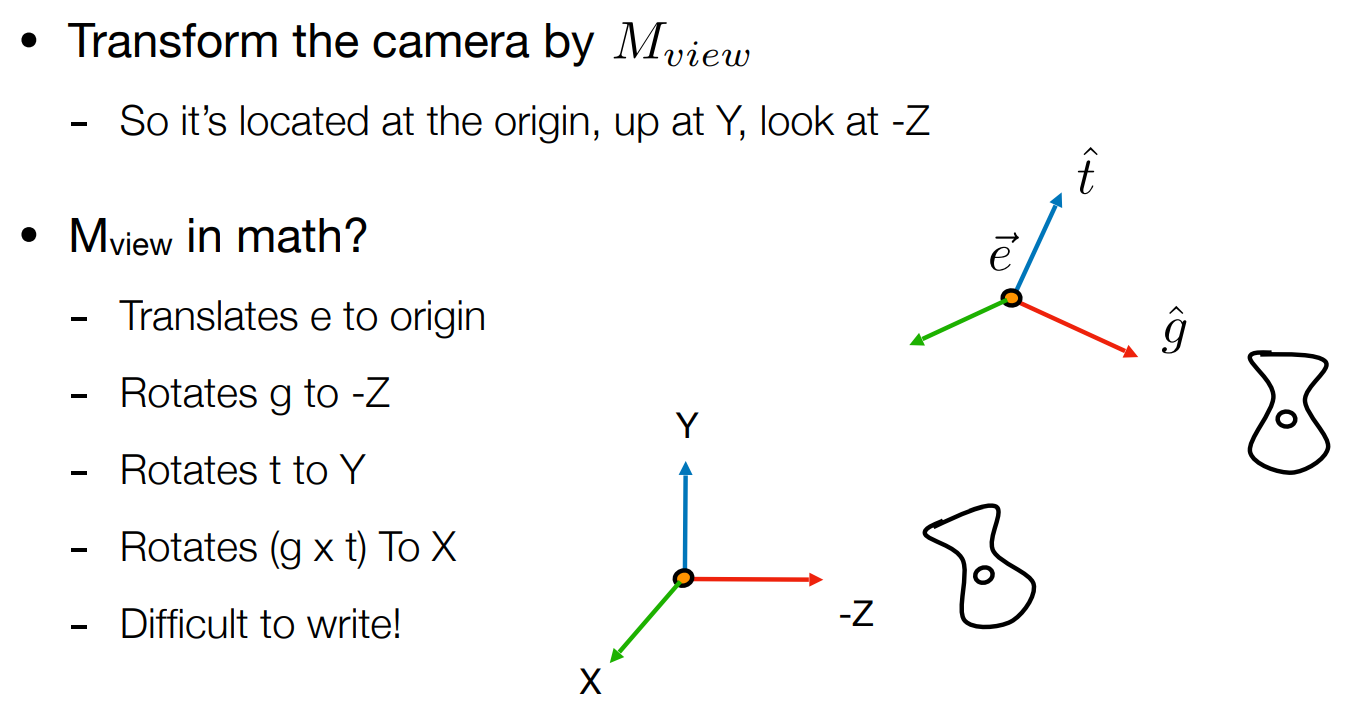
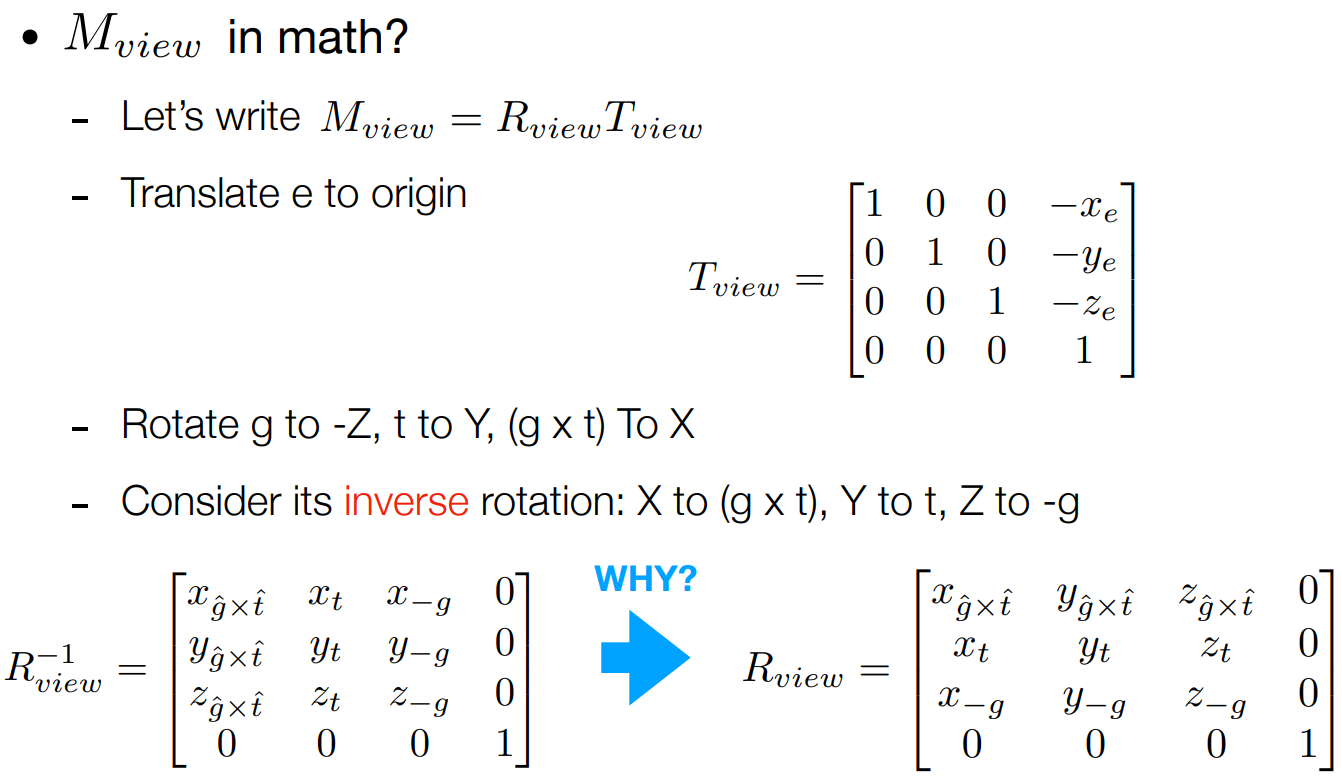
CG中将一个物体从3D世界投影到2D世界的过程分为4步：

1. 模型变换(model)：旋转平移缩放(都是刚性的)
2. 视图变换(view/camera)：从相机的观察点看场景
3. 投影变换(project)：将相机通过透视原理看到的场景映射到正则(canonical)空间中
   1. 透视变换(perspective)：将相机看到的frustum通过透视原理变换成cuboid
   2. 正交变换(orthographic)：将cuboid映射到正则空间中
4. 视口变换(viewport)：光栅化正则空间中的场景(常涉及缩放和偏移，以是的正则空间中的场景适应屏幕大小位置)

视图变换：

**我们往往假设我们是往z轴的负方向看向model的**，这和张林CV课有所不同。





对于线性变换矩阵Mview的作用过程，有以下两种理解方式：

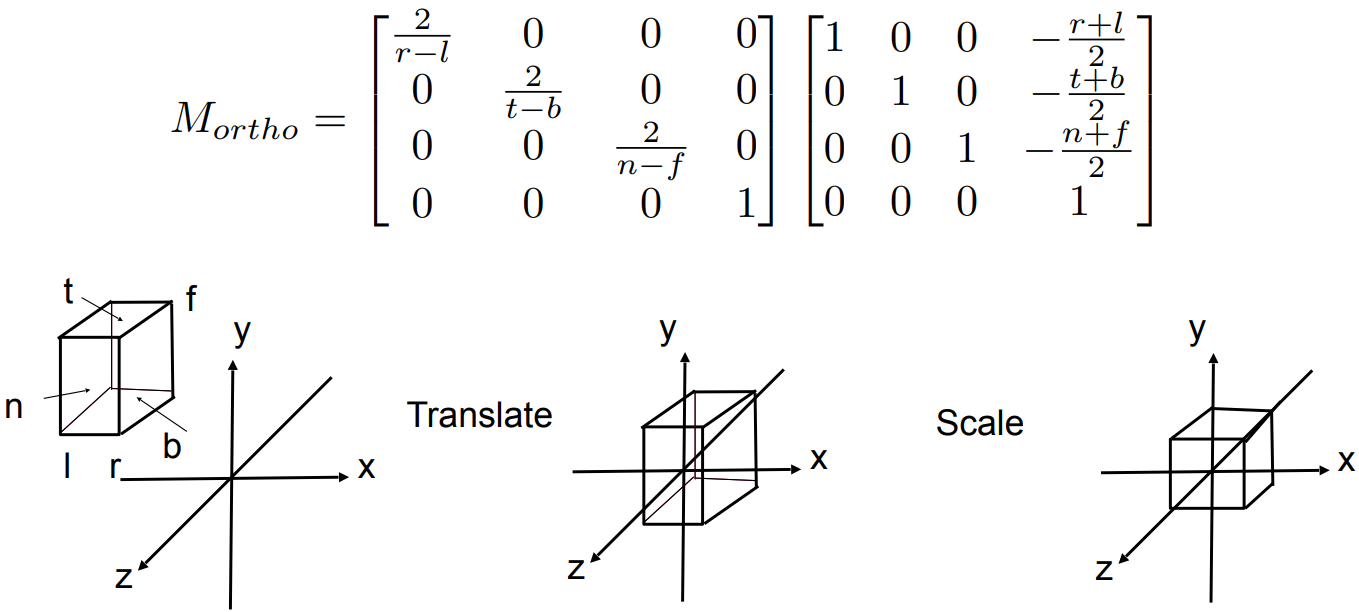
1. 复杂的理解方式：一开始，Model的坐标本来就是在x-y-z坐标系下表示的，我们要把它的坐标转换为在g-t-gxt坐标系下的表示方式。转换的句子为Mview，Mview实际上就代表了x-y-z坐标系下的各个轴在g-t-gxt坐标系下的表示方式，大概可以说Mview的每个列向量就是x-y-z坐标系的每个轴用g-t-gxt坐标系的轴线性组合的系数。
2. 简单的理解方式：g、t、gxt三根轴线和model的坐标都是在x-y-z坐标系下表示的，我们要把g,t,gxt三根轴线变换到与x-y-z轴重合，随便把model也随着g,t,gxt转换过来。于是，g,t,gxt的转换矩阵Mview也就和model的转换矩阵相同了。我们归根结底转换的不是g,t,gxt轴，而是model。

投影变换：

用一个透视投影矩阵作用一个场景的过程实际上是一个MVP(model-view-project)。

透视投影矩阵会将model变换到一个canonical(正则/规范/标准)空间中，即(-1,1)^3的空间。

透视投影矩阵Mpersp的组成部分包括正交投影矩阵Mortho，要先推导出Mortho:

