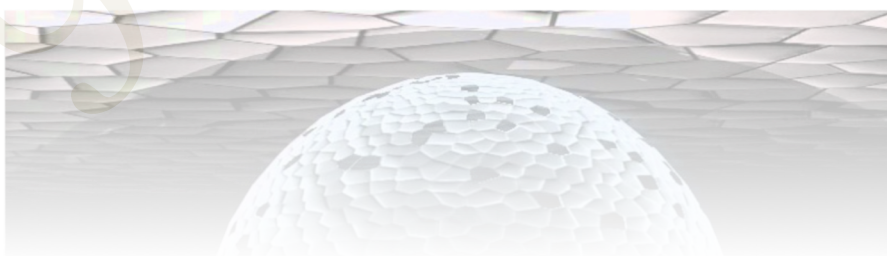
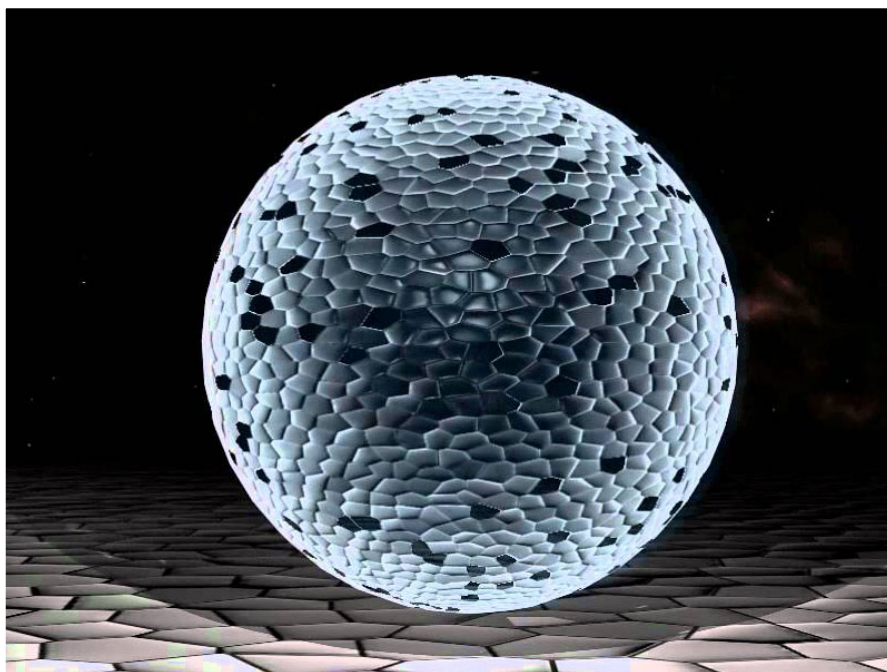


Unity3d Shader 编程

98jy.net

化石 编著



目录

写在前面	5
SHADER 理论	6
SHADER 在引擎中的作用	7
3D 管线	10
SHADERLAB	15
亲手制作第一个 UNITY SHADER	16
SHADERLAB	21
SHADERLAB 的 PROPERTIES 实验	47
理解 SHADER 和 MATERIAL 的关系	53
光照模型支持	59
自定义光照模型（方程）	60
HALF LAMBERT 模型	64
RAMP TEXTURE	66
常见实例	75
材质贴图和凹凸贴图	76
高光效果	83
环境映射和 CUBEMAP	91

透明、ALPHA 混合(ALPHA BLEND)	99
ALPHA 测试 (ALPHA TEST)	107
顶点方法 (VERTEX FUNCTION) 和顶点色	111
多 PASS (MULTI-PASS) 渲染	119
轮廓(SILHOUETT SHADER)	121
更漂亮的轮廓核心之模糊理论	125
模糊的实现	131
基于模糊的轮廓 (高亮)	135
X-RAY (X 光, X 射线) 基于 RAMP TEXTURE 的版本	139
X-RAY (X 光, X 射线)增强版	144
遮挡 SHADER (在被遮挡后显示不同效果)	148
透明和玻璃	154
后期	157
后期(POST PROCESS)概述	158
后期处理之灰度画面以及通用的后期框架	159
重影 (DOUBLE VISION)	167
老照片	170
后记	173

98jy.net

写在前面

本书是为 98jy.net 增加初创内容时所做。本书的作用，主要是系统的介绍 Unity Shader 的 ShaderLab 和使用 Unity 制作 Shader 的过程。由于时间关系，很多 shader 的代码细节对初学者来说有阅读困难之处，最好是有了 cg、HLSL 或者 GLSL 的基础后用来对照着学习 Unity 的 ShaderLab。如果是初学者，前几章读起来是没有问题的，到后面讲述例子（自定义光照方程）的时候，必须要配合相关的书籍阅读。如果对 Shader 感兴趣的，可以到 <http://98jy.net> 上关注相关信息。

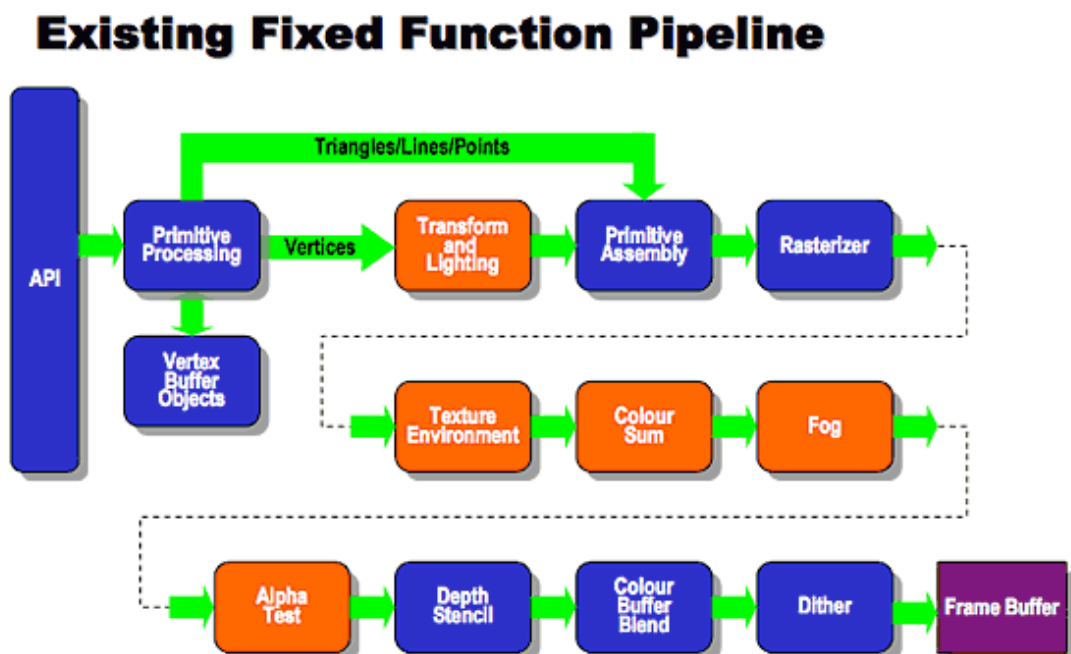
Shader 理论

学习 Shader 编程，对 3D 流水线的熟悉是首要的，因此这里会讲述 3D 流水线。然后我们会通过亲自动手来“感知” Unity Shader 的编写过程。从而引出我们对 Unity 的 ShaderLab 的介绍。要说的是，ShaderLab 的理论部分，因涉及细节很多，初始只需要浏览到“知道有这么个内容”就好。

Shader 在引擎中的作用

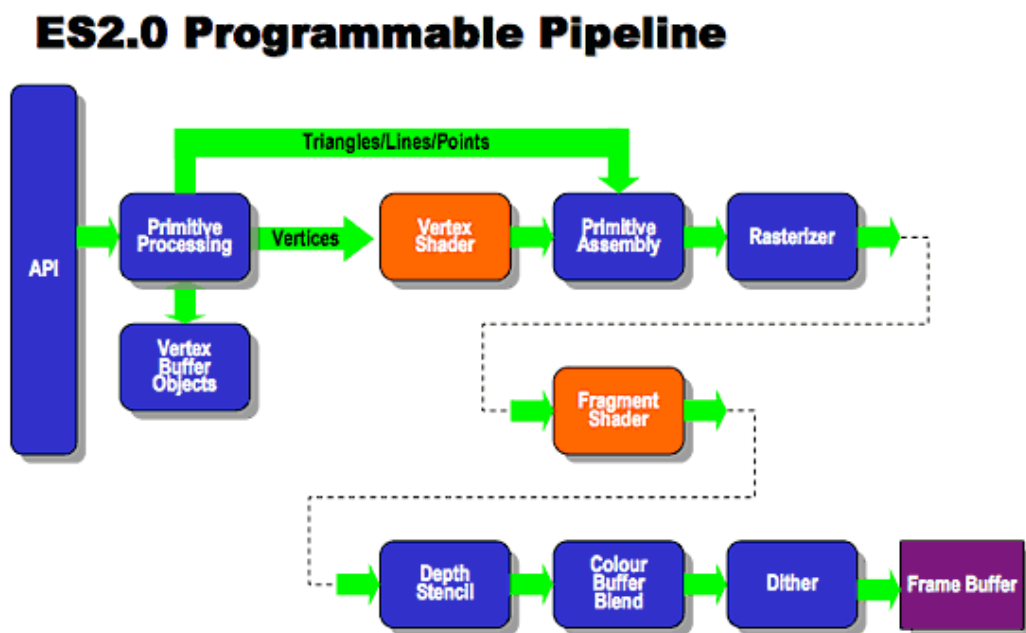
shader，中文一般译作“着色器”，白话一点就是“给物体上色”，所以 shader 的作用也就是计算并决定最后物体的颜色是什么。

每一个 3D 引擎，在实现的时候，会通过 API 层让用户准备好物体的数据，然后交由 GPU 进行处理。处理的目标或者说是核心任务就是根据各种设置来得到物体最后表现出来的色彩。GPU 处理的时候，会遵循所谓的流水线（也叫“管线”，下面有内容介绍）。下图是早期的 GPU 遵循的管线（摘自 OpenGL 的标准，为保持原汁原味，没有对其进行翻译），专业名词把它叫做“固定管线”：



在这个管线中，橙色的部分是用户可以通过 GPU 进行配置的。在着色这个层次上，它表现在仅能通过配置一些地方，“不要采用光照”，“请启用雾效”这些很高层次的控制，而并不能控制到每个物体的细节，譬如，这个物体跟这种光究竟是采用什么样的算法。也因此，使用这种固定管线的程序，做出来的东西，基本上都是看起来塑料感比较重（可以通过贴图等方式进行一些弥补）。究其原因，主要是物体的细节部分，比如金属物体和塑料物体，在每个细节的部分对光的交互，应该有不同的算法。在固定管线中，这种变化没有办法进行程序上的控制。

可喜的是，现在的 GPU，都进入了 2.0 以后的模式，管线如下图：



这种模式下，GPU 的 Vertex Shader(其中一橙色块) 和 Fragment Shader（另一个橙色块），都是一段通过用户层可以设置的供 GPU 运行的代码。Vertex Shader 接收从 GPU 送进来的

顶点信息，并对每个顶点运行用户的顶点程序，即顶点着色程序；Fragment shader 接收 GPU 送进来的像素信息，并对每个像素运行用户的像素程序，即像素着色程序（精确的说，fragment 并不等同于 pixel，但这里简化这个说法）。通过这样的方式，用户就可以在这两段程序里放置自己的算法，对于金属，使用一种算法；对于塑料，又是另外一种算法。因为这些地方都是可以进行编程的，所以这个管线也就被叫做“可编程管线”。

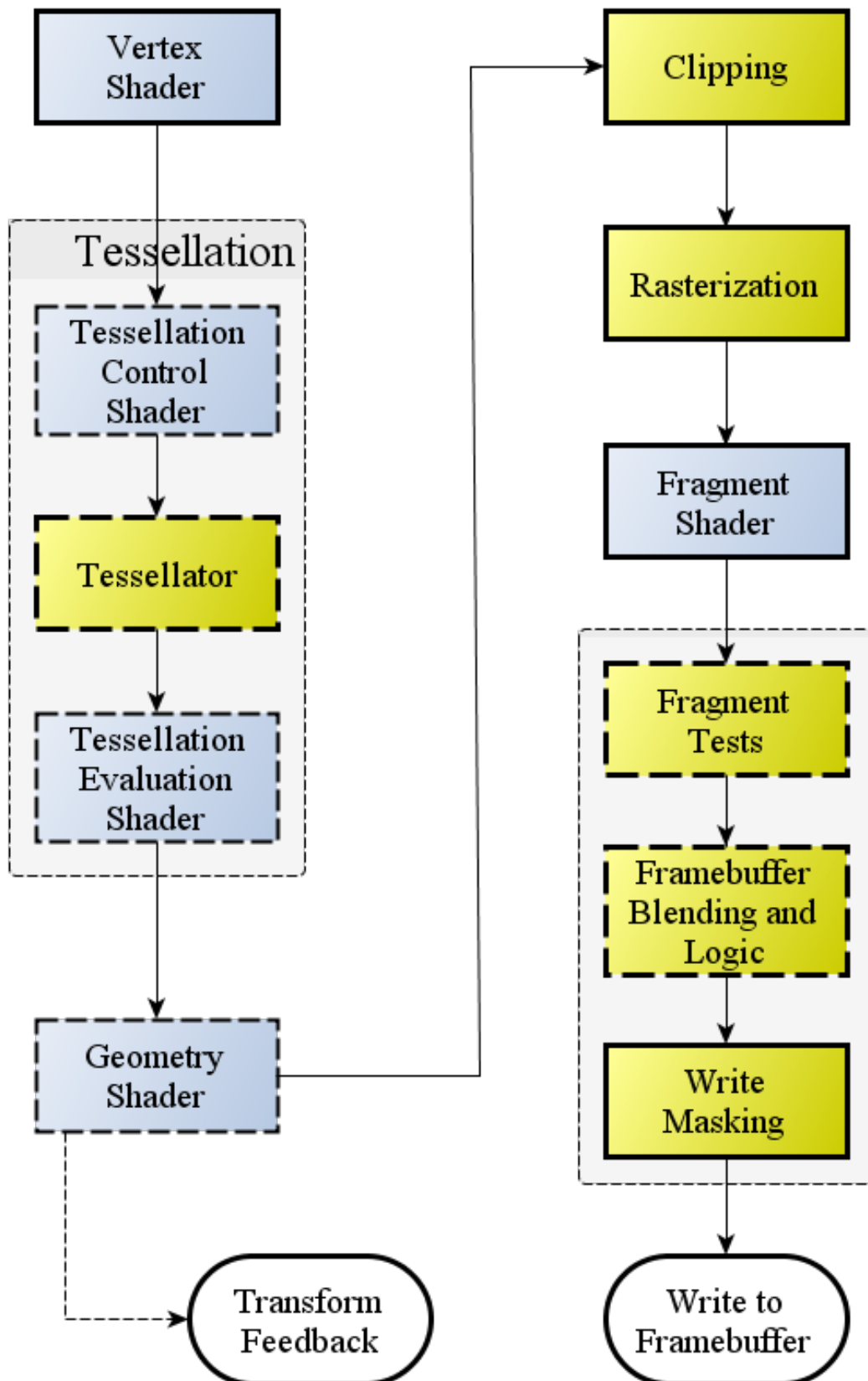
最新的可编程部分，已经不仅是 Vertex Shader 和 Fragment Shader 了。但是我们这里只是拿它做个例子，因此不详细介绍。

对于顶点着色器来说，主要就是对顶点处理后交由到管线的下一步。而像素着色器的主要作用，就是计算出像素的色彩然后交到管线的下一步。由此，在流水线来讲，shader 主要是对像素进行着色（即使是 Vertex Shader 中大多数时候并不计算颜色，但它是为了最后的颜色生成做前期的计算），也就是它们的最重要的作用。在这个角度来讲，“着色器”这个翻译非常的精确。

3D 管线

如果是学习当代的实时渲染，很多时候我们都要不停的翻阅所谓的管线图。管线图能让人了解 3D 物体从自身的数据送入开始到最后呈现在屏幕上的所有历程。在理解和解决渲染发生的问题的时候，往往能起非常大的作用。

管线基本分为固定管线和可编程管线，现在的 GPU，甚至是移动设备上，基本上都配备可编程管线的 GPU。因此，我们这里拿可编程管线讲述，直接从 OpenGL 的网站拿来图：



顺着图中的箭头方向，就是数据一步一步被处理，最后写到显示缓冲区（framebuffer）的过程，因为每个单位都是一个一个的步骤走，正如工厂中的流水线，所以就有了这个“渲染流水线（rendering pipeline）”名字。

从用户准备物体数据到最后显示，会经历大致这些流水（蓝色部分对应到图中相关组件）：



1. 用户给 GPU 喂顶点数据给流水线，顶点数据是一组属性，诸如位置、法线、颜色之类的组合信息。然后通过 API(所谓的 drawcall)让 GPU 渲染。
2. 上图中的 **Vertex Shader** 接手处理每个顶点，对 Vertex Shader 来说，得到的输入就是第一步给到的信息流中的每个顶点属性；至于输出什么，在可编程管线中，完全是看用户写的代码要做什么。当然，这个步骤输出的信息有一定约束，至少有一个“位置(position)”信息必须被输出，作为一个有效的顶点信息。
3. 元件组装（Primitive Assembly），对于 OpenGL 来说，元件有 3 种，点、线和三角形。从第二步出来的顶点数据，再结合用户喂给流水线的其他数据（告知这些顶点之间的关系），会被这个过程组装成这三个类型中的一种。镶嵌/或者说是细分（**Tessellation**）会在此后运行，让前面的 primitive 能被细分成更小的元件以满足不同的精度需求。
4. 几何体着色器（**Geometry Shader**），该着色器得到第三步出来的 primitive，它的输出是 0 个或者更多的 primitives。
5. **Transform Feedback**，在 OpenGL 这里，几何体着色器后可以吐出相关的 primitive 到所谓的缓冲物体(buffer object)中（其实就是定义好格式的一些数据结构），这些

buffer object 可以被保存用作其他的作用。如果在这里抛弃流水线后续的工作，那么整个流水线的产出就是这些东西。

6. 顺图中的实线,后续是裁剪([Clipping](#))。根据用户设置的视锥体的可视范围,primitives 会被剔除(整体在可视范围外),部分在外的则被按照规则切割成几个 primitives。在现在的 GPU 上,用户除了视锥体,也可以通过裁剪面(Clip plane)来指定额外的裁剪信息。在 3D 中,大多数是单面的 primitive,这个时候,背面朝向相机的 primitive 也会被裁剪掉。
7. 能达到光栅化([Rasterization](#))这步的物体,会被这个步骤进行光栅化,举例来说,一个三角形 primitive,会产生填充其内部的许多所谓的碎片(fragment)。fragment 跟 pixel(像素)不同的地方在于,在专业角度,它是一系列信息,用来判断对应的像素最后的数据。
8. 碎片着色器([Fragment Shader](#))得到上面处理出来的数据,然后运行用户给定的一段程序,计算出最后的像素的相关信息。它的输出是一系列的值:颜色和深度信息。
9. fragment shader 的输出进入 [Fragment Tests](#) 这个阶段,会首先进行 Alpha 测试(alpha test),继而模板测试(stencil test),如果 fragment 没有经过测试,那么就不会进入到最后的缓冲区(framebuffer);然后进行深度测试(depth test),如果通不过,就不会进入 framebuffer。【注:OpenGL 的相关文档有提到,如果 fragment shader 对 fragment 的深度不会修改,那么这个测试可以提到 fragment shader 之前,所谓的“Early Z Test”】
10. [Blending](#) 是后续的工作,如果 fragment 可以被写入 framebuffer,则这个 fragment 和已经在 framebuffer 中的值可以做一定的混合,实现不同的效果,举例,半透明效果。

11. 上一步出来的值,就准备开始往 framebuffer 中写了,写的时候可以通过 Writing Mask 来控制哪些值要写到 framebuffer, 可以控制到颜色、深度、模板值,也可以控制到颜色的某个通道。
12. 结束。

虽然这里的图是 OpenGL 的图,但是 GPU 都是一样的,所以在理解流水线上,DirectX 和 OpenGL 没有什么太大的不同。因此,我们不拿基于 DirectX 的图再重复做介绍。

ShaderLab

ShaderLab 是 Unity 自己定义的 Shader 格式，这个格式包装了可以供 OpenGL 或者 Direct3D 的 shader 编译器编译的 shader 代码，并且在这个基础上加入了很多的辅助信息，以帮助 Unity 编辑器向用户提供相应的界面来方便的操作和使用 Shader。

亲手制作第一个 Unity Shader

了解了 Shader 的功能和一些管线背景后，我们先来使用 Unity 创建和编译一个我们亲手制作的 shader：漫反射着色器。这个过程能让我们熟悉创建 shader 和应用 shader 的流程。

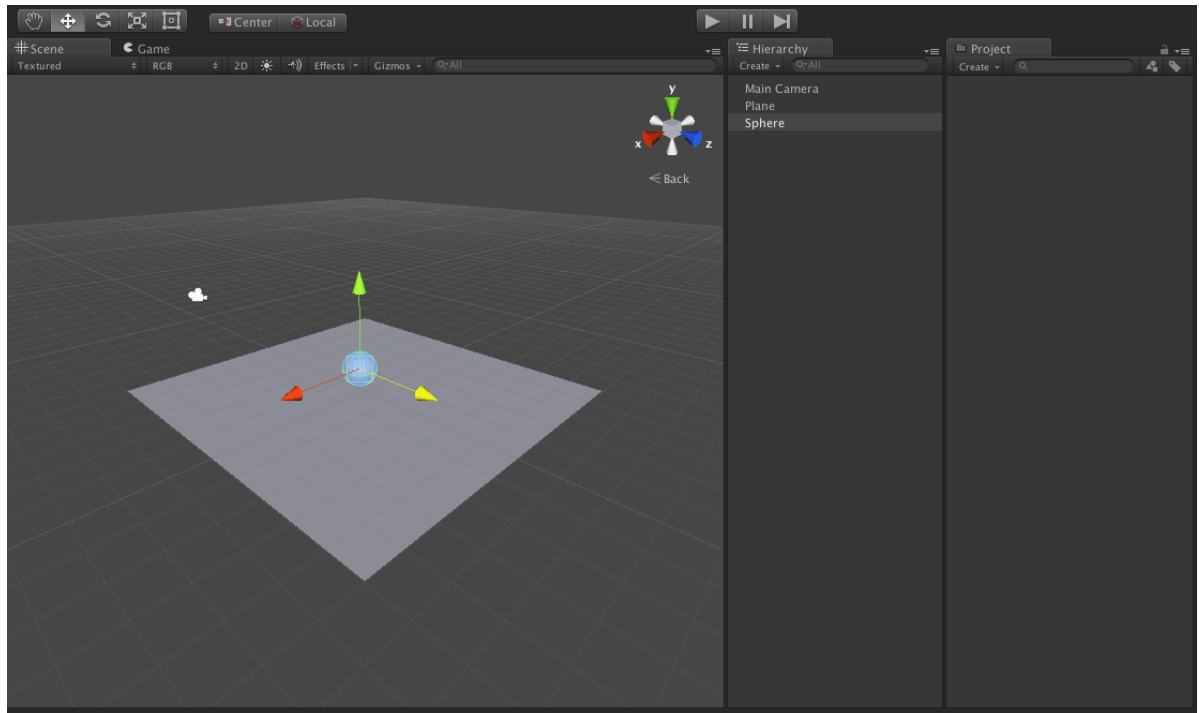
准备工作：

1. 安装 Unity 版本 4.6 以上（安装和下载可以参考 <http://98jy.net/course/12/info>）
2. 创建一个新的工程
3. 菜单 **GameObject | 3D Object | Plane** 创建出一个平面，作为我们的地面
4. 菜单 **GameObject | 3D Object | Sphere** 创建出球来，反复多次创建多个

注意：

- (1) 创建地面后选中该物体（Hierarchy 面板中），把相应的 transform 的 position 部分全部清成 0。
- (2) 所有的 sphere 在创建后都跟上面的地面物体一样，position 部分清零。
然后把球依次拖开一定距离，并放置到地面上面，可以看到整个球的部分为止。

上面的工作做完，我们得到这样的画面：



正式工作:

- 在编辑器的 Project 面板上，右键点击该面板手柄下面的 Create，弹出的菜单中选择 Shader
- 把创建出的 shader 文件的名字改成 myDiffuse
- 双击 myDiffuse 这个文件，让 MonoDevelop（任何你想用到的其他的编辑器都可）运行并且打开它，供我们编辑
- 打开的“myDiffuse”文件中，全部选中所有的代码，按 delete 清空
- 然后拷贝我们下面的代码段到该文件内容中

```
Shader "myDiffuse"
{
    Properties
```

```
{
    _MainTex ("Base (RGB)", 2D) = "white" {}
}

SubShader
{
    Tags { "RenderType"="Opaque" }
    LOD 200

    CGPROGRAM
    #pragma surface surf Lambert

    sampler2D _MainTex;

    struct Input
    {
        float2 uv_MainTex;
    };

    void surf (Input IN, inout SurfaceOutput o)
    {
        half4 c = tex2D (_MainTex, IN.uv_MainTex);
        o.Albedo = c.rgb;
        o.Alpha = c.a;
    }
    ENDCG
}
FallBack "Diffuse"
}
```

- 然后保存我们的这个文件。回到 Unity 编辑器中
- 通过 Project 面板上的 Create，创建出一个 Material 出来，重命名为 myMaterial
- 选中 myMaterial。对应的 Inspector 面板上，shader 栏位点开菜单，下拉条中选中“myDiffuse”
- 把 myMaterial 拖到（左键按住不松，移动鼠标直到鼠标达到位置才松开）Hierarchy 中的某个物体上，可以是 sphere，也可以是 plane 上面

- 选中 Hierarchy 中的我们刚才拖入 myMaterial 的物体，在对应的 Inspector 面板上确保看到它的'Mesh Renderer'部分的'Use Light Probes'下面部分是'myMaterial'

这个过程就是一个创建新的 shader，并且通过 material 把我们的 shader 应用到物体上的步骤。原理上讲，物体靠材质表现出不同的表面属性，而 shader 是材质中的一段供 GPU 运行的自动化代码，实现这种属性的特质。因此，以后每个新的 shader，都可以通过类似的步骤来使用到场景中的物体上。

上面的 shader 应用到物体上以后，我们可以发现物体的表征跟原来是一模一样的。这个原因是因为我们写的这个第一个 shader 的功能，跟 Unity 默认的 shader 是一模一样的。也就是说，虽然我们在创建物体的时候，我们并没有给它相应指定相关的 shader，但是 Unity 会保证它带上一个默认的。在 3d 应用中，每个物体都必须有一个相应的 shader 来完成最后的渲染（绘制物体的术语）。你没有去刻意选择的时候，3d 引擎本身会帮助你选择一个默认的，来保证整个程序的正常运作。

