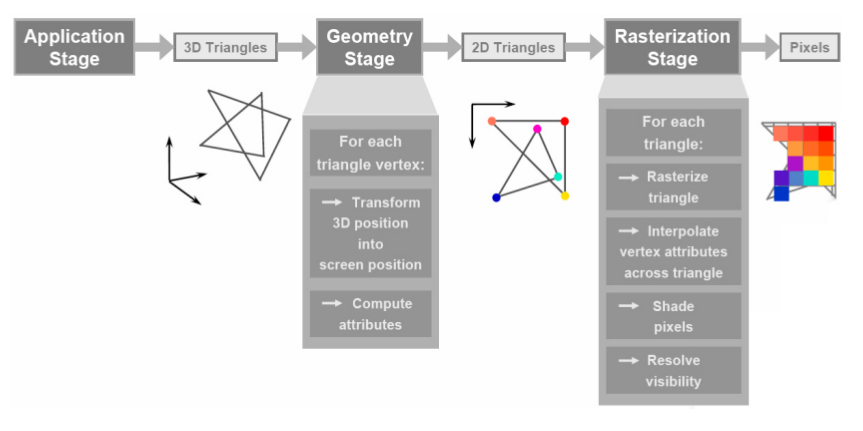
# **图形渲染基础**

作者：[Never Say Never](http://gad.qq.com/user/index?id=2273787)

链接：<http://gad.qq.com/article/detail/287648>

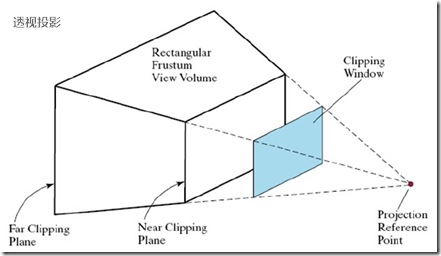
学习图形渲染必须先了解渲染管线，《Render-Time Rendering，Third Edition》书将渲染管线分成三个阶段：应用阶段、几何阶段、光栅化阶段，如下图所示。  


**应用阶段**  
应用阶段的主要任务是准备好场景数据（包括场景中的模型、光源等等）；剔除不可见物体，减少不必要的计算；设置物体的渲染状态（物体使用的材质、shader等等）。最终，应用阶段将输出渲染图元（用于渲染物体的几何信息），并传递给几何阶段。

**几何阶段**  
几何阶段的主要任务是把顶点坐标从模型空间变换到屏幕空间，然后交给光栅器进行处理。几何阶段的可编程阶段是**顶点着色器**，顶点坐标变换过程如下图所示。  
  
模型空间也被称为对象空间，是模型自己独立的空间，根据对象空间的坐标系确定模型各顶点的相对坐标。***模型空间***使用的坐标系是左手坐标系。

顶点坐标空间变换的第一步是将物体从模型空间变换到世界空间。***世界空间***可以说是场景中所有物体存在的统一空间，不同物体之间根据其在世界空间的坐标来确定相对位置。世界空间的坐标系是也左手坐标系。从模型空间变换到世界空间的变换通常叫做模型变换。

接下来顶点从世界空间变换到***观察空间***，这个变换通常叫做观察变换。观察空间也被称为摄像机空间，即以摄像机为坐标原点的度量空间。需要注意的是观察空间使用的坐标系是右手坐标系，而且摄像机的正前方指向的是坐标系的-Z轴方向。

***剪裁空间***顾名思义就是对渲染图元进行剪裁。保留位于视椎体内的图元，剔除视椎体以外的图元（摄像机看不到的图元）。视椎体是由六个平面包围成的六面体，每个面都是剪裁平面。  
  
根据投影类型不同，视椎体也有两种类型。第一种是基于透视投影的视椎体，是一种符合人眼所看到景物的规律，即近大远小。这种视椎体呈现的是金字塔形，但其顶端被一个剪裁平面削平（根据摄像机定义的远近剪裁面呈现）。另一种是基于正交投影（平行投影）的视椎体，它不符合近大远小的规律，是一个长方体形状。基于正交投影的视椎体一般在需要3D游戏的小地图或者2D游戏中应用。

从观察空间到剪裁空间的变换叫做投影变换，需要注意的是剪裁空间也是左手坐标系。最后的步骤是将顶点从剪裁空间变换到屏幕空间，这种变换被称为是屏幕映射，变换之后我们将得到像素在屏幕的真正位置。屏幕空间就是我们经常看到的屏幕二维空间，屏幕映射就是将顶点从剪裁空间映射投影到屏幕空间中。屏幕空间也是左手坐标系。

