

# Unity Shader\_2-----Learn from QianMo & CandyCat-----

主要参考：《Unity shader 入门精要 著冯乐乐》

作者博客 <http://blog.csdn.net/candycat1992/article/details/50167285>

浅墨博客 [http://blog.csdn.net/poem\\_qianmo/article/details/49556461](http://blog.csdn.net/poem_qianmo/article/details/49556461)

风宇冲博客 [http://blog.sina.com.cn/s/blog\\_471132920101d5kh.html](http://blog.sina.com.cn/s/blog_471132920101d5kh.html)

潜水的小懒猫博客[http://blog.sina.com.cn/s/articlelist\\_2312702844\\_6\\_1.html](http://blog.sina.com.cn/s/articlelist_2312702844_6_1.html)

猫都能学会的Unity3D Shader<https://onevcat.com/2013/07/shader-tutorial-1/>

<https://onevcat.com/2013/08/shader-tutorial-2/>

蛮牛教育Shader编程教程<http://edu.manew.com/course/96>

Unity官方;<http://docs.unity3d.com/Manual/SL-ShaderPrograms.html>

## 一、渲染流水线：

- 主要任务：由一个三维场景出发、生成（或者渲染）一张二维图像；

即计算机从模型等三维物体的一系列的顶点数据、纹理等信息出发，把这些信息最终转换成 一张人眼可见的图像。

概念上，可以将渲染流水线分为 3 个阶段：

- 应用阶段（CPU上进行，主要输出为渲染图元，传递给GPU进行下一步操作，CPU操作有准备场景数据阶段、提高渲染性能（culling剔除）、设置模型渲染状态）
- 几何阶段（GPU上进行，接收渲染图元，进行逐顶点和逐多边形的操作，主要任务是顶点坐标变换到屏幕空间，再由光栅器处理；主要输出为输出屏幕空间的二维顶点坐标、每个顶点对应的深度值、着色等相关信息）
- 光栅化阶段（GPU上进行，主要任务为决定每个渲染图元中的那些像素应该被绘制在屏幕上，即接收几何阶段的输出数据，对这些逐顶点数据（如纹理坐标、顶点颜色等）进行插值变换，然后进行逐像素处理）

上述渲染流水线也分为 2 个阶段：

- CPU和GPU通信，起点为CPU

1、把数据加载到显存（硬盘—>RAM（系统内存）—>VRAM（显存））

说明：显卡一般没有对内存的访问权限，但对显存有较快的访问速度；从硬盘加载数据到RAM是比较耗时的操作

2、设置渲染状态

渲染状态：使用何种着色器（顶点着色器（Vertex Shader）、片元着色器（Fragment Shader））、光照属性、材质等

3、调用DrawCall

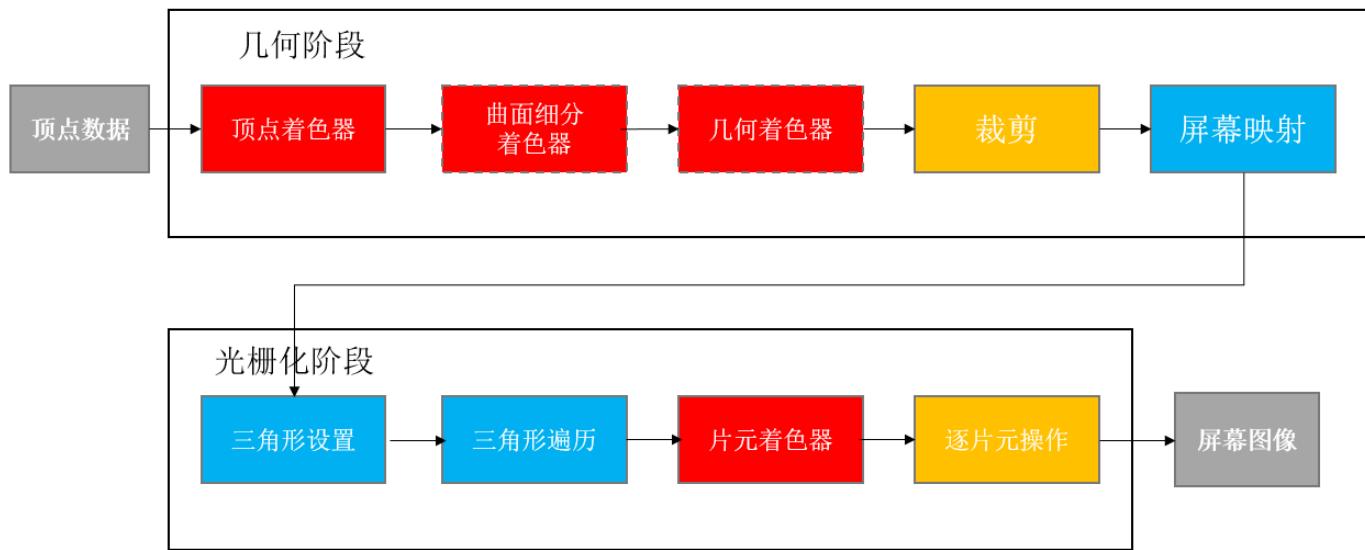
DrawCall：一个命令，有CPU发起，GPU接收该命令，GPU可以开始一个渲染过程，即进行下一阶段的GPU渲染流水线

注意：该命令仅指向一个需要被渲染的图元列表，不包括任何材质信息；

- GPU流水线（又称渲染流水线或者渲染管线，不同于上述概念性渲染流水线）

当GPU从CPU那得到渲染命令后，就会进行一系列流水线操作，最终把图元渲染到屏幕上；

这些流水线操作开发者无法拥有绝对的控制权，但开放了很多控制权。



如上图所示：颜色表示了不同阶段的可配置性或可编程性：

红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限。

实现表示该Shader必须由开发者编程实现，

虚线表示该Shader是可选的，

一般讲片元着色器默认为编程实现。

具体描述上图所示的GPU渲染管线阶段的

- 顶点着色器

#### 高度可编程

顶点着色器是渲染管线的第一个阶段，从CPU获取输入，处理单位为顶点，即输入进来的每个顶点都可以调用一次顶点着色器；

顶点着色器本身不能创建或者销毁顶点，而且也无法得到顶点与顶点间的关系，如无法得知两个顶点是否属于同一三角网格；由于该**相互独立的特性**，GPU可以利用本身的特性**并行化处理**每个顶点，大幅度提高处理速度。

顶点着色器主要需要完成工作有：坐标变换和逐顶点光照。

**坐标变换**：对顶点坐标进行相应的坐标系转换（**模型空间——>世界空间——>观察空间（相机空间）——>齐次裁剪空间（为裁剪做准备）**），主要通过与变换矩阵相乘实现（Unity内置了一些变换矩阵：如

UNITY\_MATRIX\_MVP（当前的模型·观察·投影矩阵，用于将顶点/方向矢量从模型空间变换到裁剪空间），使用方式如：“o.pos = mul(UNITY\_MATRIX\_MVP, v.vertex);”）

更多内置变换及其他内置属性请参见：

<https://docs.unity3d.com/Manual/SL-UnityShaderVariables.html>

# Transformations

All these matrices are `float4x4` type.

| Name                            | Value                                     |
|---------------------------------|---|
| <code>UNITY_MATRIX_MVP</code>   | Current model * view * projection matrix. |
| <code>UNITY_MATRIX_MV</code>    | Current model * view matrix.              |
| <code>UNITY_MATRIX_V</code>     | Current view matrix.                      |
| <code>UNITY_MATRIX_P</code>     | Current projection matrix.                |
| <code>UNITY_MATRIX_VP</code>    | Current view * projection matrix.         |
| <code>UNITY_MATRIX_T_MV</code>  | Transpose of model * view matrix.         |
| <code>UNITY_MATRIX_IT_MV</code> | Inverse transpose of model * view matrix. |
| <code>_Object2World</code>      | Current model matrix.                     |
| <code>_World2Object</code>      | Inverse of current world matrix.          |