但是，同时我们也注意到了，shader 只有通过 material（材质）作为媒介才能应用到物体上，而不能直接绑定到物体。这也就是为什么我们在创建 shader 以后必须要创建一个 material，然后从 material 中选择我们的 shader，再把 material 绑到物体上的原因。所以，对于物体来说，它具备材质属性，而 shader 只是材质中的一个组件或者部分。这个大家需要先弄清楚。

本节内容主要是让读者熟悉整个 shader 制作和应用的流程，在英文中有“Get your hands dirty”，直译为“让你的手脏（一下）”，用在编程学习方面非常适合。编程无他，唯手熟尔。对本节的内容，虽然步骤比较简单，但是学习的人都要亲自操作几遍，直到能不看书也创建一个 shader 出来为止。对于 shader 的内容本身的不理解，我们会在下面慢慢展开，欢迎进入更精彩的 shader 世界。

ShaderLab

前面我们看到了一个 Unity 的 shader，这种写法 Unity 把它称作“surface shader”。对于 Unity shader，不是每种写法都叫“surface shader”。它只是 Unity shader 支持的众多写法的其中一种，其他语法还包含 Fixed function Shader (5.0 以后开始着手取消) 和 Vertex Fragment Shader。

对于 shader 文件，写的语法 Unity 把它叫做 ShaderLab。ShaderLab 是 Unity 官方定义的对材质 (material) 描写的语法，我们的 shader 代码，不管采用哪种方式，都是这个语法描述出的某个文件中的其中一个部分。

ShaderLab 的结构如下：

```
Shader "MyShader"
{
    Properties
    {
        _MyTexture ("My Texture", 2D) = "white" {}
    }

    SubShader
    {

    }

    SubShader
    {

    }
}
```

为说明，我们把这个格式简短的描述为

```
Shader "name" { [Properties] Subshaders [Fallback] [CustomEditor] }
```

这个格式定义了一个 shader。这个 shader 会在材质的面板上的下拉菜单中（选中材质球后对应的 Inspector 窗口）用“name”来列出来，如果 name 中含有路径符号“/”，那么这个符号在材质面板上会被解释成一个子菜单。这个 Shader 中可以定义一系列的 Properties，相应的，这些属性也会在材质面板上呈现。在这后面，会跟上多个 Subshaders，然后是可选的 Fallback 和/或 CustomEditor。

下面我们对 Properties 块，Subshaders 块，Fallback 块和 CustomEditor 块的内容进行详细介绍。这块内容，除了格式，我对读者的建议是你可以先跳着看。先看完每个块的格式和例子（如果有的话），就可以先不用去读细节的部分。细节可以在后面章节讲具体的每个 shader 的时候，发现有原来没有看到过的东西或者已经遗忘的东西的时候，再翻回来看。

Properties

一个 shader 可以包含一个 Properties 块，这个块中可以放置多个属性。典型的属性如颜色，贴图或者是任意的值。

格式：Properties { Property [Property ...] }

定义了一个 Properties 块。里面可以跟多个属性。这些属性可以是以下内容：

数值和范围

```
name ("display name", Range (min, max)) = number
```

```
name ("display name", Float) = number
```

```
name ("display name", Int) = number
```

这些定义一些数值属性，每个等号后面表示的默认的取值，举例来说，Range 和 Float 值可以是“11.32”，Int 可以是 10。*Float* 是浮点数，*Int* 是整数，Range 则表示从 *min* 到 *max* 的范围。

name 是开发者给这个值起的可以在代码中访问的名字，*display name* 则是在材质的面板上的显示名字。

颜色和向量

```
name ("display name", Color) = (number,number,number,number)
```

```
name ("display name", Vector) = (number,number,number,number)
```

定义颜色值（RGBA）和向量值（xyzw），在逻辑上，这两个概念不会到一起，但是在 shader 的数学里，都是一样，因此，放在一块讲解。这里定义一个颜色属性或者一个 4D 的向量属性，等号后面跟的是默认的值，举例，“(1, 0.5, 0.7, 0.3)”。Color 关键字会让属性面板上出现一个可供用户使用的调色板按钮。Vector 关键字在面板上是四个可以填入数值的栏位。

贴图

```
name ("display name", 2D) = "defaulttexture" {}
```

```
name ("display name", Cube) = "defaulttexture" {}
```

```
name ("display name", 3D) = "defaulttexture" {}
```

定义 2D 的贴图，Cubemap 和 3D 贴图（3D Volume），等号后面都是默认的图，这些默认的图是空字符串""，或者 Unity 定义的"white"、"black"、"gray"、"bump"。在面板上，分别会出现让用户拖入 2D 图，Cube 图，3D 图的框出来。

另外要说的是，在每个属性前面，我们可以写上用方括弧括起来的属性。这些属性可以是 Unity 内定的，也可以是用户自定义的。作用主要是来控制对应的我们刚才讲的 Property 在材质面板上的显示。Unity 内定的属性列表：

[HideInInspector] - 不要在面板上显示对应的属性值

[NoScaleOffset] - 默认的贴图面板都带有 tiling 和 offset 两个栏位，这个属性让 Unity 不要显示这些栏位

[Normal] - 指示该贴图期望放进来的图是法线图

[HDR] - 指示该贴图期望放进来的是 HDR 的图

Properties 块的例子，来点感性认知。

Properties

```
{  
    _WaveScale ("Wave scale", Range (0.02,0.15)) = 0.07 // 滑杆条  
    _ReflDistort ("Reflection distort", Range (0,1.5)) = 0.5  
    _RefrDistort ("Refraction distort", Range (0,1.5)) = 0.4
```

```
_RefrColor ("Refraction color", Color) = (.34, .85, .92, 1) // 调色板
_ReflectionTex ("Environment Reflection", 2D) = "" {} // 贴图选择框
_RefractTex ("Environment Refraction", 2D) = "" {}
_Fresnel ("Fresnel (A) ", 2D) = "" {}
_BumpMap ("Bumpmap (RGB) ", 2D) = "" {}
}
```

SubShaders

一个 shader 可以包含多个 sub shader 块，而且至少需要一个。Unity 在装载 shader 的时候，会从这些 shader 块中挑选出可以在用户的机器上运行的第一个块来作为实际的 shader。如果所有的块都不能符合要求，那么 Unity 就会尝试使用 Fallback 块的 shader。

因为 GPU 能力的差异，给游戏开发的人，这里特指 shader 编程人员，带来了一些麻烦的地方。开发者希望自己的游戏能运行在最新的硬件配置上，但是又不能抛弃其他的大量用户。这也就是 Subshaders 存在的原因。开发者可以为最好的硬件写一个特别的 SubShader，然后加上其他为老一点的硬件写的 SubShader。这样 Unity 制作出来的游戏，就能支持到不同的机型。

格式：Subshader { [Tags] [CommonState] Passdef [Passdef ...] }

定义了含有可选的 Tags，可选的 CommonState 和至少一个 Pass 的 Subshader 【注：Surface Shader 看不到 Pass，那是因为 Pass 是通过 Surface 代码自动生成】

一个 subshader 可以包含多个 pass，至少需要一个。可以选择（也可以没有）设置一些作用到所有 pass 的状态（CommonState）。额外的，subshader 可以指定一些 Tags。

当 Unity 选择相应的 subshader 后，每个 pass 都会导致它对物体进行一次渲染运算（加上光照后，可能更多）。因为渲染本身是个比较消耗的事情，所以一个 subshader 中，能尽量少使用 pass 则最好。当然，在一些老旧的机器上，如果一些效果没有办法在一个 pass 内完成，我们就必须选择多个 pass。

Tags

格式: `Tags { "TagName1" = "Value1" "TagName2" = "Value2" }`

指定 *TagName1* 的值是 *Value1*，*TagName2* 的值是 *Value2*

Tags 中的内容是“关键词-值”的对。*Tags* 用来决定诸如渲染顺序和其他的一些参数。

下面的 Tag 必须在 SubShader 中体现，而不是在 Pass 中。

除了内定的 tags，用户也可以自定义，自定义的 tags 可以通过 `Material.GetTag`（这是一个 Unity 的 API 方法）来取得。

- 渲染顺序 - Queue tag

你可以通过 Queue 这个 tag 来控制物体的渲染顺序。Shader 会决定它的物体属于哪个渲染队列（Queue），通过这种方式，透明的物体总是能在不透明物体的后面渲染。

Unity 内部有 5 中预定义的队列，在这个基础上，可以产生更多的队列

- 1、Background - 这个列表队列物体在其他任何物体前渲染
- 2、Geometry（默认） - 大多数物体使用这个队列
- 3、AlphaTest - 要使用 AlphaTest 的物体使用该队列。
- 4、Transparent - 这个队列中的物体在 Geometry 和 AlphaTest 后渲染。
- 5、Overlay - 这个队列中的物体最后渲染。

这里是一个例子，使用该 shader 的物体被放置在 Transparent 队列中，

```
Shader "Transparent Queue Example"
{
    SubShader
    {
        Tags { "Queue" = "Transparent" }

        Pass
        {

        }
    }
}
```

在 Unity 内部，上面的 5 个预定义分别对应数值 1000，2000，2450，3000，4000。因此，我们对有特殊需求的物体，可以这样定义

```
Tags { "Queue" = "Geometry+1" }
```

这个例子可以让物体在所有 Geometry 队列中的物体后渲染，但是早于 AlphaTest。

- 渲染类型 - RenderType tag



RenderType 对 shader 进行归类，比如，该 shader 是属于不透明 shader，或者 shader 是属于半透明 shader。它主要用在 Unity 的 Shader 替换（可以批量的动态替换大量物体中的 shader）中。

Unity 内置以下几种值对应 RenderType：

- 1、Opaque
- 2、Transparent
- 3、TransparentCutout
- 4、Background
- 5、Overlay
- 6、TreeOpaque
- 7、TreeTransparentCutout
- 8、TreeBillboard
- 9、Grass
- 10、GrassBillboard

- DisableBatching tag

有些场合下，如果我们任由 Unity 帮助我们做 batching（优化渲染的一种技术），那么物体会被不正常的渲染。这种状况下，我们要有方式告诉 unity，不要对我们的物体做 batching。这个 tag 就是 *DisableBatching*。

它的值有 3 个：

- 1、*True* - 不要为我 batching
- 2、*False* - 默认的，可以的话，请 batching
- 3、*LODFading* - *LOD fading* 启用的时候不要 batching

- ForceNoShadowCasting tag

如果这个的值是“True”，那么对应的物体就不会产生影子。

- IgnoreProjector tag

如果它的值是“True”，那么对应的物体不会被 Projector 影响到

- CanUseSpriteAtlas tag

如果 shader 被用来渲染精灵（Unity 的 2D 中的），那么就把它设置成“False”。当对应的图被打包进入图集（atlas）的时候，这个 shader 就不起作用。

- PreviewType tag

这个能让 Unity 材质面板上的默认球形被替换成其他的物体。可以是“Plane”，和“Skybox”。

Pass 也能用 tag，这部分我们在 Pass 讲解的时候来叙述 Pass 能用的 Tag。

CommonState



CommonState 的规则对 SubShader 和 Pass（这个在后面的 passdef 中讲述）是一样的，只不过在 SubShader 中，表示这个 CommonState 会作用到 SubShader 中的每个 pass，因此算是公共的。直白点说，就是这里讲述的 CommonState 里面的东西，同样可以用到 pass 的 state 处：写在 SubShader 块里，表示该 state 影响到该 SubShader 中的所有 pass；写在 pass 中，则 state 只作用到本 pass。



CommonState 用来设置 GPU 的不同状态，比如，要不要进行 alpha 混合，雾效要不要开等等。下面是 CommonState 的列表：

- Material

格式：`Material {Material Block}`

定义物体的材质。*Material Block* 部分可以是空的，这种情况下，默认值是黑色（数值上是 0，因此对光照来说，就不起效果）。可用的列表：

- 1、*Diffuse* - 物体的漫反射光或者说是基本的颜色
- 2、*Ambient* - 物体接收到的环境光
- 3、*Specular* - 物体的高光
- 4、*Shiness* - 高光的锐度（介于 0 到 1）。0 的时候高光范围看起来比较大，1 的时候小。
- 5、*Emission* - 物体自身的光（自发光）

- *Lighting*



格式：*Lighting On* | *Off*

如果想要前面的 *Material* 块起作用，那么必须用上这里的 *Lighting On* 命令。如果是

Lighting Off，颜色就直接从下面要讲的 *Color* 命令中得到了。*Light On* 也会启用

per-vertex 光照。

- *Cull*

格式：*Cull Back* | *Front* | *Off*

控制多边形的哪个面需要被裁剪

1、Back - 默认值。裁剪掉背面

2、Front - 裁剪掉前面

3、Off - 不要裁剪

- ZTest

格式: `ZTest Less` | `Greater` | `LEqual` | `GEqual` | `Equal` | `NotEqual` | `Always`

设置深度测试的模式。默认值是 `LEqual`，这些值都算是自解释了，不赘述。

- ZWrite

格式: `ZWrite On` | `Off`

需不需要写深度缓冲。

- Fog

格式: `Fog {Fog Block}`

设置雾效的细节。Fog Block 的可用命令列表:

1、Mode `Off` | `Global` | `Linear` | `Exp` | `Exp2` - 设置雾效模式。

2、Color 色值 - 雾效颜色

3、Density 浮点数 - 雾效强度

4、Range 浮点数, 浮点数 - 设置雾效的范围

- AlphaTest

格式: AlphaTest (Less | Greater | LEqual | GEqual | Equal | NotEqual | Always |

Never | Off) 基准值

Alpha 测试设置

- Blend

格式: Blend SourceBlendMode DestBlendMode

设置 Alpha 混合的参数。

- Color

格式: Color 色值

设置颜色。这个只有在 Lighting Off 的时候会被使用到。

- ColorMask

格式: `ColorMask RGB | A | 0 | 任何 R, G, B, A 组合`

设置通道掩码, 控制通道的写入。0 表示不写任何通道。

- Offset

格式: `Offset OffsetFactor, OffsetUnits`

设置深度缓冲的偏移值

- SeparateSpecular

格式: `SeparateSpecular On | Off`

这个命令让高光最后加入计算, 这样就不会被贴图影响到 (使用 On)。只有在 Lighting On 使用的时候才有效。

- ColorMaterial

格式: `ColorMaterial AmbientAndDiffuse | Emission`

使用顶点色, AmbientAndDiffuse 替换 Material 块中的 Ambient 和 Diffuse 值; 同理, Emission 控制 Material 中的 Emission 值。

Passdef

Pass 在 Unity 中有 3 个类型

1、regular Pass - 这个是一般的，常见的

格式: `Pass { [Name and Tags] [RenderSetup] [TextureSetup] }`

- *Name and Tags*

Name 用来给 Pass 起个名字，主要的作用配合 Use Pass(Pass 中的一个类型，紧跟此处的 regular Pass 后讲述)。Use Pass 会需要名字来让 unity 知道用到哪个 pass。

Tag 的格式: `Tags { "TagName1" = "Value1" "TagName2" = "Value2" }`

下面要讲的 Tag 只能用在 Pass 块中:

LightMode tag

可取的值列表:

Always - 不应用光照

ForwardBase - 用在前置渲染 (forward rendering) 中。环境光，主方向光，顶点光，球协光 (SH Lights) 和光照图在这里应用。

ForwardAdd - 用在前置渲染中。额外的光照会被应用到，一个 light 一次 pass。

Deferred - 用在延迟渲染 (deferred rendering) 中。渲染出 g-buffer (延迟渲染过程中的一个中间贴图/缓冲)。

ShadowCaster - 把物体深度信息渲染到影子图或者深度图中。

PrepassBase - 用在延迟渲染中。渲染法线和高光。

PrepassFinal - 用在延迟渲染中。渲染最后的结果。

Vertex - 用在顶点色渲染中，当物体没有使用 lightmap 时用。

VertexLMRGBM - 用在顶点色渲染中，当物体使用 lightmap 的时候用，限 PC 和主机平台。

VertexLM - 用在顶点色渲染中，当物体使用 lightmap 的时候用，限移动平台。

RequireOptions tag

pass 可以被指示只在某些条件下才运行。这个 tag 就起这个作用。目前来说只有一个值：

SoftVegetation - 只在 Unity 的 Quality Settings 中 Soft Vegetation 是 On 的时候渲染。

- *RenderSetup*

这部分跟上面讲的 SubShader 中的 CommonState 一样，用来设置渲染状态。这里不重复。如果读者跳过上面的内容，可以翻上去大致看一下。要强调的是，pass 的 RenderSetup 和 SubShader 的 CommonState 可以设置的内容是一样的，只不过用在 SubShader 块中，就表示所有的 pass 会受影响。用在 pass 中，就只有该 pass 受影响。

- *TextureSetup*

在渲染状态后，可以用 `SetTexture` 命令来指定多张贴图

格式：`SetTexture texture property { [Combine options] }`

注意：这个设置应用在固定管线模式下的多贴图流水中，如果有像素着色器同时出现，

则这个设置会被忽略。Unity5.0 把这个部分置为 Legacy 标记了，所以这部分不再多讲。

以后都用像素着色器来替代这个功能吧。

2、Use Pass - 这个 Pass 是从其他的 Pass 拿过来的，可以减少产生重复的代码

格式：`UsePass "Shader/Name"`

这个命令能让 shader 直接用到名字为“Shader”中的“Name” pass。

3、Grab Pass - 这个 Pass 抓取当前屏幕内容到一张贴图，后续可以通过 `_GrabTexture` 来访问。

格式：`GrabPass { }` 或者 `GrabPass { "TextureName" }`

第一种形式会抓取当前屏幕内容到贴图。只不过每个用到这个 shader 的问题都会产生一次抓取。

第二种形式功能与第一个相同，不过它只会在每帧遇到第一个使用 TextureName 的物体时使用。访问的时候需要用到 TextureName。

Fallback

这个是为了 Unity 在运行的时候在所有的 Subshaders 中都找不到合适的块而设立的。它告诉 Unity，如果所有的 Subshaders 都不适合，那么，用什么 shader。

它只有两种格式：

1、Fallback "name"

挑选那个名字叫做 name 的 shader 做 fallback。

2、Fallback Off

不管发生什么，不要 fallback。

CustomEditor

如果这个部分有内容，那么 Unity 就会通过内容（也即名字）来寻找一个扩展 Shader 界面的类。如果找到的话，使用该 shader 的任何材质的面板就用这个类来绘制。

格式：

CustomEditor "name"

到这里，基本上 ShaderLab 格式的大多数东西都已经介绍过了。不管你是跳着看过来的，还是一点点的阅读过来的，我要提醒你的是，上面的内容涉及很多的细节，因此，请把这部分内容作为手册，标记好方便经常翻阅。

我要做的事情是，帮你总结一下，防止这么一大段内容看下来，你已经不知道本来在学什么了。本节内容主要是学习 ShaderLab，也就是 Unity 用来写 Shader 的语言（格式）。每个能在 Unity 中运行的 shader，都是一个符合 ShaderLab 的文件。ShaderLab 在 Unity 官方说来，一共有三种形式：

- 1、 Surface Shader
- 2、 Vertex Fragment Shader
- 3、 Fixed function Shader

第一种 shader 的例子是我们的第一个写的 shader，它的标志是 Pass 块中有 #pragma surface 字眼。第二种是 Pass 块中有 #pragma vertex 或者 #pragma fragment 字眼，举例：

Vertex fragment Shader 例子

```
Shader "Display Normals"
{
    SubShader
    {
        Pass
    }
}
```

```
CGPROGRAM

#pragma vertex vert
#pragma fragment frag
#include "UnityCG.cginc"

struct v2f
{
    float4 pos : SV_POSITION;
    float3 color : COLOR0;
};

v2f vert (appdata_base v)
{
    v2f o;
    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    o.color = v.normal * 0.5 + 0.5;
    return o;
}

half4 frag (v2f i) : COLOR
{
    return half4 (i.color, 1);
}

ENDCG

}

Fallback "VertexLit"
}
```

第三种是没有上述标记的，举例：

Fixed function shader 例子：

```
Shader "VertexLit" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,0.5)
        _SpecColor ("Spec Color", Color) = (1,1,1,1)
        _Emission ("Emmision Color", Color) = (0,0,0,0)
    }
}
```

```
_Shininess ("Shininess", Range (0.01, 1)) = 0.7
_MainTex ("Base (RGB)", 2D) = "white" { }

}

SubShader {
    Pass {
        Material {
            Diffuse [_Color]
            Ambient [_Color]
            Shininess [_Shininess]
            Specular [_SpecColor]
            Emission [_Emission]
        }
        Lighting On
        SeparateSpecular On
        SetTexture [_MainTex] {
            constantColor [_Color]
            Combine texture * primary DOUBLE, texture * constant
        }
    }
}
}
```

认识这三种方式在实际中很重要，因为由于 Unity 的开放性（它可以让用户发布自己的 shader 等到应用商店 assetstore 中），各种 shader 都可能产生，甚至是一个 Shader 中用几种方式混写的。

Vertex fragment Shader 除了 Shader 定义的 Properties 部分外，其他的语法都是跟 cg、HLSL、GLSL 一样（事实上，Unity 编译器对 surface shader 都是先翻译成内部的 cg、HLSL、GLSL 这些通用的语言然后给底层的 DirectX 或者 OpenGL 引擎再编译）。

Vertex fragment Shader 使用 cg、HLSL 的时候，用 CGPROGRAM 和 ENDCG 来标记。对于 GLSL，需要用到 GLSLPROGRAM 和 ENDGLSL 对，这里是一个例子

```
Shader "GLSL basic shader"
{
    SubShader
    {
        Pass
        {

            GLSLPROGRAM

            #ifdef VERTEX

            void main()
            {
                gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
            }

            #endif

            #ifdef FRAGMENT

            void main()
            {
                gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
            }

            #endif

            ENDGLSL
        }
    }
}
```

对于 cg、HLSL、GLSL，这些语言语法都可以在讲述 cg、HLSL、OpenGL 的书上看到，请大家自行寻找相关书籍。

在结束本节内容之前，我们还需要对 Pass 中的 #pragma 做一番解释。

对于 cg、HLSL、GLSL 语言来写的 shader 程序，都是嵌在 ShaderLab 中的文字，一般在 Pass{} 段中。比如：

```
Pass
{

    CGPROGRAM

    #pragma vertex vert
    #pragma fragment frag

    ENDCG

}
```

Cg 片段

cg 片段放置在 CGPROGRAM 和 ENDCG 中。

段落开始处可以写下用 #pragma 开始的“编译指令”，指令会告诉编译器那些功能需要被编译。

- #pragma vertex *name* - 编译方法 *name* 为顶点着色器。
- #pragma fragment *name* - 编译方法 *name* 为像素着色器。

- `#pragma geometry name` - 编译方法 `name` 为 DX10 的网格着色器. 该选项会在 `#pragma target 4.0` (下面有介绍) 使用时自动打开.
- `#pragma hull name` - 编译方法 `name` 为 DX11 下的 hull 着色器. `#pragma target 5.0` 使用时自动打开.
- `#pragma domain name` - 编译方法 `name` 为 DX11 domain 着色器. `#pragma target 5.0` 使用时自动打开.
-

其他的编译指令:

- `#pragma target name` - 为哪些目标平台编译 (目标平台下面有讲述)
- `#pragma only_renderers space separated names` - 仅为哪些渲染器编译. 默认的情况是为所有的渲染器编译 (渲染器下面有介绍)
- `#pragma exclude_renderers space separated names` - 不为这些渲染器编译.
- `#pragma multi_compile ..._ - ...` 中跟用空格隔开的各个变体的名字, 这个可以指示编译器为该 shader 的不同变体编译不同的 shader 出来 (Unity 中可以通过 `Material.EnableKeyword()` 和 `Shader.EnableKeyword()` 来选择不同的变体)
- `#pragma enable_d3d11_debug_symbols` - 为 DirectX11 生成调试信息, 这些调试信息可以用来在 Visual Studio 2012 中作为图形调试用.

每个片段必须包含至少一个顶点方法和一个像素方法. 因此 `#pragma vertex` 和 `#pragma fragment` 是必须的。

在 Unity5.0 后，早先的这些指令已经没有用了，Unity 编译的时候会忽视

```
#pragma glsl,
```

用桌面 OpenGL 平台编译着色器时,转换成 GLSL 里面的 Cg/HLSL(而不是默认设置的 ARB 顶点/片段程序)。

```
#pragma glsl_no_auto_normalization,
```

```
#pragma profileoption,
```

```
#pragma fragmentoption
```

添加选项到编译的 OpenGL 片段程序。通过 ARB 片段程序 可以查询到所允许的规范的选项列表。这个指令对顶点程序或者不是以 OpenGL 为编译目标的程序无效。

注:

Shader 目标平台

默认情况下，Unity 会按照 SM2.0 的标准来编译 shader。使用 #pragma target 能让 unity 为合适的目标生成相关的代码。
当前的目标平台：

- #pragma target 2.0 (默认) - Shader Model 2.0
Direct3D 9 Shader Model 2.0.
- #pragma target 3.0 - shader model 3.0:
Direct3D 9 Shader Model 3.0.
- #pragma target 4.0 - DX10 shader model 4.0.
Direct3D 10 Shader Model 4.0.
- #pragma target 5.0 - DX11 shader model 5.0.
Direct3D 11 Shader Model 5.0.

要说明的是所有类 OpenGL 的平台（包含移动平台）可以认为“兼容 SM3.0”。Winphone8/WinRT 平台(DX11 9.x)可以认为仅相当于 SM2.0.

渲染器

Unity 支持很多的渲染 API（比如 Direct3D 9 和 OpenGL），默认情况下，shader 会被编译出支持每个渲染器的版本。你可以使用 `#pragma only_renderers` 或者 `#pragma exclude_renderers` 指令来告诉 unity 为哪些平台编译。

当前支持的渲染器：

- d3d9 - Direct3D 9.
- d3d11 - Direct3D 11.
- opengl - OpenGL.
- gles - OpenGL ES 2.0.
- gles3 - OpenGL ES 3.0.
- metal - iOS Metal.
- d3d11_9x - Direct3D 11 9.x feature level, as commonly used on WSA/WP8 platforms.
- xbox360 - Xbox 360.
- xboxone - Xbox One.
- ps3 - PlayStation 3.
- ps4 - PlayStation 4.
- psp2 - PlayStation Vita.
- psm - PlayStation Mobile.

