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>

风宇冲博客 <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>

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；

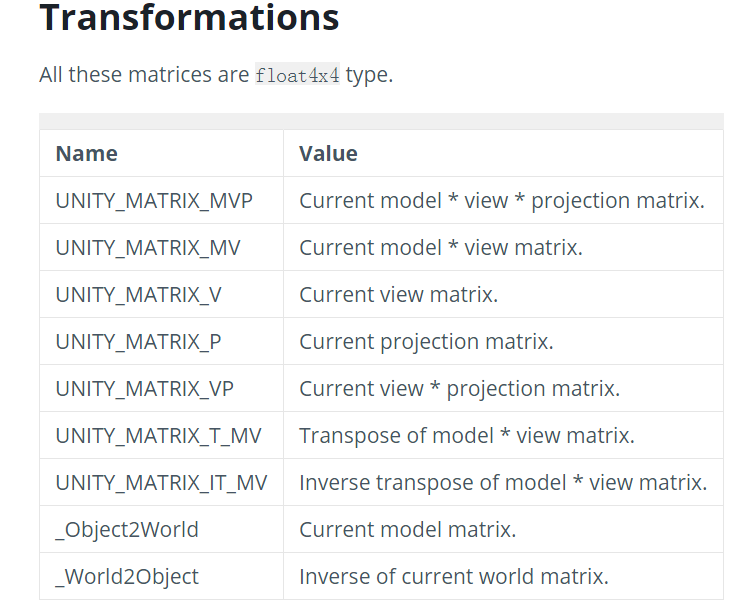
顶点着色器本身不能创建或者销毁顶点，而且也无法得到顶点与顶点间的关系，如无法得知两个顶点是否属于同一三角网格；由于该相互独立的特性，GPU可以利用本身的特性并行化处理每个顶点，大幅度提高处理速度。

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>



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](http://baike.baidu.com/view/2844853.htm)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 : POSITIVE;

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

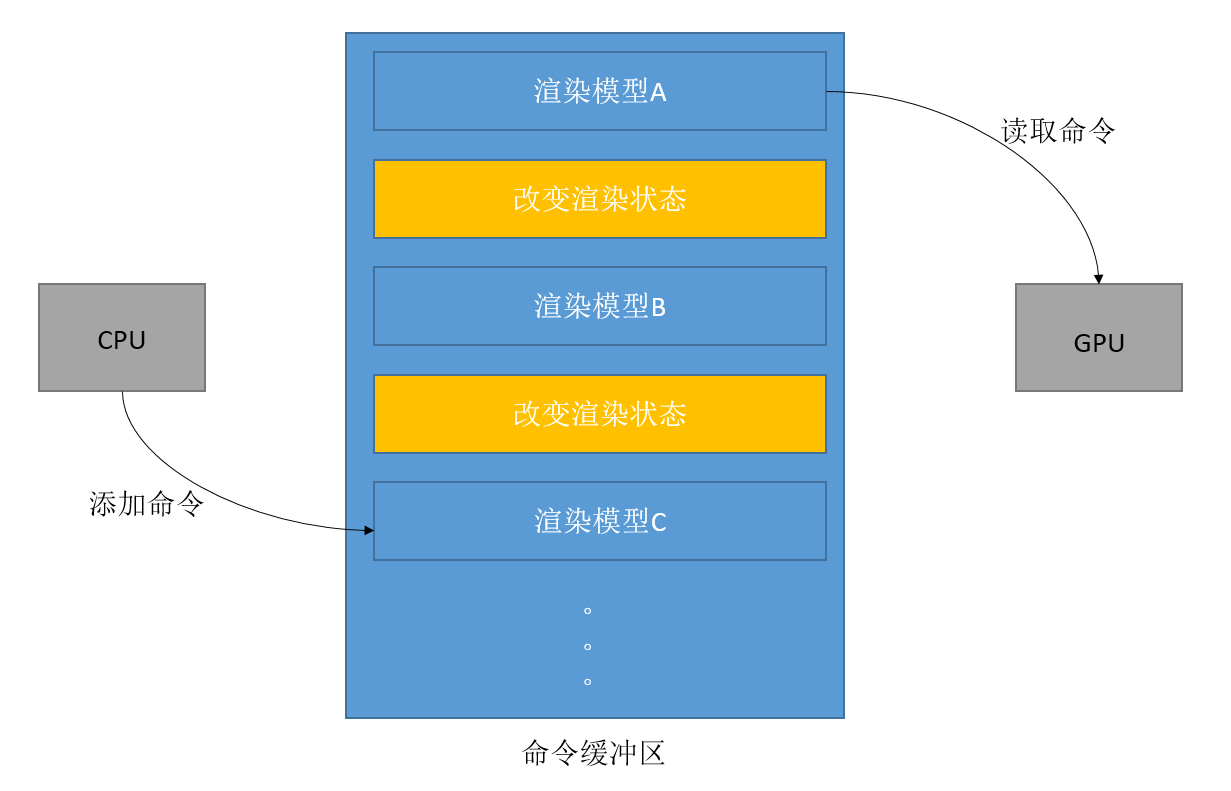
}

}

}

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，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**