逐顶点光照：

可以在顶点着色器中使用逐顶点光照方式实现标准光照模型（自反光、高光反射、漫反射、环境光）；同样的在接下来片元着色器中也有逐像素光照方法来实现光照模型

可以在顶点着色器中修改顶点颜色，顶点位置变换并输出给后续阶段

- 曲面细分着色器

细分图元

- 几何着色器

用于执行逐图元的着色操作，或者被用于产生更多的图元

- 裁剪

裁剪掉不在摄像机视野内的顶点，并剔除默写三角图元的面片；

该阶段是可配置的，如可以自定义裁剪平面来配置裁剪区域，可以通过指令控制裁剪的三角图元的正面还是背面等

- 屏幕映射

不可编程和配置，负责把每个图元的坐标（x和y）转换到屏幕（屏幕坐标系）上；

注：这些坐标均在裁剪空间下

OpenGL 和 DirectX 的屏幕坐标的区别：

OpenGL以左下角为坐标原点 (0,0)    DirectX以左上角为坐标原点 (0,0)

- 三角形设置

## 固定函数阶段

开始进入光栅化(把顶点数据转换为片元的过程)阶段，该阶段会计算光栅化一个三角网格所需的信息，为了得到每个三角网格对像素的覆盖情况，必须计算每条边上的像素坐标，此时就需要得到三角形边界的表示方式  
=====》 这样一个计算三角网格表示数据的过程就叫做三角形设置，并输出

- 三角形遍历 (也称为扫描变换)

## 固定函数阶段

检查每个像素是否被一个三角网格所覆盖，若覆盖，则生成一个对应的片元，这个过程就是三角形遍历

注意：片元不仅仅是像素，而是包含了很多状态的集合，用于计算每个像素的最终颜色；如屏幕坐标、深度信息，以及其他从几何阶段输出的顶点信息（法线和纹理坐标等）

- 片元着色器

## 高度可编程的

输入 上一阶段对顶点信息插值（逼近取近似值）得到的结果；

插值：用来填充图像变换时像素之间的空隙。

主要实现：（shader 中默认定义为 texcoord0/1/2/3... 参数）纹理采样：获取顶点着色器中输出的顶点对应的纹理坐标，该坐标再经过光栅化对三角网格的3个顶点对应的纹理坐标进行插值，最终得到其覆盖像素的片元的纹理坐标

注意：片元着色器仅影响单个片元

- 逐片元操作

## 高度可配置的，无法修改底层，可控制开启关闭功能

主要任务：1、决定每个片元的可见性，包含所有测试工作，例如深度测试、模板测试等

2、如果一个片元通过了所有的测试，就需要把这个片元的颜色值和已经存储在颜色缓冲区中的颜色进行合并，或者说混合

简单来说：

片元一般经过 模板测试（通常用于限制渲染区域，和渲染阴影、轮廓渲染等）将通过测试的片元储存到模板缓冲区，下一步进行深度测试（把当前片元与深度缓冲区中的片元进行深度值得比较），接下来可以进行合并操作（混合操作，涉及到透明物体和不透明物体的颜色值合并（如对透明通道的相加、相减、相乘等等）），最终得到混合后的图像

注意：针对半透明物体的性能下降原因：现在GPU已经是现在将深度测试放在片元着色器之前（称为 Early-Z），从而更早的剔除无法通过深度测试的片元以节省渲染时间；

然而由于透明物体需要开启透明度测试以达到实际透明效果，需要在片元着色器中对透明度进行测试（透明度测试），无法提前进行深度测试，从而导致渲染性能的下降

- 总结：

上述GPU渲染管线/流水线各阶段，在Unity中封装了很多功能，更多时候我们只需要为一个shader设置一些输入，编写顶点着色器和片元着色器，或者是Unity的表面着色器（核心还是顶点和片元着色器的变形），就能实现效果

---

下面给出一个

Vertex shader & fragment shader 大致形式

Shader "Inspect面板显示名称/子菜单名称" {

```

Properties{
    _Color("Inspect中的Shader面板显示名称", Color) = (1,1,1,1)
    .....
}

SubShader{
    pass{
        CGPROGRAM
        //定义顶点着色器和片元着色器的函数名
        #pragma vertex vert
        #pragma fragment frag
        //引用后缀为 .cginc 的文件 包含了一些常用的函数和宏
        #include "unitycg.cginc"

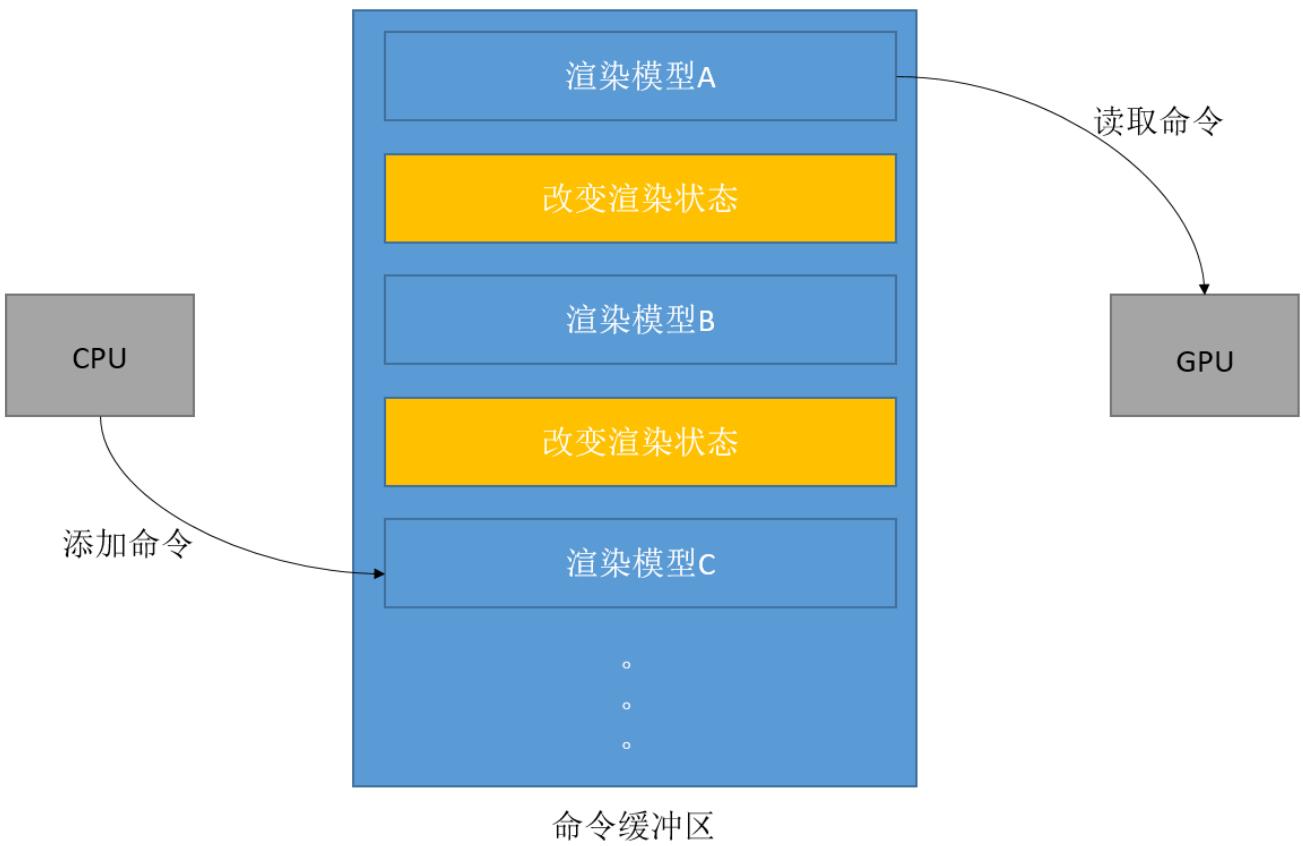
        //声明结构体
        struct v2f{
            float4 pos : POSITION;
            fixed4 color : COLOR;
        };
        //声明面板属性 (uniform)
        float4 _Color;

        v2f vert(appdata_base v)      //顶点着色器
        {
            v2f o;
            o.pos = mul(UNITY_MATRIX_MVP,v.vertex);
            o.color = fixed4(1,1,1,1);      //float4 fixed4 half4仅仅表示精度，精度有高到低
            return o;                      //输出顶点颜色
        }
        fixed4 frag(v2f IN) : COLOR
        {
            ..... //片元着色器
            return IN.color;             //输出像素颜色
        }
    ENDCG
    }
}

