猫都能学会的Unity3D Shader入门指南（一）

Unity Shader教程

动机

自己使用Unity3D也有一段时间了，但是很多时候是流于表面，更多地是把这个引擎简单地用作脚本控制，而对更深入一些的层次几乎没有了解。虽然说Unity引擎设计的初衷就是创建简单的不需要开发者操心的谁都能用的3D引擎，但是只是肤浅的使用，可能是无法达到随心所欲的境地的，因此，这种状况必须改变！从哪里开始呢，貌似有句话叫做会写Shader的都是高手，于是，想大概看看从Shader开始能不能使自己到达的层次能再深入一些吧，再于是，有了这个系列（希望我能坚持写完它，虽然应该会拖个半年左右）。

Unity3D的所有渲染工作都离不开着色器（Shader），如果你和我一样最近开始对Shader编程比较感兴趣的话，可能你和我有着同样的困惑：如何开始？Unity3D提供了一些Shader的手册和文档（比如这里，这里和这里），但是一来内容比较分散，二来学习阶梯稍微陡峭了些。这对于像我这样之前完全没有接触过有关内容的新人来说是相当不友好的。国内外虽然也有一些Shader的介绍和心得，但是也同样存在内容分散的问题，很多教程前一章就只介绍了基本概念，接下来马上就搬出一个超复杂的例子，对于很多基本的用法并没有解释。也许对于Shader熟练使用的开发者来说是没有问题，但是我相信像我这样的入门者也并不在少数。在多方寻觅无果后，我觉得有必要写一份教程，来以一个入门者的角度介绍一些Shader开发的基本步骤。其实与其说是教程，倒不如说是一份自我总结，希望能够帮到有需要的人。

所以，本“教程”的对象是

- 总的来说是新接触Shader开发的人：也许你知道什么是Shader，也会使用别人的Shader，但是仅限于知道一些基本的内建Shader名字，从来没有打开它们查看其源码。
- 想要更多了解Shader和有需求要进行Shader开发的开发者，但是之前并没有Shader开发的经验。

当然，因为我本身在Shader开发方面也是一个不折不扣的大菜鸟，本文很多内容也只是在自己的理解加上一些可能不太靠谱的求证和总结。本文中的示例应该会有更好的方式来实现，因此您是高手并且恰巧

路过的话，如果有好的方式来实现某些内容，恳请您不吝留下评论，我会对本文进行不断更新和维护。

一些基本概念

Shader和Material

如果是进行3D游戏开发的话，想必您对着两个词不会陌生。Shader（着色器）实际上就是一小段程序，它负责将输入的Mesh（网格）以指定的方式和输入的贴图或者颜色等组合作用，然后输出。绘图单元可以依据这个输出来将图像绘制到屏幕上。输入的贴图或者颜色等，加上对应的Shader，以及对Shader的特定的参数设置，将这些内容（Shader及输入参数）打包存储在一起，得到的就是一个Material（材质）。之后，我们便可以将材质赋予合适的renderer（渲染器）来进行渲染（输出）了。

所以说Shader并没有什么特别神奇的，它只是一段规定好输入（颜色，贴图等）和输出（渲染器能够读懂的点和颜色的对应关系）的程序。而Shader开发者要做的就是根据输入，进行计算变换，产生输出而已。

