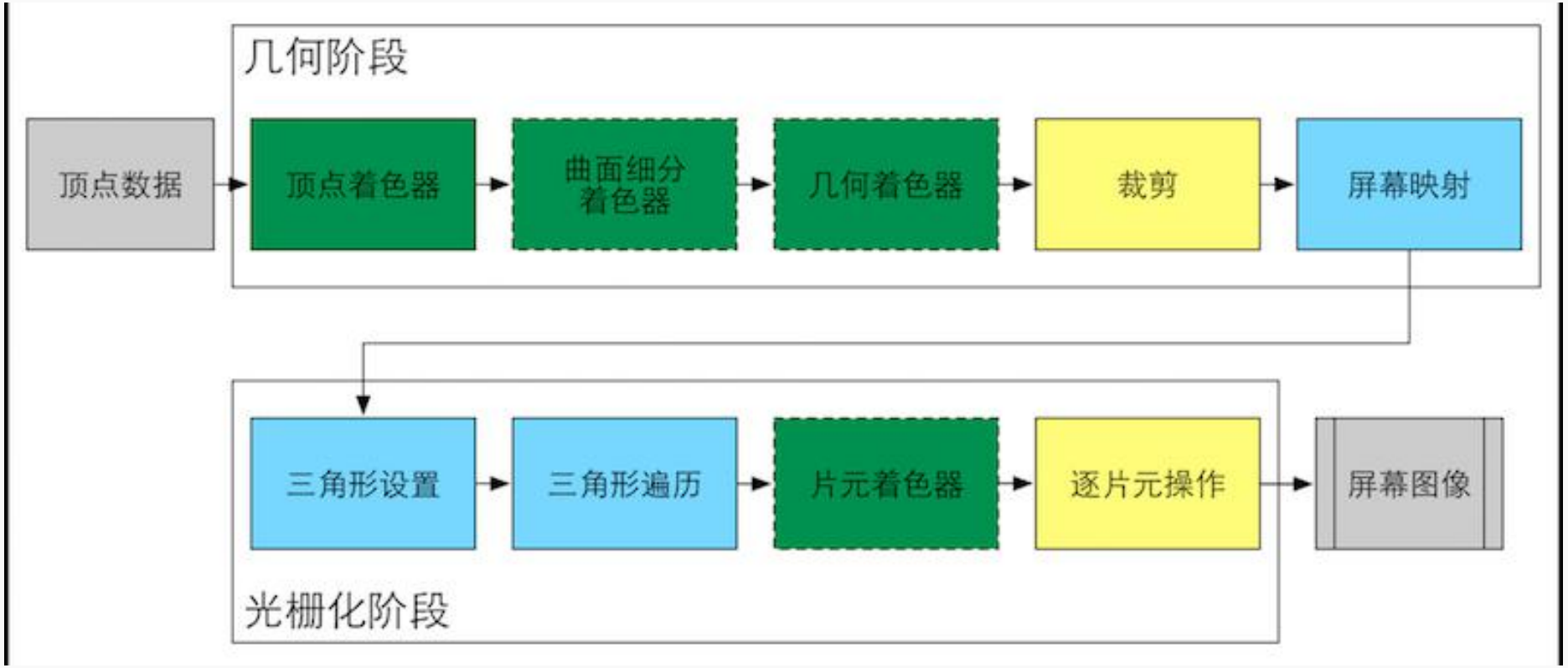
GPU流水线

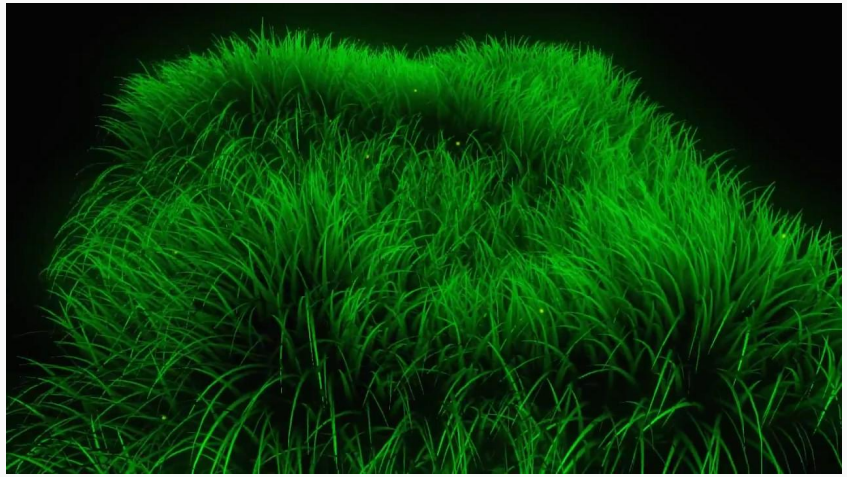


颜色表示了不同阶段的可配置性或可编程性：绿色表示该流水线阶段是完全可编程控制的，黄色表示该流水线阶段可以配置但不是可编程的，蓝色表示该流水线阶段是由GPU固定实现的，开发者没有任何控制权。实线表示该shader必须由开发者编程实现，虚线表示该Shader是可选的

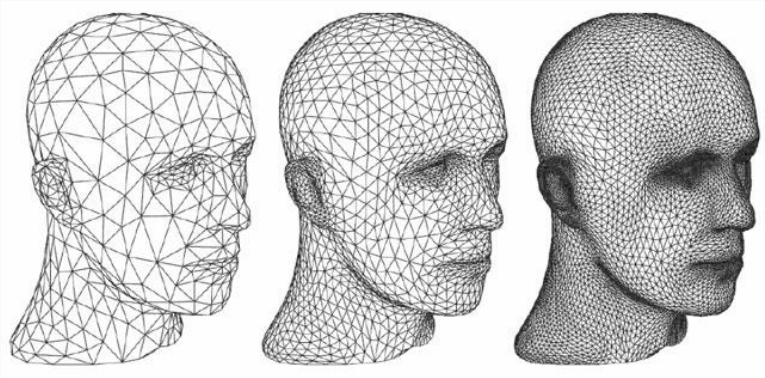
什么是Shader

- Shade一词在英文中表示“遮蔽，阴影”，Shading一词在渲染术语中可以表示“对虚拟场景进行明暗计算，最终在屏幕上显示出相应的着色”。而Shader即可表示用来做这些工作的具体单元，也就是“着色器”；
- GPU流水线上一些高度可编程的阶段，而由着色器编译出来的最终代码是会在GPU上运行的；
- 有一些特定类型的着色器，如顶点着色器，片元着色器等；
- 依靠着色器我们可以控制流水线中的渲染细节，例如用顶点着色器来进行顶点变换以及传递数据，用片元着色器来进行逐像素的渲染。

Shader类型



- 顶点着色器（Vertex Shader）
- 片元着色器（Fragment Shader）
- 曲面细分着色器（Tessellation Shader）
- 几何着色器（Geometry Shader）
- 计算着色器（Compute Shader）
- 特例——Unity专用：表面着色器（Surface Shader）



Unity中的Shader

- Unity Shader不等于真正的Shader，它实际上是Unity专门开发的一种ShaderLab语言文件（后缀名为.shader），比一般意义上的Shader更加容易编写，但是只能在Unity内部进行编译和使用。