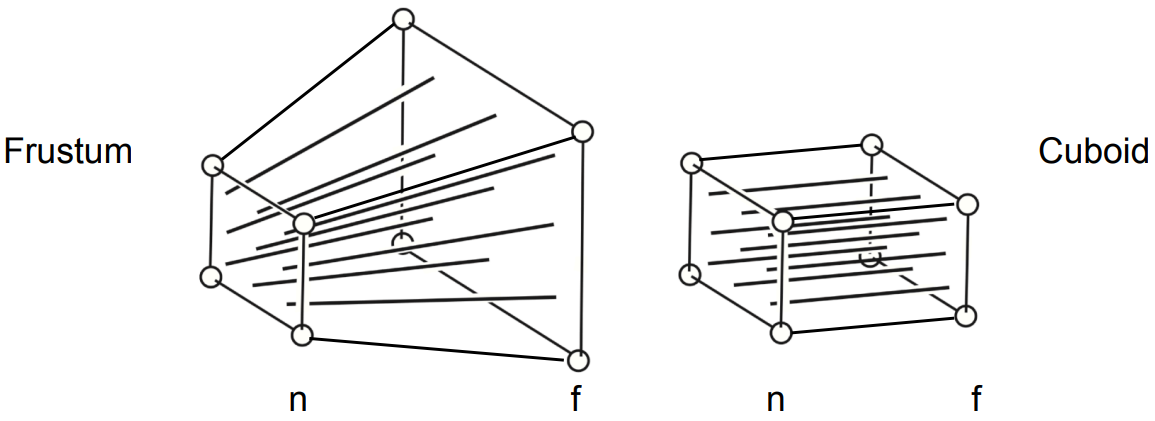


跟张林CV课上讲的3D坐标投影到2D像素平面上损失掉Z轴上信息不同，CG中透视投影矩阵作用后，要求保留Z轴信息(主要为了透视成像后保留多个model前后顺序)，所以每一步矩阵都是4\*4的。

透视投影矩阵的值跟model的near面的位置和far面的位置有关，分别记其在相机光芯坐标系Z轴上的坐标为n和f。

推导透视投影矩阵的基础假设有：

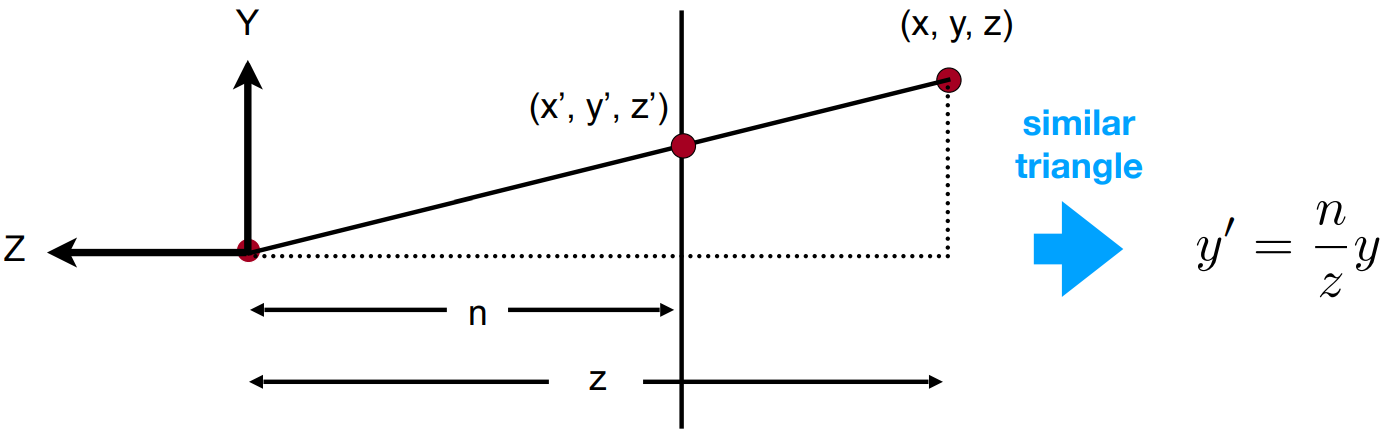
1. 从相机光芯看出去，被看到的空间是一个frustum。我们要先把frustum转换成cuboid，然后再立方投影到成像平面上。也就是说，我们要把透视投影简化成正交投影。
2. 透视投影矩阵Mpersp=Mortho \* Mpersp->ortho。Mpersp->ortho将透视投影转化成立方投影的矩阵，Mortho是正交投影的矩阵。Mortho很简单。关键在Mpersp->ortho，这也就是第1点提到的“将相机看到的frustum转换成cuboid”的矩阵。