举例来说，下面一样仅让编译器编译针对 D3D9 的版本：

```
#pragma only_renderers d3d9
```

这节课后的教程，对于写的 shader，大家可以通过特定的特征来判断 shader 用的是哪种形式。并且，本节的内容，主要的作用第一次是混个脸熟，而后主要用来做手册。

ShaderLab 的 Properties 实验

在 Unity 中，一个 shader 的属性非常重要，因为属性可以让美术人员或者用户不用会代码就能添加材质、改动数值从而改变 shader 的显示。属性通过在 Inspector 面板上提供出相关的控件出来，可以可视化的、所见即所得的实现相关的 shader 行为的修改。

为了方便，我们把第一份 shader 代码贴在下面：

```
Shader "BasicDiffuse"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }

    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert

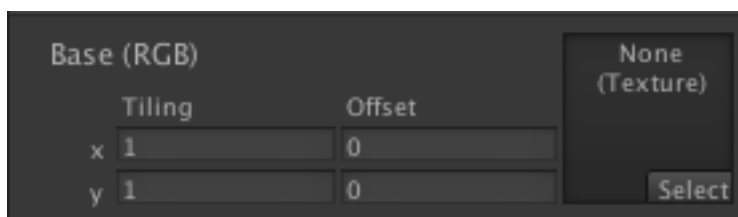
        sampler2D _MainTex;

        struct Input
        {
            float2 uv_MainTex;
        };

        void surf (Input IN, inout SurfaceOutput o)
        {
            half4 c = tex2D (_MainTex, IN.uv_MainTex);
            o.Albedo = c.rgb;
            o.Alpha = c.a;
        }
        ENDCG
    }
}
```

```
}  
    FallBack "Diffuse"  
}
```

从第 3 行的 Properties 开始到第 6 行位置，我们把这个块叫做“属性代码块”。当前，它只有一个属性：_MainTex。如果你在 Unity 中选中应用了我们这个 shader 的 material，你可以在 inspector 面板上看到一个可供选择图片的控件，如下图：



你可以通过右下角的 Select 这个按钮来选取工程里的图片，也可以把图片从 project 面板中直接拖到 select 所在的框里面。这个控件框（包含它左手边的 Base (RGB) 等），都是 shader 中的代码 Properties { _MainText ... } 让 Unity 为我们自动生成出来的。因此，我们可以使用 Unity 的这个方法，尽量把 shader 需要的外部“变量”作为属性传给 shader，这样就减少了代码的改动量。

动手实验

- 用 MonoDevelo 打开我们的第一个 shader，从 Properties 中去除原来的这一行：

```
_MainText ("Base (RGB)", 2D) = "White" {}
```

- 然后加入下面的代码，保存

```
_EmissiveColor ("Emissive Color", Color) = (1, 1, 1, 1)
```

- 回到 Unity 中等待 Unity 编译完我们的 shader。请保证你是选中的应用了我们的 shader 的 material，Inspector 面板会变成下图，点击图中的白色部分会弹出对话框供我们选择不同的颜色。



- 再回到 shader 代码中，我们再加入这么一行代码：

```
_SomeValue("Some Value", Range(0, 10)) = 2
```

- 相关面板会多出来一个滑动条出来，可以让我们左右滑动在一定范围内确定一个数值



Properties 格式再解

每个 Unity 的 shader 都被 Unity 规定了一定的格式。Properties 块就是其中的一个。这个块主要是给 shader 的程序员一个很方便的方式去让 Unity 帮助创建出符合你的代码的控件出来，而不用程序员去关心怎么显示（如果想，unity 也提供了相应的方式让程序员自由发挥）。

在 Properties 块中的代码，需要符合下面的格式：



“变量”这个名字，是在 shader 其他代码中会被引用，用来得到这个变量所对应的值。

“Inspector 显示名”，是我们这个属性在 Inspector 面板上显示出来的名字。

“类型”是变量属于的种类，同时也决定了变量在 Inspector 面板上对应的控件。

“默认值”是变量的初始化值，相应的，它的写法也跟变量的类型有关。

下面的表格描述了在 shader 中我们可以用到的“类型”：

Range(min, max)	创建一个滑动条，可以让用户选择在 min 和 max 之间的浮点数值
Color	创建一个调色板选择器，可以让用户选择颜色
2D	创建一个图片选择框，可以让用户选择贴图
Rect	创建一个 non-power-of-2 贴图选择框，功能基本跟 2D 想同
Cube	创建一个选择 Cubemap 的框
Float	创建一个让用户填入浮点数的栏位
Vector	创建 4 个栏位，让用户可以填入相应的浮点数，代表一个 Vecto4

然后看“默认值”的写法：

- Range 和 Float，一个浮点数
- Color 和 Vector，4 个用逗号隔开的数字，用()包住。对应于 Color，这四个是分别是 R、G、B、A 的归一化的值（数值范围从 0 到 1，代表 0 到 255 这个常见的色彩值范围）。对应于 Vector，是 x、y、z、w 值。
- 2D, Rect 和 Cube，可以选择空的字符串或者 Unity 内部的默认贴图名字：

"white", "black", "gray" 或 "bump"



注意：

2D、Rect、Cube 的默认值后面可以跟上 {}，类似这样的写法

```
name ("display name", 2D) = "name" { options }
```

这个部分是可选的。如果需要，这里的选择是两种模式：

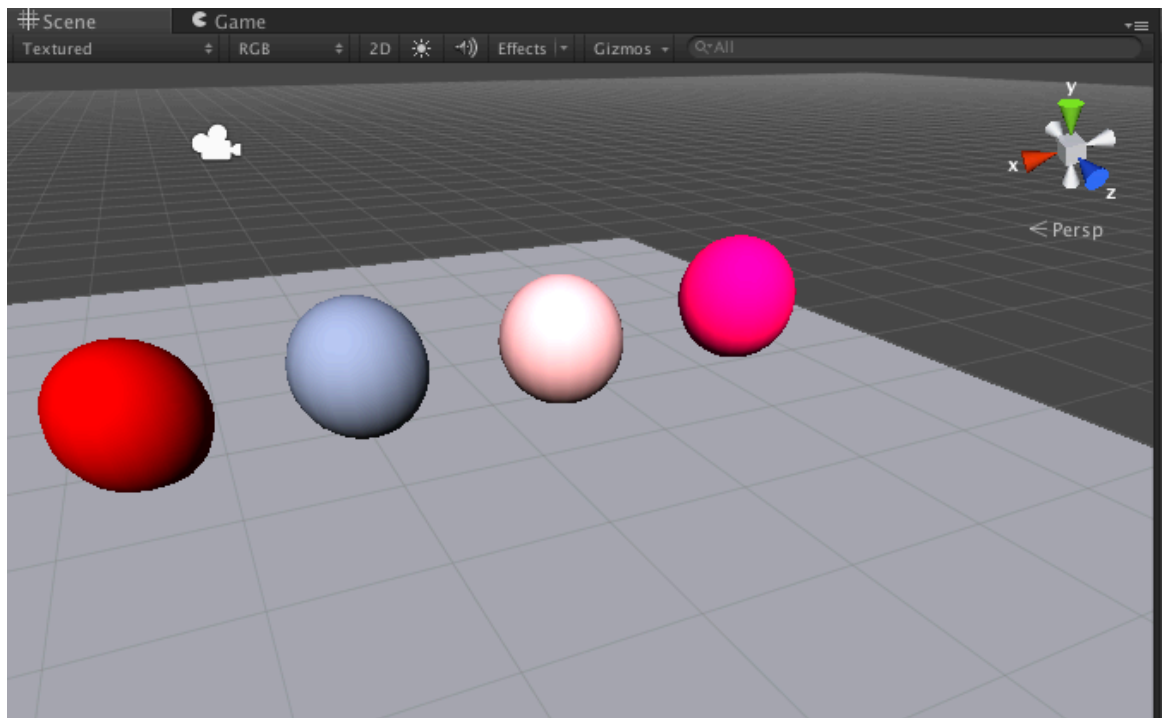


- **TexGen** *texgenmode*: 自动生成贴图的 uv 坐标。*texgenmode* 参数可以是：
ObjectLinear, *EyeLinear*, *SphereMap*, *CubeReflect*, *CubeNormal*, 都直接对应到 OpenGL 的 *texgen* 模式。如果用户用了自定义的顶点方法，那么 TexGen 会被 unity 忽略。这部分在 5.0 被标注成过时 (*legacy*) 了。

- `LightmapMode` 指示 Unity，我们这里的图会从每个 `Renderer` 组件中来，而不是直接在 `material` 中指定。

理解 Shader 和 Material 的关系

本节我们开始利用我们学到的 Properties 知识来改动代码，用一个 shader 实现如下图的几个不同的球体：



步骤：

- 使用 MonoDevelop 打开 shader 文件，内容从上节拷贝即可
- 保证 Properties 里面是如下内容

```
_EmissiveColor ("Emissive Color", Color) = (1, 1, 1, 1)  
_AmbientColor ("Ambient Color", Color) = (1, 1, 1, 1)  
_SomeValue ("Coefficient", Range(0, 10)) = 2.5
```

- 找到代码中的#pragma surface surf Lambert 这一行，回车另起一行后加入下面的三行代码

```
float4 _EmissiveColor;  
float4 _AmbientColor;  
float _SomeValue;
```

- 清除原来在这个地方的

```
sampler2D _MainTex;
```

- 找到方法 void surf(Input IN, inout SurfaceOutput o)，把里面的内容改成

```
float4 c = pow((_EmissiveColor + _AmbientColor), _SomeValue);  
o.Albedo = c.rgb;  
o.Alpha = c.a;
```

为保证你做成的最后内容跟我的一样，这里是最后的 shader 代码

```
Shader "myDiffuse"  
{  
    Properties  
    {  
        _EmissiveColor ("Emissive Color", Color) = (1, 1, 1, 1)  
        _AmbientColor ("Ambient Color", Color) = (1, 1, 1, 1)  
        _SomeValue ("Coefficient", Range(0, 10)) = 2.5  
    }  
  
    SubShader  
    {  
        Tags { "RenderType"="Opaque" }  
        LOD 200  
  
        CGPROGRAM
```

```
#pragma surface surf Lambert

float4 _EmissiveColor;
float4 _AmbientColor;
float _SomeValue;

struct Input
{
    float2 uv_MainTex;
};

void surf (Input IN, inout SurfaceOutput o)
{
    float4 c = pow((_EmissiveColor + _AmbientColor), _SomeValue);

    o.Albedo = c.rgb;
    o.Alpha = c.a;
}
ENDCG
}
FallBack "Diffuse"
}
```

到这里，我们就可以保存 shader。回到 Unity 中。回到 Unity 后，选中名字为 myDiffuse 的 material，然后按下键盘上的 Ctrl+D，复制出 material，把它改名为 myDiffuse2。同样的方式，复制出 myDiffuse3, myDiffuse4。这个方式，能让我们制作出使用同一个 shader 的不同的四个 material 出来。

在 Scene 面板中，如果你没有加满 4 个 sphere，可以现在添加。然后分别给这四个球拖入我们的 myDiffuse、myDiffuse2、myDiffuse3、myDiffuse4 这几个 material。

选中任意一个 material，你可以在 Inspector 上选择 Emissive Color 或者 Ambient Color，也可以通过拖动滑动条来改动数值，改动的过程中可以看到 Scene 中的物体会随你的改动而变化显示。同时，虽然我们只写了一个 shader 用到 4 个物体上，改动一个物体的 material 并不会影响到跟它使用不一样的 material 的物体。也就是说，我们属性里的值，其实是存在于 material 中。在我们改变相关的值的时候，unity 会把值存到 material 对应的文件中，并在运行的时候，作为 shader 需要的参数送到 shader 中。由此，虽然我们只写了一个 shader，但是因为送入的参数不同，因此物体的表现也就不同（[为那些好奇的人，这节课的附录部分介绍了怎么打开 material 文件的方式，可以看到存储的内容](#)）。

当你在 Properties 块声明一个新的属性值的时候，你也就给 shader 提供了一种让用户通过 Inspector 面板调整内部值的方式。每个变量都会存储最后用户修正的结果。ShaderLab 规定，如果代码需要在 SubShader{} 这个块中用到这些值，必须在这个块中同时声明同样名字的变量，这个方式把我们在 Inspector 上的值和最后代码的值链接起来，形成一个整体。

一旦你在 SubShader{} 块中声明这些变量后，你就可以在后续方法中使用到这些变量了。这里的 surf() 方法中，我们用到的 pow() 方法是 NVidia（英伟达，GPU 的主要厂商之一）的 cg 语言提供的一个方法，unity shader 可以用 cg 中的方法，这个方法跟其他语言中的 pow 类似，是“返回指定数字的指定次幂”。

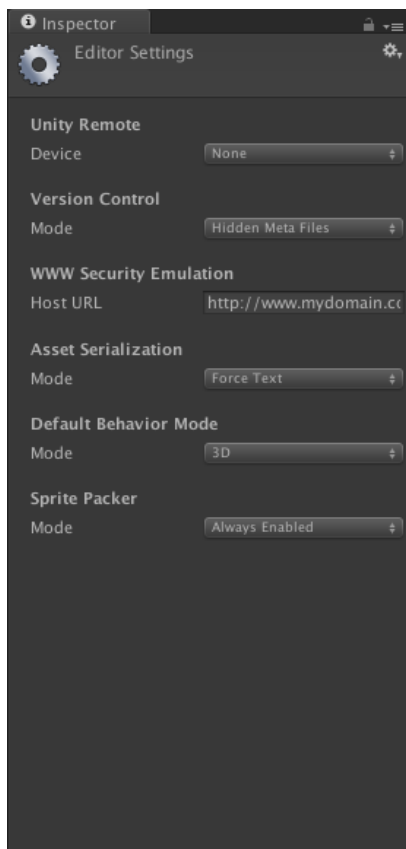
通过这个实验，我们可以了解，shader 在 3D 程序中，一般都是含在材质中的一段程序，它从材质中得到材质的必要属性，计算出相应的值来完成渲染。而材质，则是计算物体的各种属性的一个文件，shader 是它的其中一部分。

附录：

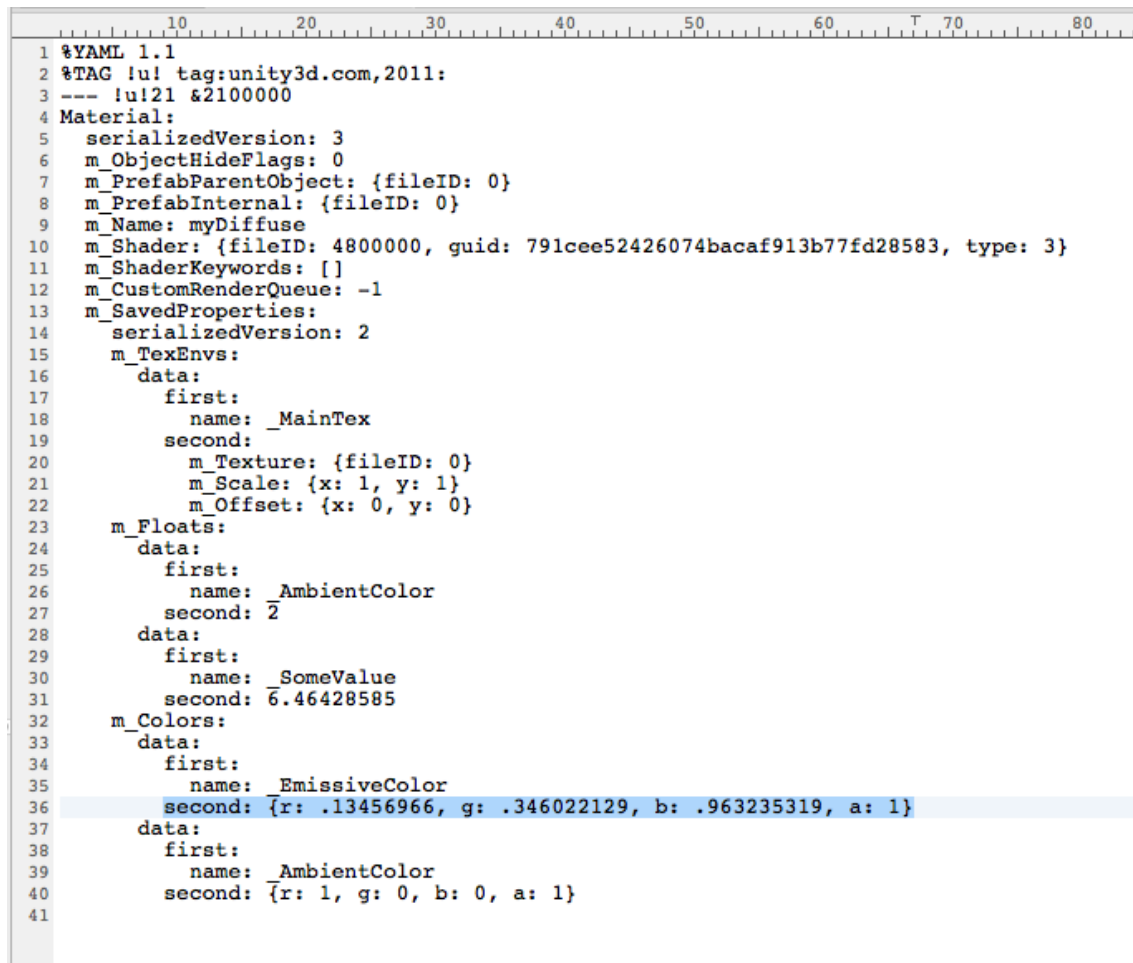
Unity 提供了把一些文件存成文本型的方式，我们可以利用这个方式来查看 material 文件的内容。

步骤：

- 菜单 Edit | Project Settings | Editor，会打开如下的面板



- 该面板的 Asset Serialization 中，Mode 点开，选择 Force Text
- 等进度条完成任务
- 硬盘上找到我们的 myDiffuse 材质（material）文件（以.mat结尾，用可以打开文本文件的软件打开（写字本、Ultraedit、Notepad++都可以）
- 查看里面的内容，我这里的如下图



```
1 %YAML 1.1
2 %TAG !u! tag:unity3d.com,2011:
3 --- !u!21 &2100000
4 Material:
5   serializedVersion: 3
6   m_ObjectHideFlags: 0
7   m_PrefabParentObject: {fileID: 0}
8   m_PrefabInternal: {fileID: 0}
9   m_Name: myDiffuse
10  m_Shader: {fileID: 4800000, guid: 791cee52426074bacaf913b77fd28583, type: 3}
11  m_ShaderKeywords: []
12  m_CustomRenderQueue: -1
13  m_SavedProperties:
14    serializedVersion: 2
15    m_TexEnvs:
16      data:
17        first:
18          name: _MainTex
19        second:
20          m_Texture: {fileID: 0}
21          m_Scale: {x: 1, y: 1}
22          m_Offset: {x: 0, y: 0}
23    m_Floats:
24      data:
25        first:
26          name: _AmbientColor
27        second: 2
28      data:
29        first:
30          name: _SomeValue
31        second: 6.46428585
32    m_Colors:
33      data:
34        first:
35          name: _EmissiveColor
36          second: {r: .13456966, g: .346022129, b: .963235319, a: 1}
37      data:
38        first:
39          name: _AmbientColor
40        second: {r: 1, g: 0, b: 0, a: 1}
41
```

图中高亮的地方可以看到我们选择的颜色值。

光照模型支持

3D 渲染中存在着多种多样的光照模型，作为一个成熟的商业引擎，Unity 本身提供两种常用的光照模型：**Lambert**（朗伯模型）和 **BlinnPhong** 模型。同时它也提供了方式，让开发者可以自己定义自己的光照模型，并且让 Unity 使用这个模型。

自定义光照模型（方程）

在我们前面的课程中，有一行代码如下

```
#pragma surface surf Lambert
```

这行代码指示 Unity 在编译我们的 shader 代码的时候，选用“Lambert”这个光照模型。它的格式是：`#pragma surface surf LightModel`。对于 Surface Shader 来说，因为用户只提供表面（surface 这个词的来历）信息，因此最后光照跟表面信息的计算需要用到的过程代码必须经由由此指定。

其中 LightModel 是 Unity 自带的光照模型的名字或者我们自己按照 Unity 的要求写好的。那么什么是光照模型呢？光照模型是反映某种物体的材质对光怎么反应的数学抽象，简单举个例子来说，你可以想象一个光子作为入射光从远处击中物体表面的某个部分，然后被反射、折射或者吸收。如果我们给定入射光以一定的数值，给定材质以一定的数值，最后发射或者折射等变化后这个光相关的数值（方向、方向上的能量）和原来的关系，就是我们的光照模型，有些地方也把这个称为光照方程。

但是这么说不能体现各种光照模型之间的差别，只是个笼统的说法。不同的光照模型，究竟区别在哪里？举个例子比较容易说明这个概念。有名的光照模型，Lambert 和 Phong（这些都是以发明这个模型的人命的名字），差别在于 Phong 比前者多考虑了物体的高光(specular)效果，因此在算式里会有计算 specular 的部分。这个是主要的差异。而 Phong 模型和

BlinnPhong 模型的差异，就小的多，主要是 Blinn 在 Phong 的基础上，用了更快速的计算 specular 的方法，但是因为改动了模型本身，而不是优化了原来的算法，所以也是另外一种模型。

光照模型或者光照方程最后都会由相关的数学公式表达出来，然后经引擎用代码变现。在 Unity 中，我们可以通过打开 Unity 安装目录下 Data\CGIncludes 的 Lighting.cginc 看到这个方程的代码，为方便，我把它拷贝如下：

```
inline fixed4 LightingLambert (SurfaceOutput s, fixed3 lightDir, fixed atten)
{
    fixed diff = max (0, dot (s.Normal, lightDir));

    fixed4 c;
    c.rgb = s.Albedo * _LightColoro.rgb * (diff * atten * 2);
    c.a = s.Alpha;
    return c;
}
```

方法名字上，比我们在#pragma 中看到的“Lambert”，多出了“Lighting”这几个字符。这个是 Unity 的规定：光照模型在#pragma 代码中不用带上“Lighting”字眼，Unity 在编译的时候会加上这个关键字后去寻找方法。因此，我们上述的#pragma surface surf Lambert 这句话，对应的方法就是 LightingLambert()。

光照模型的代码在 surface 代码运行后被 Unity 运行，surface 代码输出 Unity 定义好的一系列物体的”表面“的数值（决定物体的材质信息），然后光照模型计算光 and 该材质交互的结果输出，成为我们最后看到的颜色。

那我们可以自己写一个光照方程并且让 Unity 使用我们的自定义的方程吗？答案是肯定的。

下面我们就开始尝试写一个自己的光照模型的方法。

步骤：

- 使用我们原来的 shader 代码做基础
- 改动#pragma 一样代码中的 Lambert 到 BasicDiffuse
- 把下面的代码加入到 SubShader{ }块中

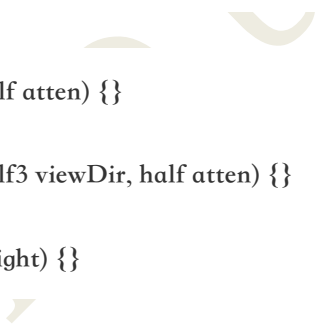
```
inline float4 LightingBasicDiffuse (SurfaceOutput s, fixed3 lightDir, fixed atten)
{
    float difLight = max(0, dot (s.Normal, lightDir));
    float4 col;
    col.rgb = s.Albedo * _LightColor0.rgb * (difLight * atten * 2);
    col.a = s.Alpha;
    return col;
}
```

- 保存该 shader，回到 Unity。

Unity 会开始编译我们的 shader，如果没有错，那么新的 shader 会应用到对应的物体上。从场景中的物体表现来看，没有任何差别（因为 BasicDiffuse 的代码逻辑跟 Lambert 的一模一

样)。但是在这个背后，Unity 的确使用了我们的光照模型"BasicDiffuse"，而不再使用它提供的"Lambert"。

Unity 用#pragma 这个预编译指令来让我们选择光照模型。有 3 中类型的关照模型的原型我们可以选择：

- 
- half4 LightingName (SurfaceOutput s, half3 lightDir, half atten) {}
 - half4 LightingName (SurfaceOutput s, half3 lightDir, half3 viewDir, half atten) {}
 - half4 LightingName_PressPass (SurfaceOutput s, half4 light) {}

第一个用在前置渲染不需要观察者方向的场合。第二个用在前置渲染需要观察者方向的场合。

第三个是 Unity 被配置成延迟渲染的时候需要的。

上面的方法中用到一个 dot()方法，这也是 cg 语言的一个库方法。用来计算两个方向之间的角度相关的值。在 Lambert 光照模型中，每个点的入射光的方向和它本身的法线位置之间的夹角，决定了反射出去的光的数值的强度。入射光和法线的方向的角度越小，结果中的数值就越大。

Half Lambert 模型

读完自定义光照模型的概念，我们这节来试验一个著名的光照模型，巩固前面的知识。

Half Lambert 光照模型是 Valve 公司在制作“半条命”游戏时发明的，用来给在比较暗的区域显示物体。总体来说，该光照模型提高了物体表面的漫反射光。下图是 Valve 的示例，左边是 Lambert 模型，右边边是 Half Lambert 模型。



使用我们原来的基础的 shader，我们把 LightingBasicDiffuse()方法改成如下：

```
inline float4 LightingBasicDiffuse (SurfaceOutput s, fixed3 lightDir, fixed atten)
{
    float difLight = dot (s.Normal, lightDir);
    float hLambert = difLight * 0.5 + 0.5;

    float4 col;
```

```
col.rgb = s.Albedo * _LightColoro.rgb * (hLambert * atten * 2);  
col.a = s.Alpha;  
return col;  
}
```

保存 shader 后，我们回到 Unity。

如果你没有对场景中的物体进行任何的改动，你可以发现现在物体整体表现比原来亮了。这个原因是因为代码

```
float hLambert = difLight * 0.5 + 0.5;
```

difLight 因为 dot()方法的关系，在[0, 1]范围，因此，hLambert 的范围在[0.5, 1]。这个算法把范围 0~1 映射到了范围 0.5~1 中，整体就提高了最后的数值。

Ramp Texture

光照模型了解后，我们可以开始真正的 shader 编写了。这节内容以 surface shader 为例，讲述一种控制漫反射的方式。

Ramp Texture 在 Valve 公司的军团要塞 2 后开始成为一种控制漫反射（diffuse）的方法。



Ramp Texture 是类似下图的一张贴图:



Ramp Texture示例

在一般的 Blinn/Phong 模型中, 我们对漫反射的系数是基于入射光和击中的点的法线的角度。

在这种情况下, 系数的计算代码大致如下:

```
float DiffuseCoeff(in float3 pos, in float3 normal, in float3 lightPos)
{
    float3 lightDir = lightPos - pos;
    lightDir.normalize();

    return max(0.0, Dot(lightDir, normal));
}
```

使用 Ramp texture, 可以用一张 1D 的材质图来做为索引表。用来控制漫反射的系数。虚拟

代码如下:

```
texture1D rampTex;
float DiffuseCoeff(in float3 pos, in float3 normal, in float3 lightPos)
{
    float3 lightDir = lightPos - pos;
    lightDir.normalize();

    float rampCoord = Dot(lightDir, normal) * 0.5 + 0.5;
    return tex1D(rampTex, rampCoord);
}
```