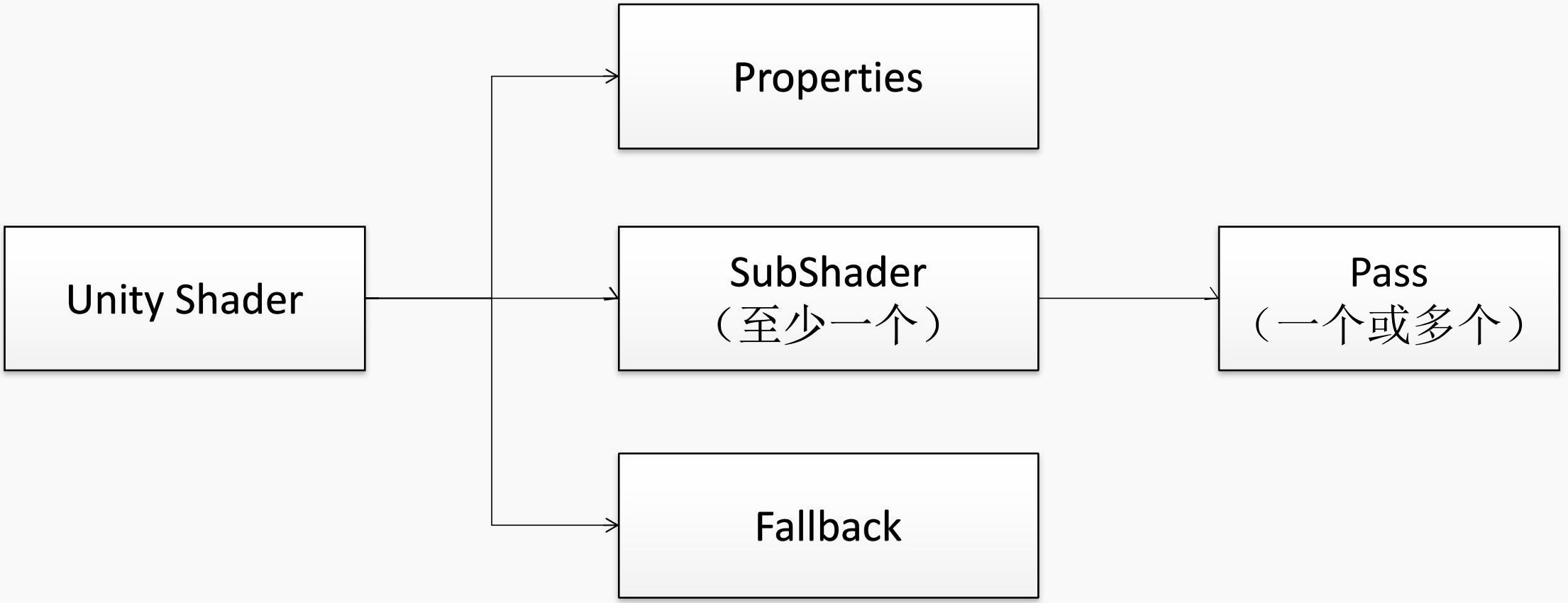
- 尽管它们使用的语法基本差不多，但ShaderLab封装了许多有用的属性和方法，在Unity Shader中几句话就能做完的操作，在传统的Shader里可能要写很长的代码，甚至要分成好几个文件。

Unity中的Shader

- Unity Shader不等于真正的Shader，它实际上是Unity专门开发的一种ShaderLab语言文件（后缀名为.shader），比一般意义上的Shader更加容易编写，但是只能在Unity内部进行编译和使用。

- 尽管它们使用的语法基本差不多，但ShaderLab封装了许多有用的属性和方法，在Unity Shader中几句话就能做完的操作，在传统的Shader里可能要写很长的代码，甚至要分成好几个文件。

顶点/片元着色器基本结构



顶点/片元着色器基本结构

Shader基本结构.txt

```
Shader "ShaderName" {
    Properties {
        //属性
    }
    SubShader {      // 针对显卡A的SubShader
            // [可选]SubShader标签，对所有Pass起效。标签示例：
            Tags {"Queue" = "Transparent" "IgnoreProjector" = "true"}
            // [可选]SubShader渲染状态，对所有Pass起效，也可以放在单个Pass。示例：
            ZWrite Off Cull Off

        Pass {      // Pass，用来进行一次渲染

            // [可选]在这里设置名字、标签、渲染状态

            // 开始CG代码片段
            CGPROGRAM
            // 该代码片段的编译指令，例如：
            #pragma vertex vert
            #pragma fragment frag

            // 这里写你自己的CG的代码

            // 结束代码片段
            ENDCG
        }
    }
    SubShader{      // 针对显卡B的SubShader
    }

    // 上述SubShader都失败的话，用于回调的Unity Shader
    Fallback "VertexLit"
}
```

