节点图表会看起来像一大碗意大利面条（spaghetti）。这就是为什么我们要用自定义节点——你只要用一个很小的 for 循环就能实现卷积核中每个像素的采样。第一步是建立一个控制采样半径的参数。

在材质编辑器中创建一个名为 Radius 的 ScalarParameter 节点，把默认值设置为 1：这个半径值决定了图像的模糊程度。在其后连接一个 Round 节点（确保卷积核的尺寸是 whole number,integer number 为 whole number 与负数的合集），并最终连在 Gaussian Blur 节点上、名为 Radius 的新输入线脚上。

现在开始写代码。你需要为每个像素进行垂直和水平共两次高斯计算。在使用自定义节点时，你无法以标准的方法实现函数，这是因为编译器会把你写的代码复制粘贴到一个函数体中，而你不能在函数内再定义函数。幸运的是，你可以利用这种复制粘贴的机制来创建全局函数。红线里是你写的代码，而编译器原封不动给你搬过去，甚至不会做更多的排版修饰。而如果你输入大括号中的内容，MyGlobalVariable 和 MyGlobalFunction()其实是不在函数体中的，这意味着他们是全局的，你可以在任意地方使用它们。一个简化的一维高斯函数如下所示。

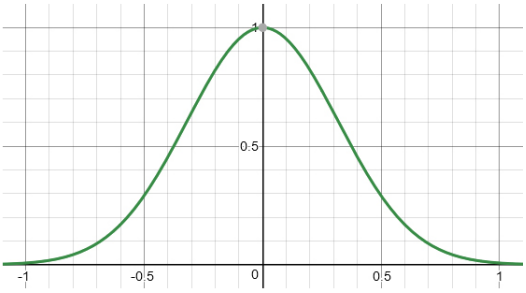
```
MaterialFloat3 CustomExpression0(FMaterialPixelParameters Parameters)
{
return 1;
}
```

```
MaterialFloat3 CustomExpression0(FMaterialPixelParameters Parameters)
{
    return 1;
}

float MyGlobalVariable;

int MyGlobalFunction(int x)
{
    return x;
}
```

$$f(x) = e^{-0.5(\pi x)^2}$$



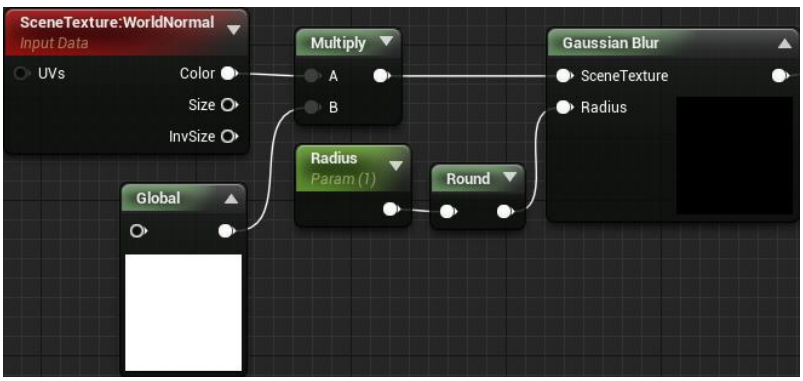
其结果是一个钟形曲线（bell curve），输入值为[-1,1]，输出值为[0,1]。

再创建一个命名为 Global 的自定义节点，粘贴如下代码，即一维高斯函数的代码形式：

```
    return 1;
}
float Calculate1DGaussian(float x)
{
    return exp(-0.5 * pow(3.141 * (x), 2));
```

为了让 Global 自定义节点起作用，你必须把它用在图表的某个地方。最简单的方法就是把它和图表的第一个节点简单相乘。这能确保全局函数在你于其他自定义节点中使用它们之前被定义。

把 Global 自定义节点的输出类型改为 CMOT Float 4。这么做的原因是你即将把它与数据类型为 float4 的 SceneTexture 节点相乘。