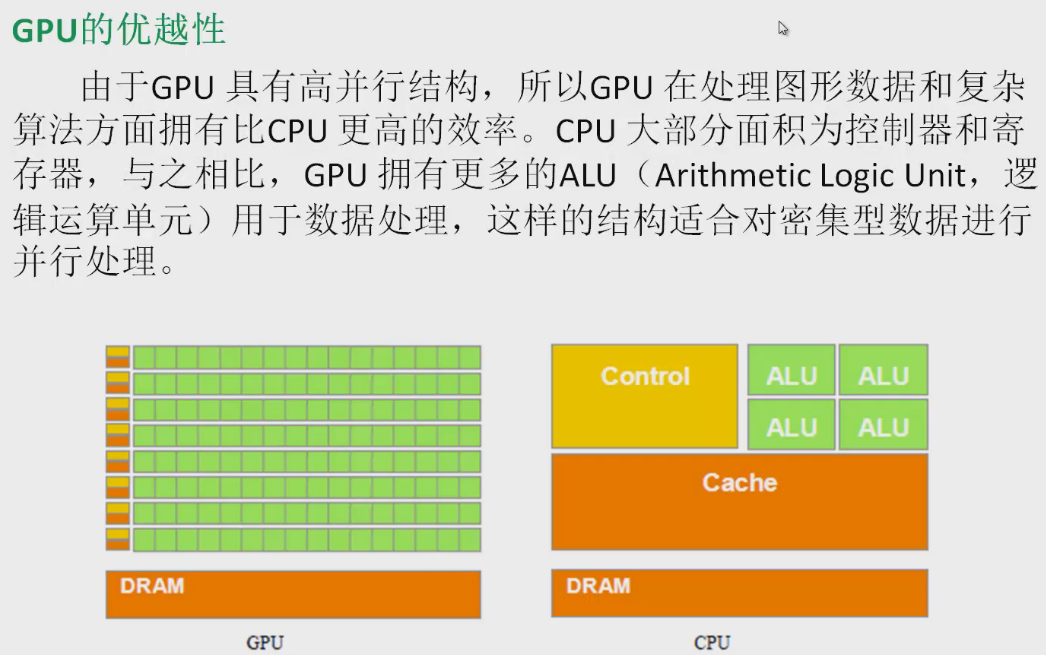
计算机渲染管线：

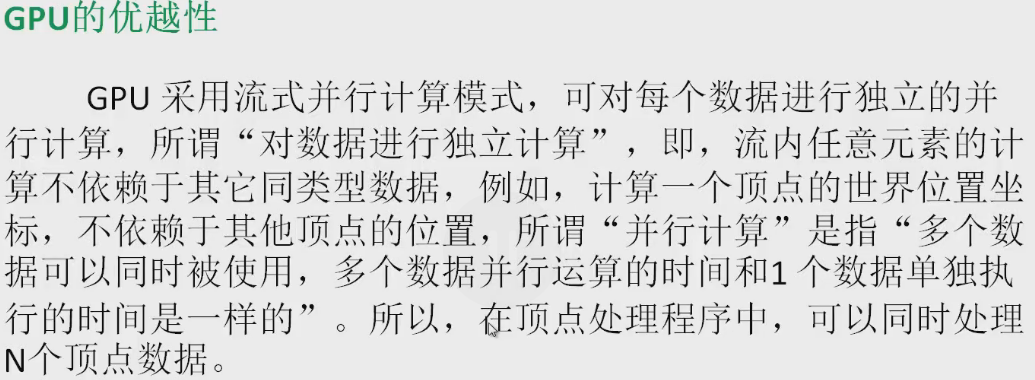
CG 标准函数库：

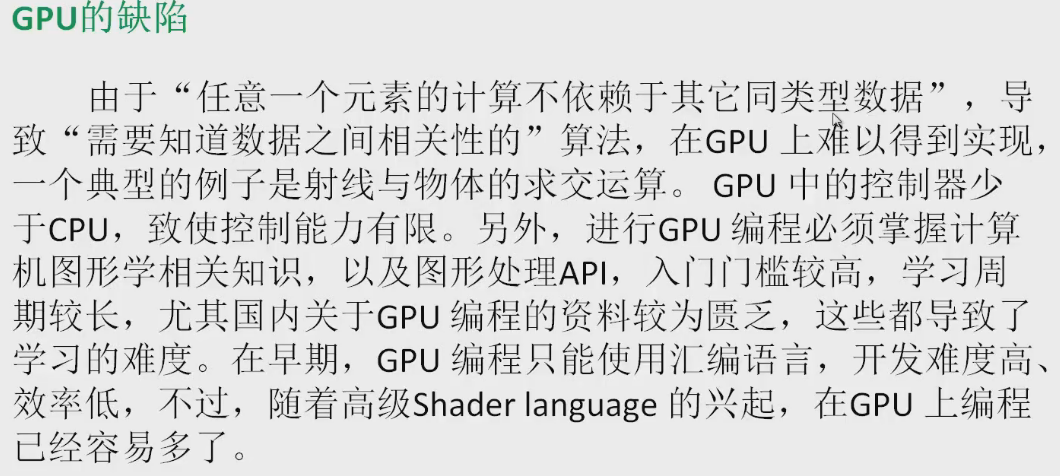
参考：<http://wenku.baidu.com/link?url=Qpanx8lvHbQtX7BchpD3gIpj-pCHHWdAXI1kUuEqM_50Ze0UGTd9dwlRUXFQ_YBnCdfN6TSUPALQQzlJOnwaxrzd_6DANjV81mLDs_JUGR7>