Properties属性

属性的标准格式：

```
Properties{
        Name { “display name”, PropertyType } = DefaultValue
        Name { “display name”, PropertyType } = DefaultValue
        // 更多属性
}
```

Properties属性

```
Properties {
    // Numbers and Sliders
    _Int ("Int", Int) = 2
    _Float ("Float", Float) = 1.5
    _Range("Range", Range(0.0, 5.0)) = 3.0
    // Colors and Vectors
    _Color ("Color", Color) = (1,1,1,1)
    _Vector ("Vector", Vector) = (2, 3, 6, 1)
    // Textures
    _2D ("2D", 2D) = "" {}
    _Cube ("Cube", Cube) = "white" {}
    _3D ("3D", 3D) = "black" {}
}
```

SubShader

```
SubShader {
    [Tags]
    [RenderSetup]

    Pass {
    }

    // Other Pass
}
```

- 每个Unity Shader文件可以包含多个SubShader块，但最少要有一个。当Unity需要加载这个Unity Shader时，Unity会扫描所有的SubShader语义块，然后选择第一个能够在目标平台上运行的SubShader。如果都不支持的话，Unity就会使用Fallback语义指定的Shader。

Tags（可选的）

标签的标准格式：

Tags { "TagName1" = "Value1" "TagName2" = "Value2" }

标签类型	说明
Queue	定义渲染顺序： （1）“Background“。值为1000。比如用于天空盒。 （2）”Geometry”。值为2000。大部分物体在这个队列。不透明的物体也在这里。这个队列内部的物体的渲染顺序会有进一步的优化（应该是从近到远，early-z test可以剔除不需经过FS处理的片元）。其他队列的物体都是按空间位置的从远到近进行渲染。 （3）“AlphaTest”。值为2450。已进行AlphaTest的物体在这个队列。 （4）”Transparent“。值为3000。透明物体。 （5）“Overlay”。值为4000。比如镜头光晕。 （6）用户可以定义任意值，比如“Queue“=”Geometry+10”

Tags（可选的）

标签类型	说明
RenderType	Unity可以运行时替换符合特定RenderType的所有Shader。与Camera.RenderWithShader或者Camera.SetReplacementShader配合使用。 （1）“Opaque”：绝大部分不透明的物体都使用这个； （2）“Transparent”：绝大部分透明的物体、包括粒子特效都使用这个； （3）”Background“：天空盒都使用这个； （4）“Overlay”：GUI、镜头光晕都使用这个； （5）还有其他可参考Rendering with Replaced Shaders；用户也可以定义任意自己的RenderType字符串。
DisableBatching	一些SubShader在使用Unity的批处理功能时会出现问题，例如使用了模型空间下的坐标进行顶点动画。可以通过该标签来直接知名是否对该SubShader进行批处理：“true”、“false”。
ForceNoShadowCasting	该SubShader是否会投射阴影：“true”、“false”。
IgnoreProjector	物理是否受Projector（Unity 的一个功能）影响，通常用于半透明物体：“true”、“false”。
CanUseSpriteAtlas	当该SubShader用于精灵（Sprites），将该标签设为“false”。
PreviewType	材质面板默认是一个球形预览，可以设置为“Plane”、“Skybox”来改变预览类型

RenderSetup（可选的）

渲染状态类型	设置指令	说明
Cull	Cull Back | Front | Off	设置剔除模式：剔除背面/正面/关闭剔除
ZTest	ZTest Less Greater | LEqual | GEqual | Equal | NotEqual | Always	设置深度测试时使用的函数
ZWrite	ZWrite On | Off	设置深度测试时使用的函数
Blend	Blend SrcFactor DstFactor	开启并设置混合模式

Pass

```
Pass{
    [Name]
    [Tags]
    [RenderSetup]

    // Other code
}
```

- 可以定义Pass的名称，例如：Name “MyPassName”，在其他Shader里就可以用Use Pass命令来使用这个Pass，提高复用性。
- Pass的渲染状态和SubShader的渲染状态语法一样，可以对Pass单独设置渲染状态。.
- Pass的标签和SubShader的标签不一样，只有两种。