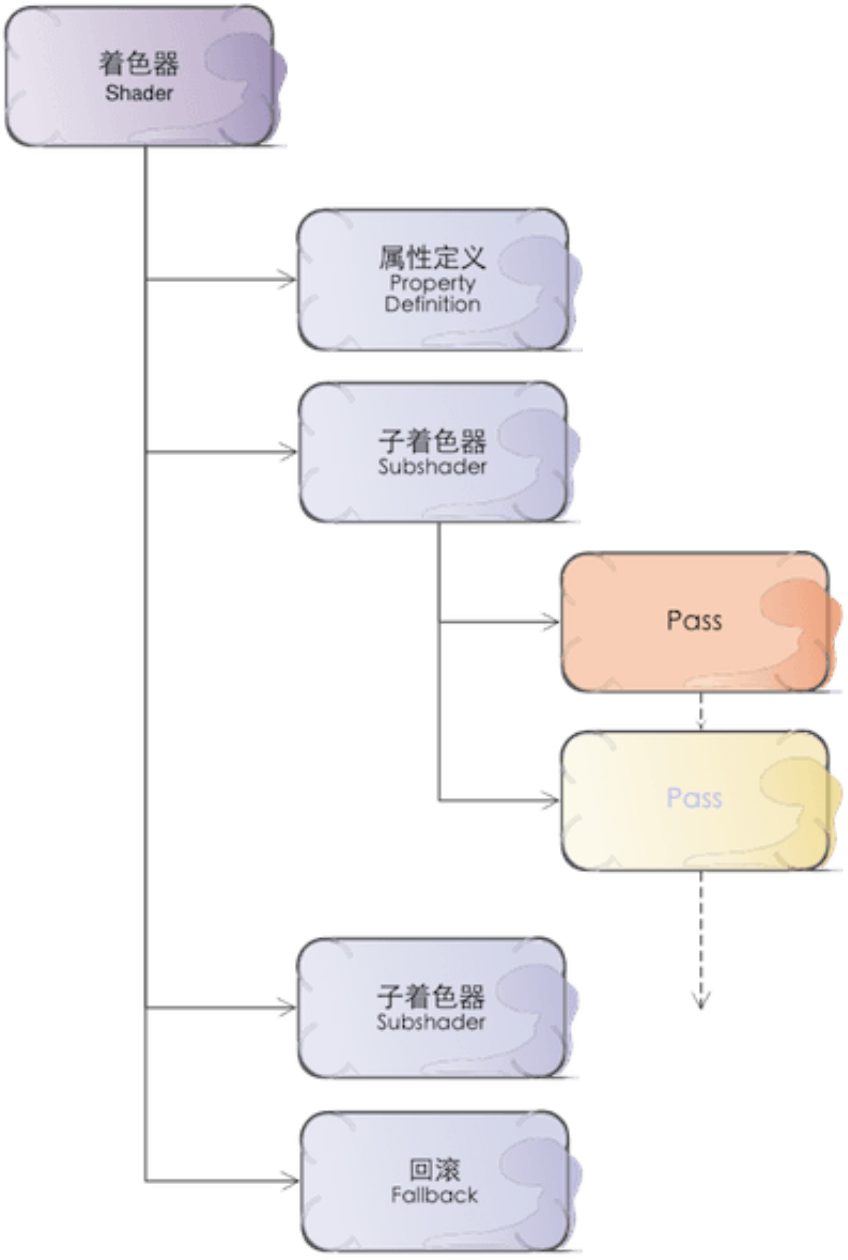
Shader大体上可以分为两类，简单来说

- 表面着色器（Surface Shader） – 为你做了大部分的工作，只需要简单的技巧即可实现很多不错的效果。类比卡片机，上手以后不太需要很多努力就能拍出不错的效果。
- 片段着色器（Fragment Shader） – 可以做的事情更多，但是也比较难写。使用片段着色器的主要目的是可以在比较低的层级上进行更复杂（或者针对目标设备更高效）的开发。