接下来我们为卷积核的每个像素采样创建 for 循环。
在 Gaussian.usf 文件中替换如下代码：

```
static const int SceneTextureId = 14; //你需要采样的场景纹理 ID，这里是 Post Process Input 0
float2 TexelSize = View.ViewSizeAndInvSize.zw;//纹素大小，用来把偏移量转换到 UV 空间中
float2 UV = GetDefaultSceneTextureUV(Parameters, SceneTextureId);//当前像素的 UV
float3 PixelSum = float3(0, 0, 0);//用于累积卷积核中每个像素的颜色值
float WeightSum = 0;//用于累计卷积核中每个像素的权重
```

随后你需要创建两个 for 循环，一个是垂直偏移，一个是水平偏移：

```
for (int x = -Radius; x <= Radius; x++)
{
    for (int y = -Radius; y <= Radius; y++)
    {
    float2 Offset = UV + float2(x, y) * TexelSize;
    float3 PixelColor = SceneTextureLookup(Offset, SceneTextureId, 0).rgb;
    float Weight = Calculate1DGaussian(x / Radius) * Calculate1DGaussian(y / Radius);
    PixelSum += PixelColor * Weight;
    WeightSum += Weight;
    }
}
```

上面的代码定义了一个以当前像素为中心的网格，其尺寸用 Radius 定义=2r+1，即如果 r=2，其尺寸就是 5x5。其次你需要计算像素的颜色和权重，即内循环中的内容：在第三行计算了像素权重，这里通过将两个一维高斯相乘来获得二维高斯，通过除以 Radius 来把 x 或 y 归一化到[-1,1]。最后你要计算其平均值，即在最末尾加上：

```
return PixelSum / WeightSum;
```

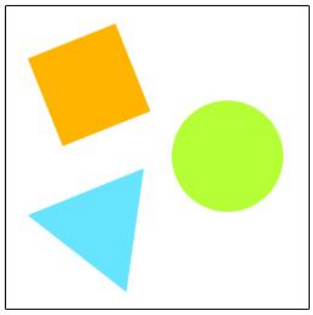
最后 Apply 就可以了！
尽管自定义节点如此强大，但是针对其局限性还是有必要做一些警告和说明：
1)渲染访问：自定义节点无法访问渲染管线中的大部分内容，比如光照信息和运动向量等。但使用**前向渲染**时会有轻微的差异；
2)引擎版本兼容问题：你写在一个引擎版本中的 HLSL 代码不能保证在另一个版本中也适用，比如前面提到，4.19 之前，你可以用 TextureCoordinate 来获取场景纹理 UVs，但 4.19 版本中你需要使用 GetDefaultSceneTextureUV()。

以下是 Epic 对优化的一些建议摘要：
使用自定义节点会阻止**常量叠算（constant folding）**，并且可能需要比完成等效版本的内置节点更多的指令。常量叠算是 UE4 在底层（under the hood）执行的优化策略，用来在必要的时候减少着色器指令数量。
比如一个表达式链 Time >Sin >Mul by parameter > Add to something 可以被 UE4 浓缩成一个简单的指令，即 final add。这样做是可行的，因为该表达式链的所有输入参数（Time/parameter）对整个 Draw Call 而言都是常量，并不会逐像素更改。但 UE4 在自定义节点中无法浓缩任何东西，这就可能产生比内置节点生成的等效版本性能更低的着色器。所以最好只在内置节点无法满足功能的情况下使用自定义节点。

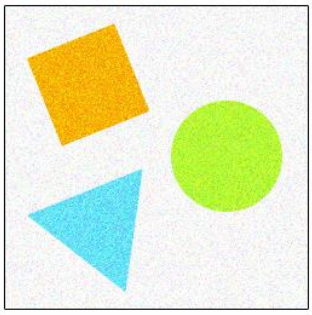
Part04 Paint Filter
非真实感渲染（Non-photorealistic rendering）包括了许多渲染技术，包括但不限于卡通渲染（Cel-Shading）、卡通描边（Toon Outline）和交叉影线（Cross Hatching）。你甚至可以让你的游戏看起来像一幅画，而 Kuwahara 滤波就是实现这一目的的技术之一。
1)从多个滤波核中计算平均值和方差
2)用最低方差输出卷积核的平均值
3)用 Sobel 来获取一个像素的局部朝向（Local Orientation）
4)基于像素的局部朝向来旋转采样的卷积核