代码 `float rampCoord = Dot(lightDir, normal) * 0.5 + 0.5;` 这句，因为 `Dot()` 方法返回的值在 -1 和 1 之间，因此，`rampCoord` 最后的范围在 0 到 1 之间（乘法改动了区间的尺寸，而加法改动了区间在轴上的位置）。

我们这节课来改动原来的 shader，自己来观察原来的 shader 在加入 Ramp Texture 后能做的事情。

- 把原来的 `BaiscDiffuse` 代码改成如下

```
inline float4 LightingBasicDiffuse(SurfaceOutput s, fixed3 lightDir, fixed atten)
{
    float difLight = dot(s.Normal, lightDir);
    float hLambert = difLight * 0.5 + 0.5;
    float3 ramp = tex2D(_RampTex, float2(hLambert, hLambert)).rgb;

    float4 col;
    col.rgb = s.Albedo * _LightColor0.rgb * (ramp);
    col.a = s.Alpha;
    return col;
}
```

- 在 Properties 中加入

```
_RampTex("Ramp texture", 2D) = "white"
```

- 在 SubShader 块中加入

```
sampler2D _RampTex;
```

为了保证你最后的结果跟我的是一样的，这里是最后的代码：

```
Shader "myDiffuse"
{
```


Properties

```
{
    _EmissiveColor ("Emissive Color", Color) = (1, 1, 1, 1)
    _AmbientColor ("Ambient Color", Color) = (1, 1, 1, 1)
    _SomeValue ("Coefficient", Range(0, 10)) = 2.5
    _RampTex("Ramp texture", 2D) = "white"
}
```

SubShader

```
{
    Tags { "RenderType"="Opaque" }
    LOD 200
```

CGPROGRAM

```
#pragma surface surf BasicDiffuse
```

```
float4 _EmissiveColor;
float4 _AmbientColor;
float _SomeValue;
sampler2D _RampTex;
```

```
inline float4 LightingBasicDiffuse(SurfaceOutput s, fixed3 lightDir, fixed atten)
{
    float difLight = dot(s.Normal, lightDir);
    float hLambert = difLight * 0.5 + 0.5;
    float3 ramp = tex2D(_RampTex, float2(hLambert, hLambert)).rgb;

    float4 col;
    col.rgb = s.Albedo * _LightColoro.rgb * (ramp);
    col.a = s.Alpha;
    return col;
}
```

```
struct Input
{
    float2 uv_MainTex;
};
```

```
void surf (Input IN, inout SurfaceOutput o)
{
    float4 c = pow((_EmissiveColor + _AmbientColor), _SomeValue);

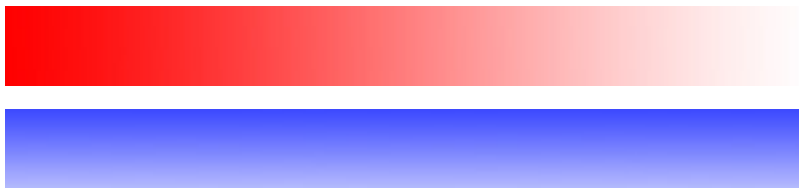
    o.Albedo = c.rgb;
```

```
        o.Alpha = c.a;  
    }  
    ENDCG  
}  
FallBack "Diffuse"  
}
```

- 保存，然后我们回到 Unity。

选中某个材质球，Inspector 面板上会出现选择图片的控件，这里我们可以拖入一张图观察结果。

对那些还不会使用绘图工具的读者，你可以用我这里给你准备好的几张图（ramp texture 生成方式可以使用 photoshop 的 gradient 工具，本节附录中有相应的步骤）



对于 Ramp Texture shader 来说，我们主要是加入了对送入的 Ramp texture 的色彩的检索，检索出来的色彩值最后能影响到我们的光照模型的输出。由此，美术人员在设计模型的时候，可以利用图来控制物体的光照。

tex2D()方法是 cg 的库方法，用来根据第二个参数提供的 uv 值(类型是 float2，分别代码水平方向的 u 和竖直方向的 v)，来检索第一个参数代表的图档中的 RGBA 值。

这里我们给定的 uv 值有 half lambert 的值决定，half lambert 越大，意味着我们索引到的像素的值的原图中越是偏向于右下角方向。美术可以利用这个特性来控制一个物体表面的漫反射（越是往右，越是往下，意为着物体的法线方向越来越接近于入射光）。

对于上述代码，有几个概念前面我们还没有讲过，这里依次解释：

- 1、代码 `LOD 200`，这行定义了对应 SubShader 的 LOD（level of detail）值，用户可以通过 Unity 的设置或者通过 API 来控制全局的 shader 的 LOD 值，这样 Unity 在运行的时候会抛弃 LOD 大于设置的值的 SubShader。是 Unity 的控制适用 SubShader 的另一种方式。
- 2、结构 `struct Input` 和 `float2 uv_MainTex`；这个涉及到 surface shader 的具体写法，可以参考下节内容。

附录：

在 Photoshop 中，我们可以几个步骤来建立 Ramp texture：



- 打开 photoshop，新建一张图
- 选中工具箱中的渐变工具（一般在橡皮擦的下面一个）
- 然后在图上从左到右拖动一根线出来
- 结果就得到一张从 photoshop 中设置的前景色过渡到背景色的图出来了

Surface Shader

Unity 自身的三种 shader 写法中，Fixed function shader 在 5.0 后基本上不再被支持；Vertex Fragment Shader 的知识主要从 cg、HLSL、GLSL 的书籍中得到；Surface Shader 大量使用的地方很多，又因为是 Unity 的发明，所以文档少，这节我们一起来学习它的相关知识。

shader 开发人员在写 shader 的时候发现自己经常在写重复的代码，而这部分完全可以交给通用的方法来做。这个就是 surface shader 被 Unity“发明”的原因。写这种 shader，开发人员只要提供物体的表面信息（surface），剩下的事情由 Unity 帮你完成。

同时，surface shader 只是一种代码生成方案，Unity 并不把它称作为一种新的语言，因为 Unity 的编译器，只是把 surface shader 翻译成相应的 Cg、HLSL，后续的工作依旧是通过 DirectX 或者 OpenGL 的 shader 编译器来真正的编译 shader 代码。

Surface Shader 怎么写

写 surface shader，我们先是

1、在 SubShader 中用 `#pragma surface surf_func` 来定义要用到的方法；在介绍 ShaderLab 格式的时候，我们提到过，这个命令不能写在 Pass 中，因为 surface shader 会被翻译成相应的 Pass。

2、上述方法第一个参数是一个 struct Input,里面可以填上 shader 需要从外部送进来的数据,比如 UV 数据; Unity 运行的时候,会根据这个结构知道该给本 shader 喂哪些数据。Input 中可以填的值:

```
float3 viewDir  
float4 变量用 COLOR 这个语义关键词来后置修饰  
float4 screenPos  
float3 worldPos  
float3 worldRefl  
float3 worldNormal  
float3 worldRefl;  
float3 worldNormal;
```

除了这些值,用户可以让 Unity 喂入 uv 值,这个只要在对应的贴图的名字前面加上 uv_这个前缀,比如,对应 MainTexture,需要用 uv_MainTexture。对应到第二套 uv,前缀是 uv2_。

3、第二个参数是一个 Unity 预定义的结构,前面用 inout 这个关键词来修饰,表示这个结构是个输入输出都会用到的东西。标准的结构如下:

```
struct SurfaceOutput  
{  
    fixed3 Albedo;  
    fixed3 Normal;  
    fixed3 Emission;  
    half Specular;  
    fixed Gloss;  
    fixed Alpha;  
};
```

Unity5, 开始也可以用这两个结构:

```
struct SurfaceOutputStandard  
{  
    fixed3 Albedo;  
    fixed3 Normal;
```

```
half3 Emission;  
half Metallic;  
half Smoothness;  
half Occlusion;  
fixed Alpha;  
};  
  
struct SurfaceOutputStandardSpecular  
{  
    fixed3 Albedo;  
    fixed3 Specular;  
    fixed3 Normal;  
    half3 Emission;  
    half Smoothness;  
    half Occlusion;  
    fixed Alpha;  
};
```

- 4、然后在这个方法中，我们会把一个 Unity 预定义的结构中的相应的值（不需要所有）填好值就完成了整个 shader 的工作。

常见实例

前面的章节大多在讲述一些 ShaderLab 的基础，在这些基础上，我们开始试验一些实例，来拓展 ShaderLab 的知识。

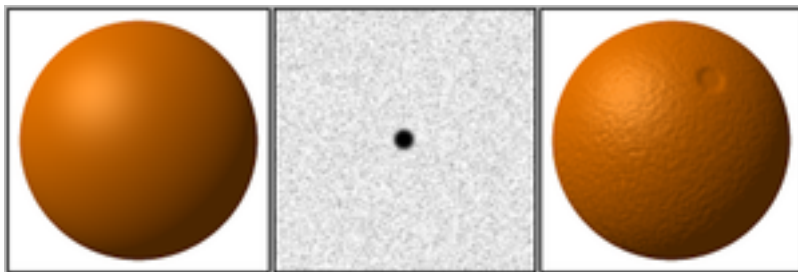
材质贴图和凹凸贴图

在前面的课程中，我们已经用到了贴图，只要 shader 的 Properties 块中用到了 2D、Cube、Rect 字眼的，我们都能在相应的 Inspector 框上看到可以选择贴图的控件。

在 shader 编程中，贴图是一张图，最早的时候通常用来表现物体表面的纹理，所以最常见的贴图也就是所谓的纹理图。木质条纹（下图）、砖墙表面，都是我们显而易见的贴图。

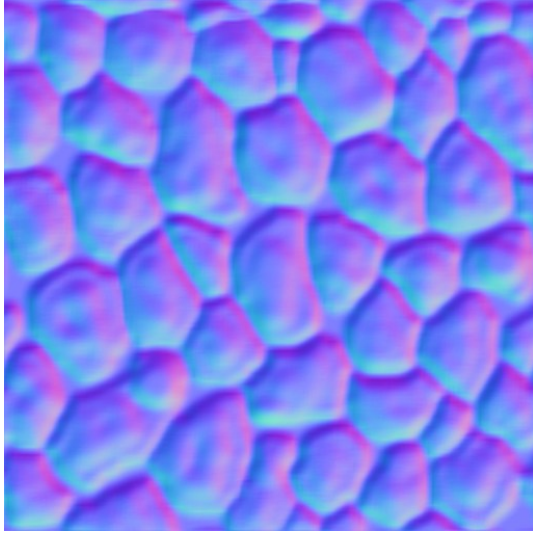


随着渲染技术的发展（渲染通俗来说，是计算机计算最后绘制出的画面的过程），产生了一些中间性的贴图，用在渲染的过程中供计算机计算一些额外的信息。常见如凹凸贴图，



上面的 3 幅图片，中间的图片用来告诉计算机物体上的凹凸状态。计算机通过对该图片的参考，混合最左边的球体输出，最后得到右边的图片，这样就让物体表面看起来有凹凸感。

而现在用的更广泛的则是一种名字叫“法线贴图（normal map）”的贴图，



这种整体倾向于蓝色的贴图里面，存储了对应物体表面的“法线”信息。计算机通过对这张图的参考，了解物体上的凹凸信息，从而在渲染的时候绘制出物体表面的凹凸感。

下面的内容我们通过自己的 shader 来实现法线贴图功能。

法线贴图, 英文名字为"normal mapping", 因为算法中会使用到数学运算 Dot(求向量的点积), 所以也称为"Dot3 bump mapping"。

在 3D 世界里面，对于物体的表面，我们如果需要让它看起来很立体，通常的方式是给这个物体的表面加入更多的面数来表现。但是这样的方式增加了图形处理器的负担（因为一个图

形处理器在渲染的过程，更多的面数意味着更多的计算量）。那么有没有稍微负担小一点的方式呢？这个答案就是法线贴图。

在下图中，场景中放置的人物没有经过法线贴图的处理，



同样的人物加入了法线贴图后，可以看到表现力更加的生动（注意红框区域的细节跟没有用法线贴图的对比）。直观来讲，用了法线贴图后，细节处理的更加有质感，层次感更加的强烈。在上图中，虽然本身的贴图做的已经非常的到位，但是因为没有办法使人物在场景中实时

的“感光”，所以相比较而言，人物相对平面化，比如右手臂上有青筋暴起，但是跟加入法线贴图的相比较，就还不够“暴”。加入法线贴图后，青筋的立体感立马就从纸上一跃而出。



下面的实验我们来自己动手做一个法线贴图并且应用到我们以前的物体上

- 新建一个 Unity 工程
- 新建一个 sphere 作为我们的物体
- 新加一个 Directional light
- 拷贝我们原来的 shader 文件到该工程
- 打开 shader 文件，全部选中代码，清空，然后拷贝下面的代码

```
Shader "myNormalMap"
{
    Properties
    {
        _MainTex("Base Texture", 2D) = "white"
        _NormalMap("Normal map", 2D) = "white"
    }

    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert

        sampler2D _MainTex;
        sampler2D _NormalMap;

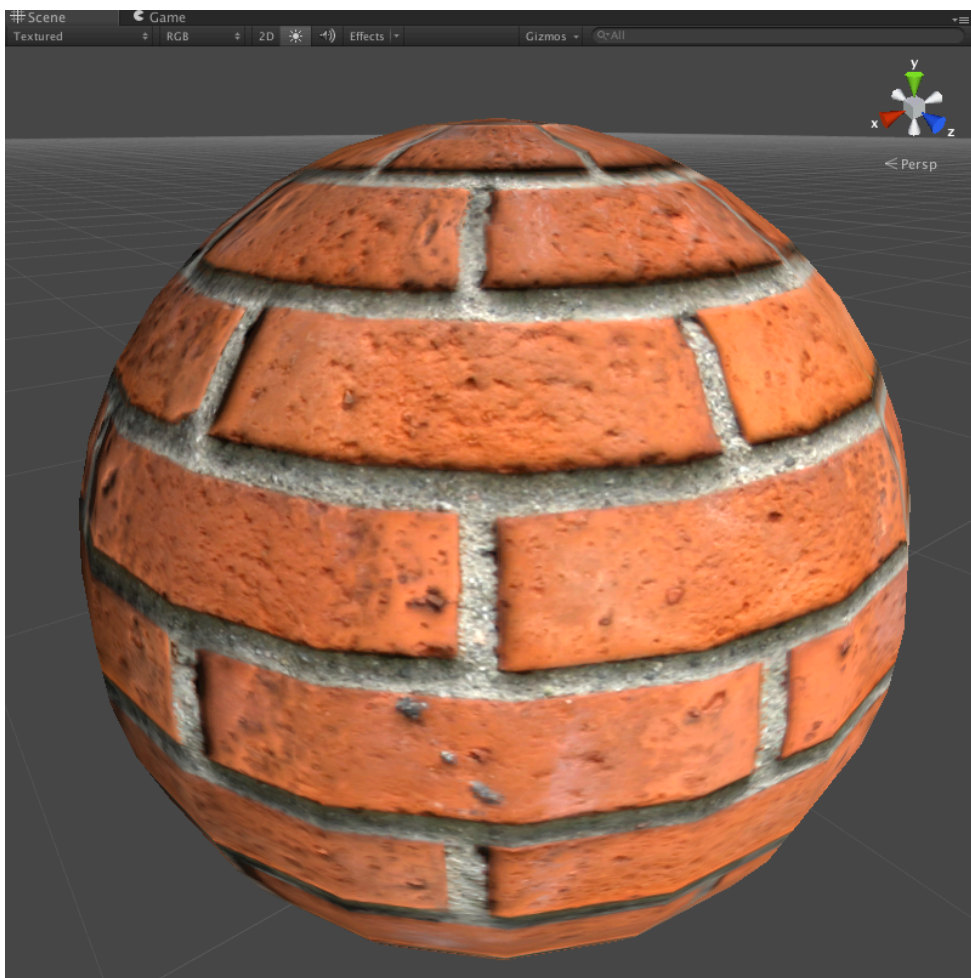
        struct Input
        {
            float2 uv_MainTex;
            float2 uv_NormalMap;
        };

        void surf (Input IN, inout SurfaceOutput o)
        {
            fixed4 c = tex2D(_MainTex, IN.uv_MainTex);

            o.Albedo = c.rgb;
            o.Alpha = c.a;
            o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_NormalMap));
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

- 保存后，为该 shader 建立一个 material。
- 把 material 拖到 sphere 上。

- 选中该 material，面板上 Base Texture 部分拖入一张物体的材质图
- 使用工具 Crazybump（请 google 之）生成一张 NormalMap，并把这个 NormalMap 拖到 material 的第二个框里。如果找不到该工具，可以通过搜索引擎之类的找到一些凹凸贴图来使用。
- 点击第二个框，在选中 None 和选中我们的 normalmap 间切换，观察场景中的物体的表面凸起随着 normalmap 选中和非选中状态下的细节变化。



Unity 的 Surface Shader，对比起其他的 shader 语言，对初学者非常友好。这里表现在它提供的 UnpackNormal 方法，会帮我们从 NormalMap 这张图中解析出法线信息出来，我们只

要调用这个 Unity 的自带的方法，然后把解析出来的法线信息赋值给 Surface Shader 的 Normal 结构就完成任务，而不用我们去管怎么解析这个包装在法线贴图中的法线信息的细节。如果有兴趣，你可以自行以 dot3 bumpmap 这样的关键词去搜索后面的算法。

```
o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_NormalMap));
```

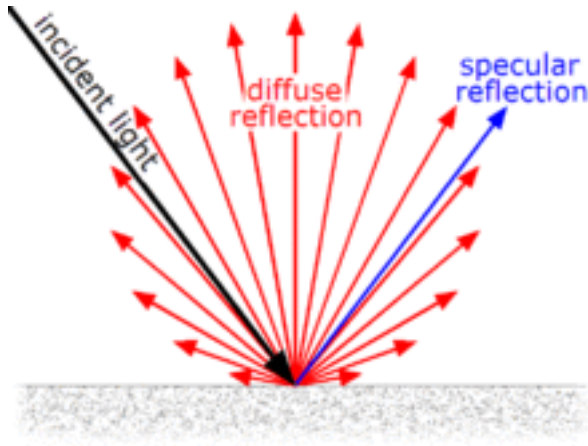
附录：

Crazybump 是一个可以通过贴图生成法线图的工具，它的官方网站在

<http://www.crazybump.com/>

高光效果

漫反射所表明物体材质可以简单描述如下：入射光击中物体上的某个点后，会在法线对应的半球范围内反射能量。而高光反射是另外一种性质，入射光击中物体上的某个点后，反射的范围局限在一个很窄的角度之内。这两种性质，可以看 wiki 上的这张图：



图中的"incident light"也就是入射光，“diffuse reflection”是红色的箭头，表示漫反射；"specular reflection"是蓝色的箭头，代表高光反射。

从图中可以看到，高光反射是随人眼位置不同而表现出很大差异的。如果人眼正好在反射的蓝色箭头上区域看向入射光击中的位置，那么看到的该点就会比较亮(反射的能量击中在该区域)；随着离开这个蓝色线的角度越来越大，那么看到该点的亮度就会越来越小（基本没有什么反射的能量在这些区域）。

可以看出来，我们如果要想实现高光（specular），那么必须要知道人眼，在 3D 中，也就是相机的位置。这个差别在我们要写的 shader 中，还意味着我们需要换掉原来的 Lambert 光

照模型，因为该光照模型根本就不考虑人眼的位置。为方便，我把原来的 Lambert 光照模型代码放在这里

```
inline fixed4 LightingLambert (SurfaceOutput s, fixed3 lightDir, fixed atten)
{
    fixed diff = max (0, dot (s.Normal, lightDir));

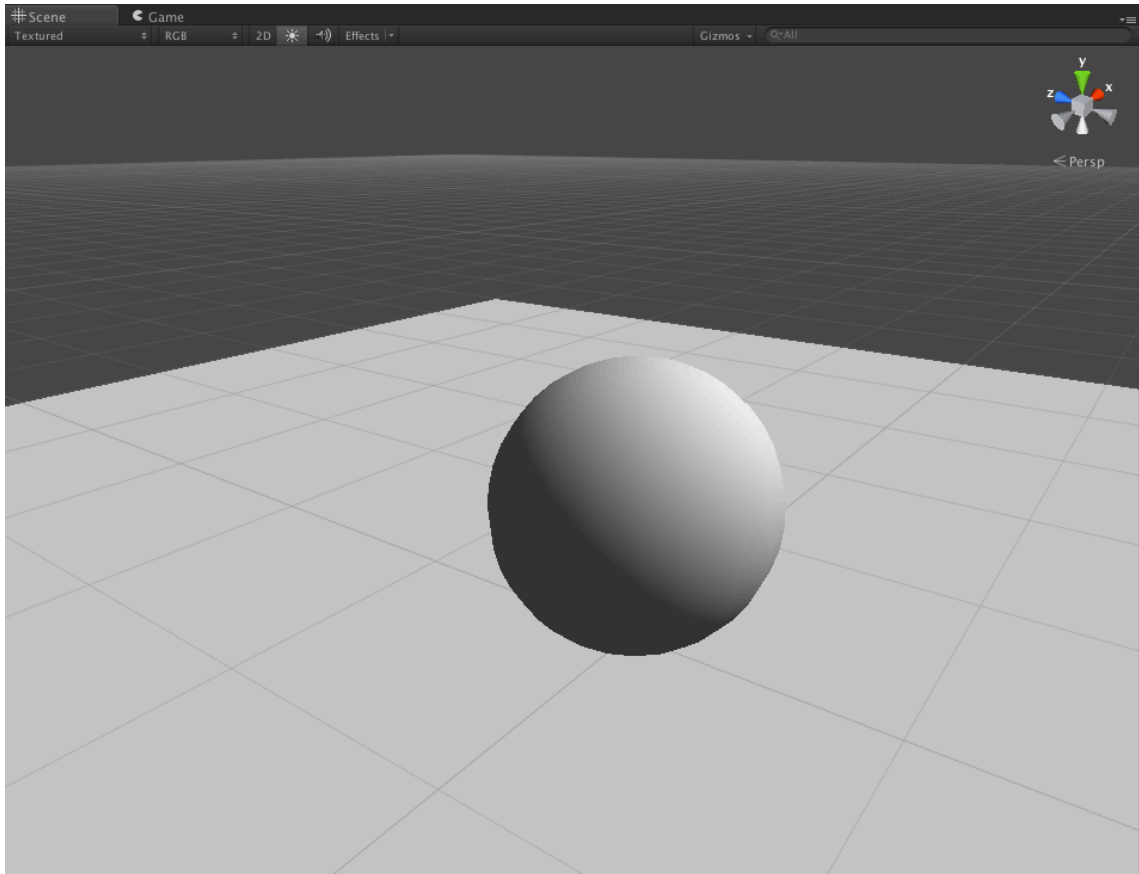
    fixed4 c;
    c.rgb = s.Albedo * _LightColor.rgb * (diff * atten * 2);
    c.a = s.Alpha;
    return c;
}
```

从上面可以看出，Lambert 模型只考虑光的方向和点的法线这些数值。如果我们继续用这个模型，就不能体现我们的高光随人眼（相机）位置变化的特征。

理论基本到这里，现在我们开始着手制作我们的高光效果

- 创建一个新的 project
- 创建一个新的 shader
- 基于该 shaer 创建一个新的 material
- 创建一个新的 sphere
- 在场景中如果没有 light 的话，加入一个 directional light

如果没有问题，我们的场景大致现在是这样



- 改动 shader 的 Properties 块

Properties

```
{
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _MainTint("Diffuse Tint", Color) = (1, 1, 1, 1)
    _Specular ("Specular Color", Color) = (1, 1, 1, 1)
    _SpecularPower ("Specular Power", Range(0, 1)) = 0.5
}
```

- 在变量声明处改成如下以对应到我们的 Properties

```
sampler2D _MainTex;
float4 _MainTint;
float4 _Specular;
float _SpecularPower;
```

- 我们需要让 Unity 使用非 Lambert 的光照模型，这里自己写一个，放在变量声明后

```
inline fixed4 LightingSpecular (SurfaceOutput s, fixed3 lightDir, half3 viewDir, fixed atten)
{
    half3 h = normalize (lightDir + viewDir);

    fixed diff = max (0, dot (s.Normal, lightDir));

    float nh = max (0, dot (s.Normal, h));
    float spec = pow (nh, s.Specular*128.0);

    fixed4 c;
    c.rgb = (s.Albedo * _LightColor.rgb * diff + _LightColor.rgb * _Specular.rgb * spec) * (atten * 2);
    c.a = s.Alpha + _LightColor.a * _Specular.a * spec * atten;
    return c;
}
```

- 改动#pragma 来使用我们的光照模型

```
#pragma surface surf Specular
```

- 最后把 surf()方法代码改成如下

```
void surf (Input IN, inout SurfaceOutput o)
{
    half4 c = tex2D (_MainTex, IN.uv_MainTex) * _MainTint;

    o.Specular = _SpecularPower;
    o.Albedo = c.rgb;
    o.Alpha = c.a;
}
```

- 保存后，回到 Unity，编译结束后我们把材质拖到物体上，观察效果。为保证你的 shader 代码跟我的一样，这里是我的 shader 代码

```
Shader "Custom/Specular"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _MainTint("Diffuse Tint", Color) = (1, 1, 1, 1)
        _Specular ("Specular Color", Color) = (1, 1, 1, 1)
        _SpecularPower ("Specular Power", Range(0, 1)) = 0.5
    }

    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Specular

        sampler2D _MainTex;
        float4 _MainTint;
        float4 _Specular;
        float _SpecularPower;

        inline fixed4 LightingSpecular (SurfaceOutput s, fixed3 lightDir, half3 view
Dir, fixed atten)
        {
            half3 h = normalize (lightDir + viewDir);

            fixed diff = max (0, dot (s.Normal, lightDir));

            float nh = max (0, dot (s.Normal, h));
            float spec = pow (nh, s.Specular*128.0);