因为是入门文章，所以之后的介绍将主要集中在表面着色器上。

Shader程序的基本结构

因为着色器代码可以说专用性非常强，因此人为地规定了它的基本结构。一个普通的着色器的结构应该是这样的：



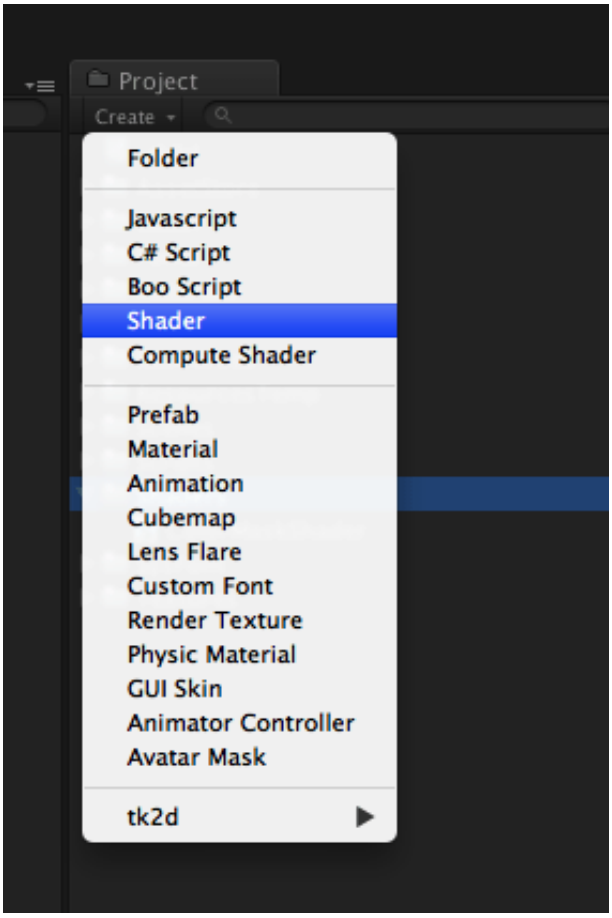
一段Shader程序的结构

首先是一些属性定义，用来指定这段代码将有哪些输入。接下来是一个或者多个的子着色器，在实际运行中，哪一个子着色器被使用是由运行的平台所决定的。子着色器是代码的主体，每一个子着色器中包含一个或者多个的Pass。在计算着色时，平台先选择最优先可以使用的着色器，然后依次运行其中的Pass，然后得到输出的结果。最后指定一个回滚，用来处理所有Subshader都不能运行的情况（比如目标设备实在太老，所有Subshader中都有其不支持的特性）。

需要提前说明的是，在实际进行表面着色器的开发时，我们将直接在Subshader这个层次上写代码，系统将把我们的代码编译成若干个合适的Pass。废话到此为止，下面让我们真正实际进入Shader的世界吧。

Hello Shader

百行文档不如一个实例，下面给出一段简单的Shader代码，然后根据代码来验证下上面说到的结构和阐述一些基本的Shader语法。因为本文是针对Unity3D来写Shader的，所以也使用Unity3D来演示吧。首先，新建一个Shader，可以在Project面板中找到，Create，选择Shader，然后将其命名为`Diffuse Texture`：