```

再认识一下*Draw Call*: 就是CPU调用图像编程接口，如OpenGL中的glDrawElements命令或者DirectX中的DrawIndexedPrimitive命令，以命令GPU进行渲染操作。

## · CPU和GPU实现并行工作



上图中黄色框的命令比较耗时

### · Draw Call 多了会影响效率

一般情况下，GPU的渲染速度远远快于CPU提交命令的速度，GPU对于渲染200个还是2000个三角网格通常没有什么区别，然而CPU发送命令包含了很多内容，需要完成如检查渲染状态等的很多准备工作，Draw Call数量很多时候，CPU就会把大量时间花费在提交Draw Call上，造成**CPU过载**。

### · 如何减少Draw Call

批处理（动态和静态）

注意：避免使用大量很小的网格

避免使用过多的材质

动态批处理的限制条件：基于Unity 5.x以上

- 1、能够进行动态批处理的网格的顶点属性规模要小于900
- 2、4.x版本中要求动态批处理的对象需要使用同一个缩放尺度
- 3、使用光照纹理（lightmap）的物体需要保证均指向光照纹理中的同一位置
- 4、由于为模型添加更多光照效果需要使用额外的pass通道，而多pass的shader会中断批处理；注意只有物体在点光源的影响范围内才需要调用额外的pass来处理

静态批处理：需要占用更多的内存来存储合并后的几何结构

实现方式：勾选上Inspector面板上的Static复选框中的 **Batching Static**

计算机渲染管线：

CG 标准函数库：

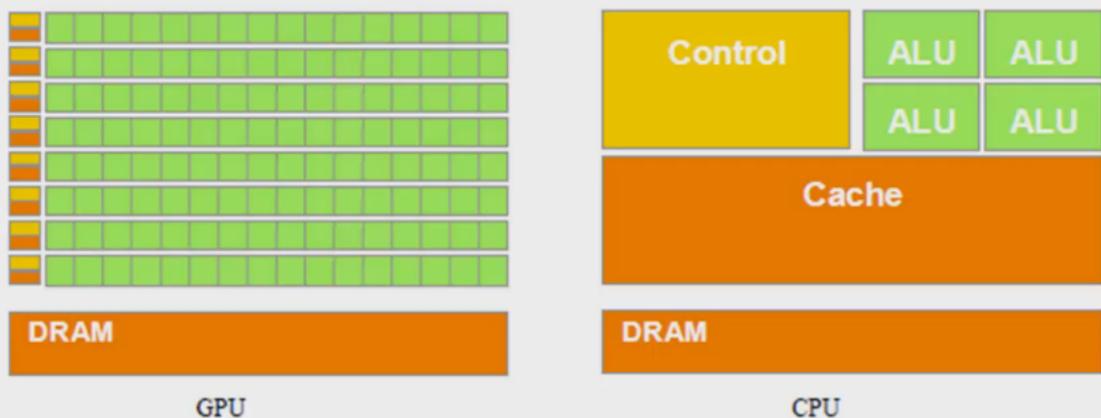
参考：[http://wenku.baidu.com/link?url=Qpanx8lvHbQtX7BchpD3glpj-pCHHWdAXI1kUuEqM\\_50Ze0UGTd9dwIRUXFQ\\_YBnCdfN6TSUPALQQzIJOnwaxrd\\_6DANjV81mLDs\\_JUGR7](http://wenku.baidu.com/link?url=Qpanx8lvHbQtX7BchpD3glpj-pCHHWdAXI1kUuEqM_50Ze0UGTd9dwIRUXFQ_YBnCdfN6TSUPALQQzIJOnwaxrd_6DANjV81mLDs_JUGR7)

GPU编程的本质：

GPU允许应用程序指定一个序列的指令进行顶点操作控制。

## GPU的优越性

由于GPU 具有高并行结构，所以GPU 在处理图形数据和复杂算法方面拥有比CPU 更高的效率。CPU 大部分面积为控制器和寄存器，与之相比，GPU 拥有更多的ALU（Arithmetic Logic Unit，逻辑运算单元）用于数据处理，这样的结构适合对密集型数据进行并行处理。



## GPU的优越性

GPU 采用流式并行计算模式，可对每个数据进行独立的并行计算，所谓“对数据进行独立计算”，即，流内任意元素的计算不依赖于其它同类型数据，例如，计算一个顶点的世界位置坐标，不依赖于其他顶点的位置，所谓“并行计算”是指“多个数据可以同时被使用，多个数据并行运算的时间和1个数据单独执行的时间是一样的”。所以，在顶点处理程序中，可以同时处理N个顶点数据。

## GPU的缺陷

由于“任意一个元素的计算不依赖于其它同类型数据”，导致“需要知道数据之间相关性的”算法，在GPU上难以得到实现，一个典型的例子是射线与物体的求交运算。GPU中的控制器少于CPU，致使控制能力有限。另外，进行GPU编程必须掌握计算机图形学相关知识，以及图形处理API，入门门槛较高，学习周期较长，尤其国内关于GPU编程的资料较为匮乏，这些都导致了学习的难度。在早期，GPU编程只能使用汇编语言，开发难度高、效率低，不过，随着高级Shader language的兴起，在GPU上编程已经容易多了。