Pass

标签类型	说明
LightMode	定义该Pass在Unity的渲染流水线中的角色： Always: 永远都渲染，但不处理光照 ForwardBase: 用于前向渲染，只有主光源、顶点光源/SH光源和Lightmap起作用 ForwardAdd: 用于前向渲染，额外的的逐像素光源也会起作用，每个光源需要进行一次Pass Deffered: 用于延迟光照，会把信息渲染到G-buffer里 ShadowCaster: 把物体的深度信息渲染到ShadowMap或者深度纹理中
RequireOptions	满足某些条件时才渲染该Pass，目前只有一个选项：“SoftVegetation”，即只有当 Quality Settings中的Soft Vegetation打开才渲染该Pass

Fallback

- 如果Shader中的所有SubShader都不能运行，那么会使用Fallback指定的Shader来渲染。

```
Fallback “ShaderName”
// 或者
Fallback Off
```

- Fallback还会影响阴影的投射。在渲染阴影纹理时，Unity会在每个Unity Shader中寻找一个阴影投射的Pass，如果Fallback中的Shader包含了这样的Pass，那么我们就不需要自己专门实现一个阴影Pass了。

一个最简单的着色器


SimpleShader.txt

```
Shader "Custom/Simple Shader" {
    SubShader {
        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            float4 vert(float4 v : POSITION) : SV_POSITION {
                return mul (UNITY_MATRIX_MVP, v);
            }

            fixed4 frag() : SV_Target {
                return fixed4(1.0, 1.0, 1.0, 1.0);
            }

            ENDCG
        }
    }
}
```

一个最简单的着色器

```
1  Shader "Custom/Simple Shader" {
2      SubShader {
3          Pass {
4              CGPROGRAM
5
6              #pragma vertex vert
7              #pragma fragment frag
8
9              float4 vert(float4 v : POSITION) : SV_POSITION {
10                 return mul (UNITY_MATRIX_MVP, v);
11             }
12
13             fixed4 frag() : SV_Target {
14                 return fixed4(1.0, 1.0, 1.0, 1.0);
15             }
16
17             ENDCG
18         }
19     }
20 }
```

由CGPROGRAM和ENDCG所包围的CG代码片段，是Shader的重点。首先，我们遇到两行重要的编译指令：

#pragma vertex name1

#pragma fragment name2

它们将告诉Unity，哪个函数包含了顶点着色器的代码，哪个函数包含了片元着色器的代码。其中name1和name2可以自己命名，在本例中被写成：

#pragma vertex vert

#pragma fragment frag

一个最简单的着色器

```
1  Shader "Custom/Simple Shader" {
2      SubShader {
3          Pass {
4              CGPROGRAM
5
6              #pragma vertex vert
7              #pragma fragment frag
8
9              float4 vert(float4 v : POSITION) : SV_POSITION {
10                 return mul (UNITY_MATRIX_MVP, v);
11             }
12
13             fixed4 frag() : SV_Target {
14                 return fixed4(1.0, 1.0, 1.0, 1.0);
15             }
16
17             ENDCG
18         }
19     }
20 }
```

这是顶点着色器代码，它是逐顶点执行的。vert函数的输入v包含了这个顶点的位置信息，这是通过POSITION语义指定的。它的返回值是一个float4类型的变量，它是该顶点在模型空间中的位置。

POSITION和SV_POSITION都是CG/HLSL语言中的语义，它们是不可省略的，这些语义将告诉系统，用户需要哪些输入，以及用户的输出是什么类型。例如这里，POSITION将告诉Unity，把模型的顶点坐标填充到v里面。SV_POSITION将告诉Unity，顶点着色器的输出是裁剪空间中的顶点坐标。