在Unity3D中新建一个Shader

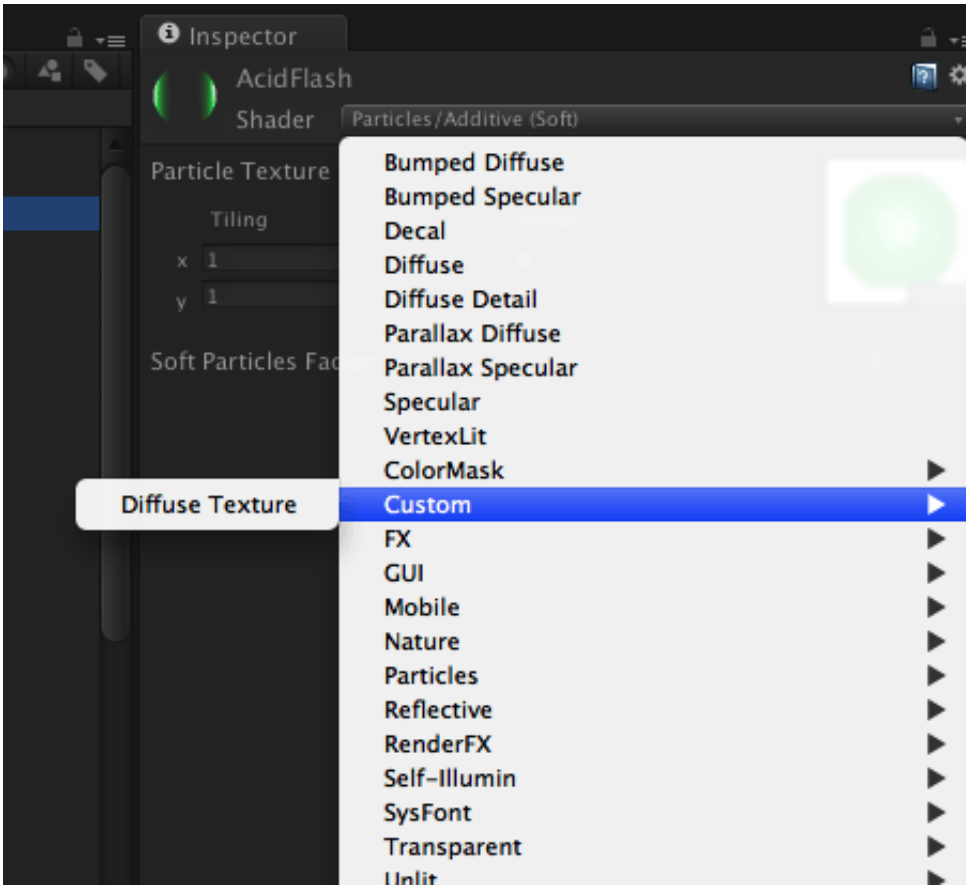
随便用个文本编辑器打开刚才新建的Shader：

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
```

```
17
18
19
20
21
22
23
24
25
26
```

```
Shader "Custom/Diffuse Texture" {
  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"
}
```

如果您之前没怎么看过Shader代码的话，估计细节上会看不太懂。但是有了上面基本结构的介绍，您应该可以识别出这个Shader的构成，比如一个Properties部分，一个SubShader，以及一个FallBack。另外，第一行只是这个Shader的声明并为其指定了一个名字，比如我们的实例Shader，你可以在材质面板选择Shader时在对应的位置找到这个Shader。



在Unity3D中找到刚才新建的Shader

接下来我们讲逐句讲解这个Shader，以期明了每一个语句的意义。

属性

在`Properties{}`中定义着色器属性，在这里定义的属性将被作为输入提供给所有的子着色器。每一条属性的定义的语法是这样的：

```
_Name("Display Name", type) = defaultValue[{options}]
```

- _Name - 属性的名字，简单说就是变量名，在之后整个Shader代码中将使用这个名字来获取该属性的内容
- Display Name - 这个字符串将显示在Unity的材质编辑器中作为Shader的使用者可读的内容
- type - 这个属性的类型，可能的type所表示的内容有以下几种：
  - Color - 一种颜色，由RGBA（红绿蓝和透明度）四个量来定义；
  - 2D - 一张2的阶数大小（256，512之类）的贴图。这张贴图将在采样后被转为对应基于模型UV的每个像素的颜色，最终被显示出来；
  - Rect - 一个非2阶数大小的贴图；
  - Cube - 即Cube map texture（立方体纹理），简单说就是6张有联系的2D贴图的组合，主要用来做反射效果（比如天空盒和动态反射），也会被转换为对应点的采样；
  - Range(min, max) - 一个介于最小值和最大值之间的浮点数，一般用来当作调整Shader某些特性的参数（比如透明度渲染的截止值可以是从0至1的值等）；
  - Float - 任意一个浮点数；
  - Vector - 一个四维数；

- defaultValue 定义了这个属性的默认值，通过输入一个符合格式的默认值来指定对应属性的初始值

（某些效果可能需要某些特定的参数值来达到需要的效果，虽然这些值可以在之后在进行调整，但是如果默认就指定为想要的值的话就省去了一个个调整的时间，方便很多）。

- Color - 以0～1定义的rgba颜色，比如(1,1,1,1)；
- 2D/Rect/Cube - 对于贴图来说，默认值可以为一个代表默认tint颜色的字符串，可以是空字符串或者”white”,”black”,”gray”,”bump”中的一个
- Float，Range - 某个指定的浮点数
- Vector - 一个4维数，写为 (x,y,z,w)

- 另外还有一个{option}，它只对2D，Rect或者Cube贴图有关，在写输入时我们最少要在贴图之后写一对什么都不含的空白的{}，当我们需要打开特定选项时可以把其写在这对花括号内。如果需要同时打开多个选项，可以使用空白分隔。可能的选择有ObjectLinear, EyeLinear, SphereMap, CubeReflect, CubeNormal中的一个，这些都是OpenGL中TexGen的模式，具体的留到后面有机会再说。

所以，一组属性的申明看起来也许会是这个样子的