总结：

- 2003年开始正式进入可编程GPU阶段
- GPU的并行处理能力强于CPU
- 目前GPU无法代替CPU

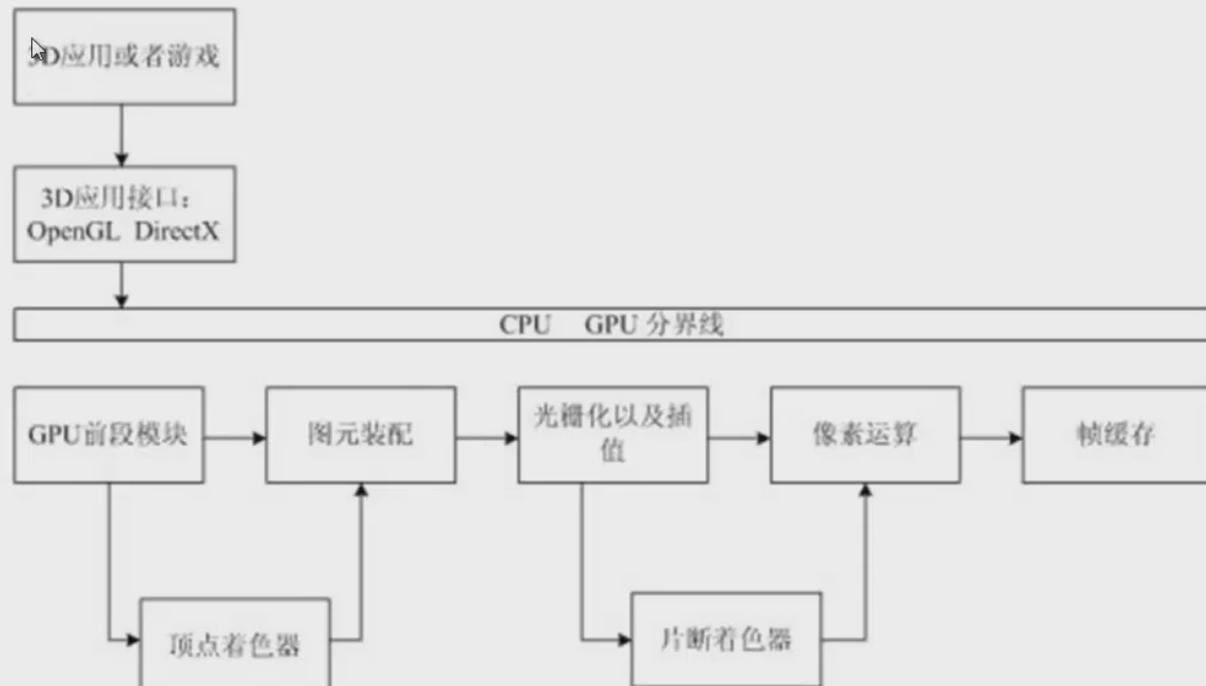
## ■ 什么是Shader

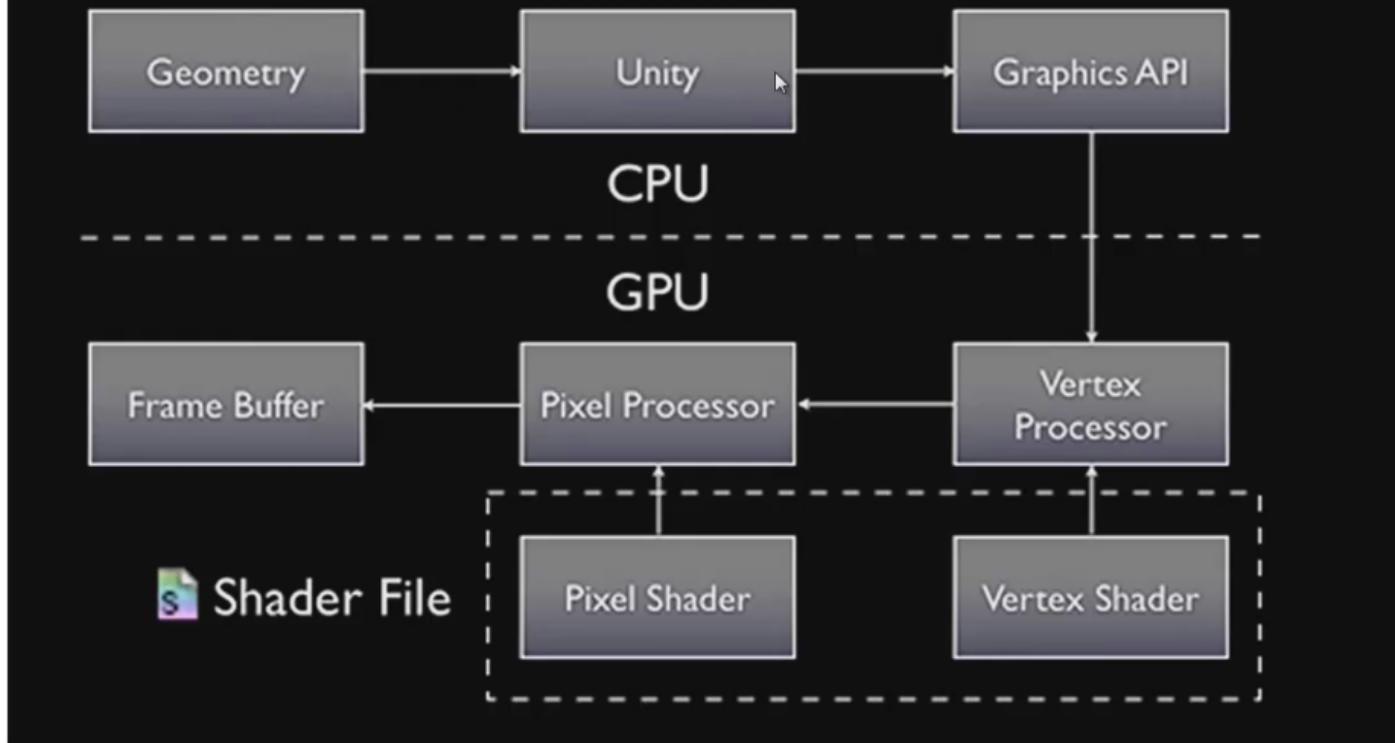
Shader，中文翻译即着色器，是一种较为短小的程序片断，用于告诉图形硬件如何计算和输出图像，过去由汇编语言来编写，现在也可以使用高级语言来编写。一句话概括：Shader是可编程图形管线的算法片段。

它主要分为两类，Vertex Shader和Fragment Shader.

## 什么是渲染管线

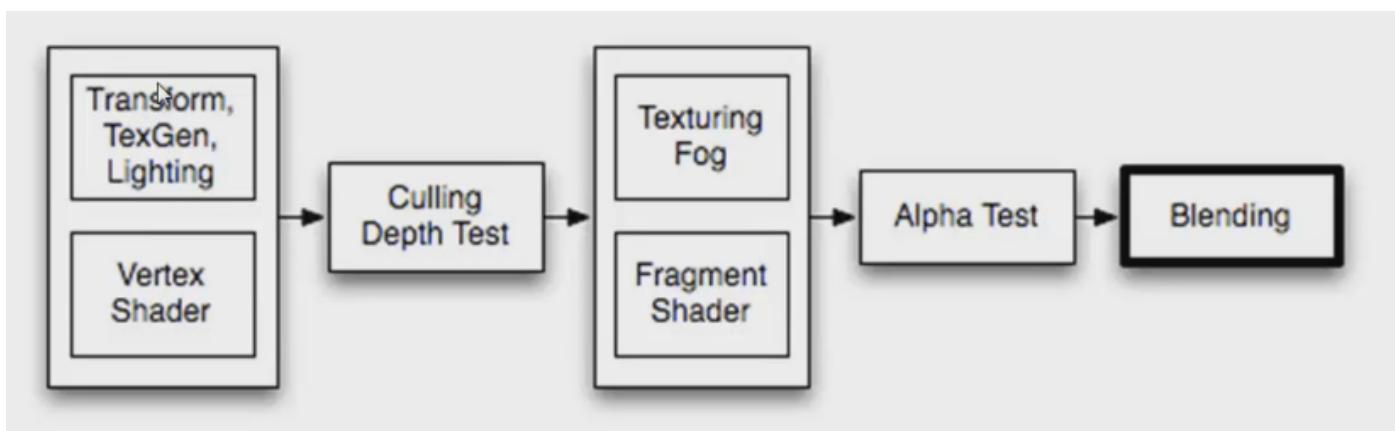
渲染管线也称为渲染流水线，是显示芯片内部处理图形信号相互独立的并行处理单元。一个流水线是一系列可以并行和按照固定顺序进行的阶段。每个阶段都从它的前一阶段接收输入，然后把输出发给随后的阶段。就像一个在同一时间内，不同阶段不同的汽车一起制造的装配线，传统的图形硬件流水线以流水的方式处理大量的顶点、几何图元和片段。