GPU编程的本质：

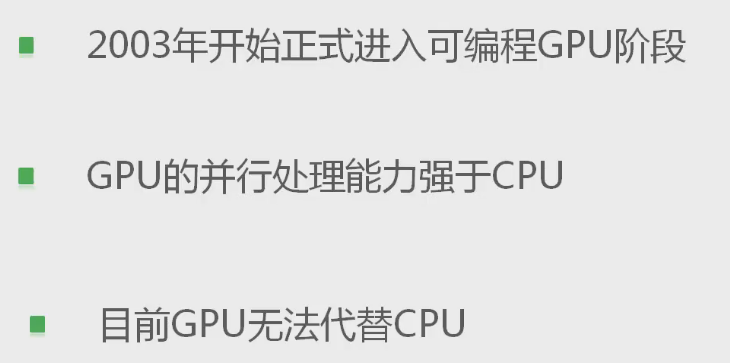
GPU允许应用程序指定一个序列的指令进行顶点操作控制。

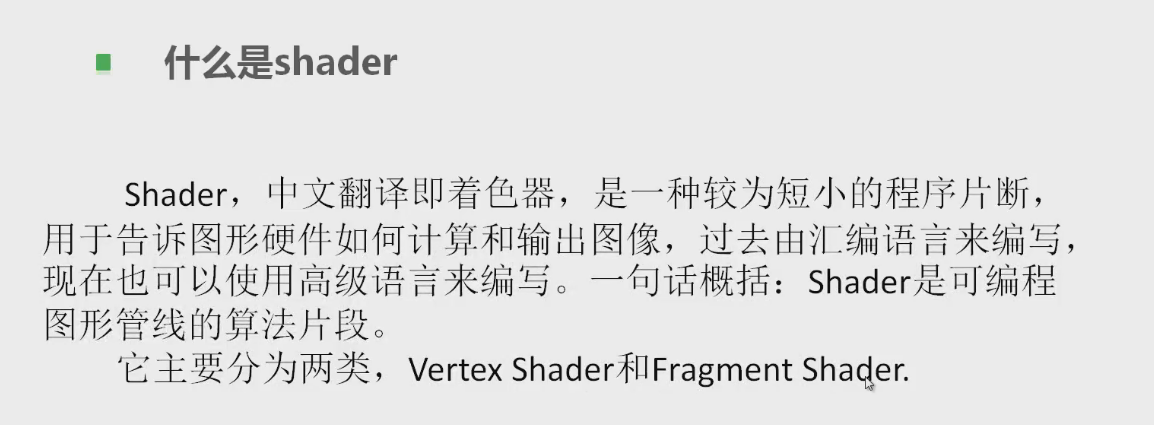


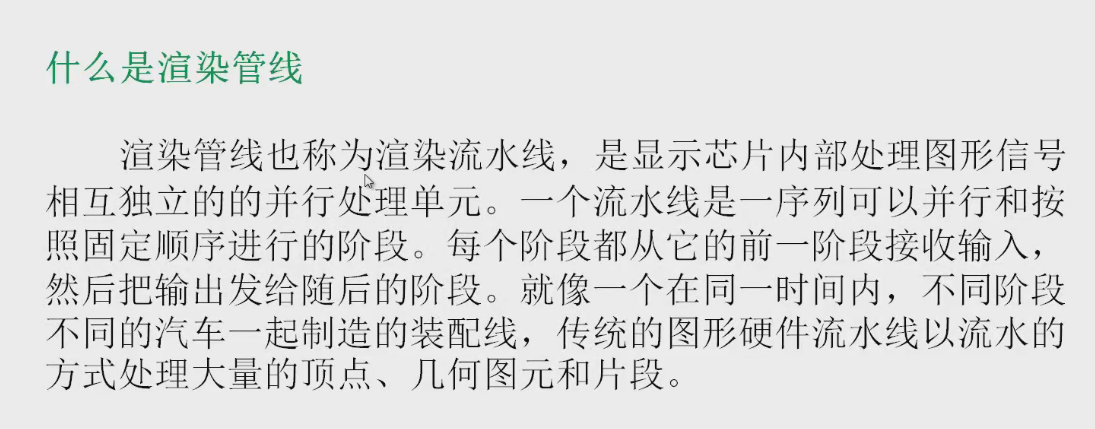


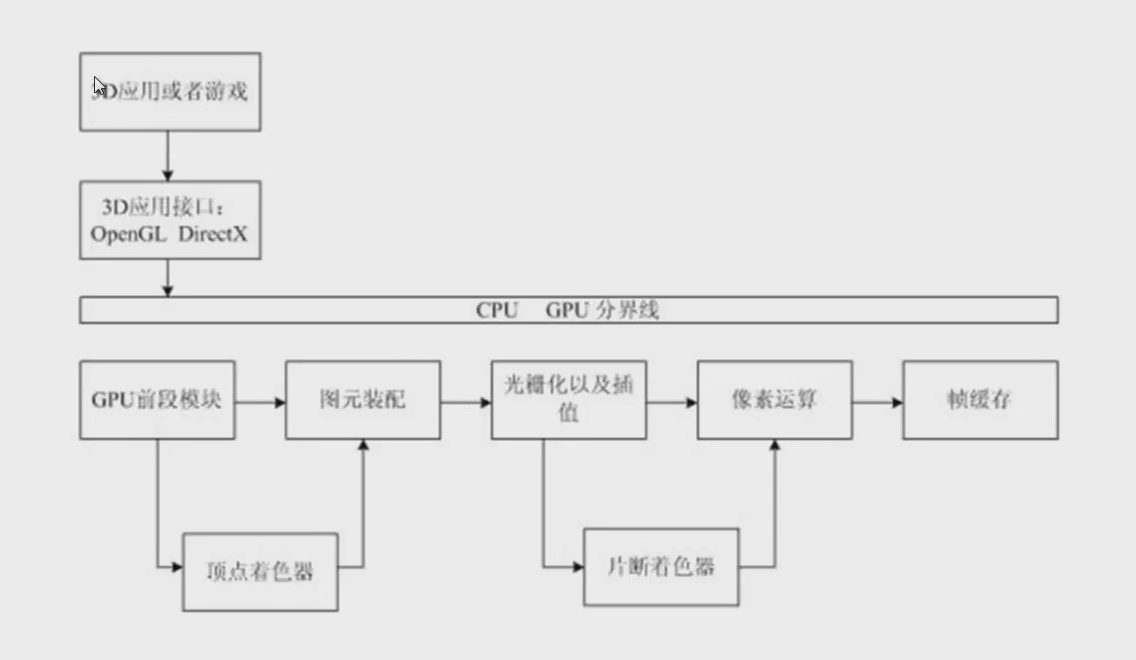


***总结：***

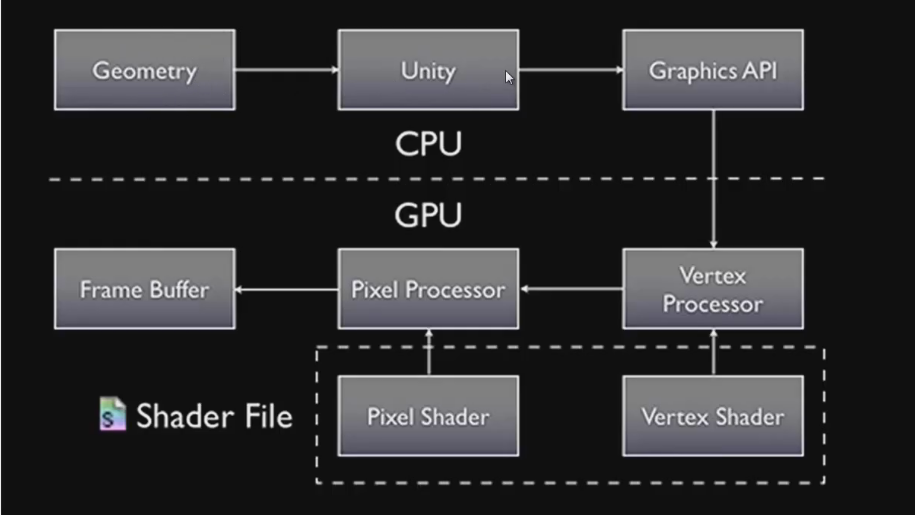








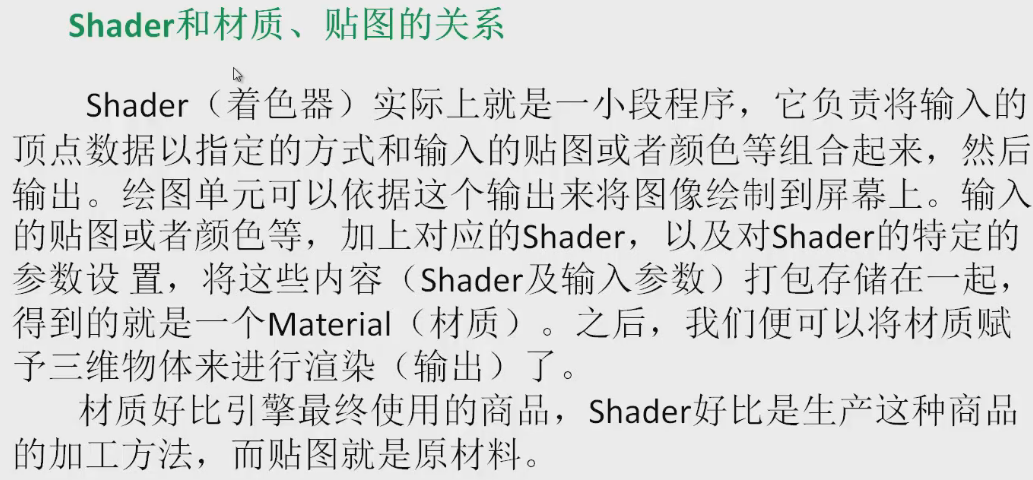
In Unity



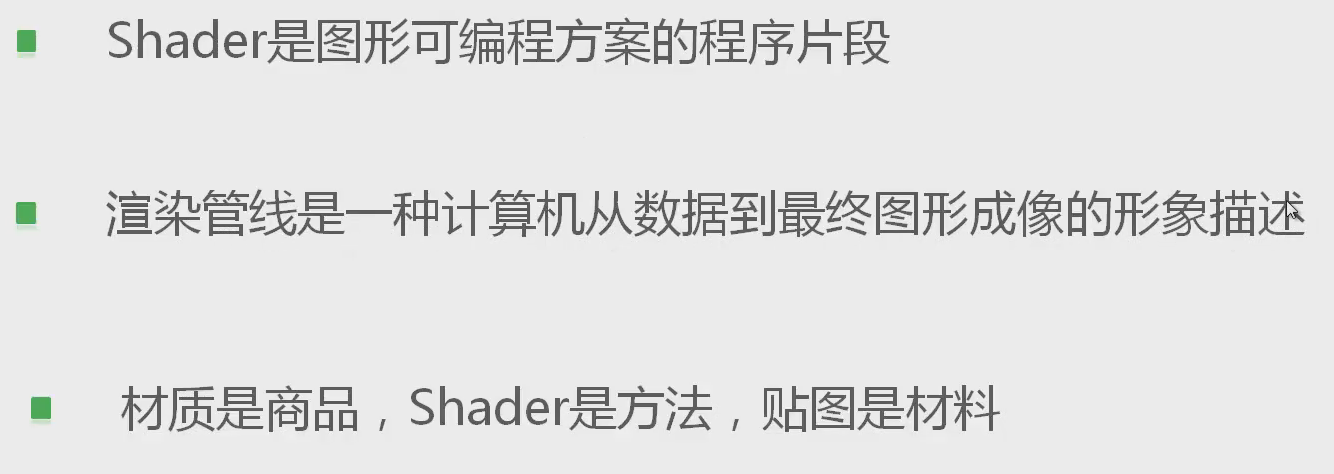