```
1
2
3
4
```

```
//Define a color with a default value of semi-transparent blue
_MainColor ("Main Color", Color) = (0,0,1,0.5)
//Define a texture with a default of white
_Texture ("Texture", 2D) = "white" {}
```

现在看懂上面那段Shader（以及其他所有Shader）的Properties部分应该不会有任何问题了。接下来就是SubShader部分了。

Tags

表面着色器可以被若干的标签（tags）所修饰，而硬件将通过判定这些标签来决定什么时候调用该着色器。比如我们的例子中SubShader的第一句

`Tags { "RenderType"="Opaque" }`
告诉了系统应该在渲染非透明物体时调用我们。Unity定义了一些列这样的渲染过程，与RenderType是Opaque相对应的显而易见的是`"RenderType" = "Transparent"`，表示渲染含有透明效果的物体时调用。在这里Tags其实暗示了你的Shader输出的是什么，如果输出中都是非透明物体，那写在Opaque里；如果想渲染透明或者半透明的像素，那应该写在Transparent中。

另外比较有用的标签还有`"IgnoreProjector"="True"`（不被Projectors影响），`"ForceNoShadowCasting"="True"`（从不产生阴影）以及`"Queue"="xxx"`（指定渲染顺序队列）。这里想要着重说一下的是Queue这个标签，如果你使用Unity做过一些透明和不透明物体的混合的话，很可能已经遇到过不透明物体无法呈现在透明物体之后的情况。这种情况很可能是由于Shader的渲染顺序

不正确导致的。Queue指定了物体的渲染顺序，预定义的Queue有：

- Background - 最早被调用的渲染，用来渲染天空盒或者背景
- Geometry - 这是默认值，用来渲染非透明物体（普通情况下，场景中的绝大多数物体应该是非透明的）
- AlphaTest - 用来渲染经过Alpha Test的像素，单独为AlphaTest设定一个Queue是出于对效率的考虑
- Transparent - 以从后往前的顺序渲染透明物体
- Overlay - 用来渲染叠加的效果，是渲染的最后阶段（比如镜头光晕等特效）

这些预定义的值本质上是一组定义整数，Background = 1000， Geometry = 2000, AlphaTest = 2450, Transparent = 3000，最后Overlay = 4000。在我们实际设置Queue值时，不仅能使用上面的几个预定义值，我们也可以指定自己的Queue值，写成类似这样：`"Queue"="Transparent+100"`，表示一个在Transparent之后100的Queue上进行调用。通过调整Queue值，我们可以确保某些物体一定在另一些物体之前或者之后渲染，这个技巧有时候很有用处。

LOD

LOD很简单，它是Level of Detail的缩写，在这里例子里我们指定了其为200（其实这是Unity的内建Diffuse着色器的设定值）。这个数值决定了我们能用什么样的Shader。在Unity的Quality Settings中我们可以设定允许的最大LOD，当设定的LOD小于SubShader所指定的LOD时，这个SubShader将不可用。Unity内建Shader定义了一组LOD的数值，我们在实现自己的Shader的时候可以将其作为参考来设定自己的LOD数值，这样在之后调整根据设备图形性能来调整画质时可以进行比较精确的控制。

- VertexLit及其系列 = 100
- Decal, Reflective VertexLit = 150
- Diffuse = 200
- Diffuse Detail, Reflective Bumped Unlit, Reflective Bumped VertexLit = 250
- Bumped, Specular = 300
- Bumped Specular = 400
- Parallax = 500
- Parallax Specular = 600

Shader本体

前面杂项说完了，终于可以开始看看最主要的部分了，也就是将输入转变为输出的代码部分。为了方便看，请容许我把上面的SubShader的主题部分抄写一遍