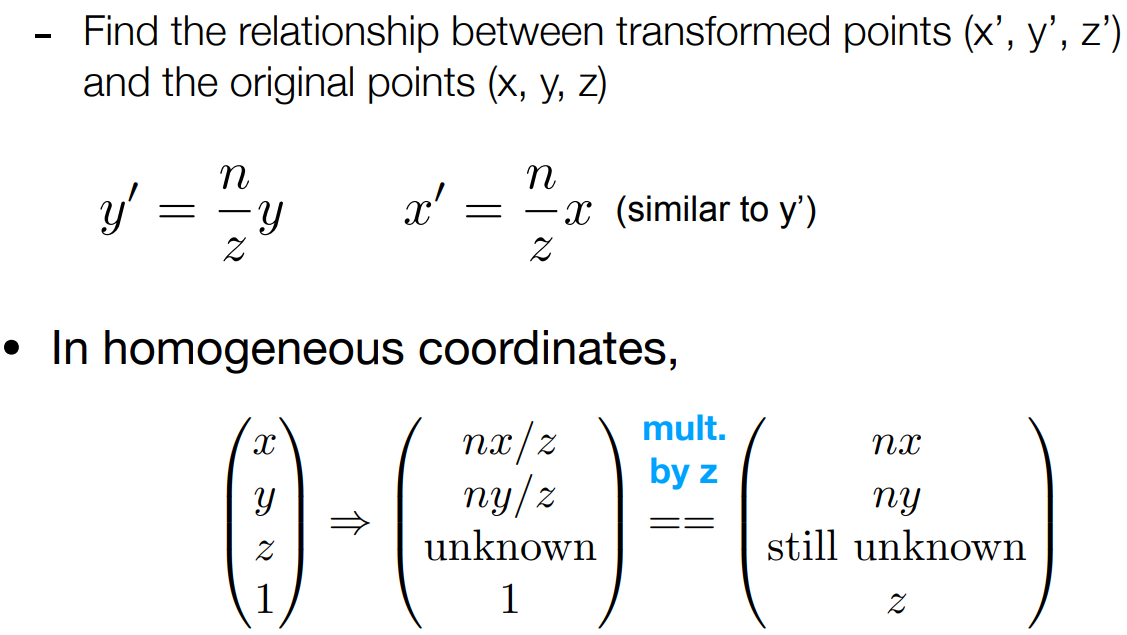


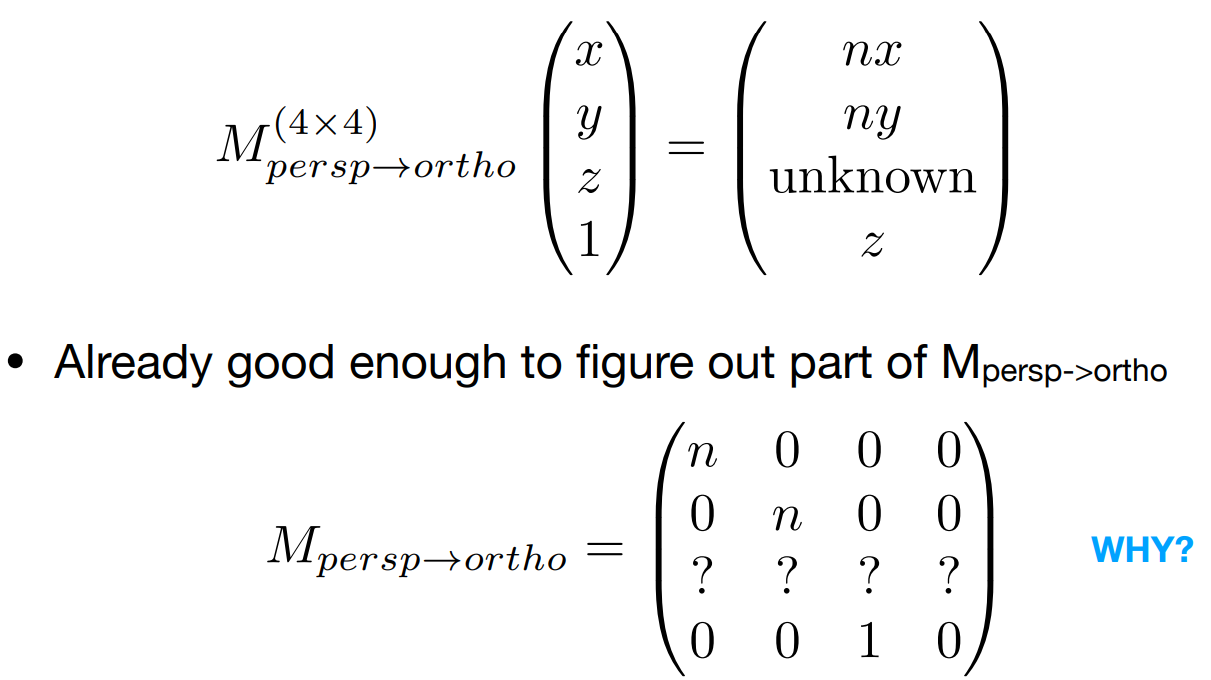


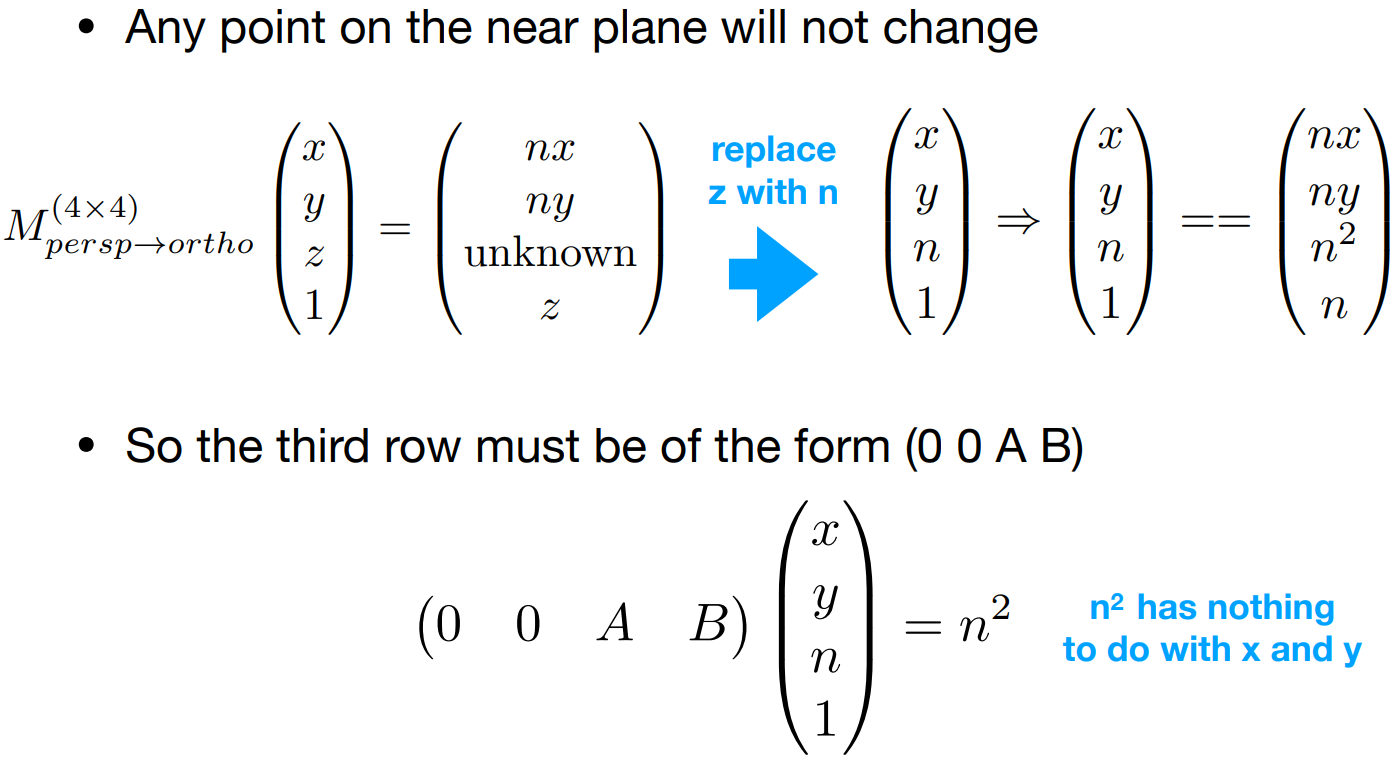
1. model的每个点从Frustum空间转换到cuboid空间的过程是一样的，所以Mpersp->ortho中的元素不能含有x,y,z
2. 要求Mpersp->ortho，我们还要对frustum->cuboid引入4点假设：
   1. n面的点x,y,z坐标都不变
   2. f面的z坐标不变
   3. n和f之间的面上的点的z坐标变化后的新位置不确定
   4. z坐标变换的过程和x,y在Mpersp->ortho上的分量无关

求解Mpersp->ortho的过程如下：

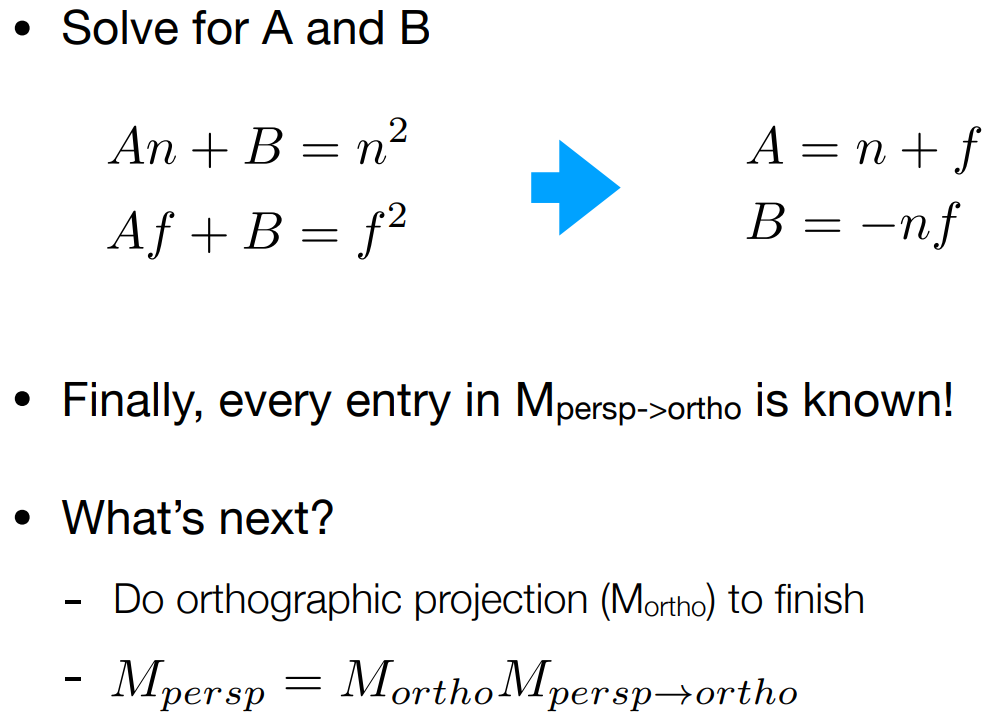




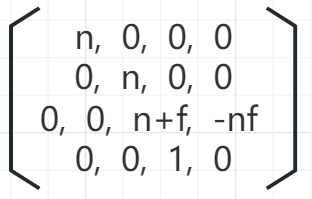








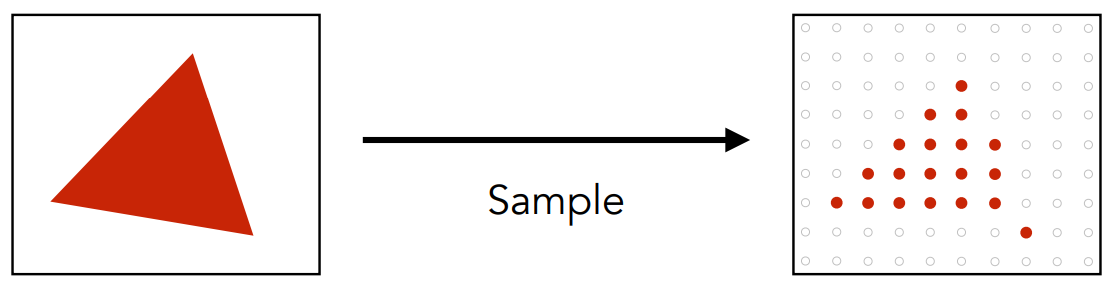
总之，最后的Mpersp->ortho我推导出来应该是：



走样Aliasing:

走样就是失真，例如用手机去拍摄电脑屏幕产生摩尔纹，又例如图像锯齿。

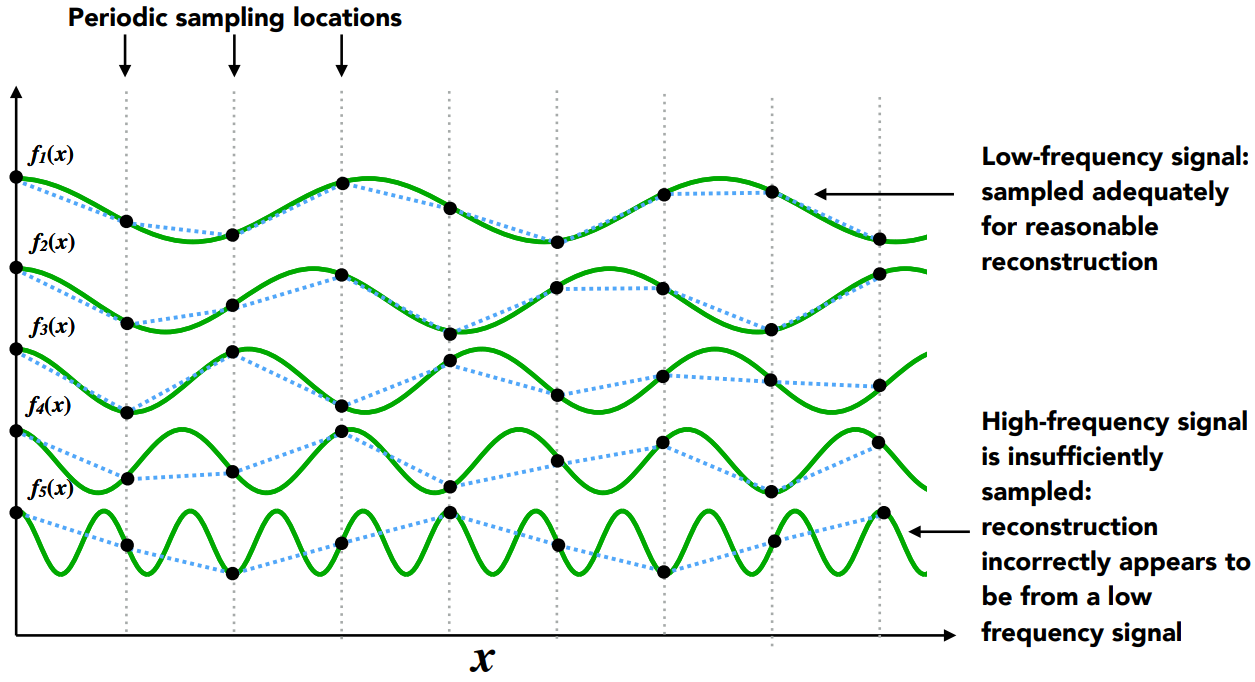
走样的根本原因是采样频率过低，例如下面，采样频率过低/像素点过稀疏，导致走样/锯齿：