低通道滤波（如 blur**（所以高斯滤波是一种低通道滤波？）查询卷积与滤波的差别**）是去除画面噪点的常用方法。

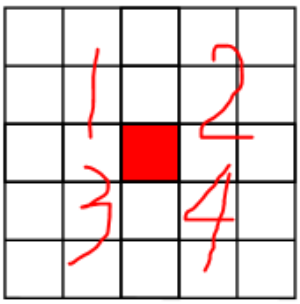
但经过模糊，噪点被去除，同时边缘也失去了锐利度。Kuwahara Filter 就是一个能够平滑画面同时保留边缘的技术。与卷积不同的是，Kuwahara 使用了四个核（Kernel），它们以当前像素为中心，在四个角上排列。下图为一个 5x5Kuwahara 滤波核。



Without Noise

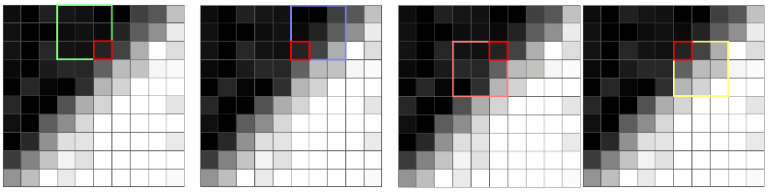


With Noise

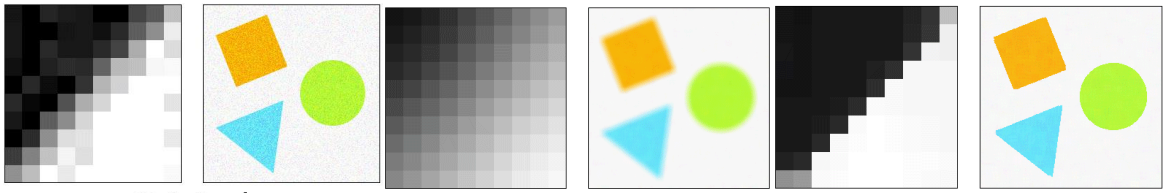


All

首先为每个滤波核计算平均值，可以实现基础的噪声平滑。同时对每个滤波核计算方差，这一般是滤波核内颜色变化的度量。比如有着相似颜色的滤波核方差会偏低，而颜色差别较大的滤波核方差则会很高（对平均值的偏离）。最终输出有着最低方差的滤波核平均值，这一机制就是 **Kuwahara 能够保留边缘的原因**。下面是个例子：
给定一张 10x10 带有噪点的灰度图，从右上到左下表示了一条边缘。选定一个像素，下图标志了相关滤波核。可以看到落在边缘上的滤波核颜色变化更大，即方差更大。通过不选择这些边缘滤波核的方式，可以避免边缘模糊。例子中的像素最终会选择绿框中的像素平均值来作为最终颜色，因为其同质化（homogeneous）最强。



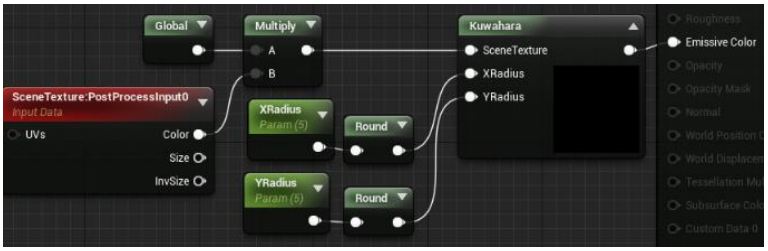
下图为 Kuwahara 滤波和盒型模糊结果：



Original　　Box Blur　　Kuwahara

恰巧的是（Incidentally），这种保留边缘（edge-perserving）的平滑特征能够让一幅图像更偏向绘画的效果。因为笔触（brush stoke）就是硬边低噪点的。Kuwahara 滤波是将真实感图像转为绘画风格的良好解决方案。

Unreal 引擎中的实现：



-2, -2	-1, -2	0, -2
-2, -1	-1, -1	0, -1
-2, 0	-1, 0	0, 0

滤波实现分为 Global.usf 和 Kuwahara.usf 两个 shader 文件。第一个文件会存储一些计算滤波核平均值与方差的函数，第二个文件则调用前一个文件的函数来进行每个滤波核的计算。