```
1
2
3
4
5
6
7
```

```
8
9
10
11
12
13
14
15
```

```
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
```

还是逐行来看，首先是CGPROGRAM。这是一个开始标记，表明从这里开始是一段CG程序（我们在写Unity的Shader时用的是Cg/HLSL语言）。最后一行的ENDCG与CGPROGRAM是对应的，表明CG程序到此结束。

接下来是是一个编译指令：`#pragma surface surf Lambert`，它声明了我们要写一个表面Shader，并指定了光照模型。它的写法是这样的

`#pragma surface surfaceFunction lightModel [optionalparams]`

- surface - 声明的是一个表面着色器
- surfaceFunction - 着色器代码的方法的名字
- lightModel - 使用的光照模型。

所以在我们的例子中，我们声明了一个表面着色器，实际的代码在surf函数中（在下面能找到该函数），使用Lambert（也就是普通的diffuse）作为光照模型。

接下来一句`sampler2D _MainTex;`，sampler2D是个啥？其实在CG中，sampler2D就是和texture所绑定的一个数据容器接口。等等..这个说法还是太复杂了，简单理解的话，所谓加载以后的texture（贴图）说白了不过是一块内存存储的，使用了RGB（也许还有A）通道，且每个通道8bits的数据。而具体地想知道像素与坐标的对应关系，以及获取这些数据，我们总不能一次一次去自己计算内存地址或者偏移，因此可以通过sampler2D来对贴图进行操作。更简单地理解，sampler2D就是GLSL中的2D贴图的类型，相应

的，还有sampler1D，sampler3D，samplerCube等等格式。

解释通了sampler2D是什么之后，还需要解释下为什么在这里需要一句对`_MainTex`的声明，之前我们不是已经在`Properties`里声明过它是贴图了么。答案是我们用来实例的这个shader其实是由两个相对独立的块组成的，外层的属性声明，回滚等等是Unity可以直接使用和编译的ShaderLab；而现在我们是在`CGPROGRAM...ENDCG`这样一个代码块中，这是一段CG程序。对于这段CG程序，要想访问在`Properties`中所定义的变量的话，必须使用和之前变量相同的名字进行声明。于是其实`sampler2D _MainTex;`做的事情就是再次声明并链接了_MainTex，使得接下来的CG程序能够使用这个变量。

终于可以继续了。接下来是一个struct结构体。相信大家对于结构体已经很熟悉了，我们先跳过之，直接看下面的的surf函数。上面的#pragma段已经指出了我们的着色器代码的方法的名字叫做surf，那没跑儿了，就是这段代码是我们的着色器的工作核心。我们已经说过不止一次，着色器就是给定了输入，然后给出输出进行着色的代码。CG规定了声明为表面着色器的方法（就是我们这里的surf）的参数类型和名字，因此我们没有权利决定surf的输入输出参数的类型，只能按照规定写。这个规定就是第一个参数是一个Input结构，第二个参数是一个inout的SurfaceOutput结构。

它们分别是什么呢？Input其实是需要我们去定义的结构，这给我们提供了一个机会，可以把所需要参与计算的数据都放到这个Input结构中，传入surf函数使用；SurfaceOutput是已经定义好了里面类型输出结构，但是一开始的时候内容暂时是空白的，我们需要向里面填写输出，这样就可以完成着色了。先仔细看看INPUT吧，现在可以跳回来看上面定义的INPUT结构体了：

```
1
2
3
```

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

作为输入的结构体必须命名为Input，这个结构体中定义了一个float2的变量…你没看错我也没打错，就是float2，表示浮点数的float后面紧跟一个数字2，这又是什么意思呢？其实没什么魔法，float和vec都可以在之后加入一个2到4的数字，来表示被打包在一起的2到4个同类型数。比如下面的这些定义：