对aliasing有两种理解：

1. 简单理解：

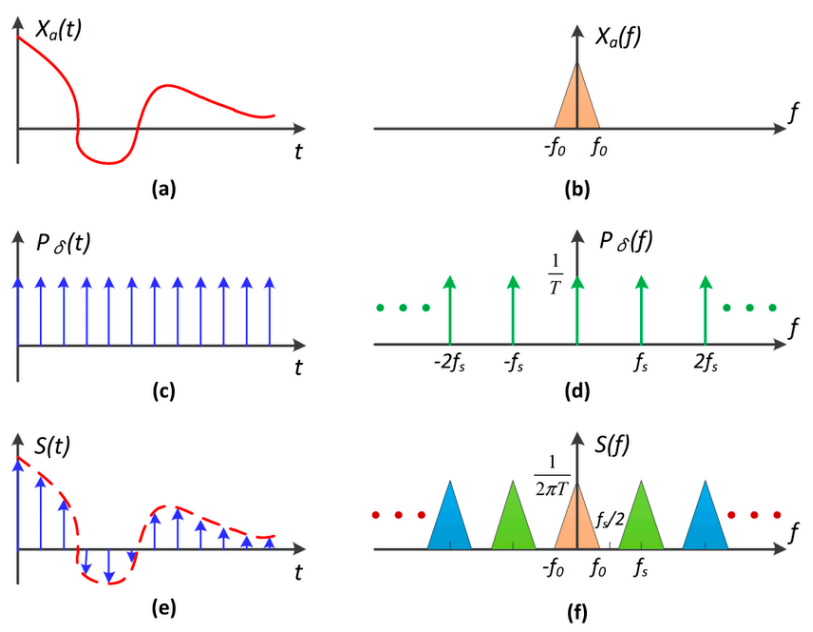
采样频率跟不上信号频率，导致采样得到的样本点拟合不出原始信号



1. 采样的本质是什么？采样的本质是对原始信号频谱的重复。

首先将采样的“冲激函数”通过傅里叶变换转换到频域，然后将原始信号的频谱的中心挨个在冲激函数的高幅值频率上重复。

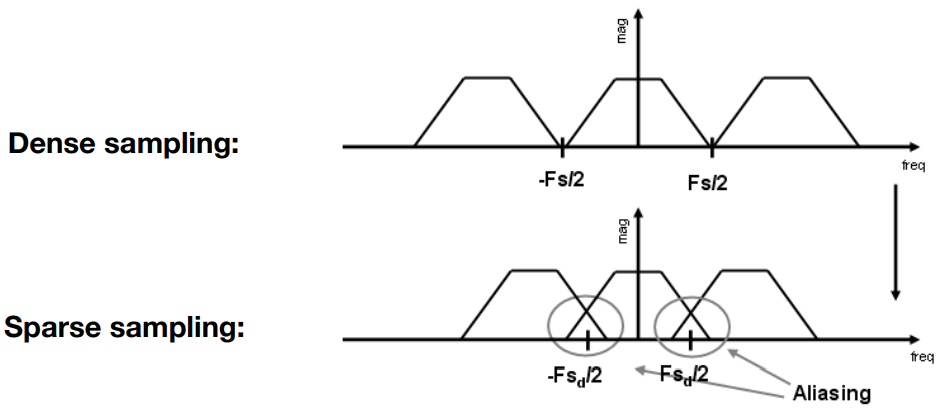
假如采样过于稀疏，冲激函数频率过低，这会导致原始信号的频谱交叠重合过多，故而走样。



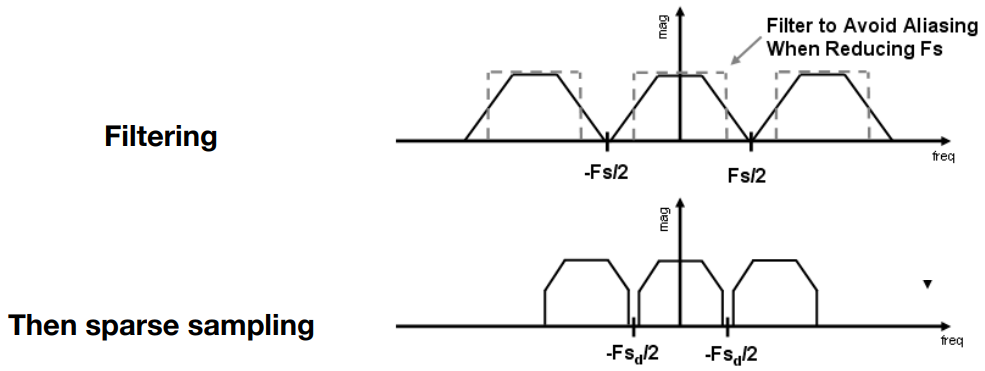
抗走样(antialiasing)的方法：

1. 提高分辨率(基于对aliasing的第一种理解)
2. 对原始信号进行低通滤波(基于对aliasing的第二种理解)：

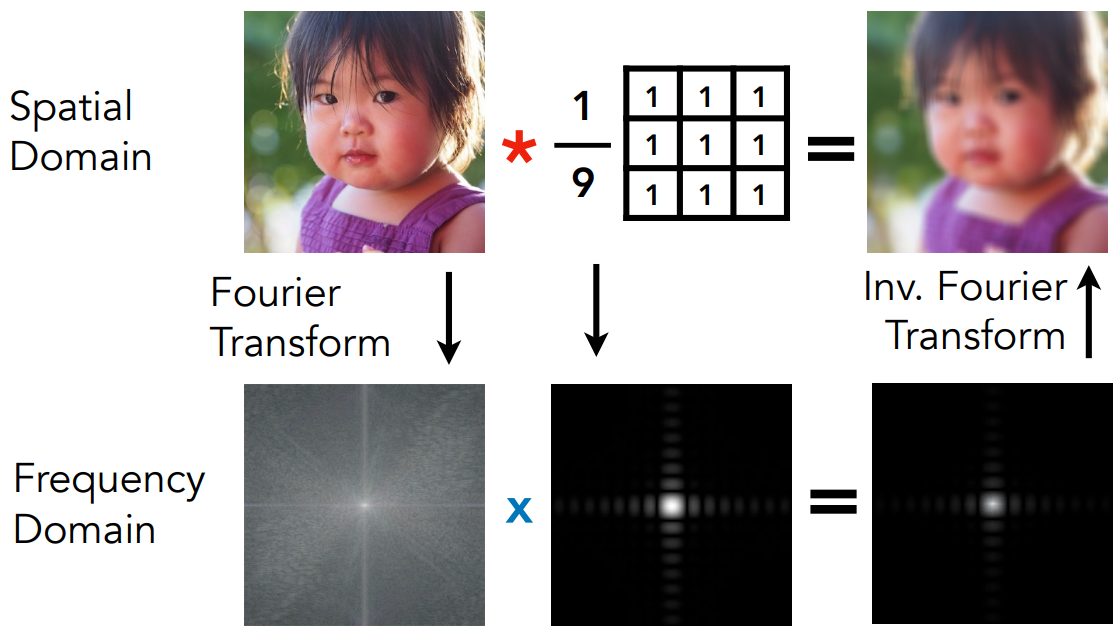
对于未打通滤波的原始信号，低频采样导致相互交叠产生aliasing的部分往往是原始信号的高频部分：



于是我们通过低通滤波裁掉高频部分，这样低频采样就不会让原始信号的频谱产生交叠：

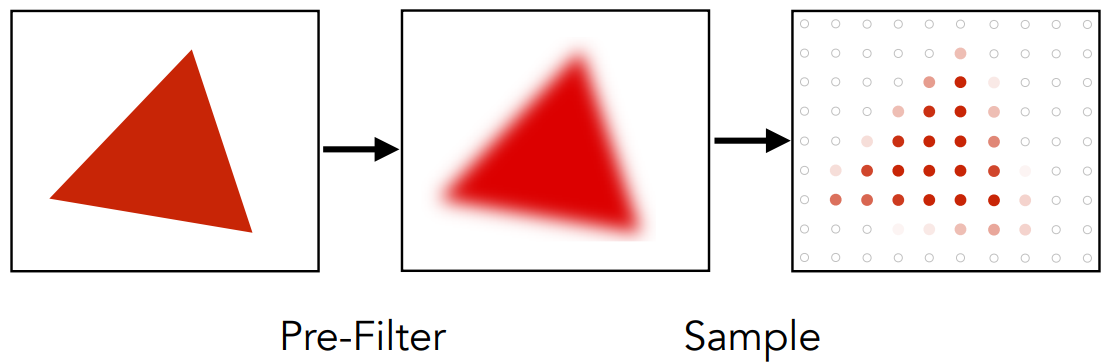


这里对滤波做一个补充说明，我们在空域用卷积对原始信号做的滤波，也可以对应转换成频率的滤波，即将空域的卷积核先补0后傅里叶变换：

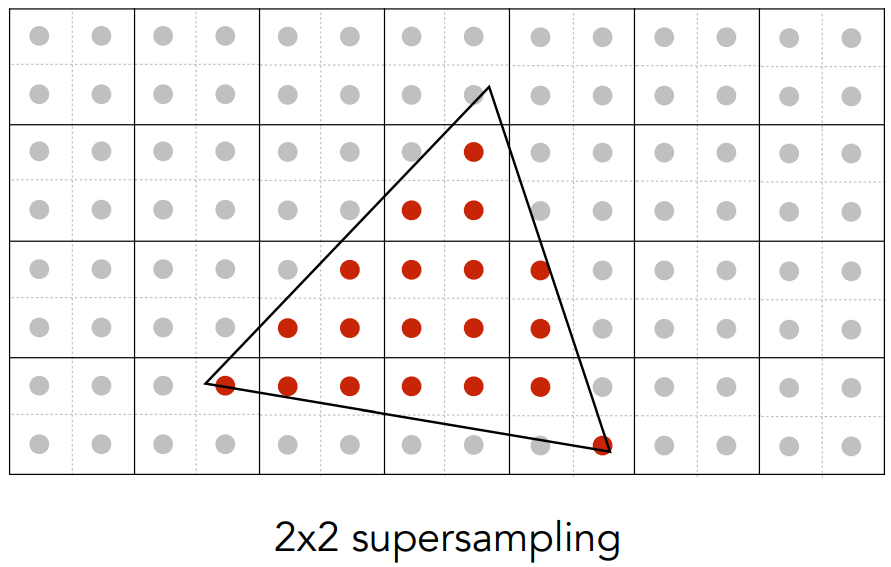


对原始信号线进行低通滤波再采样的抗走样的例子如：

* 1. 先对图片做高斯模糊后采样：



* 1. MSAA(Multisample Anti-aliasing)，看上去好像是提高采样频率，但实际上像素总数并没有变，还是先低通滤波后采样：



1. 先采样后模糊，这与第2种刚好相反，但它的效果往往很差，因为它相当于是先采样得到混叠的信号，后用低通滤波对信号频谱进行截断，这样混叠的信号就还是混叠的信号。

可见性/遮挡：

画家算法

深度缓冲算法(深度图像)

着色Shading：

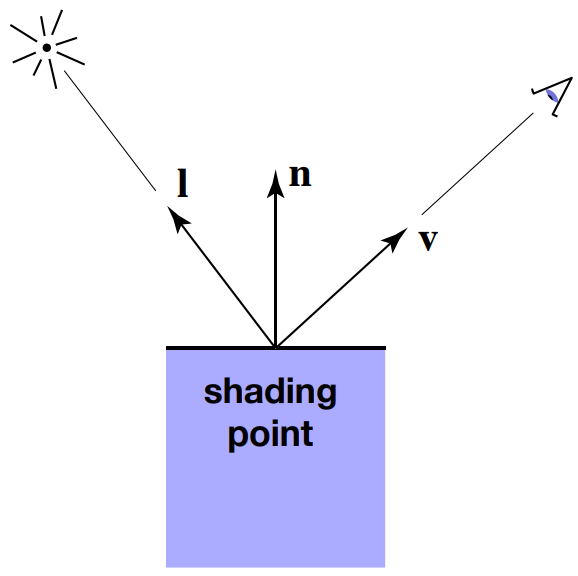
闫令琪：“就是将不同材质应用于不同的模型。着色和材质没什么区别，不同的材质就是不同的着色方法产生的结果。”