**T&L**

**|**





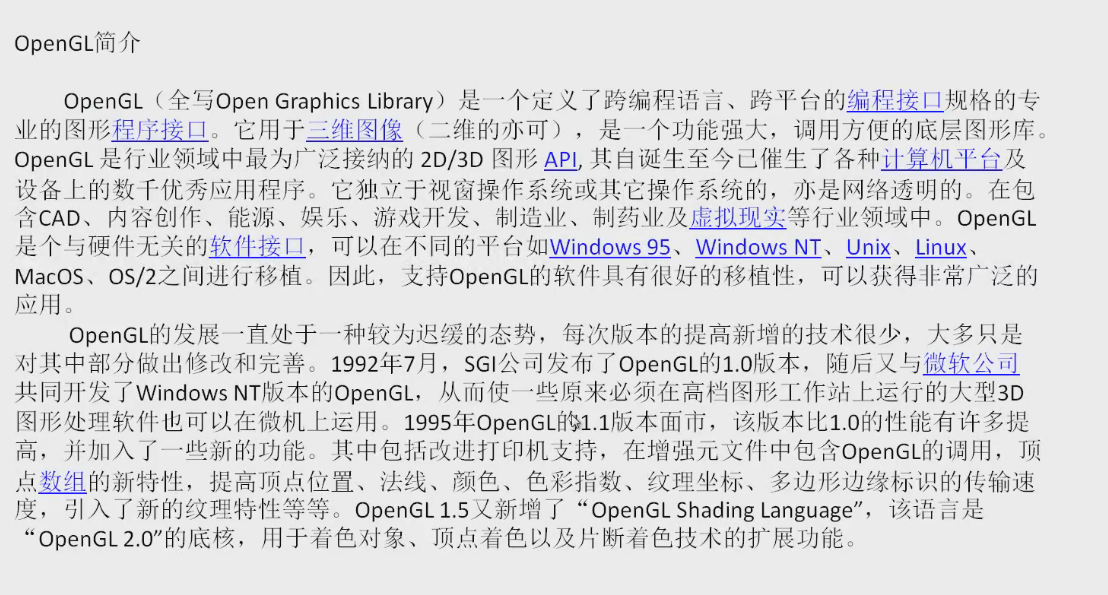
***总结：***



三大主流的GPU编程语言（shader language）：

HLSL 、 GLSL 、 CG

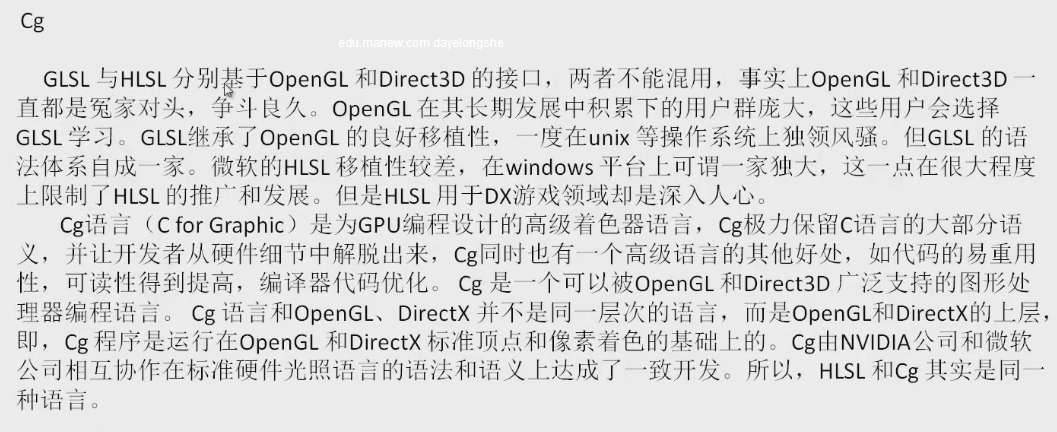
GLSL：基于OpenGL的OpenGL Shading Language



HLSL：基于DirectX的High Level Shading Language



CG：Nvidia 的 C for Graphic



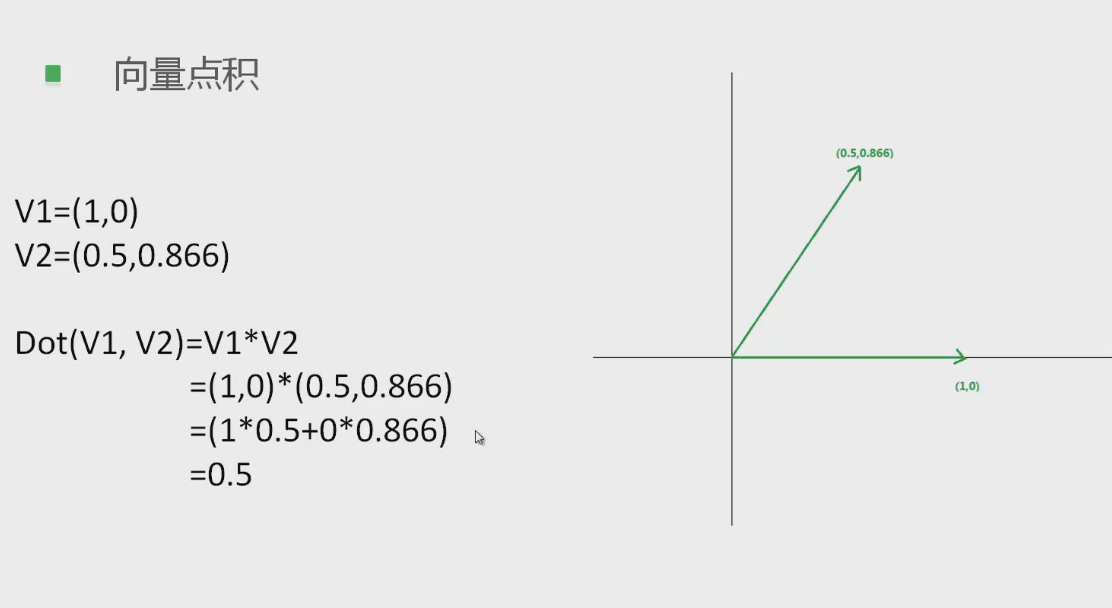
Unity中的shaderLab ： Cg/HLSL 的代码片段 ，也支持GLSL（推荐使用原生的GLSL）