一个最简单的着色器

```
1  Shader "Custom/Simple Shader" {
2      SubShader {
3          Pass {
4              CGPROGRAM
5
6              #pragma vertex vert
7              #pragma fragment frag
8
9              float4 vert(float4 v : POSITION) : SV_POSITION {
10                 return mul (UNITY_MATRIX_MVP, v);
11             }
12
13             fixed4 frag() : SV_Target {
14                 return fixed4(1.0, 1.0, 1.0, 1.0);
15             }
16
17             ENDCG
18         }
19     }
20 }
```

这是片段着色器的代码，本例中，它没有任何输入，它的输出是一个fixed4类型的变量，并且使用了CG/HLSL中的SV_Target语义进行限定，相当于告诉渲染器，把用户的输出颜色存储到一个渲染目标（render target）中，这里将输出到默认的帧缓存中。代码只有一句话，就是返回了一个表示白色的fixed4变量。

数值类型

CG/HLSL的数值类型	精度
float	32位浮点数，通常用于超精度数据和位置信息的存储
half	16位浮点数，范围-60000～+60000，通常用于位置信息
fixed	11位浮点数，范围-2.0～+2.0，通常用于颜色和单位矢量

- 例如：float4表示一个4维的32位浮点数数组（向量）
- 尽可能使用精度较低的类型，可以优化性能，对于移动平台尤其重要

模型提供的信息还可以更多吗？

- 刚才的代码中，只有顶点位置被传入到vert函数中，有时候我们想要模型上每个顶点的法线方向，纹理坐标等信息，那怎么办呢？
- Unity Shader提供了一种方式，可以将一个结构体传入到vert函数里面，定义如下（注意末尾有分号）：

```
struct a2v{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0;
};
```

- Unity支持的语义有： POSITION, TANGENT, NORMAL, TEXCOORD0, TEXCOORD1, TEXCOORD2, TEXCOORD3, COLOR等。

模型提供的信息还可以更多吗？

- 定义好a2v结构体后，把vert函数的输入参数改成：

```
float4 vert(a2v v) : SV_POSITION {
    return mul (UNITY_MATRIX_MVP, v.vertex);
}
```

- 在Unity中，这些数据都是从材质的Mesh Render组件里拿到的，在每帧调用Draw Call的时候，Mesh Render组件会把它负责渲染的模型数据发给Unity Shader，例如顶点位置，切线，法线，纹理坐标，顶点颜色等。

vert和frag之间的通信

- 刚才的片元着色器只是把模型全部填充成白色，没有用到除顶点位置之外的任何信息，我们希望把法线、纹理坐标等数据传递给片元着色器，为此需要定义一个新的结构体：

```
struct v2f {
    float4 pos : SV_POSITION;
    fixed3 color : COLOR0;
};
```

- 输出结构中必须包含一个变量，它的语义是SV_POSITION，否则，渲染器将无法得到裁剪空间中的顶点坐标
- COLOR0语义中的数据可以由用户自己定义，但一般都是存储颜色。

如何使用Properties

- 在Properties语义块中定义的属性，为了能在后续的CG代码中访问它，我们需要在CG代码片段中提前定义一个变量，这个变量的名称和类型必须与Properties语义块中的属性定义相匹配。

```
Properties {
    _Color ("Color Tint", Color) = (1, 1, 1, 1)
}
SubShader {
    Pass {
        CGPROGRAM

        #pragma vertex vert
        #pragma fragment frag

        fixed4 _Color;
```

如何使用Properties

ShaderLab属性类型	**CG变量类型**
Color, Vector	float4, half4, fixed4
Range, Float	float, half, fixed
2D	sampler2D
Cube	samplerCube
3D	sampler3D

Unity提供的内置文件

- 包含文件（include file）是类似于C++头文件的一种文件，在Unity中，它们的文件后缀是.cginc，可以从官网下载。
- 使用include指令把这些文件包含在CG片段中，就可以使用内置的变量和函数。

```
CGPROGRAM
// ...
#include "UnityCG.cginc"
// ...
ENDCG
```