T&L

|



## Shader和材质、贴图的关系

Shader（着色器）实际上就是一小段程序，它负责将输入的顶点数据以指定的方式和输入的贴图或者颜色等组合起来，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三维物体来进行渲染（输出）了。

材质好比引擎最终使用的商品，Shader好比是生产这种商品的加工方法，而贴图就是原材料。

## 总结：

- Shader是图形可编程方案的程序片段
- 渲染管线是一种计算机从数据到最终图形成像的形象描述
- 材质是商品，Shader是方法，贴图是材料

三大主流的GPU编程语言（shader language）：

HLSL、GLSL、CG

GLSL：基于OpenGL的OpenGL Shading Language

### OpenGL简介

OpenGL（全写Open Graphics Library）是一个定义了跨编程语言、跨平台的[编程接口](#)规格的专业图形[程序接口](#)。它用于[三维图像](#)（二维的亦可），是一个功能强大，调用方便的底层图形库。OpenGL是行业领域中最为广泛接纳的2D/3D图形[API](#)，其自诞生至今已催生了各种[计算机平台](#)及设备上的数千优秀应用程序。它独立于视窗操作系统或其它操作系统的，亦是网络透明的。在包含CAD、内容创作、能源、娱乐、游戏开发、制造业、制药业及[虚拟现实](#)等行业领域中。OpenGL是个与硬件无关的[软件接口](#)，可以在不同的平台如[Windows 95](#)、[Windows NT](#)、[Unix](#)、[Linux](#)、[MacOS](#)、[OS/2](#)之间进行移植。因此，支持OpenGL的软件具有很好的移植性，可以获得非常广泛的应用。

OpenGL的发展一直处于一种较为迟缓的态势，每次版本的提高新增的技术很少，大多只是对其中部分做出修改和完善。1992年7月，SGI公司发布了OpenGL的1.0版本，随后又与[微软公司](#)共同开发了Windows NT版本的OpenGL，从而使一些原来必须在高档图形工作站上运行的大型3D图形处理软件也可以在微机上运用。1995年OpenGL的1.1版本面市，该版本比1.0的性能有许多提高，并加入了一些新的功能。其中包括改进打印机支持，在增强元文件中包含OpenGL的调用，顶点[数组](#)的新特性，提高顶点位置、法线、颜色、色彩指数、纹理坐标、多边形边缘标识的传输速度，引入了新的纹理特性等等。OpenGL 1.5又新增了“OpenGL Shading Language”，该语言是“OpenGL 2.0”的底核，用于着色对象、顶点着色以及片断着色技术的扩展功能。

HLSL：基于DirectX的High Level Shading Language

## DirectX简介

DirectX，（Direct eXtension，简称DX）是由[微软公司](#)创建的[多媒体编程接口](#)。由C++[编程语言](#)实现，遵循COM。被广泛使用于[Microsoft Windows](#)、Microsoft XBOX、Microsoft XBOX 360和Microsoft XBOX ONE电子游戏开发，并且只能支持这些平台。最新版本为DirectX 12，创建在最新的Windows10。DirectX是这样一组技术：它们旨在使基于Windows的[计算机](#)成为运行和显示具有丰富[多媒体](#)元素（例如全色图形、视频、3D动画和丰富音频）的[应用程序](#)的理想平台。DirectX包括安全和性能更新程序，以及许多涵盖所有技术的新功能。[应用程序](#)可以通过使用[DirectX API](#)来访问这些新功能。

DirectX加强3D[图形](#)和[声音](#)效果，并提供设计人员一个共同的硬件驱动标准，让游戏开发者不必为每一品牌的硬件来写不同的[驱动程序](#)，也降低了用户安装及设置硬件的复杂度。从字面意义上说，Direct就是直接的意思，而后边的X则代表了很多的意思，从这一点上可以看出DirectX的出现就是为了众多软件提供直接服务的。

举例来说，以前在[DOS](#)下玩家玩游戏时，并不是安装上就可以玩了，他们往往首先要设置声卡的品牌和型号，然后还要设置[IRQ](#)（中断）、[I/O](#)（输入与输出）、[DMA](#)（存取模式），如果哪项设置的不对，那么游戏声音就发不出来。这部分的设置不仅让玩家伤透脑筋，对游戏开发者来说就更为头痛。为了让游戏能够正确运行，开发者必须在游戏制作之初，把市面上所有声卡硬件数据都收集过来，然后根据不同的API（应用编程接口）来写不同的驱动程序。这对于游戏制作公司来说，是很难完成的，所以在当时[多媒体](#)游戏很少。微软正是看到了这个问题，为众厂家推出了一个共同的应用程序接口——DirectX。只要游戏是依照[DirectX](#)来开发的，不管显卡、声卡型号如何，统统都能玩，而且还能发挥最佳的效果。当然，前提是使用的显卡、声卡的驱动程序必须支持[DirectX](#)才行。

## CG: Nvidia 的 C for Graphic

### Cg

edu.manew.com/dayelongshe

GLSL 与 HLSL 分别基于 OpenGL 和 Direct3D 的接口，两者不能混用，事实上 OpenGL 和 Direct3D 一直都是冤家对头，争斗良久。OpenGL 在其长期发展中积累下的用户群庞大，这些用户会选择 GLSL 学习。GLSL 继承了 OpenGL 的良好移植性，一度在 unix 等操作系统上独领风骚。但 GLSL 的语法体系自成一家。微软的 HLSL 移植性较差，在 windows 平台上可谓一家独大，这一点在很大程度上限制了 HLSL 的推广和发展。但是 HLSL 用于 DX 游戏领域却是深入人心。

Cg 语言（C for Graphic）是为 GPU 编程设计的高级着色器语言，Cg 极力保留 C 语言的大部分语义，并让开发者从硬件细节中解脱出来，Cg 同时也有一个高级语言的其他好处，如代码的易重用性，可读性得到提高，编译器代码优化。Cg 是一个可以被 OpenGL 和 Direct3D 广泛支持的图形处理器编程语言。Cg 语言和 OpenGL、DirectX 并不是同一层次的语言，而是 OpenGL 和 DirectX 的上层，即，Cg 程序是运行在 OpenGL 和 DirectX 标准顶点和像素着色的基础上的。Cg 由 NVIDIA 公司和微软公司相互协作在标准硬件光照语言的语法和语义上达成了一致开发。所以，HLSL 和 Cg 其实是同一种语言。