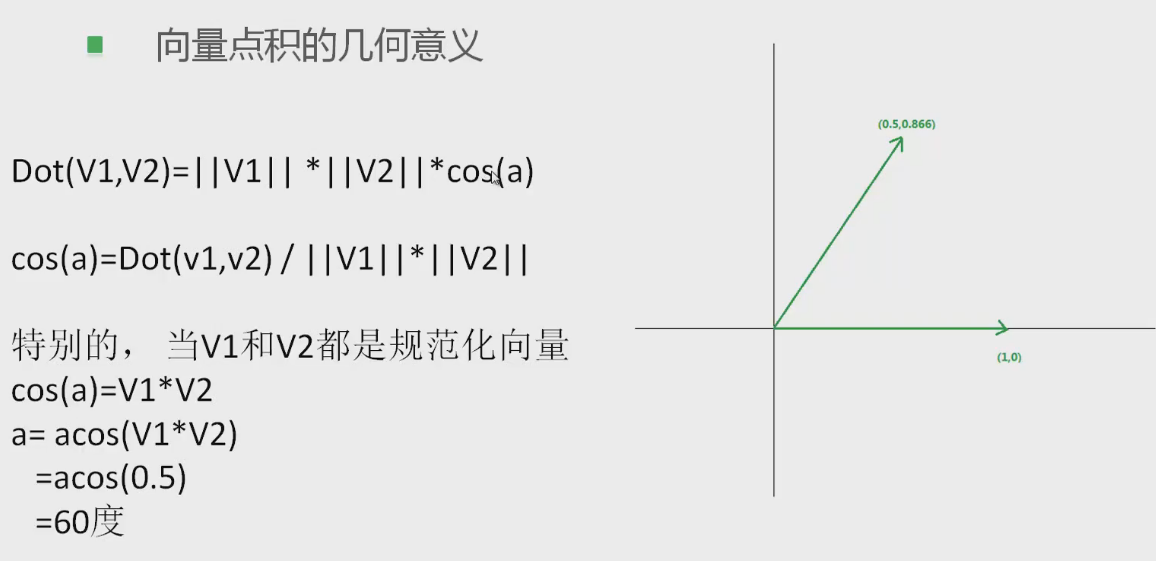
**3D数学&图形学 for Shader**

坐标系：

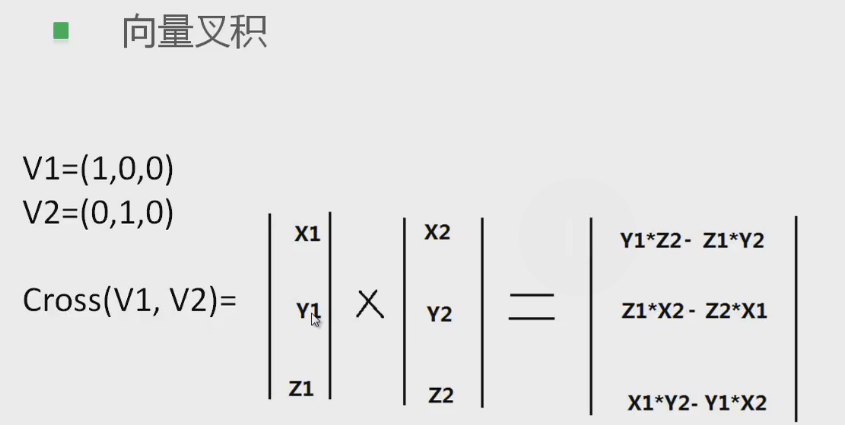
模型坐标系 ——> 世界坐标系 ——> 摄像机坐标系 ——> 屏幕投影坐标系

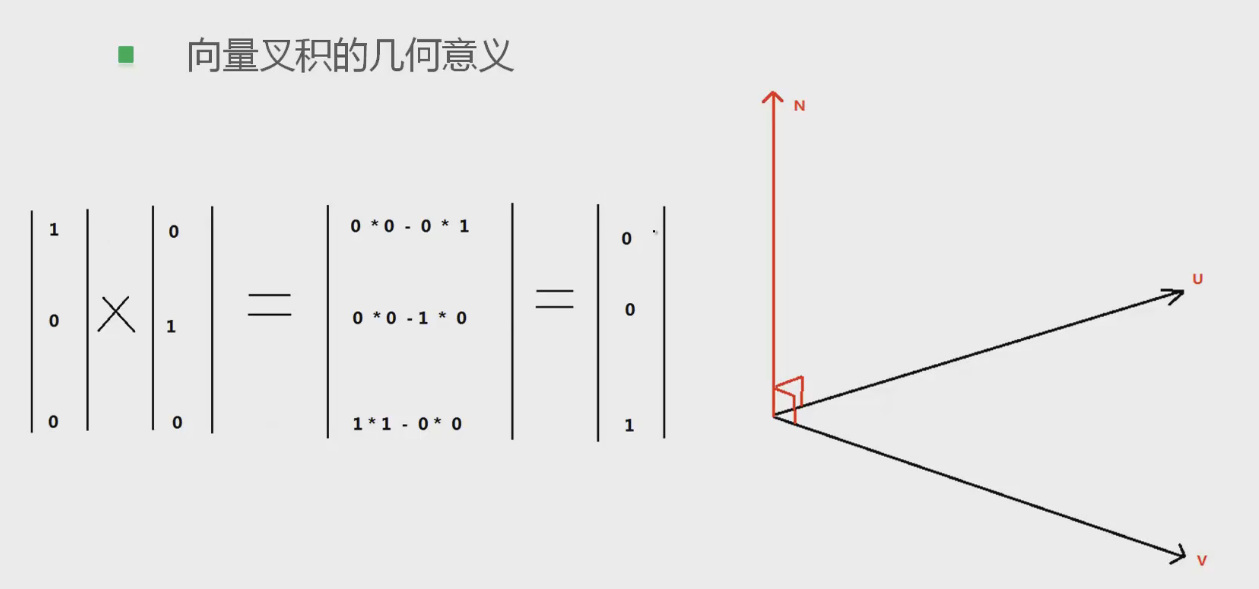
向量点积：



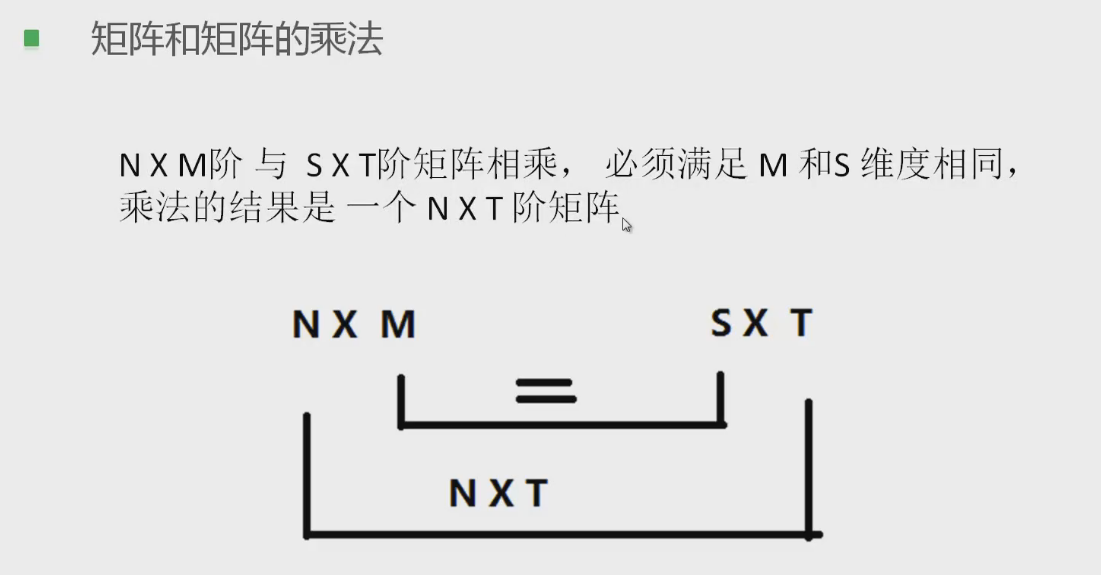


向量叉积



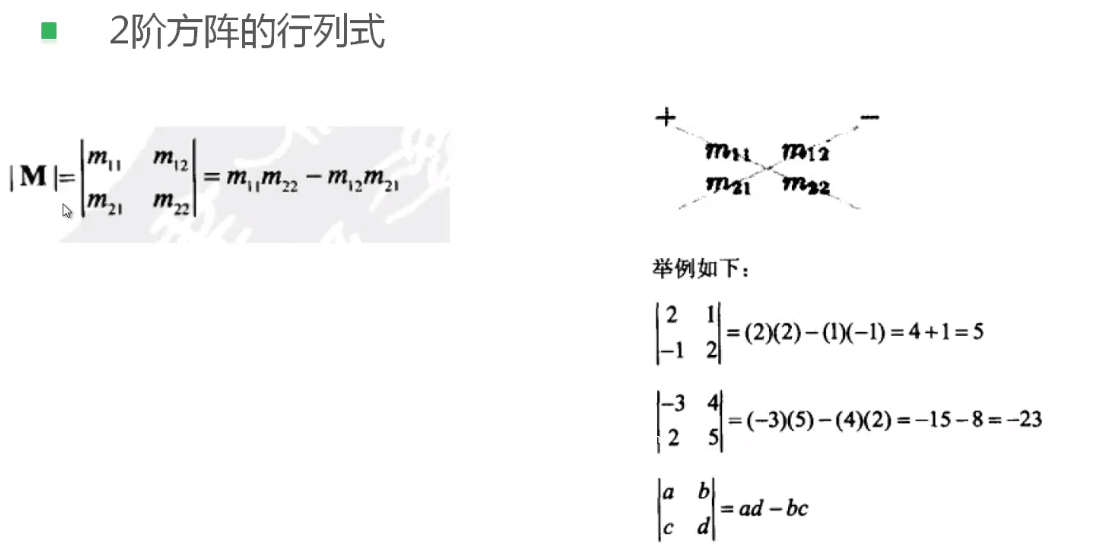


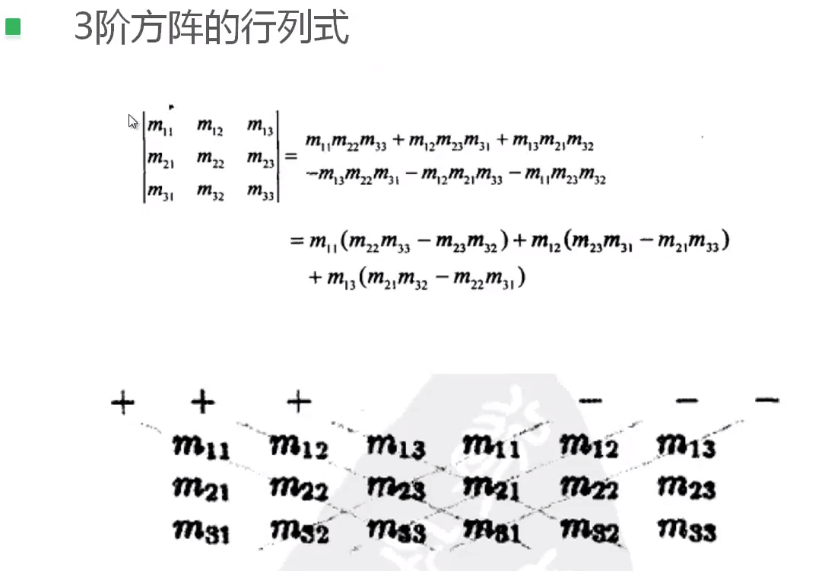
矩阵乘法

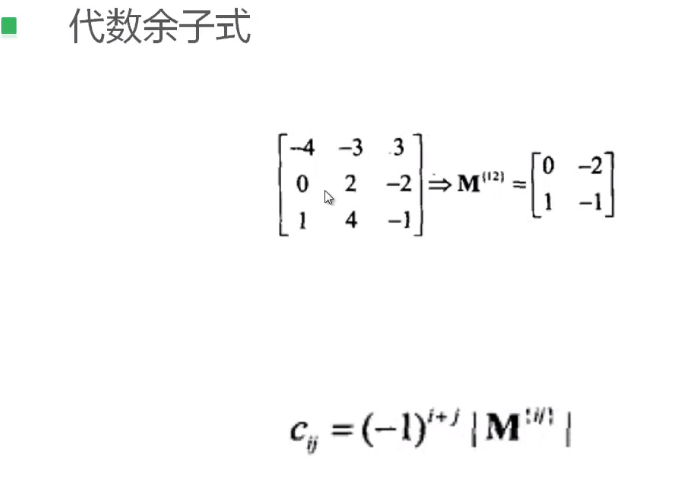