Unity提供的内置文件

常用包含文件

文件名	描述
UnityCG.cginc	包含了最常使用的函数、宏和结构体等
UnityShaderVariables.cginc	在编译Unity Shader时，会被自动包含进来。包含了许多内置的全局变量，如UNIYT_MATRIX_MVP
Lighting.cginc	包含了各种内置的光照模型，如果编写的是Surface Shader，会自动包含
HLSLSupport.cginc	在编译Unity Shader时，会自动包含，声明了许多用于跨平台编译的宏和定义

Unity提供的内置文件

- UnityCG.cginc是我们最接触的一个包含文件，该文件中有许多常用的结构体和函数，例如，我们可以直接使用里面预定义的结构体作为顶点着色器的输入和输出，而不需要自己去定义a2v。

UnityCG.cginc中一些常用的结构体

名称	描述	包含的变量
appdata_base	用于顶点着色器的输入	顶点位置、顶点法线、第一组纹理坐标
appdata_tan	用于顶点着色器的输入	顶点位置、顶点切线、顶点法线、第一组纹理坐标
appdata_full	用于顶点着色器的输入	顶点位置、顶点切线、顶点法线、四组（或更多）纹理坐标
appdata_img	用于顶点着色器的输入	顶点位置、第一组纹理坐标
v2f_img	用于顶点着色器的输出	裁剪空间中的位置、纹理坐标

Unity提供的内置文件

UnityCG.cginc中一些常用的函数

函数名	描述
float3 WorldSpaceViewDir(float4 v)	输入一个模型空间中的顶点位置，返回世界空间中从该点到摄像机的观察方向
float3 ObjSpaceViewDir(float4 v)	输入一个模型空间中的顶点位置，返回模型空间中从该点到摄像机的观察方向
float3 WorldSpaceLightDir(float4 v)	**仅可用于前向渲染中。**输入一个模型空间中的顶点位置，返回世界空间中从该点到光源的光照方向，没有被归一化
float3 ObjSpaceLightDir(float4 v)	**仅可用于前向渲染中。**输入一个模型空间中的顶点位置，返回模型空间中从该点到光源的光照方向，没有被归一化
float3 UnityObjectToWorldNormal(float3 norm)	把法线方向从模型空间转换到世界空间中
float3 UnityObjectToWorldDir(float3 dir)	把方向矢量从模型空间转换到世界空间中
float3 UnityWorldToObjectDir(float3 dir)	把方向矢量从世界空间转换到模型空间中

Unity支持的语义

从应用阶段传递模型数据给顶点着色器时Unity支持的常用语义

语义	描述
POSITION	模型空间中的顶点位置，通常是float4类型
NORMAL	顶点法线，通常是float3类型
TANGENT	顶点切线，通常是float4类型
TEXCOORDn, 如TEXCOORD0、TEXCOORD1	该点的纹理坐标，TEXCOORD0表示第一组纹理坐标，以此类推。通常是float2或者float4类型
COLOR	顶点颜色，通常是fixed4或float4类型

Unity支持的语义

从顶点着色器传递数据给片元着色器时Unity使用的常用语义

语义	描述
SV_POSITION	裁剪空间中的顶点坐标，结构体中必须包含一个用该语义修饰的变量
COLOR0	通常用于输出第一组顶点颜色，但不是必需的
COLOR1	通常用于输出第二组顶点颜色，但不是必需的
TEXCOORD0～TEXCOORD7	通常用于输出纹理坐标，但不是必需的

- 从vert传到frag这一过程中，除了SV_POSITION以外，其他语义对变量的含义没有明确要求，可以存储任意值在这些变量里。通常，如果需要把一些自定义的数据进行传递时（比如说传递高光uv和法线uv），会选用TEXCOORD0。

Unity支持的语义

片元着色器输出时Unity支持的常用语义

语义	描述
SV_Target	输出值将会存储到渲染目标（render target）中

Unity支持的语义

- 一个语义最多可以定义成float4，可以根据自己的情况来定义更小的精度类型，例如：

```
struct v2f {
    float4 pos : SV_POSTION;
    fixed3 color0 : COLOR0;
    fixed4 SpecularColor : COLOR1;
    half value0 : TEXCOORD0;
    float2 value1 : TEXCOORD1;
};
```