着色是shade，不是shadow，着色时我们只关注着色点，一个着色点是局部的，我们在对它着色时不考虑其他物体的存在，即便有其他物体遮挡了它我们也不管，所以在着色操作中，永远不可能有阴影。

一种典型的着色模型叫做Blinn-Phong模型，它基于以下几点假设：

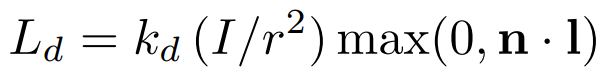
1. Blinn-Phong模型关注的是物体表面光照的接收与反射。
2. Blinn-Phong假设在环境光/漫反射/镜面高光产生之前，物体颜色是黑色的(这一点是我猜的，我加上去的)
3. Blinn-Phong模型只考虑光源到着色点的能量耗散，不关注着色点到人眼的能量耗散。
4. Blinn-Phong模型根据不同的光照程度将物体表面对光反射分为3种情况考虑：
   1. Spectual highlights(镜面高光)
   2. Duffuse reflect(漫反射)
   3. Ambient lighting(环境光)

Blinn-Phong模型对着色点的基本着色建模是这样的，其中l,n,v都是单位向量：

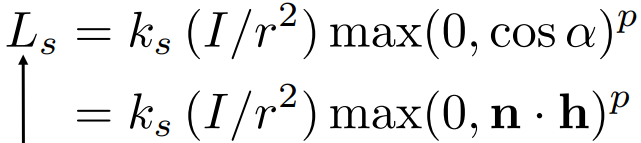


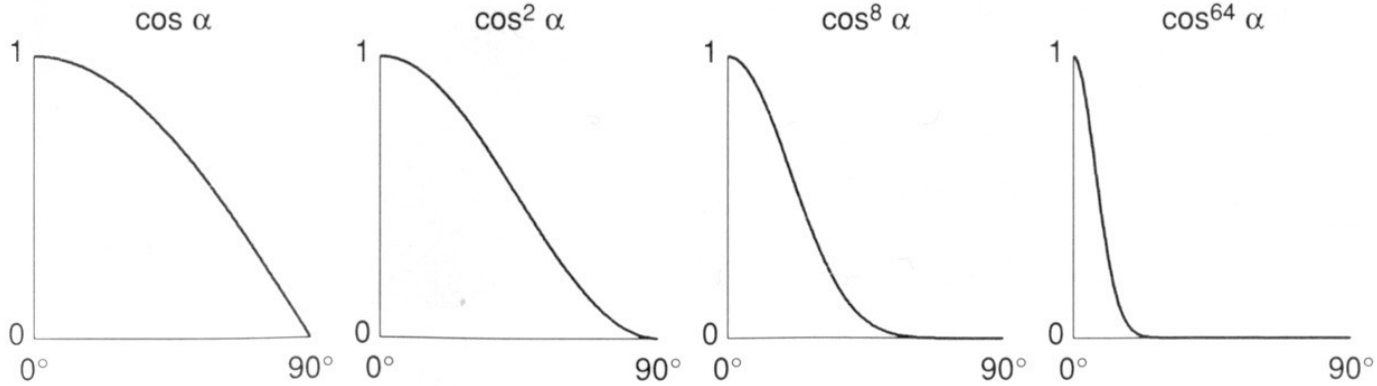
针对不同的反射光的情况，Blinn-Phong模型有以下3种公式：

1. 漫反射公式
   1. r是光源到着色点的距离，之所以是r^2是因为光波包络面是一个球体，球表面积是r^2
   2. max(0,n\*l)表示光到着色点的强度，l与n越平行，接收到的光照越强
   3. Kd是漫反射系数，它可以是一个标量，那么Ld就是灰度值，它也可以是一个RGB向量，那么Ld就是彩色
   4. Ld和v无关，这就是漫反射的精髓，四面八方都反射



1. 镜面反射公式
   1. h叫做半程向量，是l和v的正中
   2. 用n和h的接近程度来衡量人眼接收到的镜面反射光的强度大小，越接近就越符合真正的镜面反射的物理原理。那么为什么不直接用l关于n的对称向量R和v的接近程度来衡量呢？因为用半程向量更好算
   3. 相较于漫反射公式，没有用l和n的接近程度来衡量入射光的强度大小，这是一种的简化，源于镜面反射和漫反射的底层逻辑不同
   4. 多了一个系数p，p越大则对h和n偏移的容忍度越低，一般取100~200，也就是说要严格约束人眼接收到的反射光Ls接近于真正的镜面反射

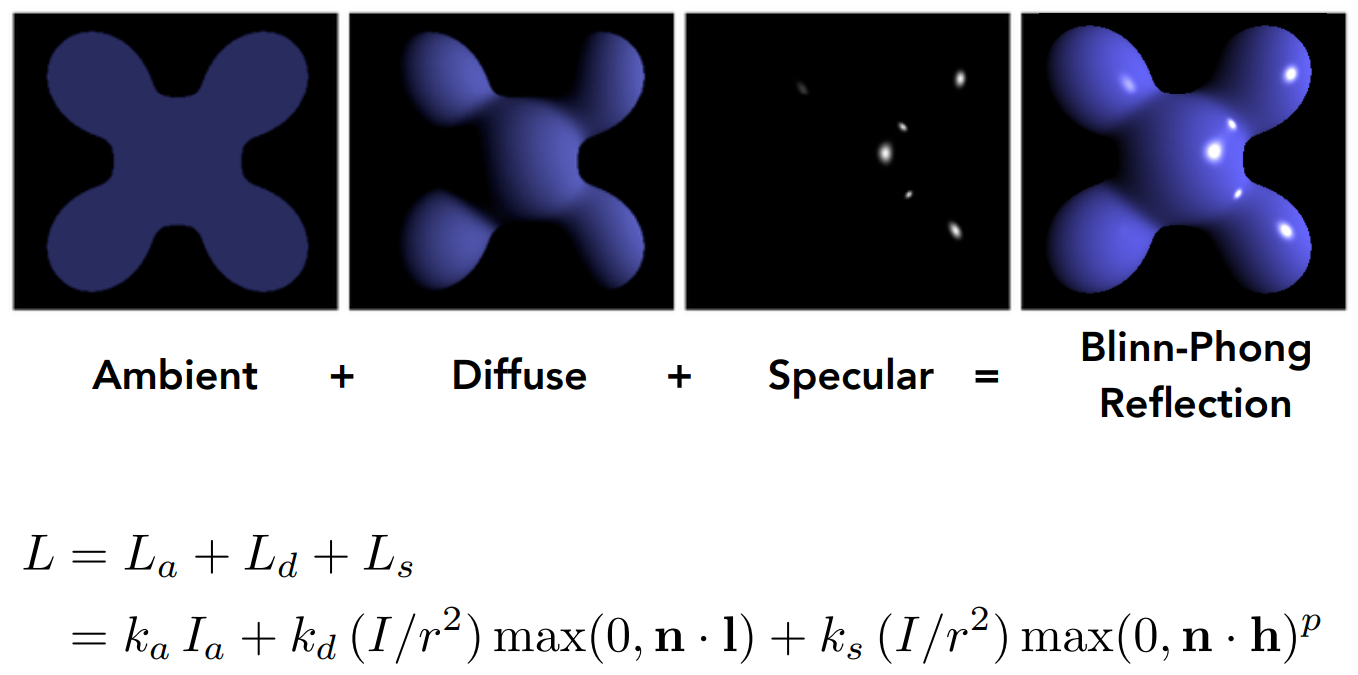




1. 环境光反射公式
   1. 该公式保证了没有地方是完全黑的，这是一个大胆的简化假设

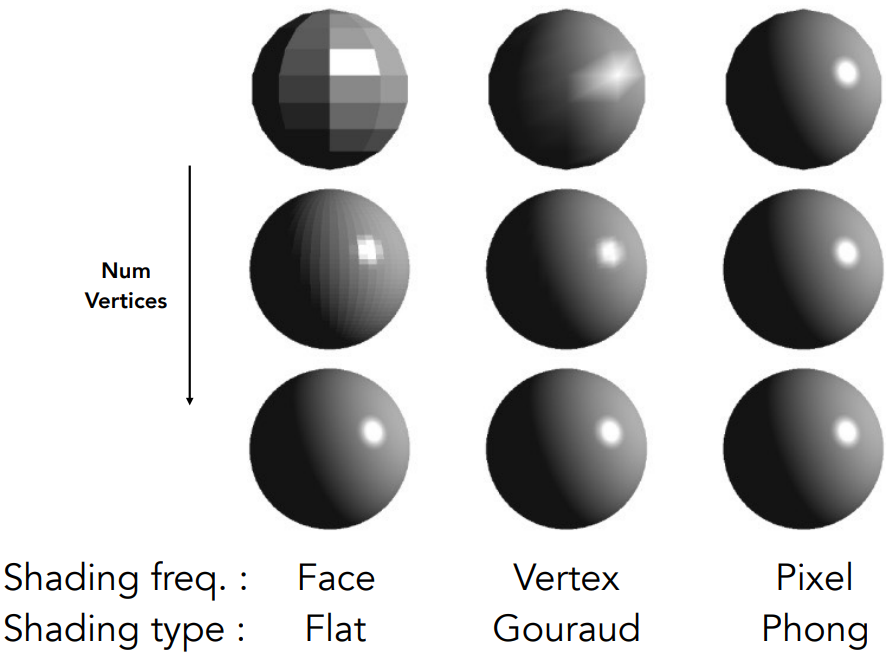


Blinn-Phong着色模型的总结如下：



着色模型建模了针对一个着色点的着色过程，那我们该如何把针对一个着色点的着色模型应用于整个3D model呢？

这里要引入一个着色频率的概念，分为逐面片着色/逐节点着色/逐像素着色：



1. 逐面片着色是对一个面片中心的点进行着色后运用于整个面片
2. 逐节点着色是对一个面片的几个角着色后将着色值通过插值运用于整个面片。它的难点在于求面片的角的节点的法向量。
3. 逐像素着色和逐节点着色接近，但更复杂，主要复杂在镜面反射光和漫反射光的处理上。求镜面反射光和漫反射光都需要法向量，对于一个面片上每个像素的法向量，我们的求法是：先求出各个角的法向量，像逐节点着色一样，然后通过插值求出每个像素的法向量，然后求每个像素着色值。

纹理texture：

纹理texture是对着色shading的改进补充。

纹理上也有像素，叫做“纹理元素/纹素”texel。

将Blinn-Phong着色模型应用于一个3D model时(不论是用什么着色频率)，可能会因为同一个3D model材质的变化(例如颜色的变化)而漫反射系数/镜面反射系数/环境光系数发生变化。