**方阵和行列式**

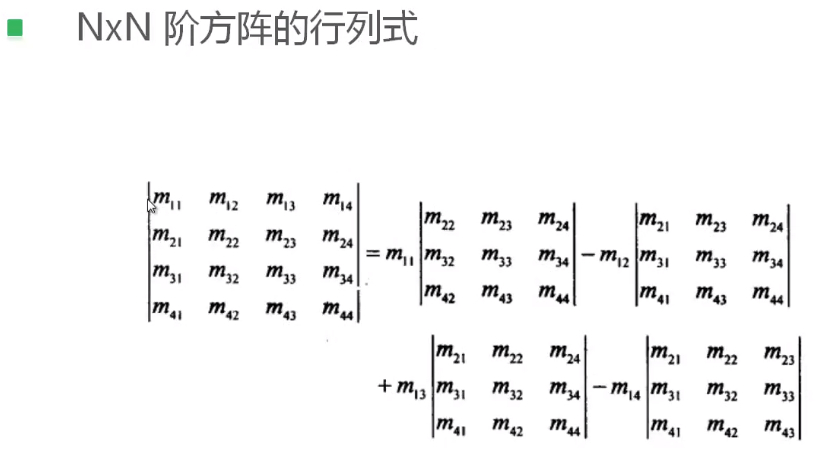
行列式的前提是方阵

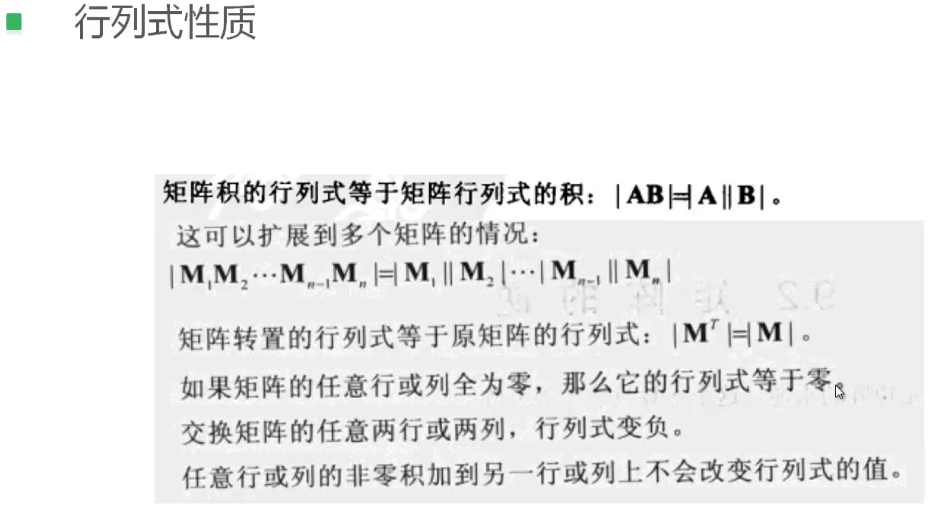






**||**



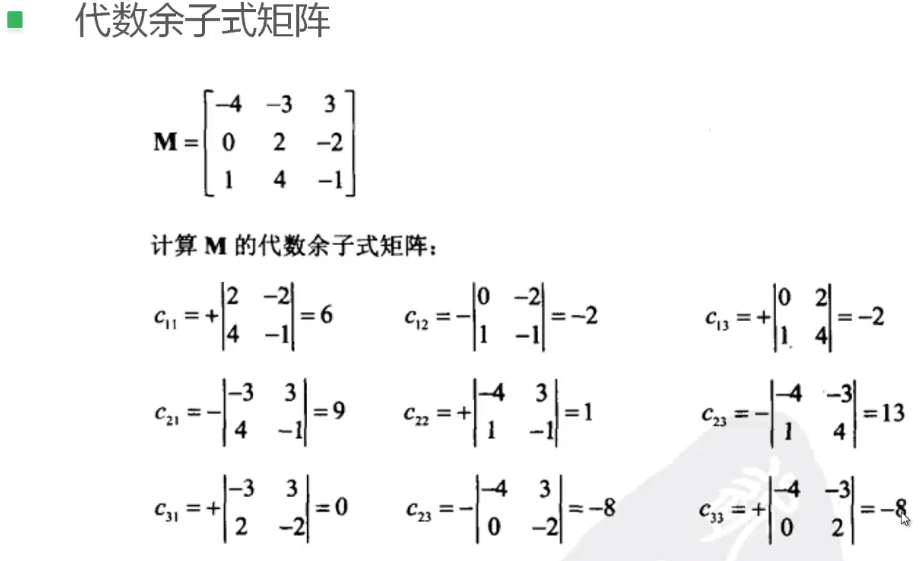


**矩阵的逆 & 单位矩阵**

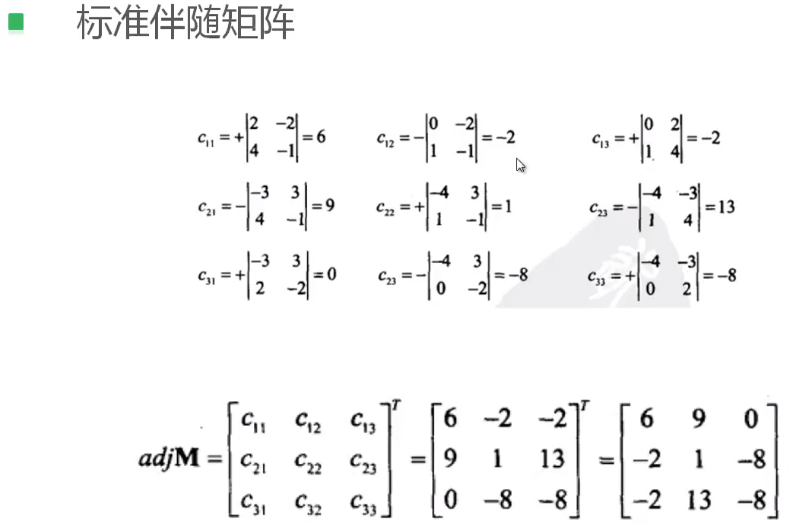


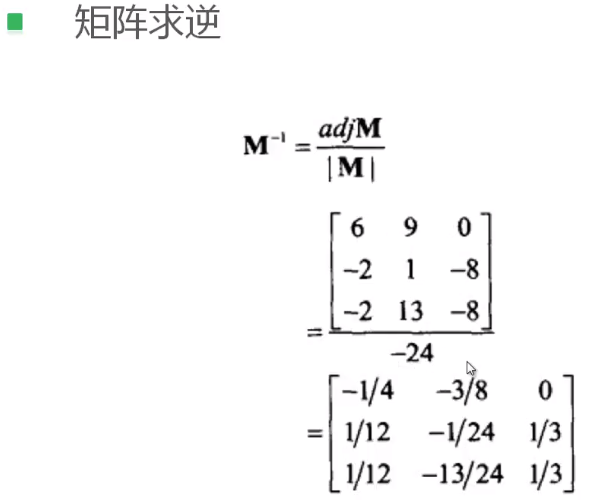
***计算矩阵的逆：***

***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***

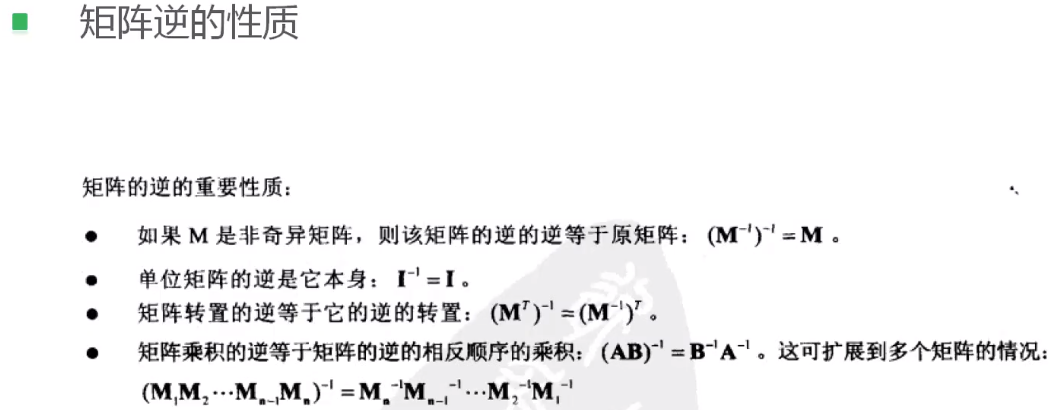