**光栅化阶段**  
光栅化阶段主要是决定每个渲染图元中的哪些图元会被绘制在屏幕上，它将几何阶段传过来的三角形转化为片段，并对逐顶点数据进行插值，再进行逐片元处理。

*片元着色器*是光栅化阶段中一个非常重要的可编程着色阶段，它的输入是上一个阶段对顶点信息插值得到的结果，输出是该片元的颜色值。这一阶段完成很多重要的渲染技术，包括纹理采样，光照计算等等。

*逐片元操作*是渲染管线的最后一步，它主要解决每个片元的可见性问题。其中包括很多个复杂的测试，比如深度测试（过滤掉被遮挡的片元）和模板测试等等。只有通过了所有测试的片元才能把其颜色值和颜色缓冲区中的颜色进行合并，并最终绘制到屏幕上。逐片元操作阶段是可配置的。

下面介绍一个非常基础的使用unity的Shaderlab渲染开发示例：

Shader "Unity Shaders/Simple Shader"｛

Properties｛

\_Color ("Color Tint", Color) = (1.0, 1.0, 0, 1.0)

//\_MainTex ("Texture", 2D) = "white" ｛｝

｝

SubShader｛

Pass｛

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

fixed4 \_Color;

struct a2v｛

float4 vertex : POSITION;

float3 normal : NORMAL;

float4 texcoord : TEXCOORD0;

｝;

// vertex shader output

struct v2f ｛

float4 pos : SV\_POSITION;

fixed3 color : COLOR0;

//float2 uv : TEXCOORD0;

｝;

v2f vert(a2v v )｛

v2f o;

o.pos = mul(UNITY\_MATRIX\_MVP, v.vertex);

o.color = v.normal \* 0.5 + fixed3(0.5, 0.5, 0.5);

return o;

｝

fixed4 frag(v2f i): SV\_Target｛

fixed3 c = i.color;

c \*= \_Color.rgb;

return fixed4(c, 1.0);

｝

ENDCG

｝

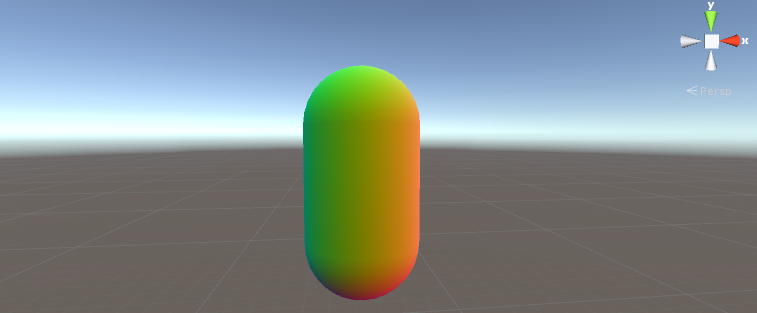
｝

｝

上面的代码中vert函数和frag函数分别对应了**顶点着色器**和**片元着色器**。顶点着色器函数的输入是一个顶点的结构体，它的任务是对顶点进行顶点空间变换和顶点颜色的计算，它的返回值是关联顶点着色器和片元着色器的“插值后的片元”结构体。片元着色器的输入是由顶点着色器传递过来的“插值后的片元”结构体，进行颜色计算后返回该片元（像素）的颜色值。

需要注意的是，顶点着色器中使用了 UNITY\_MATRIX\_MVP宏，它是模型、观察和投影变换的联合矩阵，直接将顶点从模型空间变换到剪裁空间。顶点的颜色值用的是顶点“规范化”的法线值。因为颜色值范围是【0到1】，法线值范围是【-1到1】，通过公式 normal\*0.5+0.5 就能将法线值规范到颜色值范围内，进行正确的颜色输出。

上面的实例效果图如下所示：



对于在渲染的过程中可能发生的性能问题例举如下：

1. 在片元着色器中包含大量复杂的计算，导致游戏运行缓慢。

2. 游戏场景中包含许多光源，并且包含实时光照计算，导致游戏性能下降。

3. 纹理贴图分辨率过高，占用大量的计算资源和内存，导致游戏性能下降。

针对以上可能出现的性能问题，提出以下几点方法建议：

1. 片元着色器和顶点着色器的计算不在同一个量级，要尽量多的把计算放在顶点着色器中进行。

2. 尽可能少地使用光源，多使用烘焙技术和光照贴图，这样会大大提高游戏性能。

3. 适当降低纹理贴图的分辨率，权衡好游戏画质和游戏性能。

4. 在编写Shader时，尽量避免使用分支、循环等流程控制语句。因为渲染管线中的计算基本都是在GPU中完成的，GPU的优势是并行计算，而流程控制语句很大程度上则会降低GPU的并行处理操作。