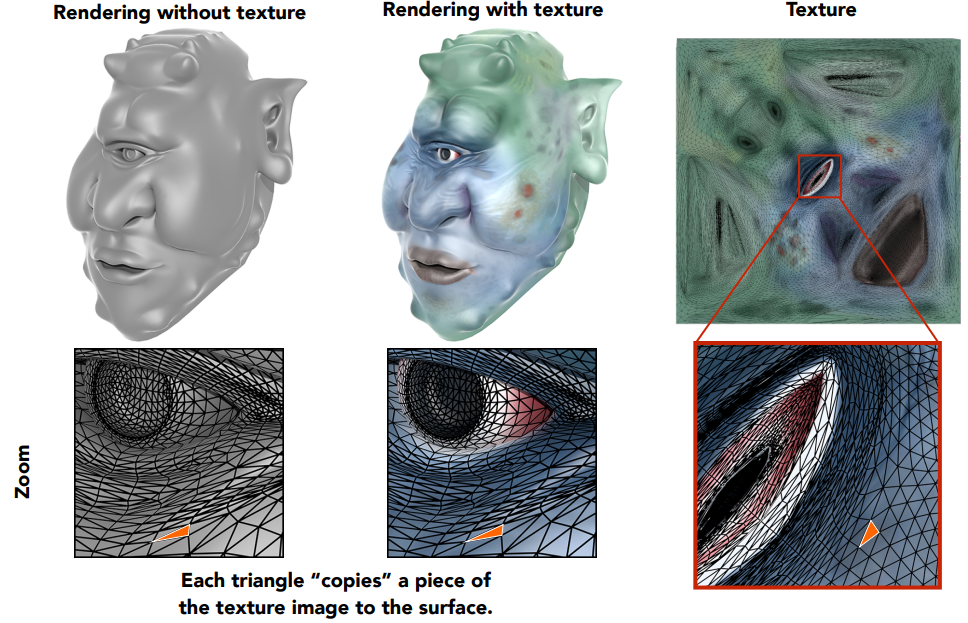
这时候，为了方便记录描述3D model表面不同部位的Kd/Ks/Ka，我们可能就需要将3D model的表面展开成一张1\*1的纹理图。

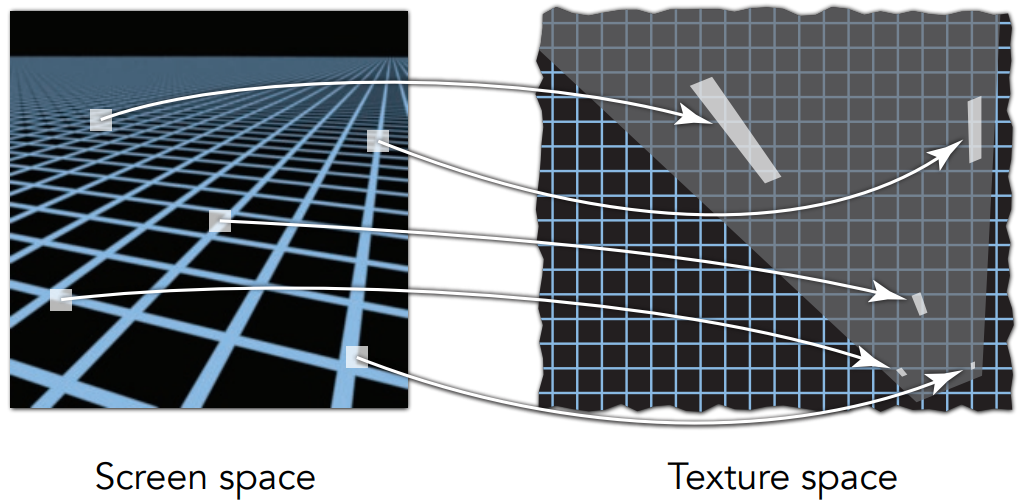
3D model表面是由一张张三角形面片构成的，纹理也是，3D model到纹理的映射是面片顶点到面片顶点的映射。

纹理不仅仅是在面片顶点记录属性，在面片内部也记录属性。

纹理图记录的不仅仅是Kd/Ks/Ka，还有其他很多属性。

现在是2023/9/4，就我目前所学，shading就是Kd/Ks/Ka都固定不变，而texture mapping就是使用Texture所记录的不同Kd/Ks/Ka所进行的shading，也就是说，shading和texture mapping本质上是一样的。但不管是shading还是texture mapping，都是对光栅化后的2D图像进行的处理(见于graphics pipline)。

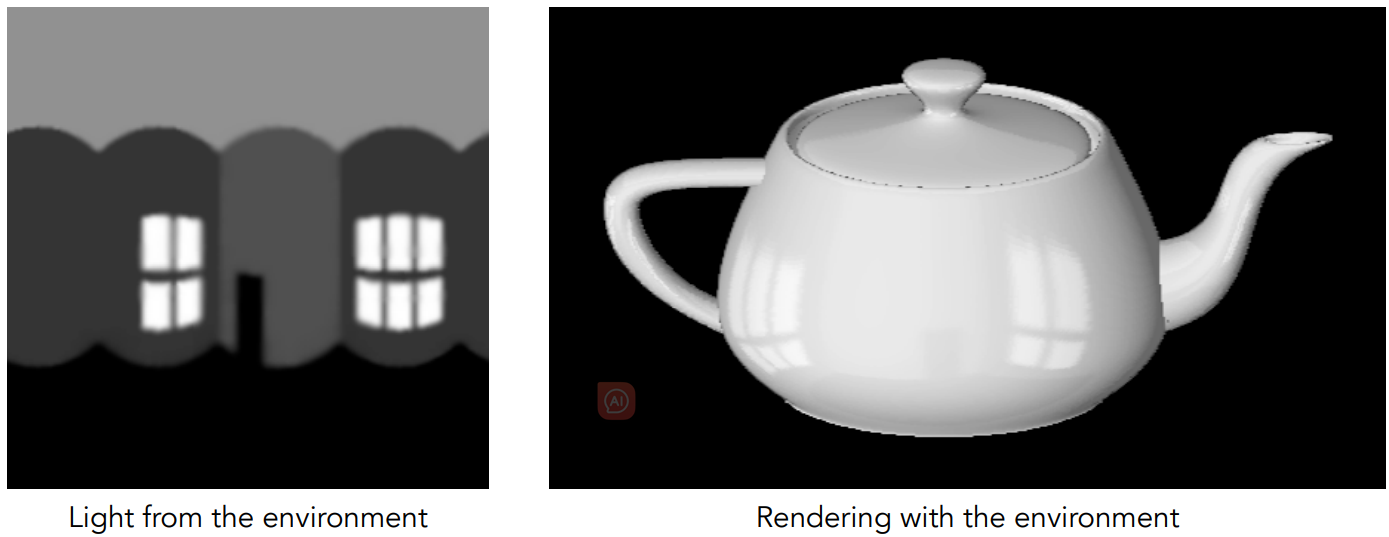




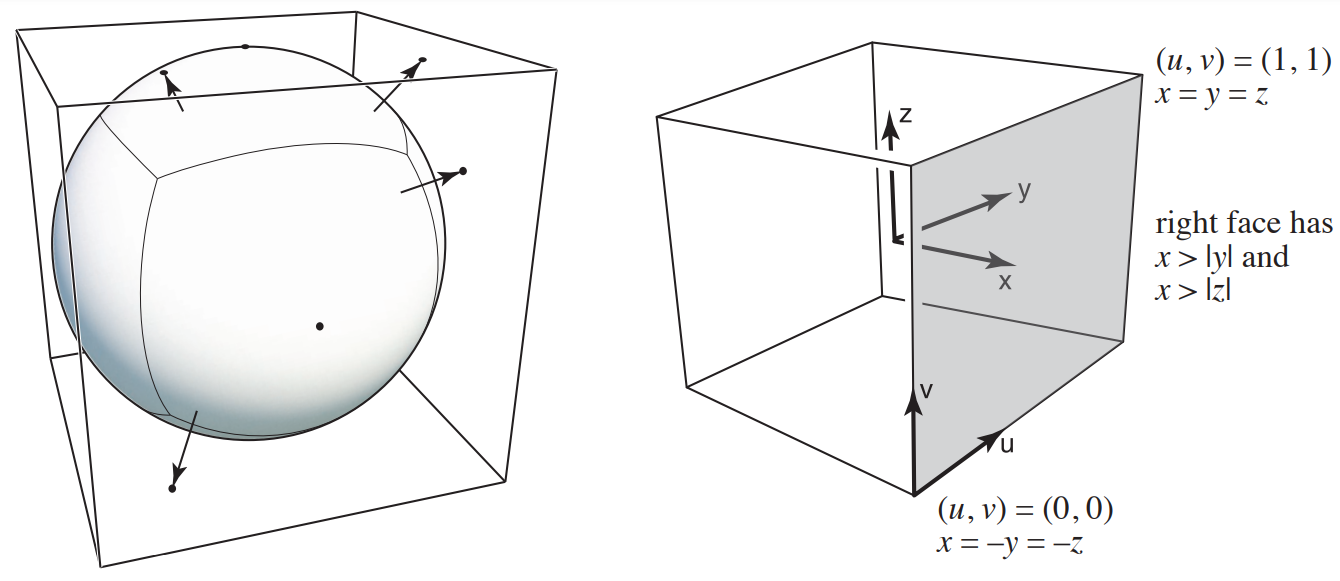
现在是2023/09/11，我刚刚学完lecture10。根据记录的属性的不同，texture可以分为很多种，常见的是用于帮助shading的记录Kd/Ks/Ka(这些系数本质上也是颜色，只不过物体最终的颜色是通过Blinn-Phong模型的公式计算出来的)的texture。当然，也有些texture更加一步到位，直接记录物体表面的最终RGB颜色，但这样有不足，就是不能适应不同的光照方向和光照强度。

texture也可能是球形的(spherical)或者正方体形(cuboid)的，为了解释理由，我这里补充一下环境光假设：

在Blinn-Phong模型中我们已经接触过环境光了，但我们当时假设环境光是均匀的，与方向无关的，所以公式是Ka\*I，但这里我再补充一下，公式里虽然没体现方向的概念，但Ka的取值与方向有关，我们在shading环境光反射的时候，要从texture查询该点对应的是哪个方向的环境光，这跟查询Ks/Kd是一样的。而且我们假设环境光的光源无限远，例如下面的例子中，虽然茶壶是在一个小房间中，但假设房间四壁打过来的环境光光源都是无限远的，所以不论房间中的茶壶如何移动，记录Ka属性的Texture都是不变的。虽然在真实物理世界中，茶壶移动了一下，那么打过来的环境光肯定不同了，texture肯定跟着变了。