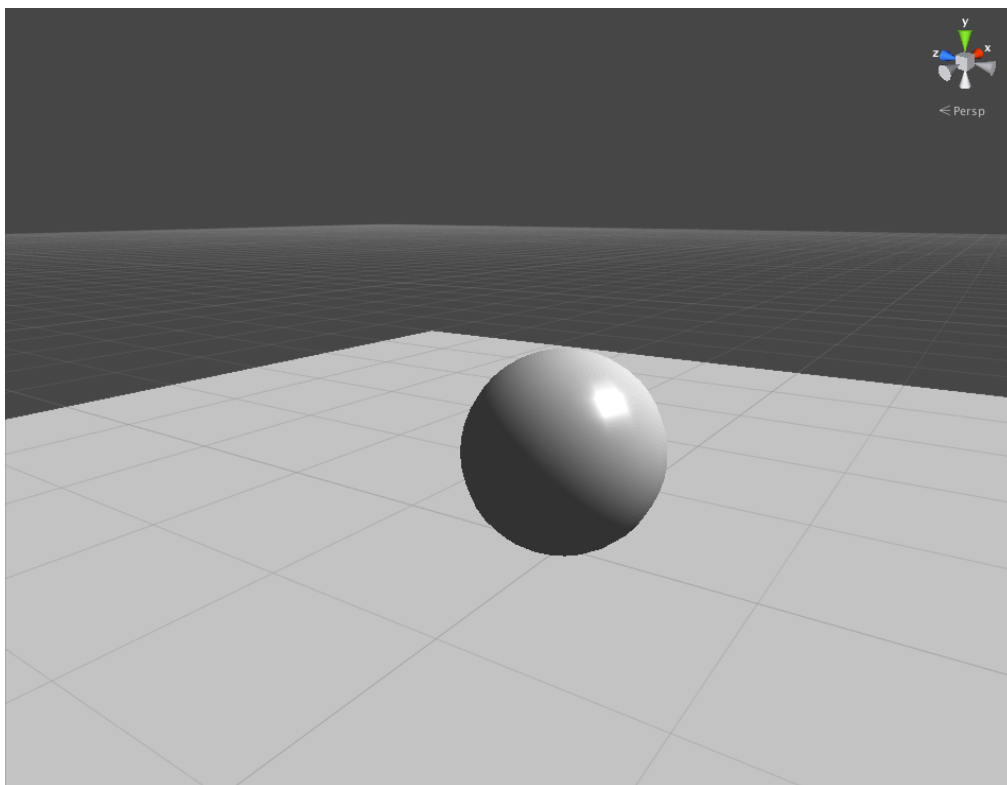
            fixed4 c;
            c.rgb = (s.Albedo * _LightColoro.rgb * diff + _LightColoro.rgb * _Spec
ular.rgb * spec) * (atten * 2);
            c.a = s.Alpha + _LightColoro.a * _Specular.a * spec * atten;
            return c;
        }

        struct Input
        {
            float2 uv_MainTex;
        }
    }
}
```

```
};

void surf (Input IN, inout SurfaceOutput o)
{
    half4 c = tex2D (_MainTex, IN.uv_MainTex) * _MainTint;

    o.Specular = _SpecularPower;
    o.Albedo = c.rgb;
    o.Alpha = c.a;
}
ENDCG
}
Fallback "Diffuse"
}
```



图中已经可以看到高光效果出来了，我们可以在场景中移动自己的观察角度，可以发现随角度的不同，原来亮的地方会变成非高光效果，同时原来没有高光效果的地方，会呈现出高光来。这个时候你再回头去看一下刚才开篇的理论，就能更透彻的明白高光要实现的是什么。

在上述的代码中,我们首先使用了给 `o.specular` 这个变量设置了我们的 `specularPower` 这个用户可以调整的变量值;然后光照模型中参数使用了相机的方向(`viewDir`,使用这种类型的函数原型的时候,unity会自动认出我们需要的是相机方向,因此会从外面把相机方向作为参数传进 shader)。光照模型的第一行代码是经典的 BlinnPhong 模型算法,通过相机方向和入射光方向算出一个“中间方向”(向量的加法符合平行四边形原则,可以得到另外一个向量),最后通过这个中间方向跟点的法线的夹角来决定我们的相机看到的光的强度。

上述代码实现的是 BlinnPhong 的模型,这个模型由微软研究院的 Blinn 在 Phong 的模型上改进,速度上比 Phong 的模型快很多,实现效果上跟 Phong 有一点差异。我们可以改动光照模型的代码成为 Phong 的如下

```
inline fixed4 LightingSpecular (SurfaceOutput s, fixed3 lightDir, half3 viewDir, fixed atten)
{
    float diff = dot(s.Normal, lightDir);
    float3 reflectionVector = normalize(2.0 * s.Normal * diff - lightDir);

    float spec = pow(max(0, dot(reflectionVector, viewDir)), s.Specular);
    float3 finalSpec = _Specular.rgb * spec;

    fixed4 c;
    c.rgb = (s.Albedo * _LightColoro.rgb * diff) + (_LightColoro.rgb * finalSpec);
    c.a = 1.0;
    return c;
}
```

自己可以在编辑器中观察，基本上，Phong 的算法出来的效果会更加平滑，而 BlinnPhong 的模型出来的效果会有棱角。但是，BlinnPhong 会快很多。

环境映射和 Cubemap

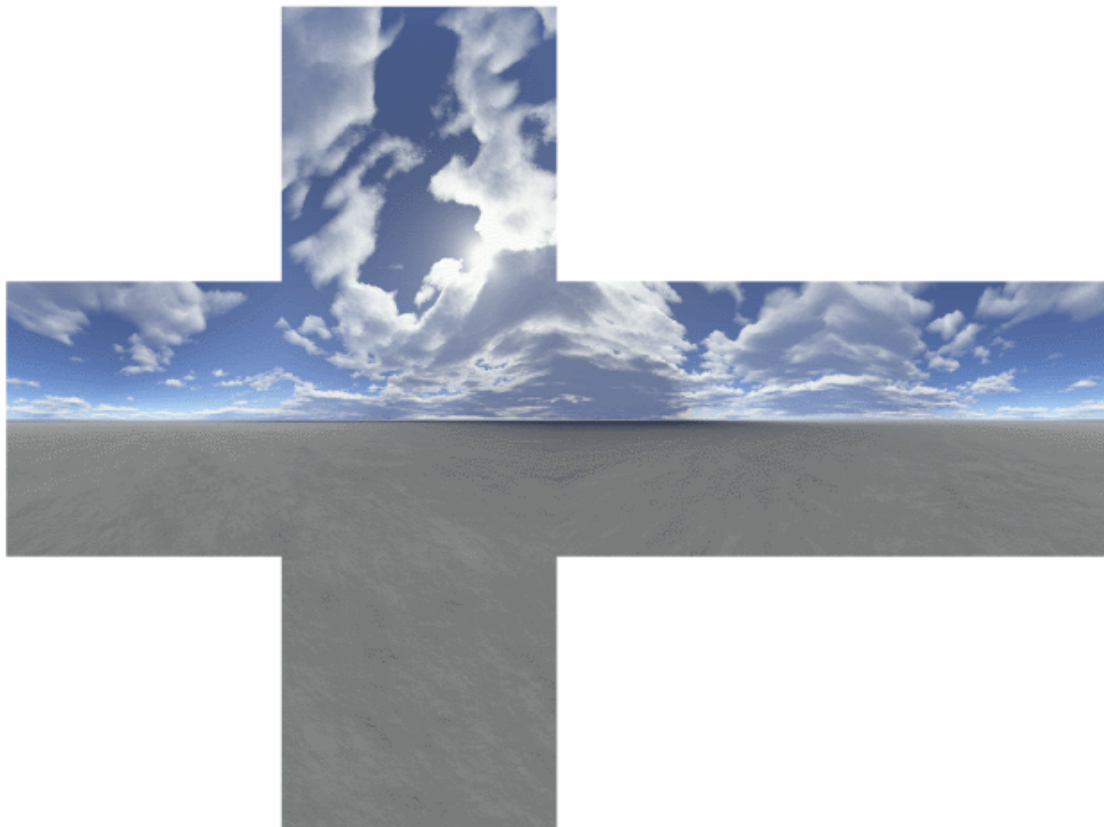
对于金属状、玻璃状物体，它们的表面能映射出周围环境。如果碰上这类的物体，用上我们今天课程的 shader，就能让这种物体显示更加的逼真。这类 shader 能模拟反射物体周围的环境。这种方式大量的使用在游戏中的窗玻璃、汽车外壳等上面，给对应的物体添加了非常形象的效果。

同一个物体，不应用环境映射和应用后的效果对比（第一张图没有用环境映射）：





在 Unity 中，我们使用 Cubemap 来帮助实现这种效果。Cubemap 在本质上也是一张贴图，



上图中表现了一个“相机”固定在某个位置，然后通过转动它本身的镜头到前、后、左、右、上、下拍下的六个画面，这六个画面按照一定的位置放在一张图里面，就形成我们需要的 cubemap 贴图。

使用 Unity 生成 Cubemap

- 建立一个场景，在场景里放入多个物体，尽量放置不同的物体形状，并且保持数量和大小能保证我们的相机在某个点拍摄的时候，六个方向上都有东西拍到。如果有真实的场景，那就最好。
- 在 Project 面板中创建一个 Editor 文件夹，因为我们需要在这里放置一个代码文件来做 cubemap 生成
- 在 Editor 文件夹中创建一个 c#脚本，名字为 GenerateCubemap
- 拷贝下面的代码到脚本中替换原来所有的内容

```
using UnityEngine;
using UnityEditor;
using System.Collections;

public class GenerateCubemap : ScriptableWizard
{
    public Transform renderFromPosition;
    public Cubemap cubemap;

    void OnWizardUpdate()
    {
        helpString = "Select transform to render from and cubemap to render into";
        isValid = (renderFromPosition != null) && (cubemap != null);
    }
}
```

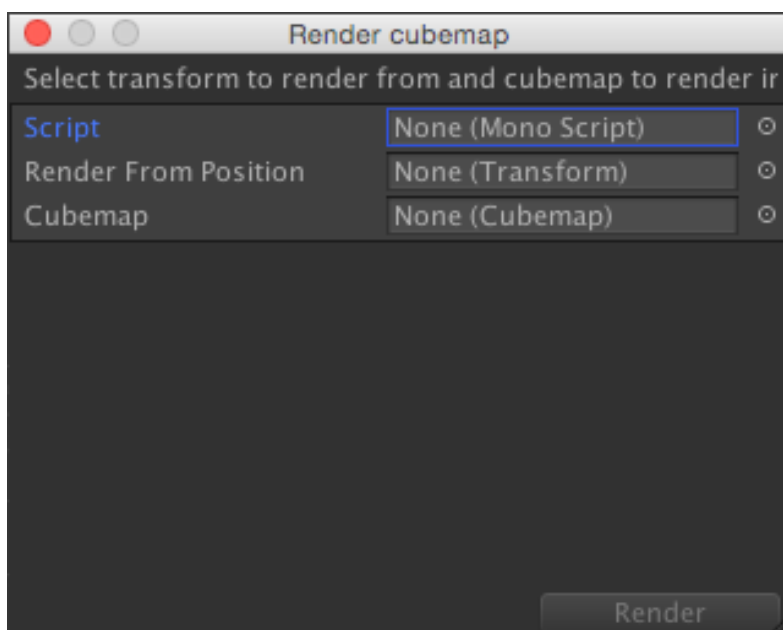
```
void OnWizardCreate()
{
    GameObject go = new GameObject("Cubemap Camera", typeof(Camera));
    go.transform.position = renderFromPosition.position;
    go.transform.rotation = Quaternion.identity;

    go.camera.RenderToCubemap(cubemap);

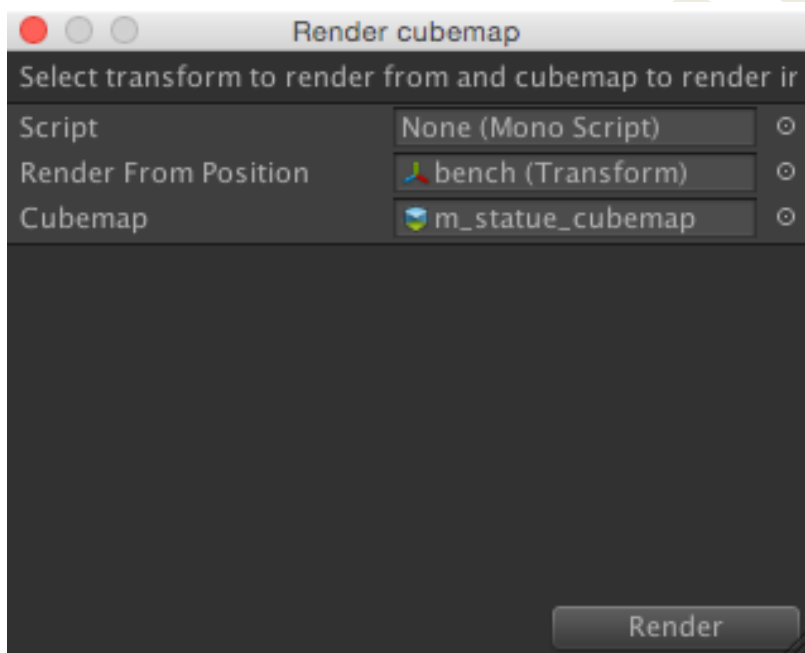
    DestroyImmediate(go);
}

[MenuItem("Tool/Render into Cubemap")]
static void RenderCubemap()
{
    ScriptableWizard.DisplayWizard("Render cubemap", typeof(GenerateCubemap), "Render");
}
}
```

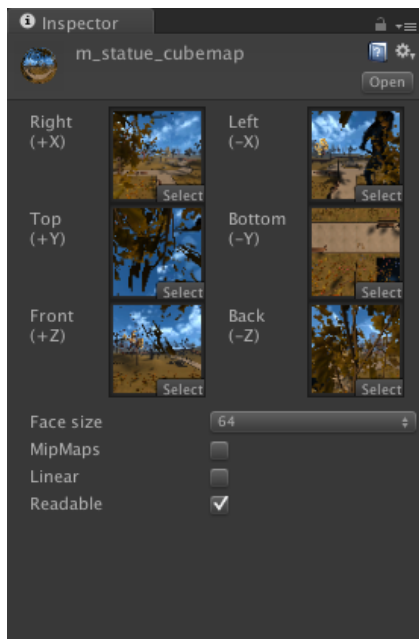
- 保存后回到 Unity。等待一小会，如果没有编译错误。我们应该在 Unity 的主菜单上看到多了一个 Tool，点开会出现下拉菜单“Render into Cubemap”。【注：如果没有错误，但是等好久也看不到这个菜单，请关闭 Unity 再打开】
- 点击该菜单，出现如下对话框



- 第一栏 Script 我们就照原样不动
- 第二栏我们从场景中拖入一个物体进去，这个物体的位置代表拍照相机的位置，所以，如果场景中的物体位置都不符合你的预期，你可以通过在场景中创建一个临时的物体，比如，sphere，来作为这个 Transform
- 第三栏的 cubemap，我们要在 Project 面板上的 Create 下拉菜单中找到 cubemap，然后点击后创建一个 cubemap 文件出来【注：5.0 Cubemap 的创建应该在 legacy 中】。然后把这个文件往这个栏位拖入
- 如果没有问题，这个时候我们的 Render 按钮应该亮起来了



- 点击这个按钮。然后去 Project 面板查看 cubemap 的文件，对应的 Inspector 上应该会出现类似这样的画面



- 图中的 6 个图片随场景不同而不同，表达了刚才相机在的位置从 6 个方向拍摄到的场景。这样我们的 cubemap 就制作好了。

然后我们写一个使用 Cubemap 的 shader 来应用环境映射

刚才我们做的事情，只是把物体周围的环境通过快照取了下来，并且存在了一张所谓的 Cubemap 贴图里面。现在我们来制作一个 shader 来支持把环境“映射”到物体的表面上。

- 创建一个新的 shader "Envmap"，擦去里面的原有内容
- 把下面的代码拷贝到这个文件中

```
Shader "myEnvmap"
{
    Properties
    {
        _MainTex("Base texture", 2D) = "white"
```

```
        _Cubemap("Env map", CUBE) = "" {}
    }

    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert

        sampler2D _MainTex;
        samplerCUBE _Cubemap;

        struct Input
        {
            float2 uv_MainTex;
            float3 worldRefl;
        };

        void surf(Input IN, inout SurfaceOutput o)
        {
            fixed4 c = tex2D(_MainTex, IN.uv_MainTex);
            o.Albedo = c.rgb;

            fixed4 reflcol = texCUBE (_Cubemap, IN.worldRefl);
            reflcol *= c.a;
            o.Emission = reflcol.rgb;
            o.Alpha = reflcol.a;
        }
        ENDCG
    }
}
```

- 保存后，回到 Unity 中。
- 给我们的 shader 创建一个 material，在 material 的 Inspector 面板上，就可以看到供选择 Cubemap 的控件

- 在 Cubemap 控件中拖入我们的 cubemap，这个时候不放置物体本来的贴图，已经可以看到环境映射在物体上的效果

这个 shader 代码，主要是用到了 texCUBE 这个 Unity 内置的方法，这个方法第一个参数是我们的 cubemap 文件，第二个参数是 unity 内置的基于世界坐标系的反射方向，在 Unity 渲染物体的时候，如果我们有 Input 结构需要这个东西，只要在 Input 结构中写上 worldRefl 就可以，unity 会帮助把物体某个点的反射方向作为参数送入 shader，我们只要使用就可以。

texCUBE 取出的值，我们把它放置到 SurfaceOutput 的 Emission 变量里面，出了 surf()这个方法后，光照方程会被运行，Emission 的值会在此过程保持，最后出了光照方程后（Unity 在光照方程中主要计算某个点的光照颜色），Unity 会把 Emission 的颜色做加法加到顶点的颜色上，合成最后的输出到显卡缓存的色彩，也就是我们最后看到的颜色（这里主要回答一些同学提出的为什么在 Lighting.cginc 中看不到 Emission 被使用的问题）。

透明、Alpha 混合(alpha blend)

这节课我们开始学习 shader 中透明的使用，透明这个性质在渲染的时候是很多材质，如玻璃、头发或者树叶，必须要用到的。

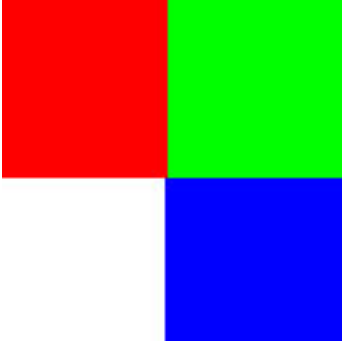


首先我们来创建一个最简单的使用透明(transparent)的 shader，在这个过程中建立起透明在物体渲染中的重要性。于此同时，在这个过程中，我们会了解在 Unity 的 surface shader 中，控制透明的几个关键点。

开始试验

- 创建一个新的工程
- 在场景中放入一个 sphere、一个 plane、一个 directional light
- 创建一个新的 shader 和一个新的 material，并且让 material 用上我们的新的 shader

- 把下面的图存到你的工程里面，名字为 rgb（右键，图像另存为，到你本地，然后复制进 unity 工程的 assets 目录）



- 把原来的 shader 代码清除，然后加入下面的代码

```
Shader "Custom/Transparent"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }

    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert alpha
        sampler2D _MainTex;

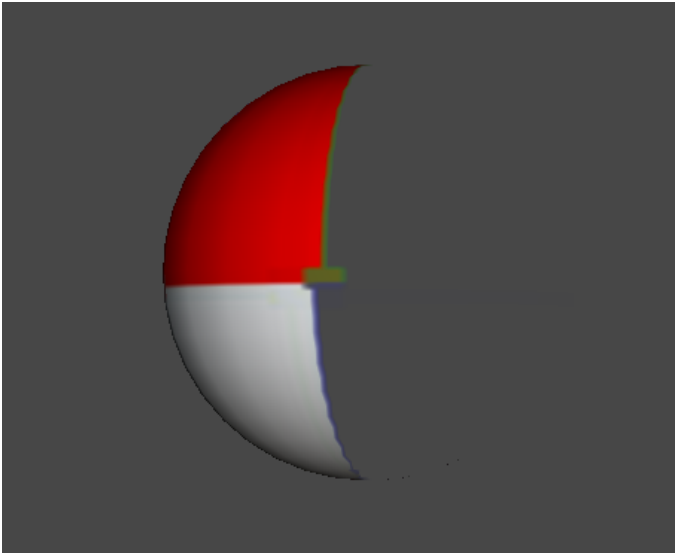
        struct Input
        {
            float2 uv_MainTex;
        };

        void surf (Input IN, inout SurfaceOutput o)
        {
            half4 c = tex2D (_MainTex, IN.uv_MainTex);
```



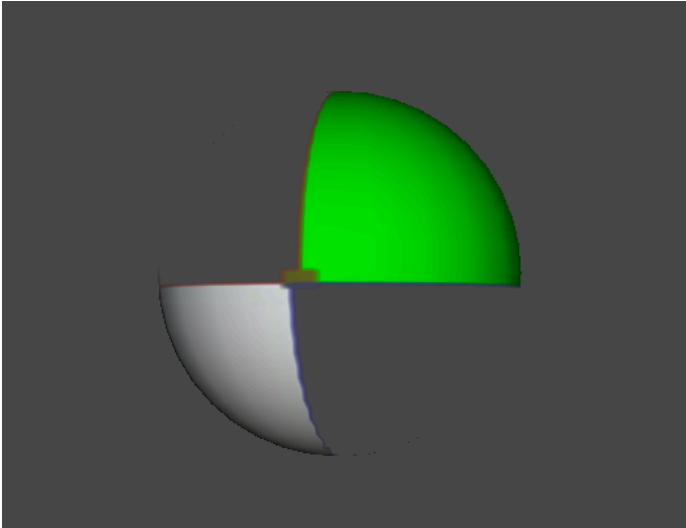
```
        o.Albedo = c.rgb;
        o.Alpha = c.b;
    }
    ENDCG
}
Fallback "Diffuse"
}
```

- 保存后回到 unity，把 material 加到我们的 sphere 上，然后选中 material，Inspector 面板上拖入 rgb 这个图
- 如果没有任何问题，这个时候我们的球应该呈现如图



从图中来看，绿色部分和蓝色部分包住的球体没有显示出来，也就是变成“透明”了。

- 改动 shader 中的代码的最后一行，把 "c.r" 改成 "c.g"，则截图如下



- 你可以尝试改动最后一行到 c.b，自行观察结果，这里不截图了。

流程分析

这次我们的`#pragma`一行跟原来的有一些差异，多了最后的一个 `alpha` 字眼。在以前的课程中，我们已经知道这个后面可以跟光照模型的信息。然后，这里是它的完整原型

```
#pragma surface surfaceFunction lightModel [optionalparams]
```

上述格式中，`#pragma` 和 `surface` 是关键词，不能更改。`surfaceFunction` 用你自己给 `surface` 方法起的名字代替，比如我们一直用的 `surf`。`lightModel` 用内置的 `Lambert` 或者 `BlinnPhong` 或者自定义的，自定义的方式在前面的课程已经讲过。

`[optionalparams]`这里用[]括起来，表明这个字段是可选的，只在特定的情况下才需要写。我们这里用到了 `alpha` 这个字段。`alpha` 字段用在这里指示 Unity 编译器我们的 `shader` 需要启用 `alpha` 混合(`alpha blend`)的功能，这样 `unity` 引擎会为我们打开这个开关。这样我们的 `shader`

输出的颜色会跟原来的显示缓冲的颜色进行一个比原来的 shader 多出来的步骤，利用颜色中的 α (alpha)值，进行混合。

在我们原来的 shader 中，很多时候，我们在 surf()方法中，会给 o.Alpha 赋上值。但是实际上，没有打开 alpha 混合，那么这个 alpha 对后续的色彩根本就不起作用，会被 GPU 忽略。

因此，这也是用透明的第一个要点：

用上 `#pragma xxx alpha`

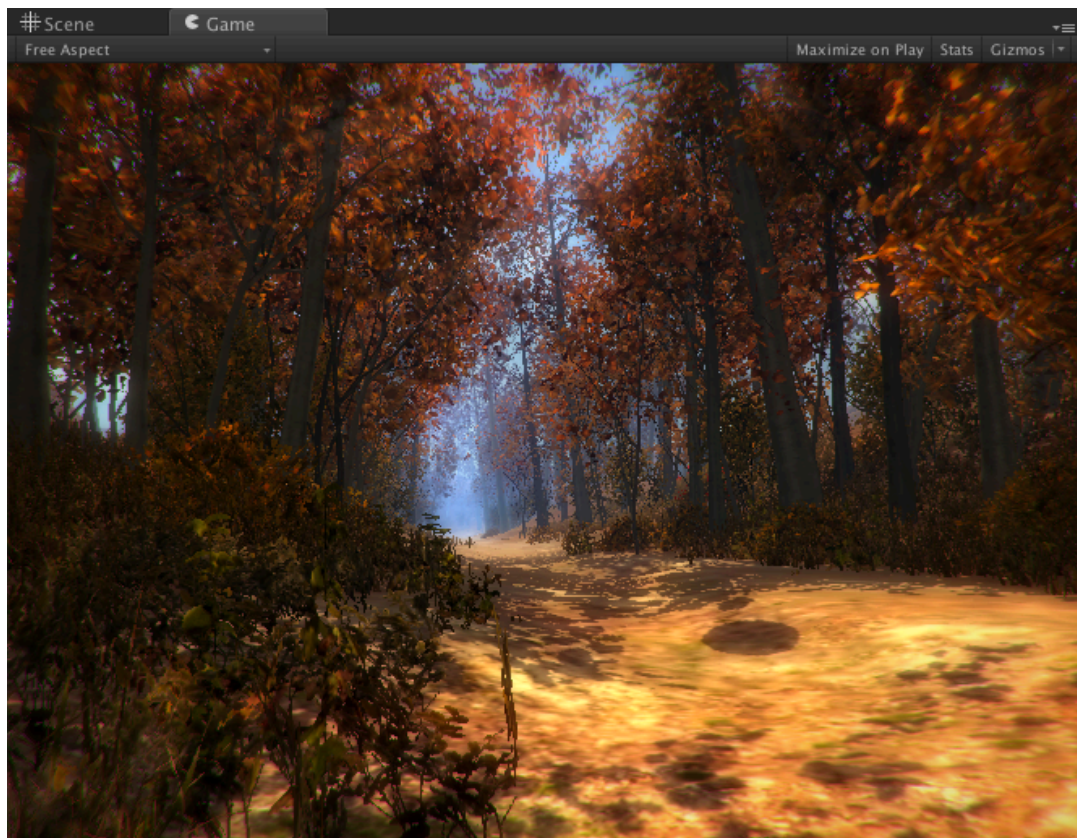
在我们上面的两个图对应的 surface shader 中，我们分别对 surf()方法的输出用了 `o.Alpha = c.r` 和 `o.Alpha = c.b`。c 是贴图中的每个像素的值，我们制作的测试图，基本上分为 4 个区域（红色部分、蓝色部分、白色部分、绿色部分），红色对应的每个像素取出后，对应到(1, 0, 0)。蓝色对应到(0, 0, 1)，绿色对应到(0, 1, 0)，白色是(1, 1, 1)。让我们第一次用 `o.Alpha = c.r` 的时候，surface shader 最后丢给 Alpha 混合阶段的值，对应到四个部分，除了原本红色和白色的区域是 1 以外，其他的两个部分都是 0。结果混合后，我们就只能看到红色和白色部分。换成 `o.Alpha = c.b` 后，除了蓝色和白色部分，其他的两个部分对应的颜色输出 Alpha 值都是 0，所以我们只看到蓝色和白色部分。如果我们把代码改成 `o.Alpha = 0`；那么我们整个 sphere 都会消失。这就是使用透明的第二个要点：

surface shader 的 alpha 值一定要放置好正确的值。0 会让相应的 texel(对应于贴图中的每个像素，渲染到显示缓存的每个最终单位叫做 texel)消失。

如果我们把现在的 shader 的 o.Alpha 值改成 0.5，在理论上我们应该看到我们的 sphere 变成半透明的。所以现在开始直接改动代码