转置得：

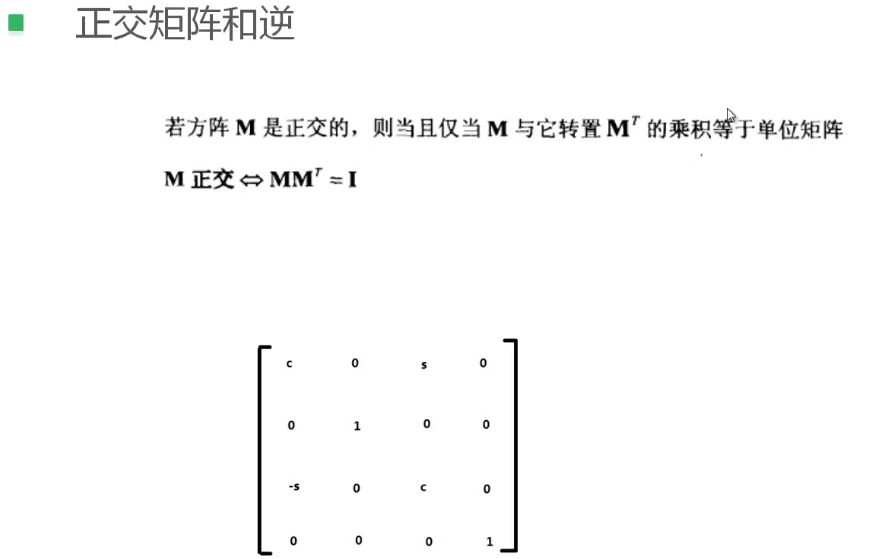




矩阵的逆 = 标准伴随矩阵 / 行列值

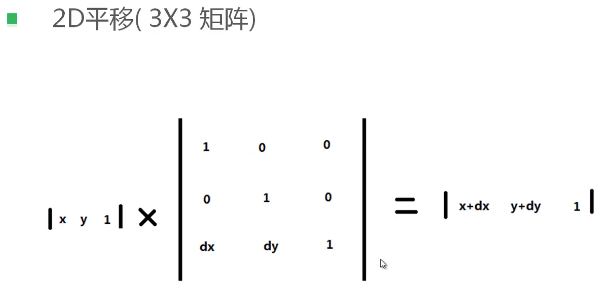


矩阵作用：可以撤销之前的矩阵变换



**矩阵变换**

***平移矩阵***



向量也需要扩展（满足矩阵乘法）

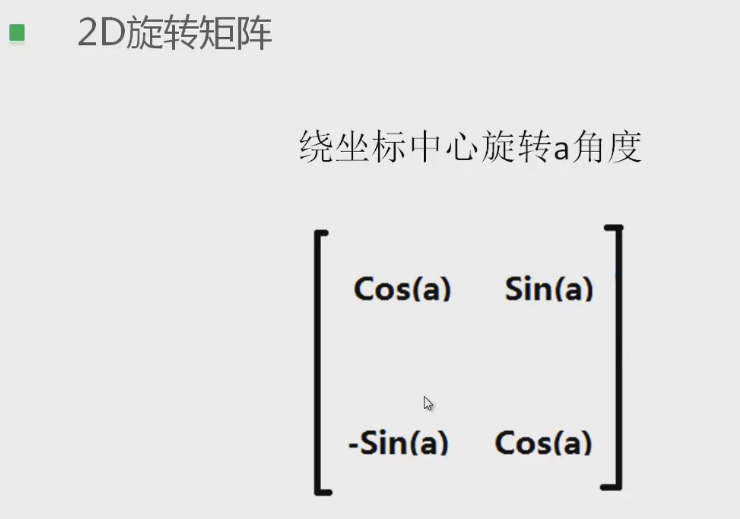
需要满足扩展的分量需要满足为1，保证第二次做变换仍可以正确平移



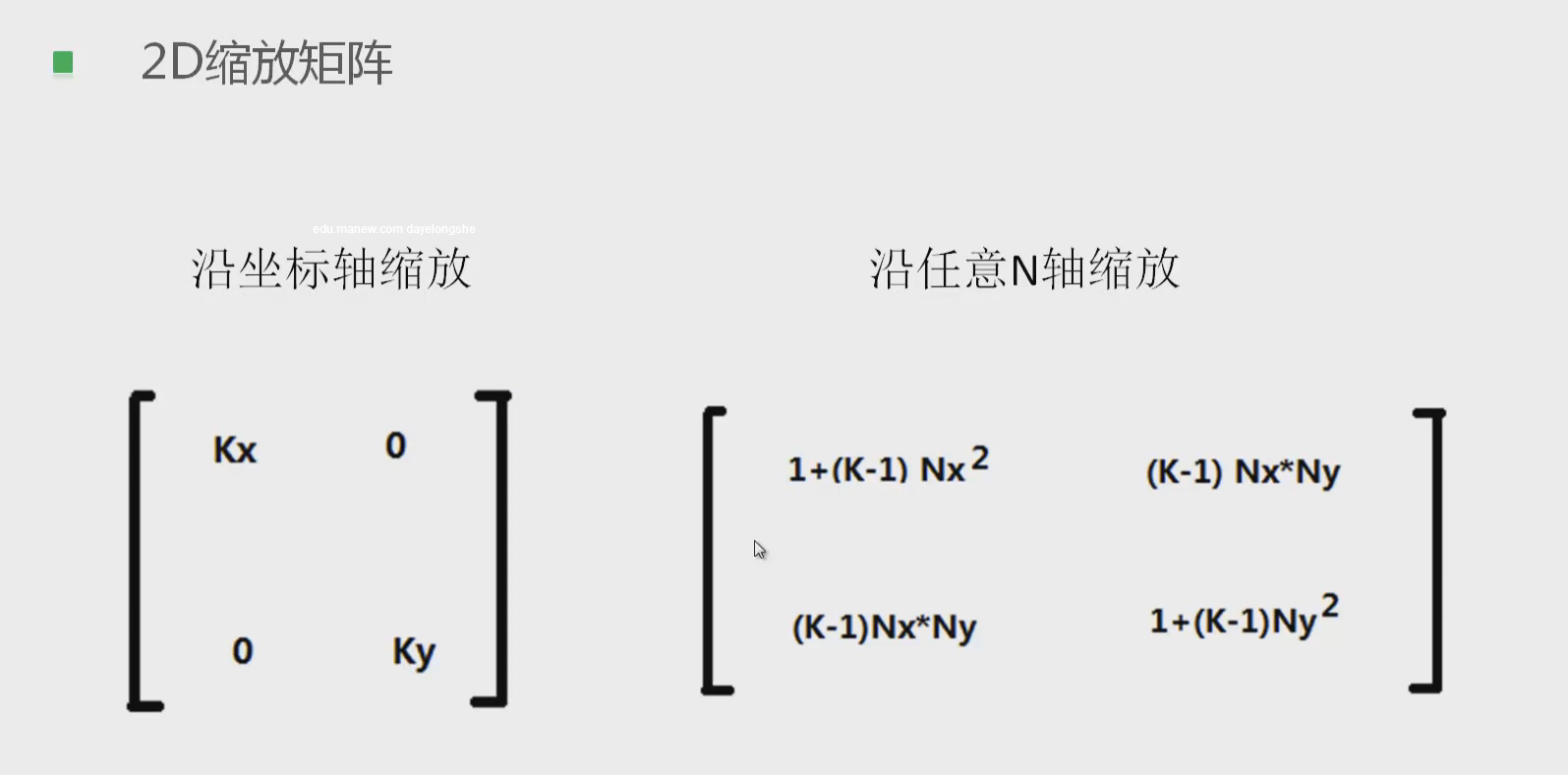
***旋转缩放矩阵***

（左乘 一个 需要变换的对象 矩阵）