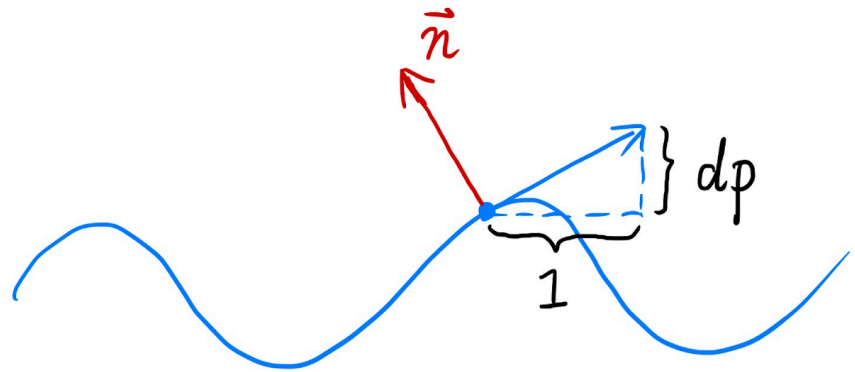
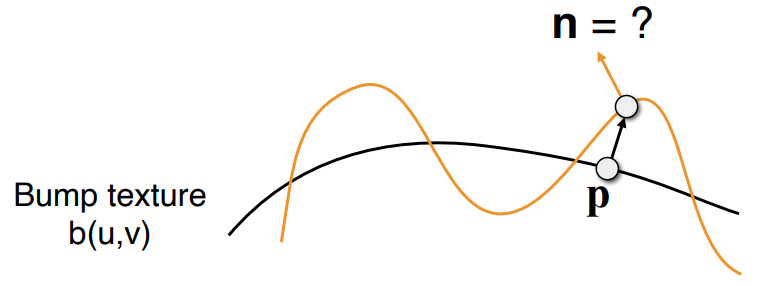


为了专门服务于环境光假设(不同方向的环境光Ka可能不同)，texture的形状也不一定是方形的，也可以是球形的或是正方体形的。球形的优点是随便想要某个方向的环境光Ka都能很快查询到，缺点是你若是把它展开，它的上下部分就会出现扭曲失真，就像是展开世界地图一样。正方体形的优点是若是展开成六个平面不会失真，但是查询某个方向的Ka比较麻烦，因为你还要先事先确定目标Ka是在哪个面上。



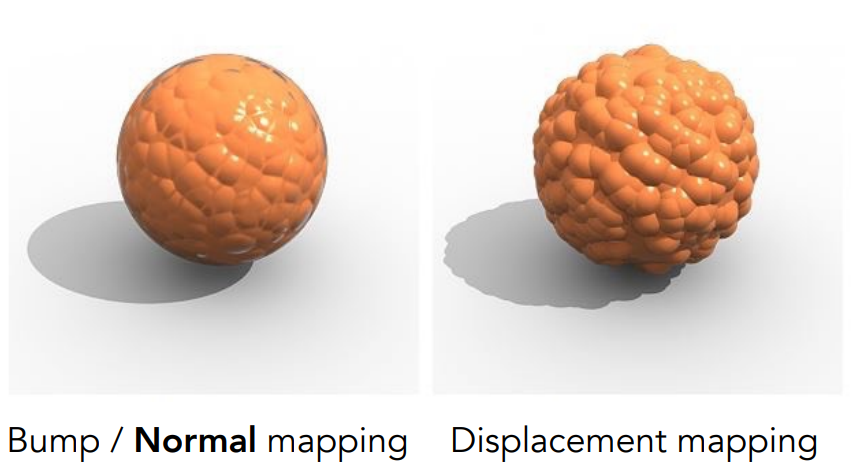
texture还有个典型功用。

有些物体表面是凹凸不平的，要表示这种物体所需要的三角形面片数量都极多，对于计算和存储都是很大的开销。于是我们可以在建模时只建立简单的大致的模型，然后在texture中记录表面各个点的“relative height”，也就是实际物体表面和大致模型表面之间凹凸的偏差，这种叫做bump texture：



有了relative height，你就能计算物体表面任一点的法线，代入Blinn-Phong模型，就能渲染出物体表面凹凸不平的效果。由上图你可知计算法线要先计算切线，由relative height计算切线的过程有一定偏差，那为什么texture不直接记录法线？当然可以，但是这样会消耗更多存储空间。若是textrue真的记录法线，一般也不是记录世界坐标系下的法线，而是将原本的法线假定为(0,0,1)，然后记录新的偏移的法线。

记录relative height的texture叫做Bump Mapping，记录法线的texture叫做Normal Mapping，效果基本一样。它们的效果如下图所示：



但是Bump/Normal mapping有缺点就是：1.它本质上是用表面变化偏移的法线来shading，并没有真的改变3D物体的形状，是欺骗人的眼睛，所以你看它一到球形边缘就露馅；2.由于并没有真的改变表面的形状，所以在shading后如果想做进一步的阴影投影，就会失真，你看上图左右两图的阴影区别。

为了解决Bump/Normal mapping的区别，我们引入了displacement mapping，即上面右图的效果。displacement mapping的texture和bump mapping texture的值是一模一样的，但是displacement mapping会真的改变物体表面的relative height，于是这就要求3D model表面的小三角形特别多，多到采样频率高于texture的小三角形数目。这又会给存储和计算带来巨大的开销。于是directX库开发了一种不开源的算法叫做动态曲面细分dynamic tessellation，它不要求原始3D model的面片数目高于texture，而是在texture贴图过程中自动打碎3D Model的面片，凹凸多的地方就碎成更多面片，凹凸少的地方就碎成更少面片，这是种自适应的算法。

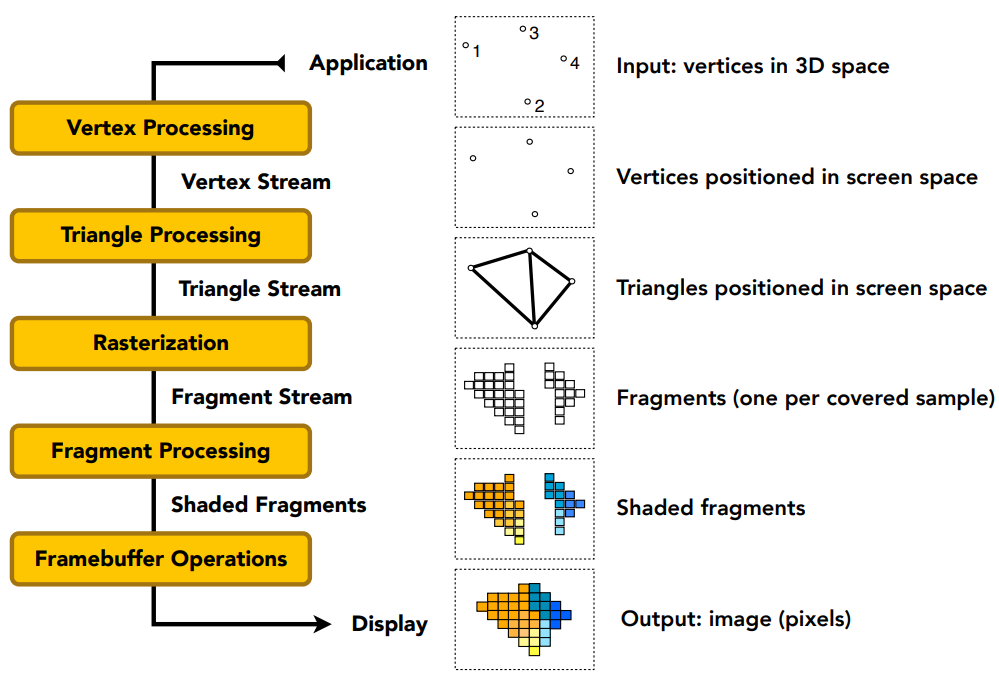
有些texture还是实心球体，也就是说它记录了一个3D model表面和内部的所有属性。

有些texture还用于专门记录阴影，你先对一个3D model进行简单shading，然后再把阴影texture贴上去。

图形管线graphics pipeline：

图形管线指的是把3D model转换为2D图像的基本流程。

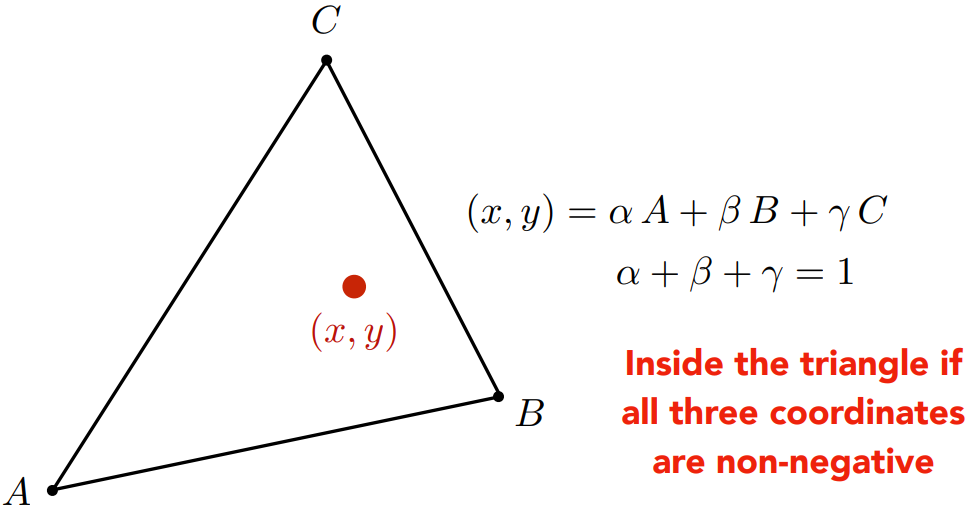
这里的fragment是openGL里的概念，在不考虑MSAA采样点的情况下，你可以当它是像素pixel。



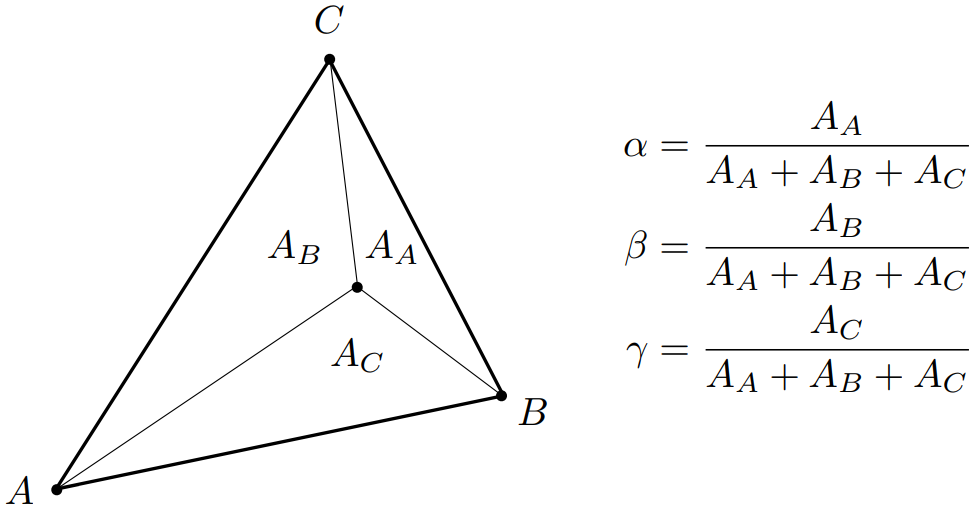
1. model/view/perspective transformation都是Vertex processing
2. 计算frame buffer上的某个像素在不在某个三角形中，以及Z-buffer算法是Rasterizer
3. 着色既有vertex processing又有fragment processing，这是因为不同的着色频率，Gouraud和Phong着色会用到vertex processing
4. Texture mapping也是既有vertex processing又有fragment processing