```
o.Alpha = 0.5;
```

保存后我们看结果，为了看出 sphere 的透明效果，我这里把 sphere 放在了一个真实的场景中，这里是出来的截图



上图的路中间的地面阴影可以猜出 sphere 的位置，但是我们的 sphere 本身似乎消失了，或者你在自己的 Unity 中可以移动相机四出查看，你可能会发现 sphere 会若有若无。这个是什么呢？

原因在于使用透明的第三个要点：

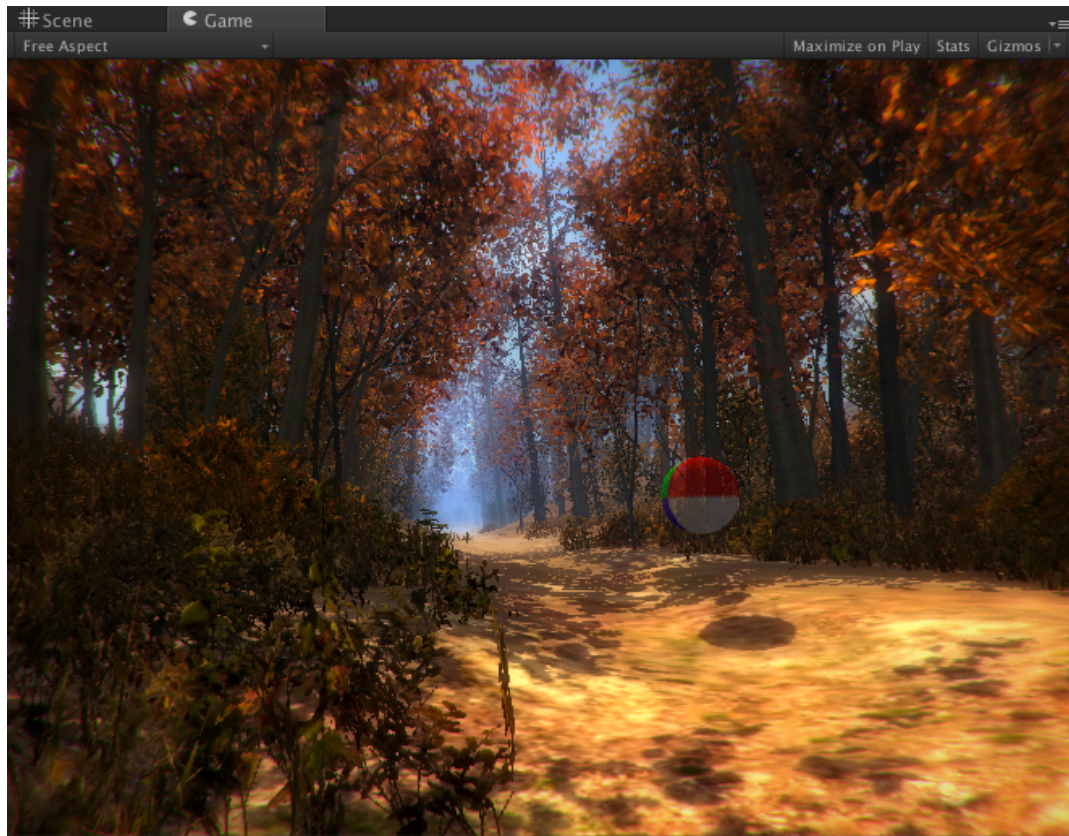
一定让透明的物体在不透明物体后面渲染

透明物体的渲染背后是：shader 每个输出的 texel 都会跟原来在显示缓存里的 texel 做 alpha 混合。因此，我们可以想象一下，如果是透明物体先绘制，那么这个时候显示缓存里还没有存在本来应该存在的非透明物体的 texcel 值，那么该 texel 最后的结果就是一个错误的值，因为它混合了不该混合的颜色。

Unity surface shader 用 SubShader 中的 Tags{} 块的某个属性让我们控制物体的渲染顺序，这个属性叫做"Queue"。要修正我们刚才的问题，我们只要这么把原先的 Tags{} 块改成如下：

```
Tags { "Queue" = "Transparent" }
```

然后我们就可以看到透明的 sphere 在场景中的模样



上图用的 alpha 值是 0.5，因此我们可以透过 sphere 看到后面的树干。

Alpha 测试 (alpha test)

前面的课程里讲到了 Alpha 混合, 这个过程会让 texcel 写入缓存的时候跟原来缓存中的数据做一定的加减乘除等运算, 用来表现物体的透明度。它的指令大致如下:

```
#pragma surface surf Lambert alpha
```

这个编译器指令来打开 alpha 混合。另外一个相似的指令是 alphatest, 举例

```
#pragma surface surf Lambert alphatest:_value
```

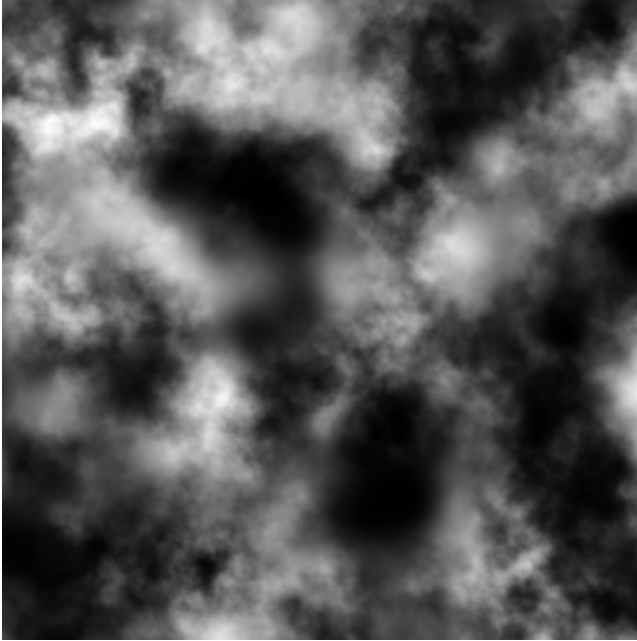
alphatest 是一个关键字(keyword), 后面跟上冒号(:), 冒号后面跟的是一个数值。这也是开关命令, 当存在的时候, Unity 知道我们这个 shader 需要 alphatest 这个过程, 就会帮助打开 alphatest 这个环节(这个环节在我们刚学的 alpha 混合环节前面一步, 如果忘记, 可以在“3D 管线章节读到相关的内容”)

alphatest 开关如果打开, GPU 会对 shader 输出的每个单位(texel)的 alpha 值进行判别, 只有符合条件的才会被送入下面的环节, 不符合条件的就直接抛弃, 不再参与整个流程。在这里的命令里面, Unity 是实现所谓的 cutoff 效果, 所以这个条件直白的说就是“小于_value 的值都会被抛弃”。

动手实验

- 新建工程

- 场景里加入一个 sphere
- Projects 里面加入 material、shader，并且把 material 和 shader 关联
- 保存下面的这张图到工程



- 拷贝下面的代码到我们的 shader 中替代原来的内容

```
Shader "Custom/Alphatest"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _Cutoff("Value", Range(0, 1)) = 0.5
    }

    SubShader
    {
        CGPROGRAM
        #pragma surface surf Lambert alphatest:_Cutoff

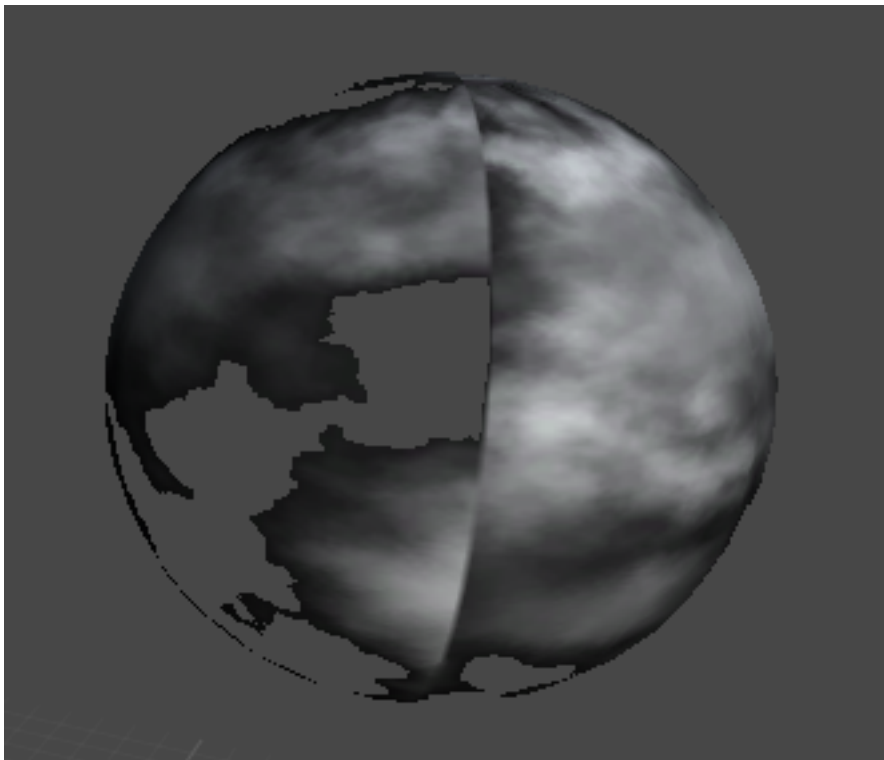
        sampler2D _MainTex;

        struct Input
```



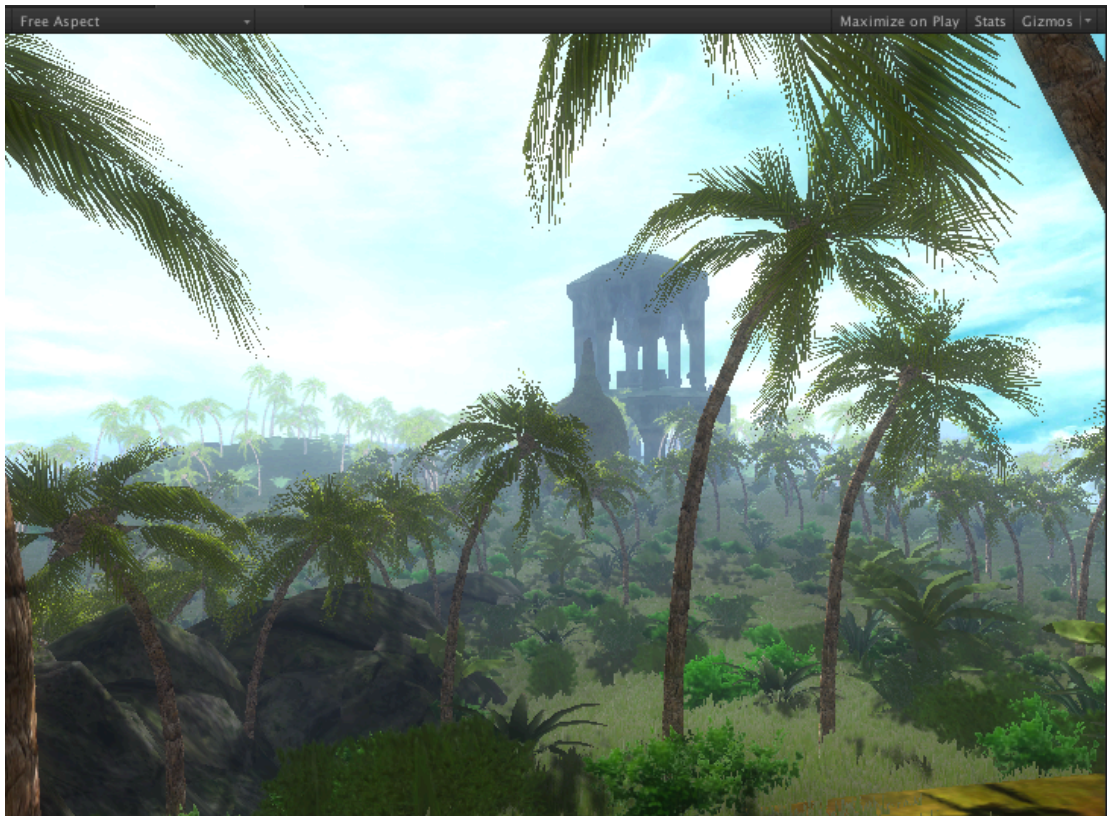
```
{  
    float2 uv_MainTex;  
};  
  
void surf (Input IN, inout SurfaceOutput o)  
{  
    half4 c = tex2D(_MainTex, IN.uv_MainTex);  
    o.Albedo = c.rgb;  
    o.Alpha = c.r;  
}  
ENDCG  
}  
Fallback "Diffuse"  
}
```

- 保存后回到 unity
- 让 sphere 用上我们的 material
- 选中 material，拉动它的_Cutoff 控件观察效果。
- 下面的图是我拉动控件后的某一张，随_Cutoff 的不同，这个图“破损”的范围也不同



这份代码中开启了 alphatest 开关，并且在 surf()方法中用用户给出的图的红色值作为 surface shader 输出的 Alpha 值。因此，如果图中的红色值（R 通道）小于 _Cutoff 值，那么相关的 texel 的生命就在整个流程中终止了，不会再参与到后面的流程中。所以形成图中的“残破”现象。

alphatest 常用在树叶、毛发的渲染场合，而且经常跟 alpha 混合一起使用达到比较完美的输出。前者用于先取出 alpha 值非常小的部分，后者用来实现边缘的半透明化。



顶点方法 (Vertex Function) 和顶点色

前面的课程中，我们看到的 surface shader 里面都只有一个用 #pragma 指定的方法，该方法让我们把所谓的 surface 信息按照一定的方式填好，然后交给 unity 做光照处理，最后根据我们对整个流程的控制（比如控制 Alphatest、Alpha 混合）处理合成最后的画面。

这种 Unity 创造的 shader 格式，让写 shader 的人一旦熟悉，写个 shader 就非常的方便，不必要像其他如 HLSL、GLSL、cg 一样，每个 shader 都得写很多的代码。你只要告诉 Unity“物体的表面（surface 一词的来历）信息”就行了。

在方便的同时，Unity 也注重到了功能的强大。今天我们的课程就来初步探索它对顶点方法的控制，有了这个特性后，我们能在 shader 里面得到顶点的坐标，据此可以改动顶点的属性，实现特定的功能：资源比较紧张的硬件上，可以使用这种方式用很少量的贴图，得到不同的效果。比如下图中一个早期国外游戏截图：



泥土部分和草地部分用的是同样的贴图（可以从纹理上看出来），用上顶点色后就能在感官上给人不同的地貌。

动手实验

- 新建立一个工程
- 新建一个 material
- 新建一个 shader
- 把 material 跟 shader 关联起来
- shader 里面拷贝以下代码代替原来的内容

```
Shader "Custom/grassandmud"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }

    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert vertex:vert

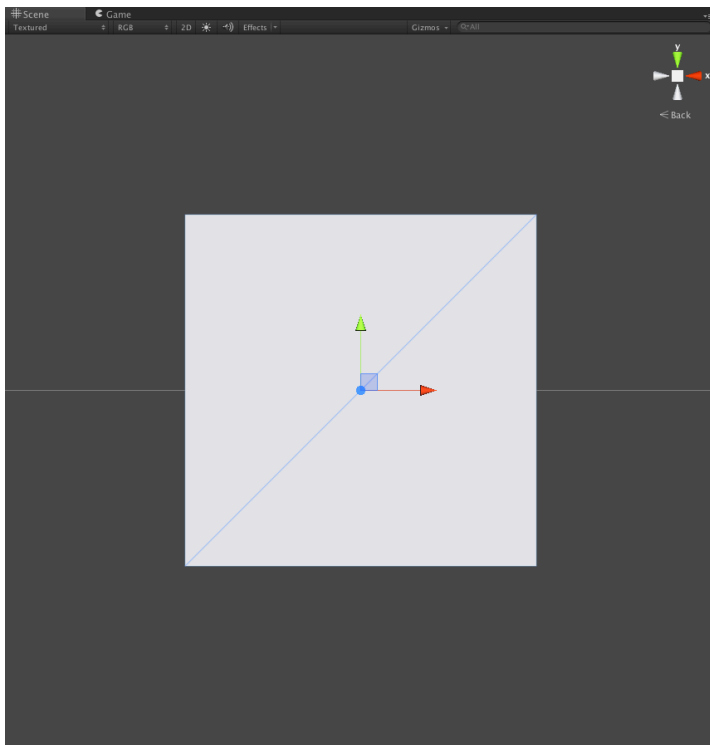
        sampler2D _MainTex;

        struct Input
        {
            float2 uv_MainTex;
            float4 vertColor;
        };

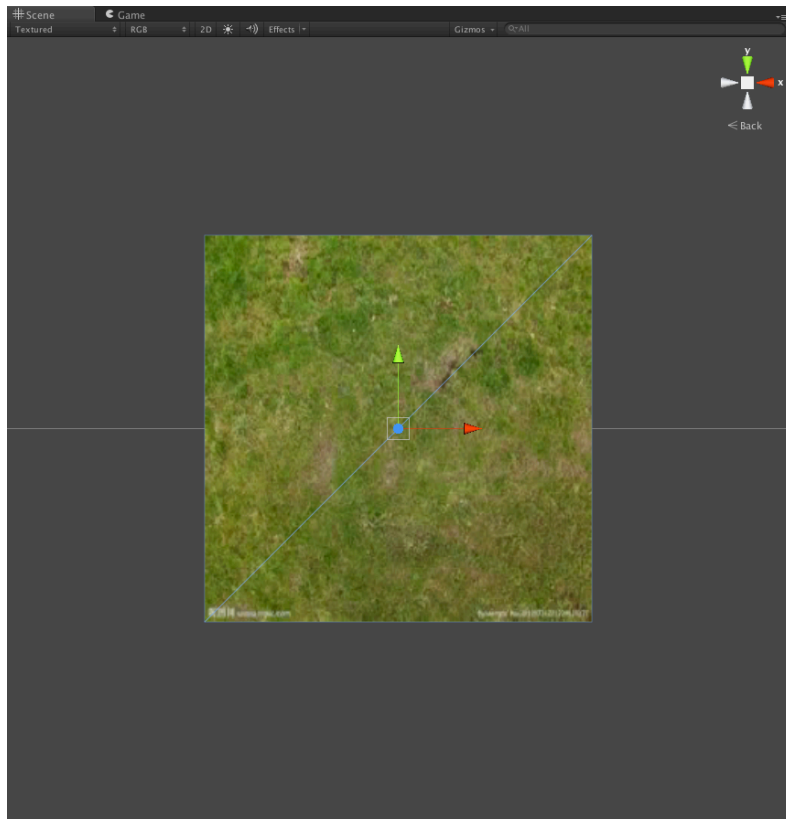
        void vert(inout appdata_full v, out Input o)
        {
            UNITY_INITIALIZE_OUTPUT(Input,o);
            o.vertColor = v.color;
        }
    }
}
```

```
void surf (Input IN, inout SurfaceOutput o)
{
    o.Albedo = IN.vertColor.rgb * tex2D(_MainTex, IN.uv_MainTex).rgb;
}
ENDCG
}
Fallback "Diffuse"
}
```

- 新建一个 Quad。这个物件它内部含有 4 个顶点，组成两个三角形，供我们在下面动态的改动它的顶点属性。



- 找一张贴图，放在我们的 Quad 上能看起来像草地的，放到我们的工程中。
- 把 material 中的贴图换成上面的。这个时候 Quad 大致看起来是这样



- 创建一个 C#脚本文件
- 把下面的内容覆盖原来的代码

```
using UnityEngine;
using System.Collections;

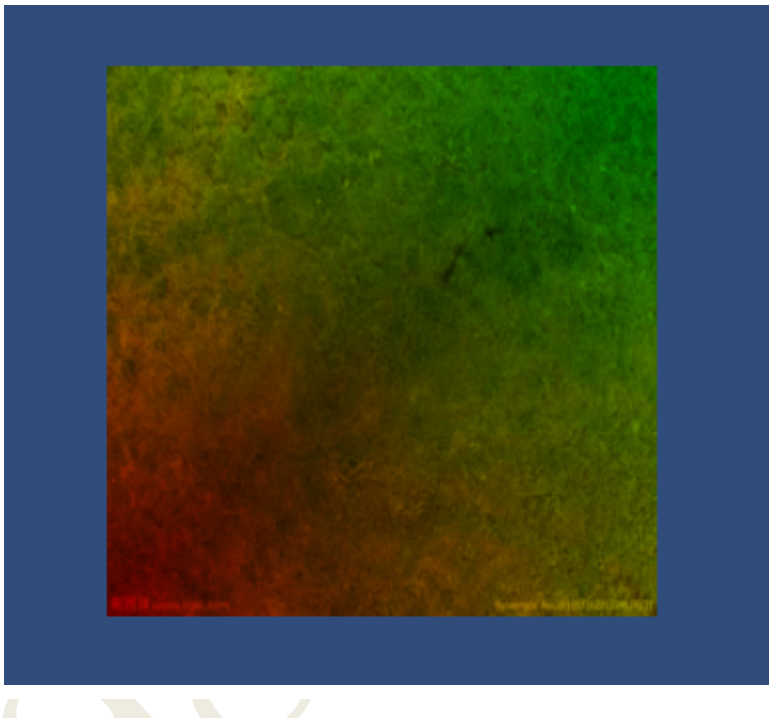
public class MeshColor : MonoBehaviour
{
    void Start ()
    {
        Mesh mesh = GetComponent<MeshFilter> ().mesh;
        Vector3[] vertices = mesh.vertices;
        Color[] colors = new Color[vertices.Length];

        colors [0] = Color.red;
        colors [1] = Color.green;
        colors [2] = new Color(199, 195, 0);
        colors [3] = new Color(199, 195, 0);
```

```
        mesh.colors = colors;
    }

}
```

- 保存该代码, 并且把该代码作为 Quad 对象的组件 (从 project 拖到 Quad 的 Inspector 上)
- 点击 Unity 的 Play 按钮查看结果



上面的的 shader 代码中, 我们用到这样一行

```
#pragma surface surf Lambert vertex:vert
```

在光照模型方法后我们加了 `vertex:vert`, 其中 `vertex` 是关键词, 后面跟冒号, 再跟一个我们定义的名字, 对应到下面的一个方法。这个用来告诉 Unity 编译我们的 shader 的时候, 我们

需要对顶点做一些相关的操作，所以请它在编译的时候，把我们指定的顶点方法编译，以便让 Unity 引擎执行该 shader 程序的时候用上我们的顶点方法。

再来看指定的这个方法的内容

```
void vert(inout appdata_full v, out Input o)
{
    UNITY_INITIALIZE_OUTPUT(Input,o);
    o.vertColor = v.color;
}
```

第一个参数的类型是 appdata_full，它的定义在 UnityCG.cginc 中可以看到，为了方便，我把内容拷贝在下面

```
struct appdata_full
{
    float4 vertex : POSITION;
    float4 tangent : TANGENT;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
    float4 texcoord1 : TEXCOORD1;
    fixed4 color : COLOR;
#ifdef SHADER_API_XBOX360
    half4 texcoord2 : TEXCOORD2;
    half4 texcoord3 : TEXCOORD3;
    half4 texcoord4 : TEXCOORD4;
    half4 texcoord5 : TEXCOORD5;
#endif
};
```

这个是从应用程序往 shader 中送入的数据的结构，如位置(vertex)、法线(normal)，加上我们这节课在介绍的顶点色(color)。

第二个参数类型是 Input，是我们定义的一个结构

```
struct Input
{
    float2 uv_MainTex;
    float4 vertColor;
};
```

与我们经常用的 Input 结构不同的地方在于：

- 1、我们加了一个 float4 类型的变量定义
- 2、这个结构在 vert 方法中用了 out 修饰，表示 vert 方法输出的格式会基于这个结构。

为什么需要这么一个结构？在 vert 方法中用作输出，在 surf 方法中用作输入，这里涉及到渲染的流程（属于叫渲染管线）：shader 总是先对每个顶点进行计算（这里如果我们用顶点方法，那么我们的顶点方法就会被启用）；顶点计算出后按照一定的格式输出给下面的流程，到某一步就是我们放置的 surf 方法，它接收的输入可以进行一定的控制（我们这里用了顶点方法的输出）。

对于我们上面的一个实验来说，整个 shader 相关流程如下：

- 1、shader 准备对一个顶点进行计算。因我们自定义顶点方法，因而 vert 被调用
- 2、vert 的第一参数决定该顶点需要带进来哪些属性，我们得到这个属性里的顶点色，把这个顶点色放入到 Input 结构中，以供后续的 surf 使用；

3、我们在这里就知道了原先顶点的颜色，然后我们可以把顶点的颜色跟贴图的颜色合成，形成一种新的视觉效果。

对于顶点的颜色这个属性，大多数的 3D 建模软件，比如 3ds max，都能在建模的时候指定。

不过我们在这个例子里面，因为用到的是 Unity 本身的四变形体，所以通过了代码来指定每个顶点的颜色，相关的代码是 C#里面的

```
void Start ()
{

    Mesh mesh = GetComponent<MeshFilter> ().mesh;
    Vector3[] vertices = mesh.vertices;
    Color[] colors = new Color[vertices.Length];

    colors [0] = Color.red;
    colors [1] = Color.green;
    colors [2] = new Color(199, 195, 0);
    colors [3] = new Color(199, 195, 0);

    mesh.colors = colors;
}
```

也就是 Mesh 结构中，有 colors 属性，这个属性就对应到每个顶点的颜色，可以很方便的进行更新和修改。

多 pass (multi-pass) 渲染

在 shader 中，同样的材质和资源，可能一次渲染并不能做出相应的效果出来。因此，GPU 的渲染能支持所谓的多个 pass。简单来说，在一个 shader 被执行的时候，有两个不同的程序段可以被依次执行到，举个例子，假如在 shader 中有程序段 A 和程序段 B，该 shader 被执行的时候，会先执行程序段 A，执行完 A 后再执行 B。

到现在为止，我们一直使用的都是 Surface Shader，并没有见过 pass。这里也正好可以看到 pass 的使用。

在 SubShader{} 块中，我们可以使用 Pass{} 块来指定一个 pass 的区域。因此，一个两个 pass 的 shader，看起来会是如下的结构