第一步是 Global.usf 的实现：

float4 GetKernelMeanAndVariance(float2 UV, float4 Range)//**函数的 Signature**

为了在网格中进行采样，需要两个 for 循环：一个是水平偏移，一个是垂直偏移。变量 Range 的前两个通道存储了水平循环的边界，后两个存储着垂直循环的边界。比如你要采样半径为 2 的滤波器左上方滤波核，其范围 Range = float4(-2, 0, -2, 0);对其偏移量计算如上图示。半径为 r，滤波核边长为 2r+1。

右上代码第一行获取了采样像素的偏移并将其转换到 UV 空间中。第二行获取偏移的 rgb 值。右下角为简化后的方差公式。下面为平均值与方差计算代码。

```
for (int x = Range.x; x <= Range.y; x++)
{
    for (int y = Range.z; y <= Range.w; y++)
    {

    }
}
```

```
float2 Offset = float2(x, y) * TexelSize;
float3 PixelColor = SceneTextureLookup(UV + Offset, 14, false).rgb;
```

$$Variance = \frac{\sum x^2}{Samples} - Mean^2$$

```
Mean += PixelColor;
Variance += PixelColor * PixelColor;
Samples++;
```

```
Mean /= Samples;
Variance = Variance / Samples - Mean * Mean;
float TotalVariance = Variance.r + Variance.g + Variance.b;
return float4(Mean.r, Mean.g, Mean.b, TotalVariance);
```

第二步是 Kuwahara.usf 的实现：

首先创建变量如下：

```
float2 UV = GetDefaultSceneTextureUV(Parameters, 14);
float4 MeanAndVariance[4];
float4 Range;
```

随后需要为每个滤波核调用 GetKernelMeanAndVariance()：左上、右上、左下、右下，比较输出最终颜色

```
Range = float4(-XRadius, 0, -YRadius, 0);
MeanAndVariance[0] = GetKernelMeanAndVariance(UV, Range);

Range = float4(0, XRadius, -YRadius, 0);
MeanAndVariance[1] = GetKernelMeanAndVariance(UV, Range);

Range = float4(-XRadius, 0, 0, YRadius);
MeanAndVariance[2] = GetKernelMeanAndVariance(UV, Range);

Range = float4(0, XRadius, 0, YRadius);
MeanAndVariance[3] = GetKernelMeanAndVariance(UV, Range);
```

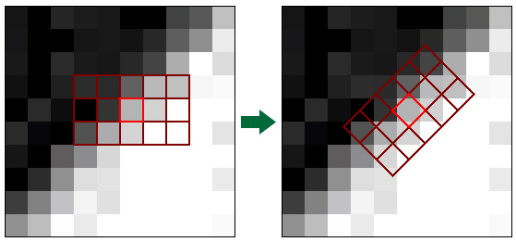
```
// 1
float3 FinalColor = MeanAndVariance[0].rgb;
float MinimumVariance = MeanAndVariance[0].a;

// 2
for (int i = 1; i < 4; i++)
{
    if (MeanAndVariance[i].a < MinimumVariance)
    {
        FinalColor = MeanAndVariance[i].rgb;
        MinimumVariance = MeanAndVariance[i].a;
    }
}

return FinalColor;
```

//排错排半天发现是用了中文逗号……

如果仔细看会发现图片里有些块状的区域，这是使用轴对称（axis-aligned）滤波核的副作用。Directional Kuwahara 滤波可以有效解决这个问题，这种情况下，滤波核是与像素朝向对齐的。因为你可以把卷积核当作矩阵，所以在描述尺寸维度的时候，往往用 HeightxWidth 而不是平时（conventional）用的 WidthxHeigh。为了计算局部朝向，我们需要 Sobel 算子：Sobel 使用 Gx 和 Gy 两个滤波核，前者计算水平方向的梯度（gradient），Gy 计算垂直方向的梯度。下面是个 3x3 灰度图的例子：