Unity 中的 shaderLab： Cg/HLSL 的代码片段，也支持 GLSL（推荐使用原生的 GLSL）

## 3D数学&图形学 for Shader

坐标系：

模型坐标系 —> 世界坐标系 —> 摄像机坐标系 —> 屏幕投影坐标系

## ■ 向量点积

$$V1 = (1, 0)$$

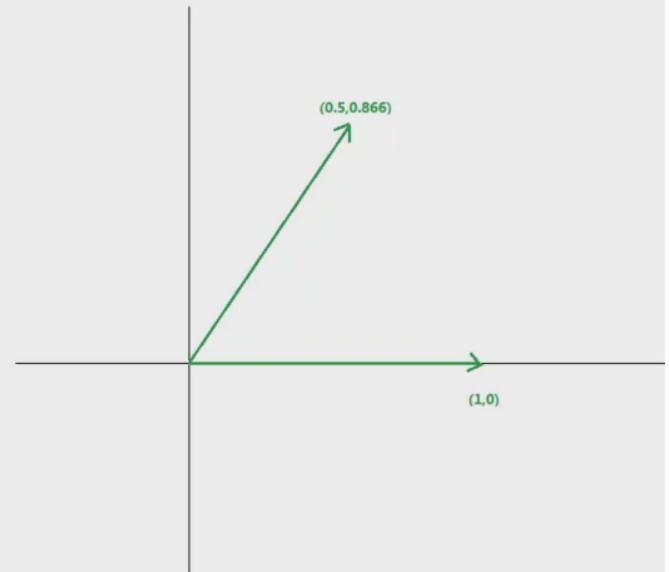
$$V2 = (0.5, 0.866)$$

$$\text{Dot}(V1, V2) = V1 \cdot V2$$

$$= (1, 0) \cdot (0.5, 0.866)$$

$$= (1 \cdot 0.5 + 0 \cdot 0.866) \quad \downarrow$$

$$= 0.5$$



## ■ 向量点积的几何意义

$$\text{Dot}(V1, V2) = \|V1\| * \|V2\| * \cos(a)$$

$$\cos(a) = \text{Dot}(v1, v2) / \|V1\| * \|V2\|$$

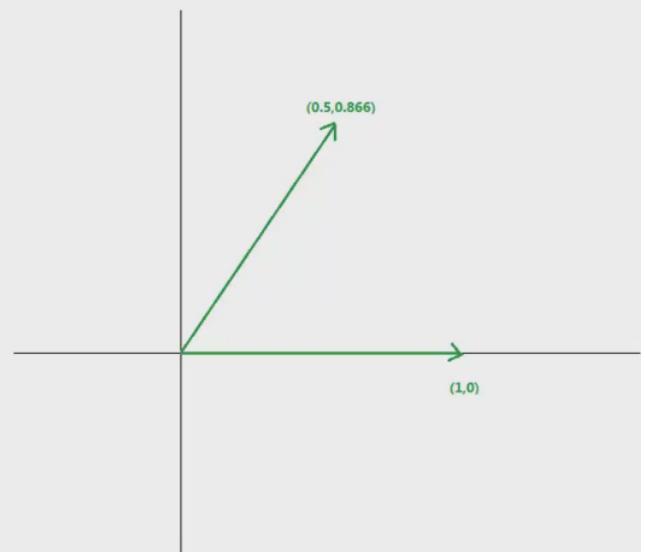
特别的，当V1和V2都是规范化向量

$$\cos(a) = V1 \cdot V2$$

$$a = \arccos(V1 \cdot V2)$$

$$= \arccos(0.5)$$

$$= 60^\circ$$



## ■ 向量叉积

$$V1 = (1, 0, 0)$$

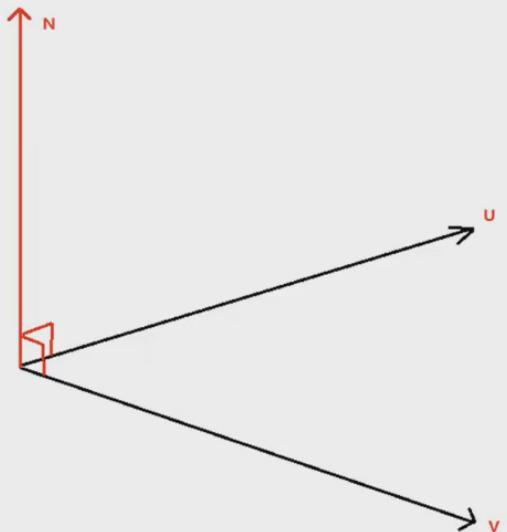
$$V2 = (0, 1, 0)$$

$$\text{Cross}(V1, V2) =$$

$$\begin{vmatrix} x_1 \\ y_1 \\ z_1 \end{vmatrix} \times \begin{vmatrix} x_2 \\ y_2 \\ z_2 \end{vmatrix} = \begin{vmatrix} y_1 * z_2 - z_1 * y_2 \\ z_1 * x_2 - z_2 * x_1 \\ x_1 * y_2 - y_1 * x_2 \end{vmatrix}$$

### ■ 向量叉积的几何意义

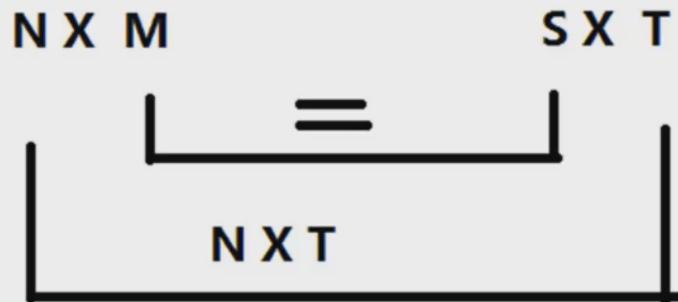
$$\begin{vmatrix} 1 \\ 0 \\ 0 \end{vmatrix} \times \begin{vmatrix} 0 \\ 1 \\ 0 \end{vmatrix} = \begin{vmatrix} 0 * 0 - 0 * 1 \\ 0 * 0 - 1 * 0 \\ 1 * 1 - 0 * 0 \end{vmatrix} = \begin{vmatrix} 0 \\ 0 \\ 1 \end{vmatrix}$$



矩阵乘法

## ■ 矩阵和矩阵的乘法

$N \times M$  阶与  $S \times T$  阶矩阵相乘，必须满足  $M$  和  $S$  维度相同，  
乘法的结果是一个  $N \times T$  阶矩阵。



## 方阵和行列式

行列式的前提是方阵

### ■ 2阶方阵的行列式

$$|\mathbf{M}| = \begin{vmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{vmatrix} = m_{11}m_{22} - m_{12}m_{21}$$



举例如下：

$$\begin{vmatrix} 2 & 1 \\ -1 & 2 \end{vmatrix} = (2)(2) - (1)(-1) = 4 + 1 = 5$$

$$\begin{vmatrix} -3 & 4 \\ 2 & 5 \end{vmatrix} = (-3)(5) - (4)(2) = -15 - 8 = -23$$

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

## ■ 3阶方阵的行列式

$$\begin{aligned}\triangleright \begin{vmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{vmatrix} &= m_{11}m_{22}m_{33} + m_{12}m_{23}m_{31} + m_{13}m_{21}m_{32} \\ &\quad - m_{13}m_{22}m_{31} - m_{12}m_{21}m_{33} - m_{11}m_{23}m_{32} \\ &= m_{11}(m_{22}m_{33} - m_{23}m_{32}) + m_{12}(m_{23}m_{31} - m_{21}m_{33}) \\ &\quad + m_{13}(m_{21}m_{32} - m_{22}m_{31})\end{aligned}$$



## ■ 代数余子式

$$\begin{bmatrix} -4 & -3 & 3 \\ 0 & 2 & -2 \\ 1 & 4 & -1 \end{bmatrix} \Rightarrow \mathbf{M}^{(12)} = \begin{bmatrix} 0 & -2 \\ 1 & -1 \end{bmatrix}$$

$$c_{ij} = (-1)^{i+j} |\mathbf{M}^{(i,j)}|$$

||

## ■ NxN 阶方阵的行列式

$$\begin{vmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{vmatrix} = m_{11} \begin{vmatrix} m_{22} & m_{23} & m_{24} \\ m_{32} & m_{33} & m_{34} \\ m_{42} & m_{43} & m_{44} \end{vmatrix} - m_{12} \begin{vmatrix} m_{21} & m_{23} & m_{24} \\ m_{31} & m_{33} & m_{34} \\ m_{41} & m_{43} & m_{44} \end{vmatrix} \\ + m_{13} \begin{vmatrix} m_{21} & m_{22} & m_{24} \\ m_{31} & m_{32} & m_{34} \\ m_{41} & m_{42} & m_{44} \end{vmatrix} - m_{14} \begin{vmatrix} m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \\ m_{41} & m_{42} & m_{43} \end{vmatrix}$$