```
Shader "Name"
{
    Properties
    {
    }

    SubShader
    {
        Tags { }

        Pass
        {

        }

        Pass
        {
```

```
    }  
  
    }  
  
    Fallback "other"  
}
```

我们马上进入下一节内容，看一个实际的多 pass 的例子。

轮廓(silhouett shader)

在游戏或者 3D 应用中，选中物体常常会显示出自己的轮廓，表示物体的选中状态。



这样的 shader 往往需要用到我们的前面所说的 multi-pass，下面我们自己动手来做这么一个 shader。

- 新建立一个工程，也可以在原有的工程里新建一个场景(File | New scene)
- 新建立一个 sphere 作为显示轮廓的物体
- 新建立一个 material 和 shader，将两者关联
- 把 material 拖到 sphere 上，让 sphere 使用我们的 material
- 打开 shader 文件，用下面的内容替代原来的内容

```
Shader "Outlined/Silhouette"
{
    Properties
    {
        _OutlineColor ("Outline Color", Color) = (0,0,0,1)
    }
}
```

```
    _Outline ("Outline width", Range (0.0, 0.03)) = .005
}

SubShader
{
    Tags { "Queue" = "Transparent" }

    Pass
    {
        Name "BASE"
        Cull Back
        Blend Zero One

        SetTexture [_OutlineColor]
        {
            ConstantColor (0,0,0,0)
            Combine constant
        }
    }

    Pass
    {
        Name "OUTLINE"
        Tags { "LightMode" = "Always" }
        Cull Front

        Blend One OneMinusDstColor

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        #include "UnityCG.cginc"

        struct appdata
        {
            float4 vertex : POSITION;
            float3 normal : NORMAL;
        };

        struct v2f
        {
            float4 pos : POSITION;
```

```
        float4 color : COLOR;
    };

    uniform float _Outline;
    uniform float4 _OutlineColor;

    v2f vert(appdata v)
    {
        v2f o;
        o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

        float3 norm = mul((float3x3)UNITY_MATRIX_IT_MV, v.normal);
        float2 offset = TransformViewToProjection(norm.xy);

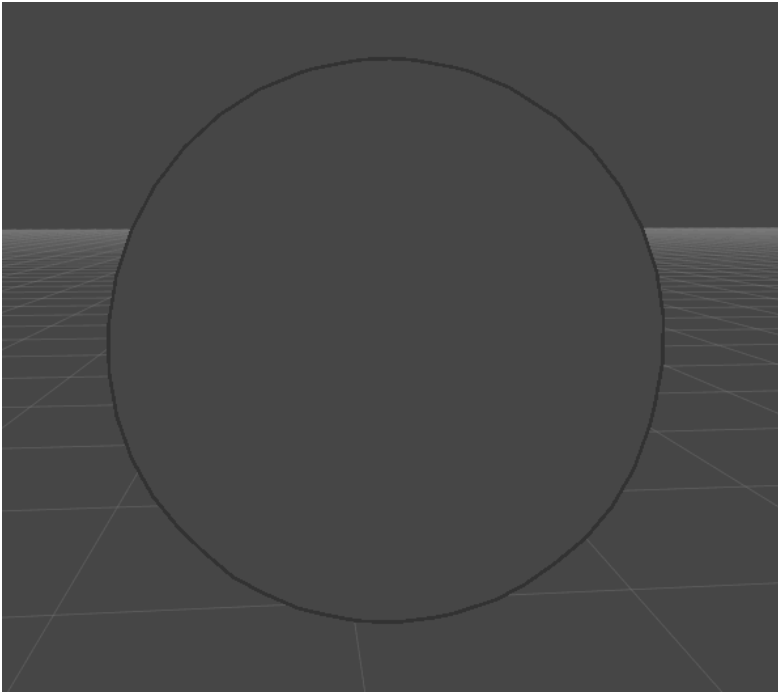
        o.pos.xy += offset * o.pos.z * _Outline;
        o.color = _OutlineColor;
        return o;
    }

    half4 frag(v2f i) :COLOR
    {
        return i.color;
    }
    ENDCG

}

Fallback "Diffuse"
}
```

- 保存，回到 unity，如果没有错的话，现在的 sphere 应该跟下图相似



上图可以看出来，我们的球体周围形成了一道框线，也就是我们要的轮廓。在 material 的 Inspector 面板上，我们可以通过拖动滑动条改变轮廓线的粗细，还可以通过使用调色板改动轮廓线的颜色。

该 shader 使用两个 pass，第一个 pass 用了 Fixed Function 语法，第二个是 VertexFragment 语法。第一个 pass 渲染物体的正面部分，在深度缓冲对应的 sphere 部分就留下了正面部分的深度值；第二个 pass 渲染物体的背面部分，同时让每个点的位置都朝自己的法线方向做了伸展，对于那些经过伸展后还和正面部分重合的位置的点，因为深度缓冲不能通过检测，就会被抛弃；而本来在球体边缘部分的点，因为伸展的缘故，深度缓冲值能够通过检测，因此就留下来，最后形成轮廓。

更漂亮的轮廓核心之模糊理论

用前面讲过的轮廓算法出来的效果,如果用在物体的高亮上面,一般不如商业游戏中做的好。

因为商业游戏中更漂亮的轮廓,并不是靠我们前面的算法做出来的。而且,我们前面的轮廓算法,也对付不了立方体(Box)这样的物体。在 Box 这类物体中,在法线方向伸展顶点会导致 Box 面跟面之间有裂缝。



上图中,我们可以看到,商业游戏中会用发光(Glow)这种方式来制作轮廓。Glow 或者 Bloom 这种算法,其中的核心就是模糊(Blur),这节内容我们着重这个点的理论。

模糊在图像处理中是非常重要的操作。它的核心算法在于:在目标像素点的附近取像素点,并且给这些点赋予一定的权重(weight),然后把每个像素点的值和它的权重得到的结果相加,得到目标像素点的值。举例来说,我们在处理图像的时候,当前准备往缓冲里写某个像素的

值，因为加入模糊，所以我们需要把当前的像素值和它相应区域的值加起来做一定的加权平均，这样计算出来的值才写到缓冲中。

语义上的算法如下：

```
function Blur (source, dest, radius)
{
    for (y = 0; y < height; ++y)
    {
        for (x = 0; x < width; ++x)
        {
            total = 0
            for (ky = -radius; ky <= radius; ++ky)
                for (kx = -radius; kx <= radius; ++kx)
                {
                    total += source(x + kx, y + ky)
                    dest(x, y) = total / (radius * 2 + 1) ^ 2
                }
        }
    }
}
```

这个方法对每个“源像素点”进行“模糊”，然后存入“目标像素点”中。其中唯一的参数是半径，这个半径决定了每个目标像素点在值由源像素点周围多少范围内的像素值来衡量。

大多数时候，我们其实不用简单的平均数，而是根据采样像素点离源像素点的位置来加权平均计算。使用加权平均，离源像素点近的像素就得到更大的照顾，出来的效果就更符合实际。

在上面的虚拟代码中,我们可以了解,如果对一幅为 width * height 这样的图进行模糊的话,运行次数 $\text{fetchtime} = \text{width} * \text{height} * \text{radius} * \text{radius}$ 。这里往往是被优化成两个过程,第一个过程先对水平方向进行模糊,第二个过程在第一个过程的基础上,对竖直方向进行模糊。

```
function BlurHorizontal (source, dest, radius)
{
    for (y = 0; y < height; ++y)
    {
        for (x = 0; x < width; ++x)
        {
            total = 0
            for (kx = -radius; kx <= radius; ++kx)
            {
                total += source(x + kx, y)
                dest(x, y) = total / (radius * 2 + 1)
            }
        }
    }
}

function BlurVertical (source, dest, radius)
{
    for (x = 0; x < width; ++x)
    {
        for (y = 0; y < height; ++y)
        {
            total = 0
            for (ky = -radius; ky <= radius; ++ky)
            {
                total += source(x, y + ky)
                dest(x, y) = total / (radius * 2 + 1)
            }
        }
    }
}

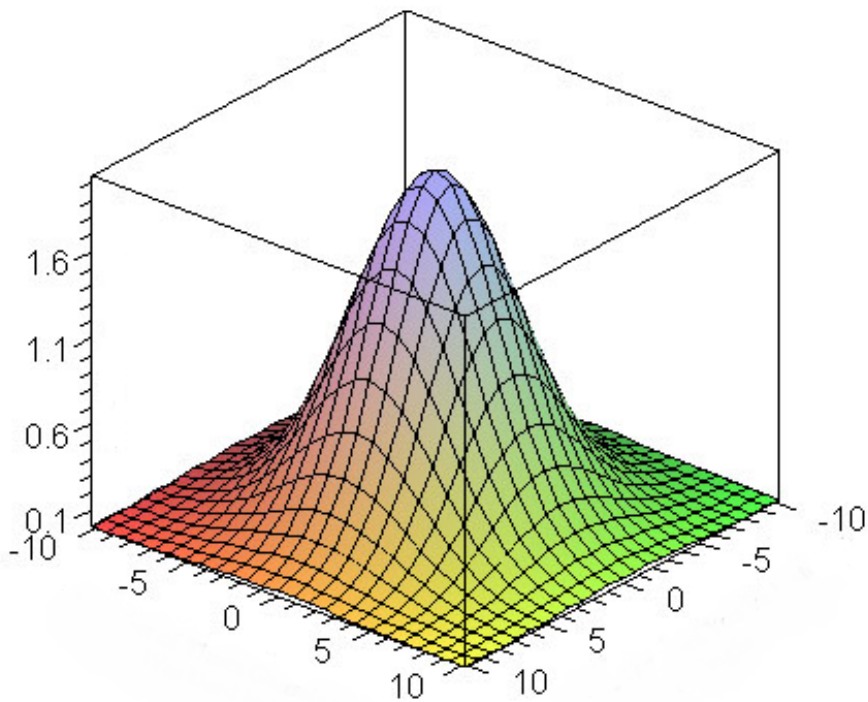
function Blur (source, dest, radius)
{
    BlurHorizontal(source, temp, radius)
```

```
BlurVertical(temp, dest, radius)
}
```

大家可以自行比较一下两个算法（出来的结果是一样的）的运行次数的差别。下面是几种有名的模糊算法

- 高斯模糊

模糊算法中常见的应该算是高斯模糊了，在 Photoshop 软件中有专门的菜单。在专业名词中，我们一般把这样的算法叫滤镜，因此高斯模糊在英文上也叫做 Gaussian filter。二维的高斯滤镜，是个立体的铃铛型的模型



这个图中心的最高点，如果把它当做是源像素点的位置的话，也就是它对应的目标像素点的值（铃铛覆盖到的整个区域）。

对于高斯模糊，我们需要下图这张图表

Index N	Coefficients												Sum = 2 ^N										
0								1					1										
1								1		1			2										
2								1		2		1	4										
3								1		3		3	1	8									
4								1		4		6	4	1	16								
5								1		5		10	10	5	1	32							
6								1		6		15	20	15	6	1	64						
7								1		7		21	35	35	21	7	1	128					
8								1		8		28	56	70	56	28	8	1	256				
9								1		9		36	84	126	126	84	36	9	1	512			
10								1		10		45	120	210	252	210	120	45	10	1	1024		
11								1		11		55	165	330	462	462	330	165	55	11	1	2048	
12								1		12		66	220	495	792	924	792	495	220	66	12	1	4096

比如，如果每个像素点都是靠周围的 9 个点来得到结果的话，我们需要找到中间的三角形横向部分数字对应的个数超过或者等于 9 的，按上图的实例，Index 8 以后的都可以用的上。

具体用法是，选择一行后，该行中间的数和 Sum 这列的数的商(Coefficients / Sum)就是该像素点的权重。这一行的中间数对应的是中心像素点，左右散开则是对应到左边（上边）/右边（下边）的像素点。



- 盒子滤镜 (Box Filter)

Box Filter 是常用的模糊方式，跟高斯滤镜不同，它给于每个像素点同样的权重。

核心算法是：

$\frac{1}{9}$ $\frac{1}{9}$ $\frac{1}{9}$

$\frac{1}{9}$ $\frac{1}{9}$ $\frac{1}{9}$

$\frac{1}{9}$ $\frac{1}{9}$ $\frac{1}{9}$

也就是对于每个目标像素点，它最后的值是对应的源像素点的周围（包含它自身）的 9 个像素点的综合评估值。

这些大致就是模糊的理论部分，更多的变种万变不离其宗。唯一要提到的是，在实现的时候，对于边缘的部分，也就是采样点不在图内的部分，一般可以用源像素点本身代替。在 shader 编程中，这块地方倒是硬件能帮助做掉。

模糊的实现

前面的课程讲到了几种模糊的理论，这节课我们来尝试实现一种模糊的方法，这个方法使用了前面没有讲过的一种方式，可以对源图进行缩小，然后在缩小的图上进行多次相关的类似 BoxFilter 般的采样，最后得到不错的模糊效果。结合 Unity 本身的流程和模糊理论，我们在步骤上先理一下思路。要实现把源图进行模糊，我们需要这么几个步骤：

1、把场景渲染到我们能读取的 buffer 中。这一步，我们可以通过 RenderTexture 来完成（在主相机的 OnPreRender() 里面，我们创建一个只渲染需要模糊的物体的相机（动态创建相机，参数除 RenerTexture 和 CullingMask 外，都从主相机拷贝，需要模糊的物体动态改变其 Culling Layer）。

2、得到源图的 RenderTexture 后，我们创建另外 2 个 RenderTexture，宽度和高度这两个 RenderTexture 都使用源图的一半。

3、在主相机的 OnRenderImage() 中，我们通过自己的 shader，用类似 BoxFilter 的方式把源图缩小拷贝到上述的 RenderTexture 的一个中，然后根据迭代的次数，使用类似 BoxFilter 的方式把非源图的 RenderTexture（也就是缩小的图）进行模糊处理（这里用到 Unity 的 Graphics.Blit() 方法）。

4、把缩小图的最终内容通过 Graphics.Blit() 拷贝到帧缓冲中，这里因为帧缓冲本身跟我们源图的大小相同，因此有个放大的过程，同样起了一定的“模糊”作用。

这样就能得到我们的最后的结果。

整个过程处理涉及的代码比较多，这里贴出模糊的核心的 shader 代码：

```
Shader "MyBlur"
{
    Properties
    {
        _MainTex ("" , 2D) = "" {}
    }

    SubShader
    {
        Pass
        {
            ZTest Always
            Cull Off
            ZWrite Off
            Lighting Off
            Fog { Mode Off }

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            uniform sampler2D _MainTex;
            uniform half4 _MainTex_TexelSize;
            uniform half _OffsetScale;

            struct v2f
            {
                float4 pos : POSITION;
                half2 uv[4] : TEXCOORD0;
            };

            v2f vert (appdata_img v)
            {
                v2f o;
                o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

                half2 offs = _MainTex_TexelSize.xy * _OffsetScale;
```



```
o.uv[o].x = v.texcoord.x - offs.x;
o.uv[o].y = v.texcoord.y - offs.y;

o.uv[1].x = v.texcoord.x + offs.x;
o.uv[1].y = v.texcoord.y - offs.y;

o.uv[2].x = v.texcoord.x + offs.x;
o.uv[2].y = v.texcoord.y + offs.y;

o.uv[3].x = v.texcoord.x - offs.x;
o.uv[3].y = v.texcoord.y + offs.y;

return o;
}

fixed4 frag(v2f i) : COLOR
{
    fixed4 color1 = tex2D(_MainTex, i.uv[o]);
    fixed4 color2 = tex2D(_MainTex, i.uv[1]);
    fixed4 color3 = tex2D(_MainTex, i.uv[2]);
    fixed4 color4 = tex2D(_MainTex, i.uv[3]);
    fixed4 color;
    color.rgb = max(color1.rgb, color2.rgb);
    color.rgb = max(color.rgb, color3.rgb);
    color.rgb = max(color.rgb, color4.rgb);
    color.a = (color1.a + color2.a + color3.a + color4.a);

    return color;
}

ENDCG
}

Fallback off
}
```

上面的代码用 Vertex Fragment 语法，`_MainTex_TexelSize` 是 Unity 内置的值，如它的字面意思，代表的是一个 texel 的尺寸，具体的值在 Unity 中，是一个 Vector4 结构：Vector4(1/

width, 1/ height, width, height)。【注：这里的 y 部分在某些时候可能带负号，主要是 Direct3D 中对 RenderTexture 做抗锯齿的时候可能翻转 y 的方向】

基于模糊的轮廓（高亮）

有了模糊基础后，我们其实就了解了当代商业游戏中用的非常多的轮廓高亮的核心算法。剩下的就是组合模糊部分和清晰部分，得到最终的结果。这里举例来阐述，



对于上图中的人物高亮，我们可以使用以下步骤得到：

- 1、在 `OnPreRender()` 中，把需要高亮的单位放置到特定的 layer，为叙述方便，我们把这个 layer 叫做 "highlightlayer"
- 2、把所有在 highlightlayer 上的物体用我们前面讲过的方式，使用一种单一的颜色（最后高亮的颜色），渲染到一个 `RenderTexture` 中，作为一个模板（因为 `RenderTexture` 默认色是黑色，因此渲染后只要不是黑色的部分，就是渲染的物体）。
- 3、对上面的 `RenderTexture`，用我们前面讲过的模糊方式得到一张模糊过的 `RenderTexture`。

4、使用一个用来合成的 shader，把作为模板的 RenderTexture 和模糊过的 RenderTexture 作为参数传入，该 shader 对每个 texel 做检测：发现在模板中对应部分非黑色，那么意味着该部分本来就是高亮物体的实体部分，直接用物体的本色来渲染；反之，则是高亮（也就是模糊部分），用模糊图里的 texel 代替。

下面部分是用来组合的 shader 的代码：

```
Shader "Composite"
{
    Properties
    {
        _MainTex ("", 2D) = "" {}
        _BlurTex ("", 2D) = "" {}
        _StencilTex ("", 2D) = "" {}
    }

    SubShader
    {
        Pass
        {
            ZTest Always
            Cull Off
            ZWrite Off
            Lighting Off
            Fog { Mode off }

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct v2f
            {
                float4 pos : POSITION;
                half2 uv[2] : TEXCOORD0;
```

```
};

uniform sampler2D _MainTex;
uniform sampler2D _BlurTex;
uniform sampler2D _StencilTex;
float4 _MainTex_TexelSize;

v2f vert (appdata_img v)
{
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP, v.vertex);

    o.uv[o] = v.texcoord.xy;
    o.uv[I] = v.texcoord.xy;

    if (_MainTex_TexelSize.y < o)
        o.uv[I].y = I - o.uv[I].y;

    return o;
}

fixed4 frag(v2f i) : COLOR
{
    fixed4 framebuffer = tex2D(_MainTex, i.uv[o]);
    fixed4 stencil = tex2D(_StencilTex, i.uv[I]);

    if (any(stencil.rgb))
    {
        return framebuffer;
    }
    else
    {
        fixed4 blurred = tex2D(_BlurTex, i.uv[I]);
        fixed4 color;
        color.rgb = lerp(framebuffer.rgb, blurred.rgb, saturate(blurred.
a - stencil.a));

        color.a = framebuffer.a;
        return color;
    }
}

ENDCG
}
```

```
}  
Fallback Off  
}
```

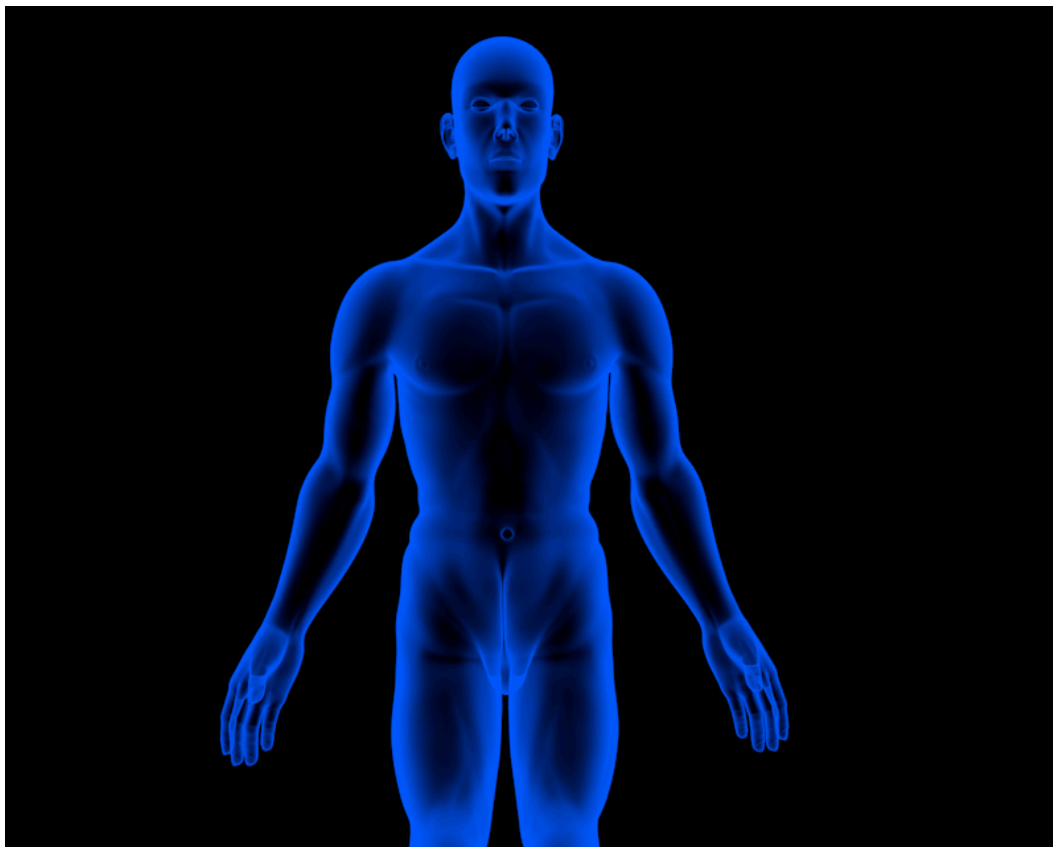
上面的代码中，`appdata_img` 在 `UnityCG.cginc` 中可以找到，是个 `struct` 结构：

```
struct appdata_img  
{  
    float4 vertex : POSITION;  
    half2 texcoord : TEXCOORD;  
};
```

代码中我们也看到对 `_MainTex_TexelSize` 带负号的处理。

X-Ray (X 光, X 射线) 基于 Ramp Texture 的版本

x 射线效果除了常用的医学软件，还用在很多场合，游戏中被遮挡的物体、工程中需要透视查看的物体部位等。



实现这种 shader，简单的版本可以用 ramp texture 来实现。举个例子，我们可以写一个用 Ramp Texture 的 shader，最后应用到场景中时，实现如下效果：





在这样的效果中，我们使用下面这样的 ramp texture：



这里是实现的 shader 代码：

```
Shader "My Shaders/XRay"
{
    Properties
    {
        _Color ("Tint (RGB)", Color) = (1,1,1)
        _RampTex ("Facing Ratio Ramp (RGB)", 2D) = "white" {}
    }
}
```



```
SubShader
{
    Cull Off
    ZWrite Off
    Tags { "Queue" = "Transparent" }
    Blend One OneMinusSrcColor

    Pass
    {

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        #include "UnityCG.cginc"

        float4 _Color;
        sampler2D _RampTex;

        struct v2f
        {
            float4 pos : SV_POSITION;
            float2 uv : TEXCOORD0;
        };

        float4 _RampTex_ST;

        v2f vert (appdata_base v)
        {
            v2f o;
            o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
            float3 viewDir = normalize(ObjSpaceViewDir(v.vertex));
            o.uv = float4( abs(dot(viewDir,v.normal)), 0.5, 0.0, 1.0 );
            return o;
        }

        half4 frag (v2f i) : COLOR
        {
            half4 texcol = tex2D (_RampTex, i.uv);
            return texcol * _Color;
        }
    }
}
```

```
        ENDCG

    }

}

Fallback "VertexLit"

}
```

上面代码中，我们主要对 `ObjSpaceViewDir()` 做一下介绍。这个方法在 `UnityCG.cginc` 中有源代码，如下

```
// Computes object space view direction
inline float3 ObjSpaceViewDir( in float4 v )
{
    float3 objSpaceCameraPos = mul(_World2Object, float4(_WorldSpaceCameraPos.
xyz, 1)).xyz * unity_Scale.w;
    return objSpaceCameraPos - v.xyz;
}
```

这段代码就两行，第一行把在世界坐标中的相机位置通过“世界转本地”矩阵，转入物体的本地化位置，然后跟参数 `v` 做了相减，得到一个相机的方向信息（该向量没有归一化）。

这个方法介绍完毕，我们就可以明白了，`normalize()`的调用是因为上面的方法缺少这一步，但为什么要用这个方法，是因为后面的法线不需要化成世界坐标下的，而只要拿出来用就可以：因为这里的相机（视线）方向已经是本地化坐标下的了。

Ramp Texture 控制的 X-Ray（X 射线）着色器，重点是这么一句：

```
o.uv = float4( abs(dot(viewDir,v.normal)), 0.5, 0.0, 1.0 );
```

o.uv 用视线和法线的夹角来指导在 RampTexture 上的 x 位置, y 位置固定为 0.5 (因此也就是个 1D texture), 因此, 越是夹角大的地方 (一般是边缘), 颜色就越往左边取。

有了这个特性后, 我们就可以通过使用不同的 RampTexture 来控制显示效果。最后要说的是, 对于 X-Ray 来说, 因为物体背面部分也要显示, 因此在开始的时候 Cull Off 是必须的。

另外, 在上面的代码中, 我们可以看到 `_RampTex_ST` 这样的字眼, 这个带 `_ST` 后缀的变量, 存放的是对应的变量 (除去后缀即它对应的变量) 的 `tiling` 和 `offset` 的值, 我们在本书内容中讲过许多这类变量和定义, 这些都没有在官方文档里定义, 在 `UnityCG.cginc` 中, 可以看到这样的字眼:

```
#define TRANSFORM_TEX(tex,name) (tex.xy * name##_ST.xy + name##_ST.zw)
```

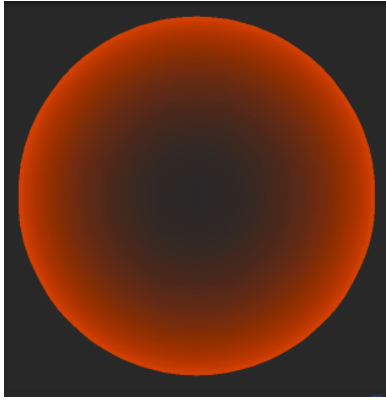
由此知道, `xy` 部分存的是 `tiling` 的值, `offset` 存放 `offset`。

X-Ray (X 光, X 射线)增强版

前面讲述了使用 Ramp Texture 做渲染控制的 X-Ray 着色器,这节课我们讲述另外一种方式,这种方式用不到 Ramp Texture, 并且可以控制相应的颜色和“边缘化”的程度,出来的效果如图:



它对应的材质球的截图:



在 Ramp Texture 中，我们根据物体表面的法线跟视线之间的夹角来确定该物体表面在 1D 材质图上的索引位置，从而得到该处的色彩信息。因而，我们可以通过改变 Ramp Texture 上的色彩来控制 X-Ray（X 射线）的最终呈现效果。

在这个过程中，我们发现这个规律，对于被判别为“边缘”的部分，它的法线如果写成向量 uvw 的话，w 分值往往接近于 0。相反，被判别为“中心”的部分，它的法线的 w 分值越来越接近于 1。利用这个特点，我们因此这个 shader 的代码如下：