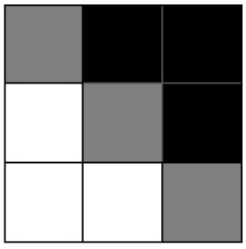


Gx

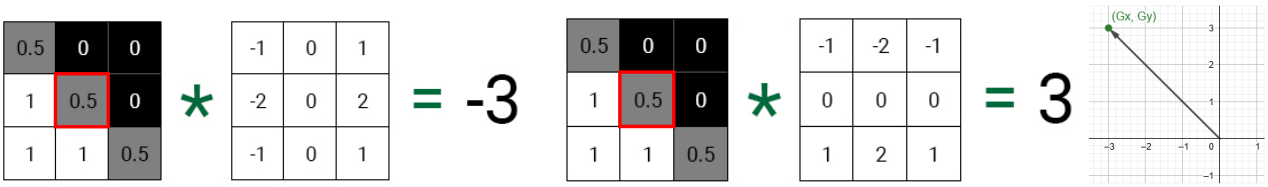
-1	0	1
-2	0	2
-1	0	1

Gy

-1	-2	-1
0	0	0
1	2	1



首先两个各做一次卷积，把结果在坐标轴中画出来，你会发现向量方向与边缘一致，就可以获得旋转角度。



第三步来获取局部方向：

在 Global.usf 中添加函数 float GetPixelAngle(float2 UV){}

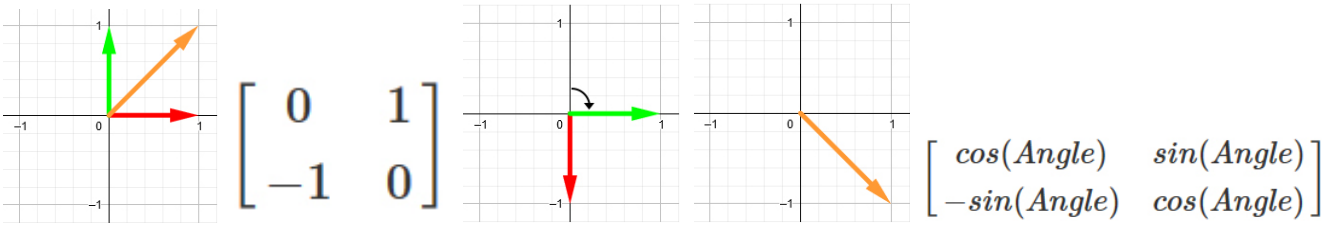
```
float GradientX = 0;
float GradientY = 0;
float SobelX[9] = {-1, -2, -1, 0, 0, 0, 1, 2, 1};
float SobelY[9] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
int i = 0;
```

然后用 SobelX 和 SobelY 来执行卷积。第三行**将图像转换为灰度值**，将图像作为整体来计算梯度，而不是拆分为单个颜色通道。

```
for (int x = -1; x <= 1; x++)
{
    for (int y = -1; y <= 1; y++)
    {
        // 1
        float2 Offset = float2(x, y) * TexelSize;
        float3 PixelColor = SceneTextureLookup(UV + Offset, 14, false).rgb;
        float PixelValue = dot(PixelColor, float3(0.3,0.59,0.11));

        // 2
        GradientX += PixelValue * SobelX[i];
        GradientY += PixelValue * SobelY[i];
        i++;
    }
}
return atan(GradientY / GradientX);
```

第四步用矩阵来旋转滤波核：



首先在 GetKernelMeanAndVariance()开放 2x2 矩阵接口，并修改内循环的第一行。mul()会执行矩阵乘法操作，会让偏移量愿意当前像素为中心进行旋转。

```
float4 GetKernelMeanAndVariance(float2 UV, float4 Range, float2x2 RotationMatrix)
```

```
float2 Offset = mul(float2(x, y) * TexelSize, RotationMatrix);
```

在 Kuwahara.usf 变量列表中添加变量如下：

```
float Angle = GetPixelAngle(UV);
float2x2 RotationMatrix = float2x2(cos(Angle), -sin(Angle), sin(Angle), cos(Angle));
```

第一行是计算当前像素的朝向角，第二行创建旋转矩阵。将每个 GetKernelMeanAndVariance()调用修改成：

```
GetKernelMeanAndVariance(UV, Range, RotationMatrix)
```