## ■ 行列式性质

矩阵积的行列式等于矩阵行列式的积:  $|AB| = |A||B|$ 。

这可以扩展到多个矩阵的情况:

$$|M_1 M_2 \cdots M_{n-1} M_n| = |M_1| |M_2| \cdots |M_{n-1}| |M_n|$$

矩阵转置的行列式等于原矩阵的行列式:  $|M^T| = |M|$ 。

如果矩阵的任意行或列全为零, 那么它的行列式等于零。

交换矩阵的任意两行或两列, 行列式变负。

任意行或列的非零积加到另一行或列上不会改变行列式的值。

## 矩阵的逆 & 单位矩阵

### ■ 逆的定义

$$M(M^{-1}) = M^{-1}M = I$$

计算矩阵的逆:

前提该矩阵为方阵, 且行列式不为0, 如果行列式为0, 则该矩阵为奇异矩阵

## ■ 代数余子式矩阵

$$\mathbf{M} = \begin{bmatrix} -4 & -3 & 3 \\ 0 & 2 & -2 \\ 1 & 4 & -1 \end{bmatrix}$$

计算  $\mathbf{M}$  的代数余子式矩阵:

$$c_{11} = + \begin{vmatrix} 2 & -2 \\ 4 & -1 \end{vmatrix} = 6 \quad c_{12} = - \begin{vmatrix} 0 & -2 \\ 1 & -1 \end{vmatrix} = -2 \quad c_{13} = + \begin{vmatrix} 0 & 2 \\ 1 & 4 \end{vmatrix} = -2$$

$$c_{21} = - \begin{vmatrix} -3 & 3 \\ 4 & -1 \end{vmatrix} = 9 \quad c_{22} = + \begin{vmatrix} -4 & 3 \\ 1 & -1 \end{vmatrix} = 1 \quad c_{23} = - \begin{vmatrix} -4 & -3 \\ 1 & 4 \end{vmatrix} = 13$$

$$c_{31} = + \begin{vmatrix} -3 & 3 \\ 2 & -2 \end{vmatrix} = 0 \quad c_{32} = - \begin{vmatrix} -4 & 3 \\ 0 & -2 \end{vmatrix} = -8 \quad c_{33} = + \begin{vmatrix} -4 & -3 \\ 0 & 2 \end{vmatrix} = -8$$

转置得:

## ■ 标准伴随矩阵

$$c_{11} = + \begin{vmatrix} 2 & -2 \\ 4 & -1 \end{vmatrix} = 6 \quad c_{12} = - \begin{vmatrix} 0 & -2 \\ 1 & -1 \end{vmatrix} = -2 \quad c_{13} = + \begin{vmatrix} 0 & 2 \\ 1 & 4 \end{vmatrix} = -2$$

$$c_{21} = - \begin{vmatrix} -3 & 3 \\ 4 & -1 \end{vmatrix} = 9 \quad c_{22} = + \begin{vmatrix} -4 & 3 \\ 1 & -1 \end{vmatrix} = 1 \quad c_{23} = - \begin{vmatrix} -4 & -3 \\ 1 & 4 \end{vmatrix} = 13$$

$$c_{31} = + \begin{vmatrix} -3 & 3 \\ 2 & -2 \end{vmatrix} = 0 \quad c_{32} = - \begin{vmatrix} -4 & 3 \\ 0 & -2 \end{vmatrix} = -8 \quad c_{33} = + \begin{vmatrix} -4 & -3 \\ 0 & 2 \end{vmatrix} = -8$$

$$adj\mathbf{M} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}^T = \begin{bmatrix} 6 & -2 & -2 \\ 9 & 1 & 13 \\ 0 & -8 & -8 \end{bmatrix}^T = \begin{bmatrix} 6 & 9 & 0 \\ -2 & 1 & -8 \\ -2 & 13 & -8 \end{bmatrix}$$

## ■ 矩阵求逆

$$\mathbf{M}^{-1} = \frac{adj\mathbf{M}}{|\mathbf{M}|}$$

$$= \begin{bmatrix} 6 & 9 & 0 \\ -2 & 1 & -8 \\ -2 & 13 & -8 \end{bmatrix}$$

$$= \begin{bmatrix} -1/4 & -3/8 & 0 \\ 1/12 & -1/24 & 1/3 \\ 1/12 & -13/24 & 1/3 \end{bmatrix}$$

## ■ 矩阵逆的性质

矩阵的逆的重要性质：

- 如果  $\mathbf{M}$  是非奇异矩阵，则该矩阵的逆的逆等于原矩阵： $(\mathbf{M}^{-1})^{-1} = \mathbf{M}$ 。
- 单位矩阵的逆是它本身： $\mathbf{I}^{-1} = \mathbf{I}$ 。
- 矩阵转置的逆等于它的逆的转置： $(\mathbf{M}^T)^{-1} = (\mathbf{M}^{-1})^T$ 。
- 矩阵乘积的逆等于矩阵的逆的相反顺序的乘积： $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$ 。这可扩展到多个矩阵的情况：  
 $(\mathbf{M}_1\mathbf{M}_2 \cdots \mathbf{M}_{n-1}\mathbf{M}_n)^{-1} = \mathbf{M}_n^{-1}\mathbf{M}_{n-1}^{-1} \cdots \mathbf{M}_2^{-1}\mathbf{M}_1^{-1}$

矩阵作用：可以撤销之前的矩阵变换

## ■ 正交矩阵和逆

若方阵  $\mathbf{M}$  是正交的，则当且仅当  $\mathbf{M}$  与它转置  $\mathbf{M}^T$  的乘积等于单位矩阵

$$\mathbf{M} \text{ 正交} \Leftrightarrow \mathbf{MM}^T = \mathbf{I}$$

$$\begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## 平移矩阵

---

### ■ 2D平移( 3X3 矩阵)

$$\begin{vmatrix} x & y & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{vmatrix} = \begin{vmatrix} x+dx & y+dy & 1 \end{vmatrix}$$

向量也需要扩展 (满足矩阵乘法)

需要满足扩展的分量需要满足为1, 保证第二次做变换仍可以正确平移

### ■ 3D平移( 4X4 矩阵)

$$\begin{vmatrix} x & y & z & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{vmatrix} = \begin{vmatrix} x+dx & y+dy & z+dz & 1 \end{vmatrix}$$

## 旋转缩放矩阵

---

(左乘一个需要变换的对象 矩阵)

2D旋转矩阵

## ■ 2D旋转矩阵

绕坐标中心旋转 $a$ 角度

$$\begin{bmatrix} \cos(a) & \sin(a) \\ -\sin(a) & \cos(a) \end{bmatrix}$$

2D缩放矩阵

### ■ 2D缩放矩阵

沿坐标轴缩放

$$\begin{bmatrix} K_x & 0 \\ 0 & K_y \end{bmatrix}$$

沿任意N轴缩放

$$\begin{bmatrix} 1+(K-1) N_x^2 & (K-1) N_x * N_y \\ (K-1) N_x * N_y & 1+(K-1) N_y^2 \end{bmatrix}$$

3D旋转矩阵 (绕X 绕Y 绕Z)