```
Shader "Xray"
{
    Properties
    {
        _Color("_Color", Color) = (0,1,0,1)
        _Inside("_Inside", Range(0,1) ) = 0
        _Rim("_Rim", Range(0,2) ) = 1.2
    }

    SubShader
    {
        Tags { "Queue" = "Transparent" }
        LOD 200

        Pass
        {
```

```
Cull off
Zwrite off
Blend oneminusdstcolor one

CGPROGRAM

#pragma vertex vert
#pragma fragment frag

#include "HLSLSupport.cginc"
#include "UnityCG.cginc"

struct v2f_surf
{
    half4 pos      : SV_POSITION;
    fixed4 finalColor : COLOR;
};

uniform half4 _Color;
uniform half _Rim;
uniform half _Inside;

v2f_surf vert (appdata_base v)
{
    v2f_surf o;

    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    half3 uv = mul( (float3x3)UNITY_MATRIX_IT_MV, v.normal );
    uv = normalize(uv);
    o.finalColor = lerp(half4(0,0,0,0),_Color,saturate(max(1- pow (uv.z,_Rim),
_Inside)));

    return o;
}

fixed4 frag (v2f_surf IN) : COLOR
{
    return IN.finalColor;
}

ENDCG
}
```

```
FallBack "VertexLit"  
}
```

上述的代码我们只要看 vert 方法中的最后一行，

```
o.finalColor = lerp(half4(o,o,o,o),_Color,saturate(max(1- pow (uv.z,_Rim),_Inside)));
```

这里是对此的分析：

- uv.z 是归一化过的，因此值区间在 0 到 1 之间（包含 0 和 1）
- 因此 pow(uv.z, _Rim)的值区间在 0 到 1 之间
- 因此 max 方法的第一个参数 1-pow(uv.z, _Rim)的值区间在 1 到 0 之间
- saturate 方法是一个 cg 的方法，能让返回值规范在 0 到 1 区间内（该方法详细可见[这里](#)）
- lerp 值 cg 的方法，意义为第一个参数可以看做参数三为 0 的时候取的值，如果参数 3 为满值 1，则取第二个参数。如果第三个参数在 0 到 1 之间，那么根据比例取第一参数和第二参数的中间值。

因此，如果我们都取默认值，_Rim = 1.2， _Inside = 0 的话。如果是边缘化的地方，因 uv.z 为 0，max 返回的结果就是 1，saturate 返回的结果就是 1，因而插值后的结果就是我们指定的颜色。如果是中间部分，因 uv.z 为 1，max 返回结果是 0，saturate 返回的结果就是 0，因而插值后的结果就是 shader 代码定下的颜色黑色（0，0，0，0）。

遮挡 shader（在被遮挡后显示不同效果）

在固定视角的游戏中，因为场景的层次关系，经常有角色或者 NPC 会进入墙或者其他物体的背面而被遮挡，在这个处理中，shader 经常发挥很大的作用。举例来说，有如下场景：



上图中，主角完全被山体挡住，为了显示这个效果，主角会被显示成整个呈现 x 射线处理过的效果。这种完全被遮挡的时候，我们只要用相应的 shader（比如前面讲过的 x-ray）处理就好。

问题在于，大多数时候，相关的人物会有部分被遮挡（如下图），这个时候我们怎么处理呢？



下面的内容我们来讲述这个特殊的 shader。

在正常的渲染流程中，GPU 都用所谓的深度测试（Z-Test）来指导一个 texel 最终需不需要绘制到帧缓冲中。默认情况下，只有更靠近相机（也就是深度测试中发现的具备更小的 z 值）的 texel 才会进入帧缓冲，否则该 texel 会在渲染管线中被提前抛弃（参考“3D 管线”一节内容）。可喜的是，这个流程在现代的可编程模式下，都是可控的：我们可以设置 GPU 怎么测试我们的 texel。

利用这个特性，我们要做的事情是，对我们要处理的物体，shader 中要具备两个 Pass，一个 Pass 处理本来通不过的 texel，让这些 texels 通过，进行 frag 方法着色，绘制我们觉得被遮挡的时候的信息。另外一个 Pass 正常处理就可。

通过上面的方法，在物体被渲染的时候，如果部分可见，部分不可见，那么就能呈现上面第二张图想表现的效果出来。

这里是我们实现的效果图：



图中的人物部分被树遮挡，被遮挡的部分用蓝色渲染了出来，正常部分是灰色的。

相关的 shader 代码如下：

```
Shader "MyCharacter"
{
    Properties
    {
        _Color ("Main Color", Color) = (1,1,1)
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _OccludeColor ("Occlusion Color", Color) = (0,0,1)
    }

    SubShader
```

```
{
    Tags {"Queue"="Geometry+1"}

    Pass
    {
        ZWrite Off
        Blend One Zero
        ZTest Greater
        Color [_OccludeColor]
    }

    Pass
    {
        Tags {"LightMode" = "Vertex"}
        ZWrite On
        Lighting On

        Material
        {
            Diffuse [_Color]
            Ambient [_Color]
        }

        SetTexture [_MainTex]
        {
            ConstantColor [_Color]
            Combine texture * primary DOUBLE, texture * constant
        }
    }
}
FallBack "Diffuse", 1
}
```

上面的代码，我们完全用 Fixed Function 格式写成。因此我们对部分代码进行一番解释：

- `Tags {"Queue"="Geometry+1"}`

这句指示使用该 shader 的物体必须在 Geometry 这个队列里面的物体渲染完毕后渲染。

按照一般的 Unity 规则，也就是在非透明队列物体后，AlphaTest 队列物体前。这句话保证了该物体先让非透明物体渲染好，因此能知道自己被遮盖的部分。

- *Color [_OccludeColor]*

这句话设置物体的颜色为后面方括弧中指定的颜色

- *Material { Diffuse [_Color] Ambient [_Color] }*

指定物体的材质属性，块里面可以指定 Diffuse, Ambient, Specular, Shiness, Emission 这些色值。用于跟光照计算颜色。

- *SetTexture { ... }*

字面意思是“分配一张贴图”，实际效果是会把块的左花括弧前的[]中的贴图跟块中的作为参数传进块里面进行合成

- *ConstantColor [_Color]*

定义一个可在同一个块中用的色值

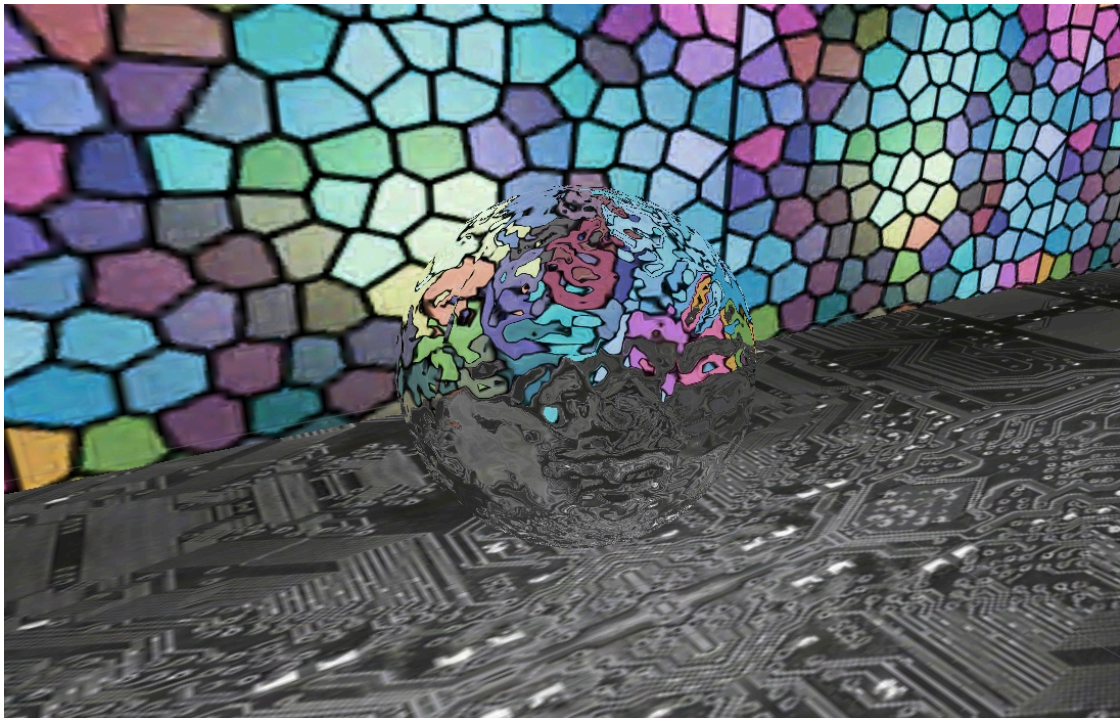
- *Combine texture * primary DOUBLE, texture * constant*

逗号隔开的部分对应于 RGB 通道和 Alpha 通道，也就是 RGB 部分让送入的 texture 和光照算法（由 Material 块决定）后的结果做乘法，出的结果翻倍。Alpha 部分由送入的 texture 的 alpha 部分和由 ConstantColor 指定的值的 alpha 部分做乘法得到。这

里如果没有逗号，那么 RGB 部分的算法和 Alpha 部分的算法是一样的。

透明和玻璃

玻璃在游戏应用中，弹性很大，有基本上跟现实中的原理一样实现的，有使用非常简便的方式实现的。最简单的就是用上透明，让东西看起来有点那么个意思。如果是真实一点的，必须注意到玻璃本身具备：透明、反射、高光、本身的色彩等等属性。



下面是我们用 surface shader 写的一个可以在 unity 中运行的着色器程序，支持玻璃自身的贴图、颜色，支持玻璃的反射（不是实时反射，用了 cubemap 来让这个 shader 效率高一些），支持玻璃的高光。

```
Shader "Custom/Transparent Specular"
{
    Properties
    {
```

```
_MainTex ("Base (RGB)", 2D) = "white" {}  
_Colour ("Transparency (A only)", Color) = (0.5, 0.5, 0.5, 1)  
_Cube ("Reflection Cubemap", Cube) = "_Skybox" { TexGen CubeReflect }  
_ReflectColour ("Reflection Colour", Color) = (1,1,1,0.5)  
_ReflectBrightness ("Reflection Brightness", Float) = 1.0  
_SpecularMap ("Specular / Reflection Map", 2D) = "white" {}  
_RimColour ("Rim Colour", Color) = (0.26,0.19,0.16,0.0)  
  
}
```

SubShader

```
{  
    Tags  
    {  
        "Queue"="Transparent"  
        "IgnoreProjector"="True"  
        "RenderType"="Transparent"  
    }  
}
```

CGPROGRAM

```
#pragma surface surf BlinnPhong alpha
```

```
sampler2D _MainTex;  
sampler2D _SpecularMap;  
samplerCUBE _Cube;  
fixed4 _ReflectColour;  
fixed _ReflectBrightness;  
fixed4 _Colour;  
float4 _RimColour;
```

```
struct Input  
{  
    float2 uv_MainTex;  
    float3 worldRefl;  
    float3 viewDir;  
};
```

```
void surf (Input IN, inout SurfaceOutput o)  
{  
    half4 c = tex2D (_MainTex, IN.uv_MainTex);  
    o.Albedo = c.rgb;  
  
    o.Alpha = _Colour.a * c.a;
```

```
half specular = tex2D(_SpecularMap, IN.uv_MainTex).r;
o.Specular = specular;

fixed4 reflcol = texCUBE (_Cube, IN.worldRefl);
reflcol *= c.a;
o.Emission = reflcol.rgb * _ReflectColour.rgb * _ReflectBrightness;
o.Alpha = o.Alpha * _ReflectColour.a;

half intensity = 1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
o.Emission += intensity * _RimColour.rgb;
}
ENDCG
}
FallBack "Diffuse"
}
```

上面的 shader 在 unity4.6 环境中编译通过，代码的相关知识点，如 Cubemap、Rim、高光的一些理论，都可以在前面的内容中找到。

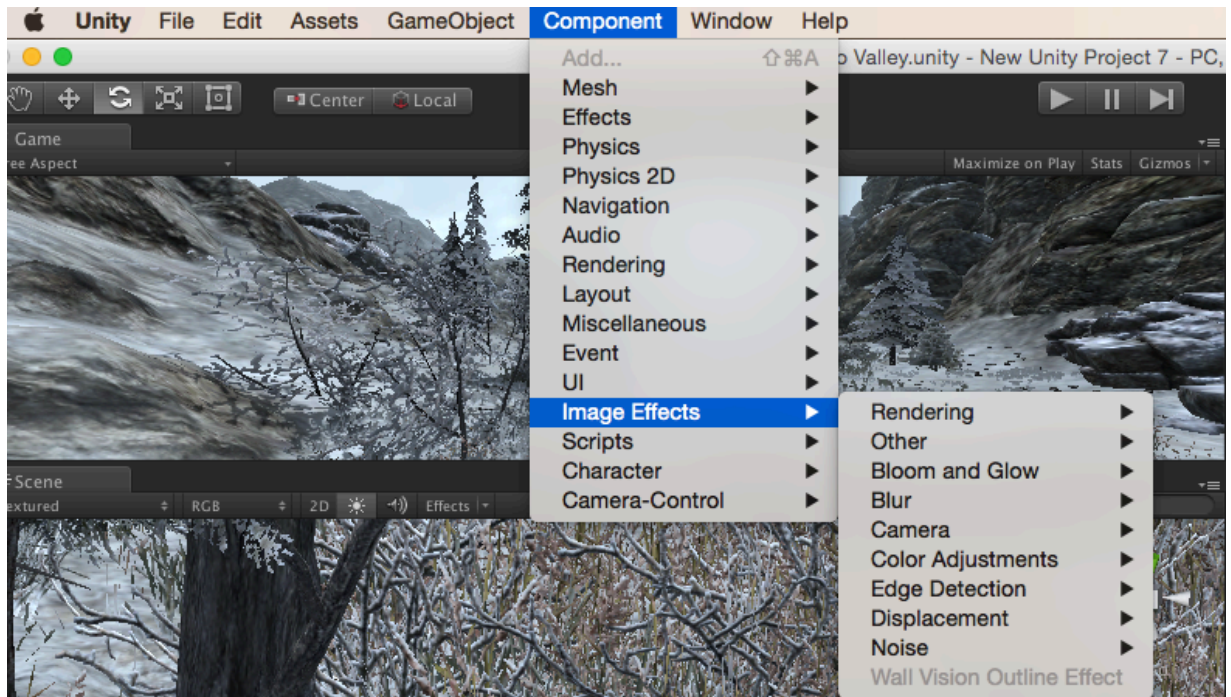
后期

商业 3D 引擎缺少后期（处理）就不会完美了。后期是在计算上非常简便，但是效果却非常强大的图像处理方式。在流程上，场景绘制输出后，2D 的缓冲区可以被作为源，让相应的计算发生在这个源上面，实现不同的效果。

后期(Post Process)概述

后期处理 (Post Process) 严格来说是渲染流程的其中一步：在视频缓冲区的内容完成后，我们对这部分的内容进行传统的 2D 图像处理，然后输出给用户看到最后的画面。因此，这个过程的特点是在 2D 空间内进行的，基本上涉及的主要是图像处理的算法。

因为后期处理是对像素做的处理，因此后期处理主要用像素着色器来控制。在市面上的大多数 3D 引擎中，都会对后期处理提供支持。而且一般会按照常见程度内置默认的后处理。在 Unity 中，用户可以通过导入包 "Image Effects" 来开启这个功能。导入这个包后，在主菜单 "Components" 下，会多出子菜单 "Image Effects"，里面置放了多种默认的后处理。



上面的截图使用的是 unity4.6，其他版本应该大同小异。

后期处理之灰度画面以及通用的后期框架

后期处理的第一个案例，我们来看 Unity 自带的一个，从而了解整个后期的流程和 shader 在其中的角色和位置。

这个案例能让彩色的画面变成灰度的，如下面两张图





上面的第二幅画面是 Unity 场景中，在主相机上挂接后期处理的 Greyscale 就能得到。如果主相机上没有这个组件，那么画面就是第一幅图。

尝试这个后期的步骤如下，在实现我下面的步骤前，你应该有了自己的一个工程，主相机能看到画面中的某部分画面。确保这个前提条件后，加入以下步骤：

- 确认主菜单 Component 下有 Image Effects。如果没有，使用菜单 Assets | Import Package | Image Effects(Pro Only)导入关联的包。
- 在 Hierarchy 中选中主相机，然后使用菜单 Component | Image Effects | Color Adjustments | Grayscale
- 确保上面的过程在主相机上加了一个 Component "Gray Effect (Script)"

如果没有问题，这个时候 Game 窗口的画面会在组件被加上的瞬间成为灰色的。

下面我们从上面加入的组件（脚本）来分析整个流程。在 Inspector 上 Grayscale 组件的右上角的下拉三角点击，出现的上下文菜单中选"Edit script"，让关联的编辑器打开这个脚本。

为方便描述，我把脚本的代码复制到下面

```
using UnityEngine;

[ExecuteInEditMode]
[AddComponentMenu("Image Effects/Color Adjustments/Grayscale")]
public class GrayscaleEffect : ImageEffectBase {
    public Texture textureRamp;
    public float rampOffset;

    // Called by camera to apply image effect
    void OnRenderImage (RenderTarget source, RenderTexture destination) {
        material.SetTexture("_RampTex", textureRamp);
        material.SetFloat("_RampOffset", rampOffset);
        Graphics.Blit (source, destination, material);
    }
}
```

一共才几行代码，主要的工作就是使用 Graphics.Blit()，对 source 对应的图像进行处理，放置到 destination 中，处理的时候使用 material 关联的材质。这里我们的 source 对应的图像是每帧 3D 流水线出来的画面，材质中我们指定了用来做灰度处理的 shader。

Grayscale 类继承于 ImageEffectBase，这个类也很小，所以拷贝放到这里

```
[RequireComponent (typeof(Camera))]  
[AddComponentMenu("")]  
public class ImageEffectBase : MonoBehaviour {  
    /// Provides a shader property that is set in the inspector  
    /// and a material instantiated from the shader  
    public Shader    shader;  
    private Material m_Material;  
  
    protected virtual void Start ()  
    {  
        // Disable if we don't support image effects  
        if (!SystemInfo.supportsImageEffects) {  
            enabled = false;  
            return;  
        }  
  
        // Disable the image effect if the shader can't  
        // run on the users graphics card  
        if (!shader || !shader.isSupported)  
            enabled = false;  
    }  
  
    protected Material material {  
        get {  
            if (m_Material == null) {  
                m_Material = new Material (shader);  
                m_Material.hideFlags = HideFlags.HideAndDontSave;  
            }  
            return m_Material;  
        }  
    }  
  
    protected virtual void OnDisable() {  
        if( m_Material ) {  
            DestroyImmediate( m_Material );  
        }  
    }  
}
```

ImageEffectBase 代码主要的功能就是动态的创建新的材质,并且能让用户指定一个 shader。

默认的情况下，我们都是对 Grayscale 的两个参数 TextureRamp 和 RampOffset 留空，因此这两个参数我们不管，直接看相关的 shader。

在 Insepector 面板上 Grayscale Effect 组件部分，双击 Shader 字样后的编辑框打开 shader，代码如下：

```
Shader "Hidden/Grayscale Effect" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _RampTex ("Base (RGB)", 2D) = "grayscaleRamp" {}
    }

    SubShader {
        Pass {
            ZTest Always Cull Off ZWrite Off
            Fog { Mode off }
        }

        CGPROGRAM
        #pragma vertex vert_img
        #pragma fragment frag
        #pragma fragmentoption ARB_precision_hint_fastest
        #include "UnityCG.cginc"

        uniform sampler2D _MainTex;
        uniform sampler2D _RampTex;
        uniform half _RampOffset;

        fixed4 frag (v2f_img i) : SV_Target
        {
            fixed4 original = tex2D(_MainTex, i.uv);
            fixed grayscale = Luminance(original.rgb);
            half2 remap = half2 (grayscale + _RampOffset, .5);
            fixed4 output = tex2D(_RampTex, remap);
            output.a = original.a;
            return output;
        }
    }
}
```

```
}  
ENDCG  
  
}  
}  
  
Fallback off  
  
}
```

`#pragma` 指定了顶点处理方法，但是代码中找不到该方法。原来这个方法是放置在 UnityCG.cginc 中的，代码如下

```
v2f_img vert_img( appdata_img v )  
{  
    v2f_img o;  
    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);  
    o.uv = MultiplyUV( UNITY_MATRIX_TEXTURE0, v.texcoord );  
    return o;  
}
```

这个方式用在后期处理的所有 shader 中，它的作用主要是把顶点化入到投影空间（3D 渲染管线需要用户的代码这么做）。

因此，正如我们前面讲过的，后期处理的真正代码都在像素着色器 frag 方法中，

- 第一行代码从输入的 source 中得到每个像素 RGBA 值
- 第二行用到一个方法，这个放在在 UnityCG.cginc 中定义

```
// Converts color to luminance (grayscale)  
inline fixed Luminance( fixed3 c )  
{
```



```
return dot( c, fixed3(0.22, 0.707, 0.071) );  
}
```

dot 方法使用 3D 的应该熟悉，所以化成简单的公式，就是

$$c.R * 0.22 + c.G * 0.707 + c.B * 0.071$$

这个公式是把彩色信息转化成灰度信息的数学公式。因此，第二行是把彩色转成了灰度。

- 第三行用第二行的灰度（因为默认后面的 RampTexture 是 0），和 0.5 构造了一个 2 维的值 half2
- 第四行用上面的这个 2 维值做索引，得到在 RampTex 这个图中的色彩值，因为我们没有提供该图，默认是 unity 自带的灰度图
- 第五行把上面索引查到的值的 alpha 值改成源图中的 alpha 值
- 第六行就可以输出结果了

由整个过程看出，如果我们确定只需要输出灰度图，那么可以简化该 shader，第二行的结果可以直接作为结果输出。Unity 这里是为了让用户可以自定义灰度信息的最终反映结果，用户可以提供一张 1 维的图作为 RampTexture，用来改变灰度信息影响到的最终结果。

看完这个案例，我们总结一下后期的整个流程：

1. 需要一个挂接在相机上的脚本文件，该文件能使用 `Graphics.Blit()` 来把源图改变（为什么需要挂在相机上？因为 `OnRenderImage()` 只有在相机上的脚本中才会被 Unity 在渲染的时候调用）
2. 需要提供一个 shader，里面的像素着色器能把每个像素处理成最后的结果。

重影 (Double Vision)

重影在“魔兽世界”某个任务（原谅我已经记不清了，找了半天没有找到这个任务名）中被使用，大致的情形是接了这个任务后，主角会进入变身豹子（没记错的话）状态，同时整个屏幕会呈现出梦境般的感觉。这个屏幕特效是多种后期的混合，其中一种后期就是我们今日的主题：重影。

重影是什么，一图胜千言：



能做出这种效果的 shader 源代码：

```
Shader "DoubleVision"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _displace ("Displace", Vector) = (0.7, 0.0, 0.0, 0.0)
    }

    SubShader
    {
        Pass
        {
            ZTest Always Cull Off ZWrite Off
        }
    }
}
```

```
Fog { Mode off }

CGPROGRAM

#pragma vertex vert_img
#pragma fragment frag
#pragma fragmentoption ARB_precision_hint_fastest
#include "UnityCG.cginc"
#include "Colorful.cginc"

sampler2D _MainTex;
fixed2 _displace;

fixed4 frag(v2f_img i):COLOR
{
    fixed4 c = tex2D(_MainTex, i.uv);

    c += tex2D(_MainTex, i.uv + half2(_displace.x * 16, _displace.y * 1
6)) * 0.6f;
    c += tex2D(_MainTex, i.uv + half2(_displace.x * 24, _displace.y * 2
4)) * 0.4f;
    c += tex2D(_MainTex, i.uv + half2(_displace.x * 32, _displace.y * 3
2)) * 0.2f;

    c *= 0.5f;

    return c;
}

ENDCG

}

Fallback off
}
```

除了像素着色器代码，其他的基本都是一样的。对于重影图像，算法主要是每个像素的目标值，都是原位置的像素和附近像素的加权平均值，具体的公式大家可以在代码中看出来：

```
destColor = (srcColor + srcColor1 * 0.6 + srcColor2 * 0.4 + srcColor3 * 0.2) * 0.5
```

其中的 srcColorN 可以通过参数进行控制。

根据时间变换参数的重影结合颜色调整就可以做出我们在前面提到的魔兽世界中的效果的重影部分。

老照片

这个部分我们来看另外一种后期的实现算法，先上图看这个后期的效果：



这种氛围给人的感觉是我们在看一副老照片。在游戏或者影视作品中，经常用来表达一种处在回忆中的感觉。在技术上，专业名词叫“Sepia tone”，sepia 取“深褐色”之意，tone 去“色调”的意思。所以中文翻译过来也即“深褐色的色调”。

在彩照出现之前，照片都是黑白色的。这个时候出现一种化学处理技术，能把照片中的银转成硫化物，这样的方式能延长照片的寿命，同时也会给照片带来这种色调。

因此，我们这里的算法，

- 首先是把在缓冲区的图像转化成黑白（灰度）的
- 然后在这个基础上加入相应的颜色（深褐色）

下面是相关的 shader 源代码：

```
Shader "Sepia tone"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }

    SubShader
    {
        Pass
        {
            ZTest Always
            Cull Off
            ZWrite Off

            CGPROGRAM
            #pragma vertex vert_img
            #pragma fragment frag
            #include "UnityCG.cginc"

            uniform sampler2D _MainTex;

            fixed4 frag (v2f_img i) : SV_Target
            {
                fixed4 original = tex2D(_MainTex, i.uv);

                // 灰度
                fixed Y = dot (fixed3(0.299, 0.587, 0.114), original.rgb);

                // 加入深褐色
                fixed4 sepiaConvert = float4 (0.191, -0.054, -0.221, 0.0);
                fixed4 output = sepiaConvert + Y;
```

```
        output.a = original.a;

        return output;
    }
    ENDCG

}

Fallback off

}
```


后记

学习 Shader 是个长期的过程，因为这门学问的知识还在发展之中。虽然 shader 被译作“着色器”，它英文的本意也是上色，但是因为是程序，所以它随着参与的人越来越多，出现的可能性也越来越广泛。如果按照功能写，可能没有哪本书能写完所有的 shader 功能。所以，原理的了解很重要，图形学、3D 知识都是极好的补充。

更多教程内容和开发学习，请上 <http://98jy.net>

3D 交流 QQ 群：291633884（时间有限，书中错误之处请入群轻拍）