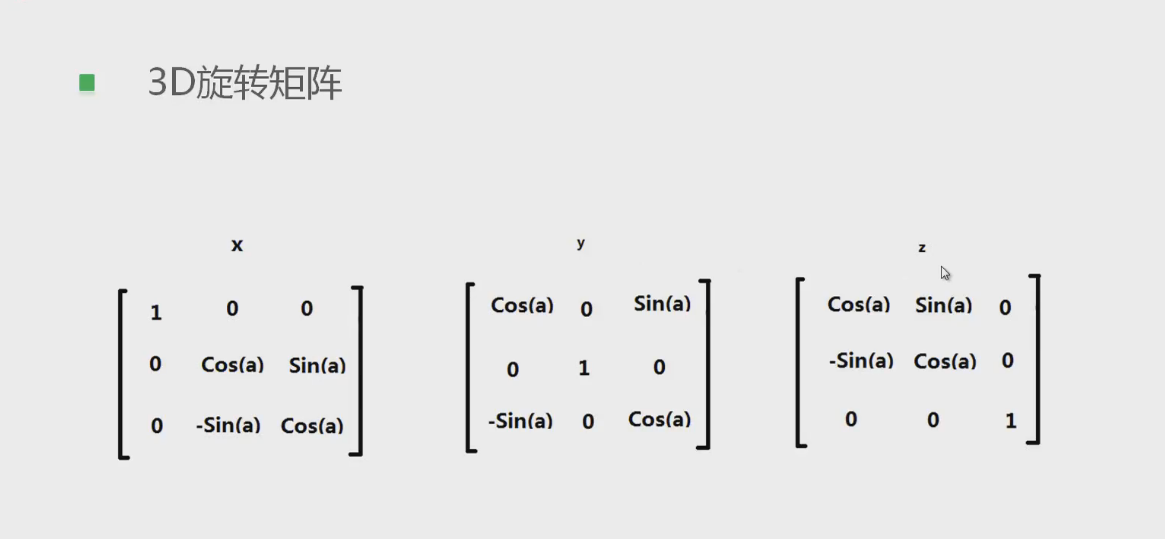
2D旋转矩阵



2D缩放矩阵



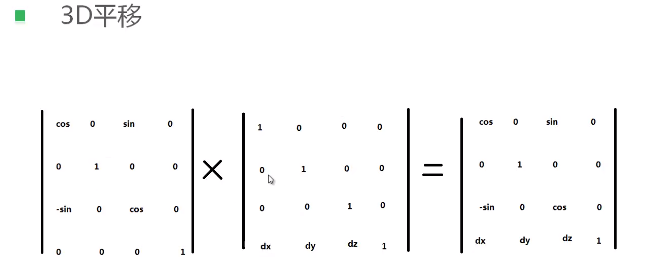
3D旋转矩阵（绕X 绕Y 绕Z）



绕任意轴n 缩放

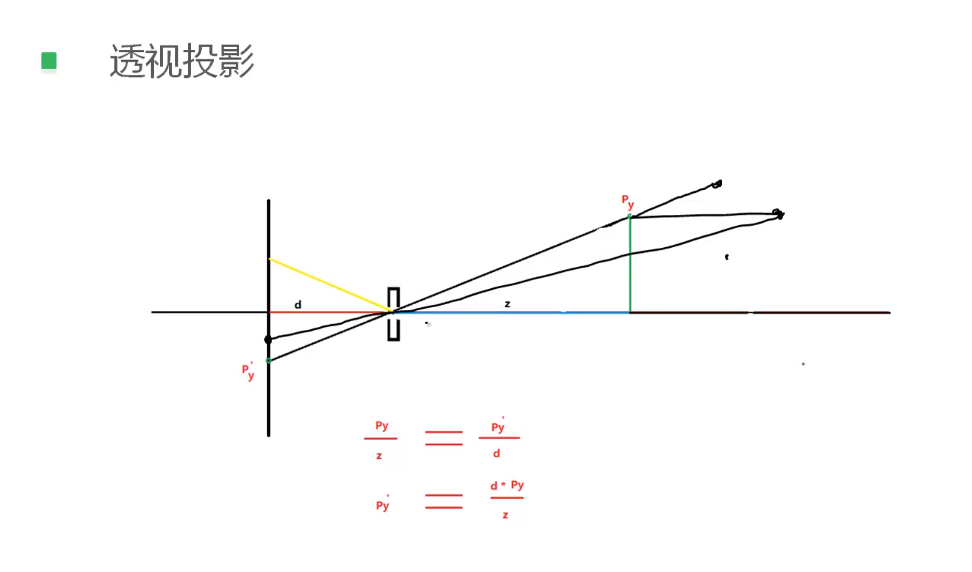


***平移和旋转矩阵的结合***



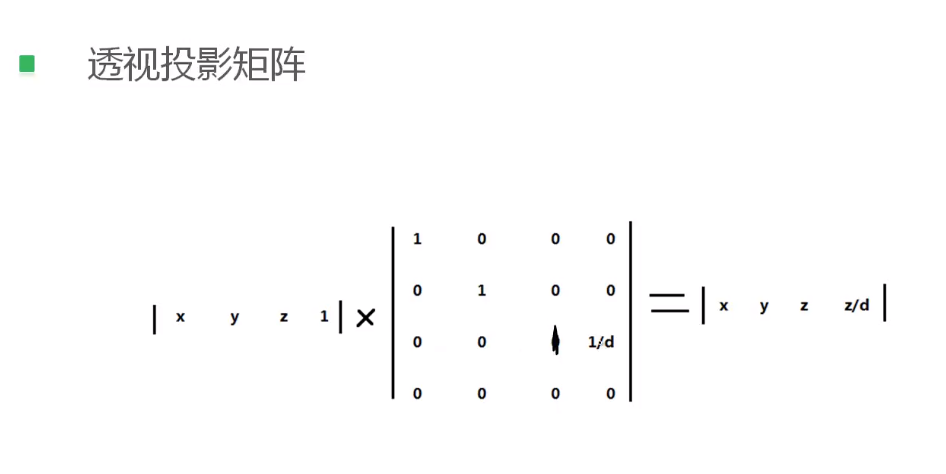
**旋转矩阵 （Y） X 平移矩阵**

***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））***



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）