## ■ 3D旋转矩阵

$$\begin{array}{c} \text{x} \\ \left[ \begin{array}{ccc} 1 & 0 & 0 \\ 0 & \cos(a) & \sin(a) \\ 0 & -\sin(a) & \cos(a) \end{array} \right] \end{array} \quad \begin{array}{c} \text{y} \\ \left[ \begin{array}{ccc} \cos(a) & 0 & \sin(a) \\ 0 & 1 & 0 \\ -\sin(a) & 0 & \cos(a) \end{array} \right] \end{array} \quad \begin{array}{c} \text{z} \\ \left[ \begin{array}{ccc} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{array} \right] \end{array}$$

绕任意轴n 缩放

## ■ 3D缩放矩阵

$$S(n, k) = \begin{bmatrix} p' \\ q' \\ r' \end{bmatrix} = \begin{bmatrix} 1 + (k-1)n_x^2 & (k-1)n_x n_y & (k-1)n_x n_z \\ (k-1)n_x n_y & 1 + (k-1)n_y^2 & (k-1)n_y n_z \\ (k-1)n_x n_z & (k-1)n_z n_y & 1 + (k-1)n_z^2 \end{bmatrix}$$

平移和旋转矩阵的结合

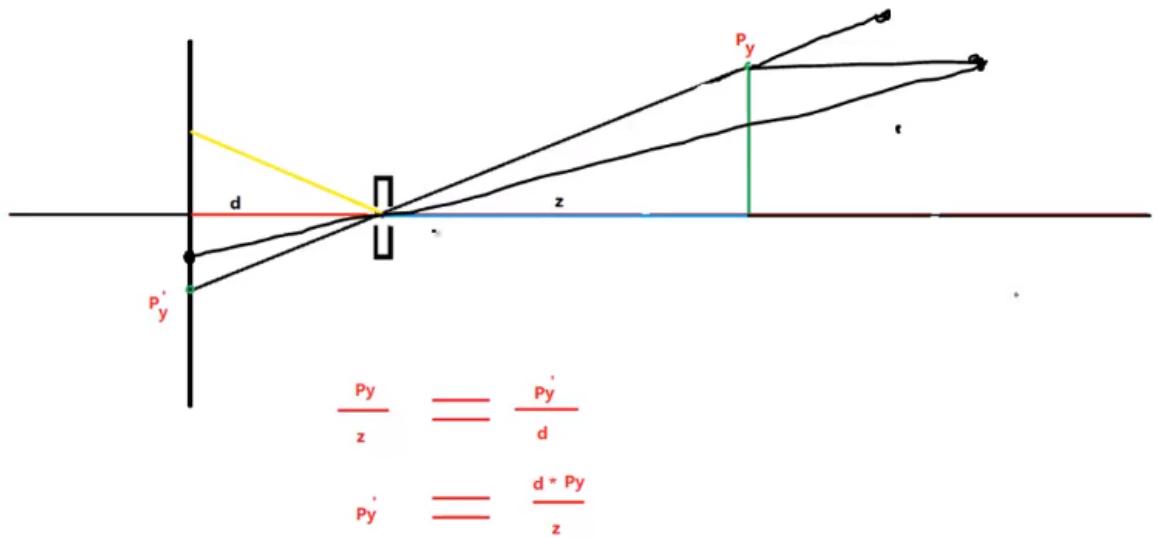
## ■ 3D平移

$$\begin{vmatrix} \cos & 0 & \sin & 0 \\ 0 & 1 & 0 & 0 \\ -\sin & 0 & \cos & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{vmatrix} = \begin{vmatrix} \cos & 0 & \sin & 0 \\ 0 & 1 & 0 & 0 \\ -\sin & 0 & \cos & 0 \\ dx & dy & dz & 1 \end{vmatrix}$$

旋转矩阵 (Y)      X      平移矩阵

透视投影：

## ■ 透视投影



(投影焦距

(一般使用等距投影 对象到相机的距离 == 相机到投影面的距离))

## ■ 透视投影矩阵

$$\begin{vmatrix} x & y & z & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/d & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix} = \begin{vmatrix} x & y & z & z/d \end{vmatrix}$$

1/d 表示摄像机到投影平面的距离

透视除法：

将需要显示在屏幕上的x,y与w(z/d)相除，得到投影到屏幕上的x' 和 y'

也称标准齐次除法，就是用齐次坐标系的w分量去除以x、y、z分量，

得到的坐标叫做 归一化的设备坐标 (NDC (Normalized Device Coordinates))

---

坐标变换的种类 (所有的线性变换都是仿射变换)

### ■ 变换的种类

旋转 缩放 平移 镜像 切变

投影(平行投影) 投影(透视投影)

可逆 等角 正交 刚体

线性变换

仿射变换

# 变换的种类

| 变换                | 线形 | 仿射 | 可逆 | 等角 | 正交 | 刚体 | 等长 | 等面积   | 行列式      |
|-------------------|----|----|----|----|----|----|----|-------|----------|
| 线性                | Y  | Y  |    |    |    |    |    |       |          |
| 仿射                |    | Y  |    |    |    |    |    |       |          |
| 可逆                |    |    | Y  |    |    |    |    |       | $\neq 0$ |
| 等角                |    | Y  | Y  | Y  |    |    |    |       |          |
| 正交                |    | Y  | Y  |    | Y  |    |    |       | $\pm 1$  |
| 刚体                |    | Y  | Y  | Y  | Y  | Y  | Y  | Y     |          |
| 平移                |    | Y  | Y  | Y  | Y  | Y  | Y  | Y     |          |
| 旋转 <sup>1</sup>   | Y  | Y  | Y  | Y  | Y  | Y  | Y  | Y     | 1        |
| 均匀缩放 <sup>2</sup> | Y  | Y  | Y  | Y  |    |    |    |       | $K^3$    |
| 非均匀缩放             | Y  | Y  | Y  |    |    |    |    |       |          |
| 正交投影 <sup>4</sup> | Y  | Y  |    |    |    |    |    |       | 0        |
| 镜像 <sup>5</sup>   | Y  | Y  | Y  |    | Y  |    | Y  | $Y^6$ | -1       |
| 切变                | Y  | Y  | Y  |    |    |    |    | $Y^7$ | 1        |

## 变换的组合

$$\mathbf{P}_{\text{世界}} = \mathbf{P}_{\text{物体}} \mathbf{M}_{\text{物体} \rightarrow \text{世界}}$$

$$\mathbf{P}_{\text{像机}} = \mathbf{P}_{\text{世界}} \mathbf{M}_{\text{世界} \rightarrow \text{像机}}$$

$$= (\mathbf{P}_{\text{物体}} \mathbf{M}_{\text{物体} \rightarrow \text{世界}}) \mathbf{M}_{\text{世界} \rightarrow \text{像机}}$$

$$\mathbf{P}_{\text{像机}} = (\mathbf{P}_{\text{物体}} \mathbf{M}_{\text{物体} \rightarrow \text{世界}}) \mathbf{M}_{\text{世界} \rightarrow \text{像机}}$$

$$= \mathbf{P}_{\text{物体}} (\mathbf{M}_{\text{物体} \rightarrow \text{世界}} \mathbf{M}_{\text{世界} \rightarrow \text{像机}})$$

$$\mathbf{M}_{\text{物体} \rightarrow \text{像机}} = \mathbf{M}_{\text{物体} \rightarrow \text{世界}} \mathbf{M}_{\text{世界} \rightarrow \text{像机}}$$

$$\mathbf{P}_{\text{像机}} = \mathbf{P}_{\text{物体}} \mathbf{M}_{\text{物体} \rightarrow \text{像机}}$$