```
1
2
3
4
5
6
```

```
//Define a 2d vector variable
vec2 coordinate;
//Define a color variable
```

```
float4 color;
//Multiply out a color
float3 multipliedColor = color.rgb * coordinate.x;
```

在访问这些值时，我们即可以只使用名称来获得整组值，也可以使用下标的方式（比如.xyzw，.rgba或它们的部分比如.x等等）来获得某个值。在这个例子里，我们声明了一个叫做`uv_MainTex`的包含两个浮点数的变量。

如果你对3D开发稍有耳闻的话，一定不会对uv这两个字母感到陌生。UV mapping的作用是将一个2D贴图上的点按照一定规则映射到3D模型上，是3D渲染中最常见的一种顶点处理手段。在CG程序中，我们有这样的约定，在一个贴图变量（在我们例子中是`_MainTex`）之前加上uv两个字母，就代表提取它的uv值（其实就是两个代表贴图上点的二维坐标 ）。我们之后就可以在surf程序中直接通过访问uv_MainTex来取得这张贴图当前需要计算的点的坐标值了。

如果你坚持看到这里了，那要恭喜你，因为离最后成功读完一个Shader只有一步之遥。我们回到surf函数，它的两有参数，第一个是Input，我们已经明白了：在计算输出时Shader会多次调用surf函数，每次给入一个贴图上的点坐标，来计算输出。第二个参数是一个可写的SurfaceOutput，SurfaceOutput是预定义的输出结构，我们的surf函数的目标就是根据输入把这个输出结构填上。SurfaceOutput结构体的定义如下

```
1
2
3
4
5
6
7
8
```

```
struct SurfaceOutput {
    half3 Albedo;     //像素的颜色
    half3 Normal;     //像素的法向值
    half3 Emission;   //像素的发散颜色
    half Specular;    //像素的镜面高光
    half Gloss;       //像素的发光强度
    half Alpha;       //像素的透明度
};
```

这里的half和我们常见float与double类似，都表示浮点数，只不过精度不一样。也许你很熟悉单精度浮点数（float或者single）和双精度浮点数（double），这里的half指的是半精度浮点数，精度最低，运算性能相对比高精度浮点数高一些，因此被大量使用。

在例子中，我们做的事情非常简单：

```
1
2
3
```

```
half4 c = tex2D (_MainTex, IN.uv_MainTex);
o.Albedo = c.rgb;
o.Alpha = c.a;
```

这里用到了一个`tex2d`函数，这是CG程序中用来在一张贴图中对一个点进行采样的方法，返回一个float4。这里对_MainTex在输入点上进行了采样，并将其颜色的rbg值赋予了输出的像素颜色，将a值赋予透明度。于是，着色器就明白了应当怎样工作：即找到贴图上对应的uv点，直接使用颜色信息来进行着色，over。

接下来…

我想现在你已经能读懂一些最简单的Shader了，接下来我推荐的是参考Unity的Surface Shader Examples多接触一些各种各样的基本Shader。在这篇教程的基础上，配合一些google的工作，完全看懂这个shader示例页面应该不成问题。如果能做到无压力看懂，那说明你已经有良好的基础可以前进到Shader的更深的层次了（也许等不到我的下一篇教程就可以自己开始动手写些效果了）；如果暂时还是有困难，那也没有关系，Shader学习绝对是一个渐进的过程，因为有很多约定和常用技巧，多积累和实践自然会进步并掌握。

在接下来的教程里，打算通过介绍一些实际例子以及从基础开始实际逐步动手实现一个复杂一点的例子，让我们能看到shader在真正使用中的威力。我希望能尽快写完这个系列，但是无奈时间确实有限，所以我也不知道什么时候能出炉…写好的时候我会更改这段内容并指向新的文章。您要是担心错过的话，也可以使用邮件订阅或者订阅本站的rss(虽然Google Reader已经关了- -)。