三角形重心坐标：

在三角形所处平面内，每一个点都有一个能用该三角形3个顶点的坐标的加权和表示的坐标，该坐标叫做三角形重心坐标。



由三角形重心坐标的推导：



可知：当3系数有负数时，说明该点不在三角形之内。当3系数之和不为1时，说明该点根本不在三角形所在平面内。

但是三角形重心坐标有不好的性质：

从旋转变换到欧氏变换到相似变换到仿射变换到射影变换，就只有相似变换及之前的变换能保证相似比不变，所以三角形所在空间经变换后，三角形重心坐标不变。但仿射变换和射影变换就不能保证。

比如在我们上面说的透视成像中，模型变换和视图变换就是欧氏变换，视口变换是相似变换，还好，但是投影变换(透视+正交)是仿射变换啊。

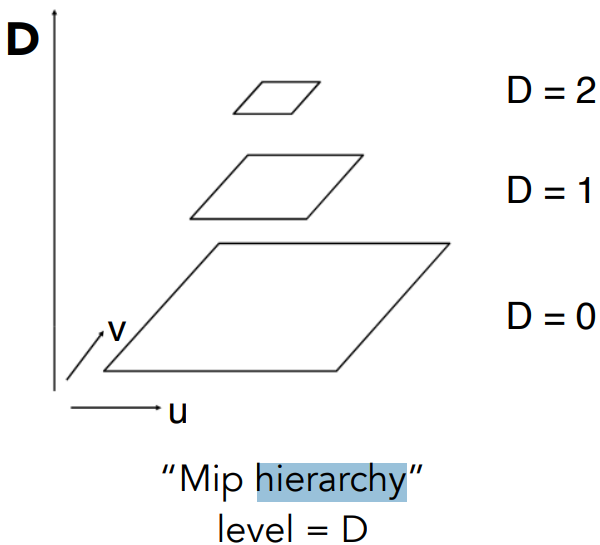
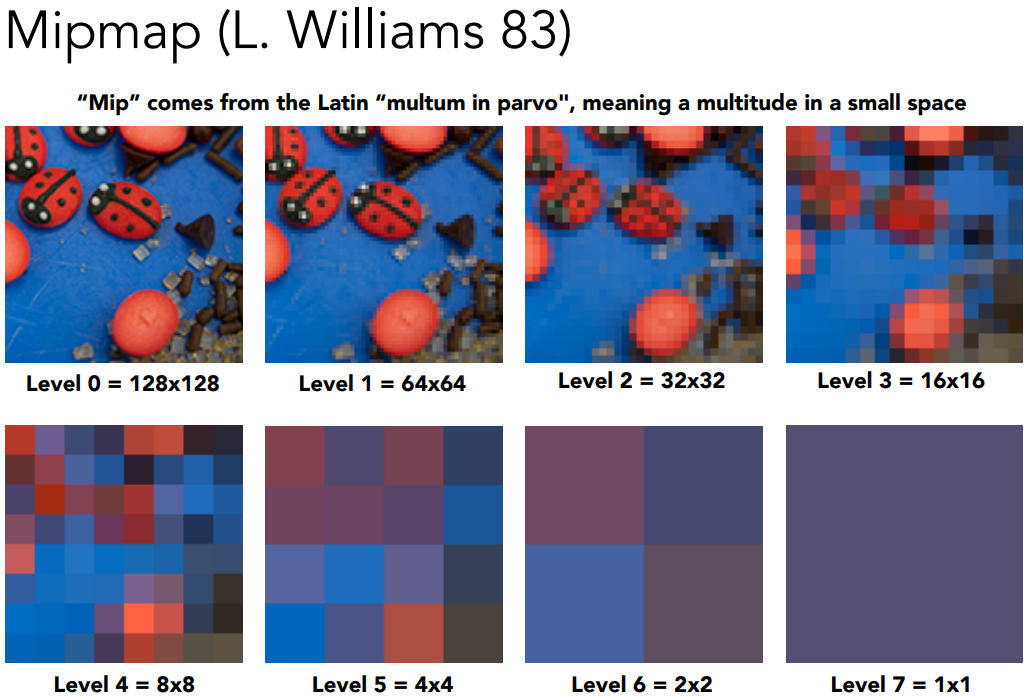
所以，假如我们知道一个由三角形面片构成的3D物体每个面片顶点上的属性(例如depth、color)，然后我们将这个3D物体透视投影到2D平面上后，我们想得到2D平面上某个三角形内部一个点的属性，我们不能在2D平面上计算得到(alpha、beta、gamma)，然后插值得到该点的属性，这是错误的，失真的。我们应该将这个点逆变换到3D空间中该物体对应三角形面片上的对应点上，然后在3D空间中计算得到该点的三角形重心坐标(alpha,beta,gamma)，然后再用这个(alpha,beta,gamma)来对属性做插值。

所以，假如我们目前有了一个物体的3D模型，但是它表面的属性是被记录在纹理中的，然后我们将该3D物体投影到了2D平面上。因为纹理与3D model表面的映射是面片顶点到面片顶点的映射，面片内部的点的映射我们不知道，所以若我们想知道2D平面上某个像素的点的属性，我们就需要做：将该2D坐标逆变换投影到3D模型上，计算该点在3D面片上的三角形重心坐标(alpha,beta,gamma)，然后根据3D model与纹理的映射找到纹理上对应的面片，然后对该面片顶点的坐标进行(alpha,beta,gamma)进行加权求和，得到目标点在纹理上的准确(u,v)坐标——于是就能直接在纹理上查询到该点的属性了。

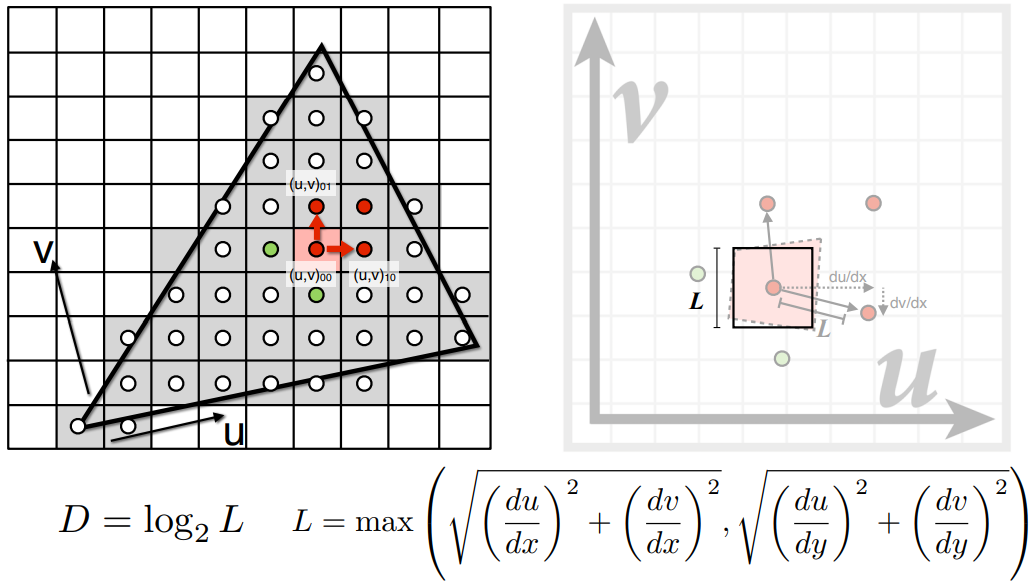
点查询与范围查询：

我们要知道屏幕空间上一个像素的属性，就需要从纹理空间中查询，假如纹理比屏幕空间小，那么像素映射到纹理上，就要通过插值查询映射点的属性，这叫做“点查询”。假如纹理比屏幕空间大，那么像素映射到纹理上，就要查询映射点周围一片点的属性的综合(通常是均值，也可能是max/min)，这叫做“范围查询”。

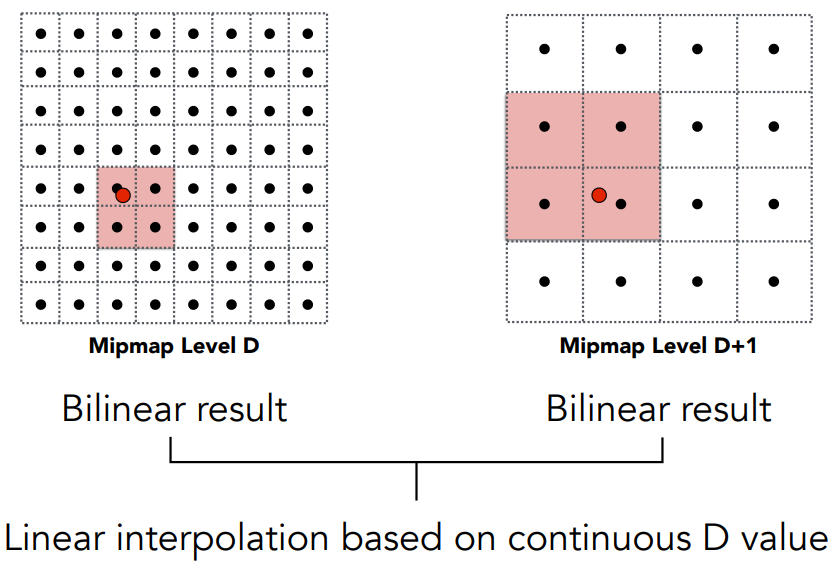
在图像学中，解决范围查询有一个fast、approx、square(仅仅适用于方形的范围查询)的算法，叫做“Mipmap”，这在CV中被称作image pyramid。见下图



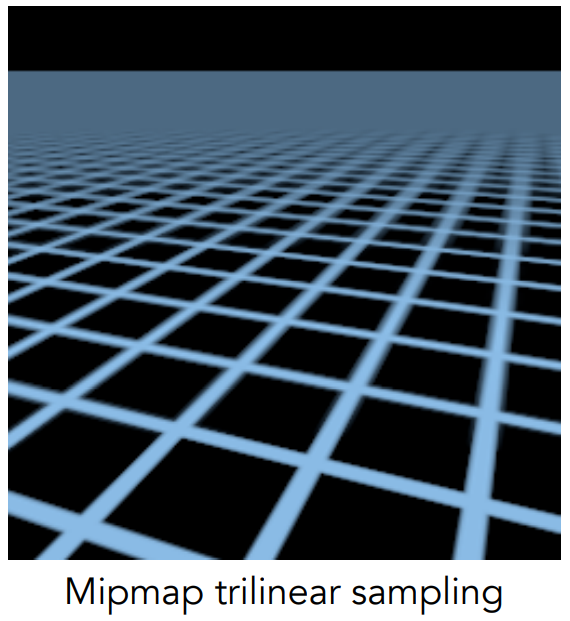
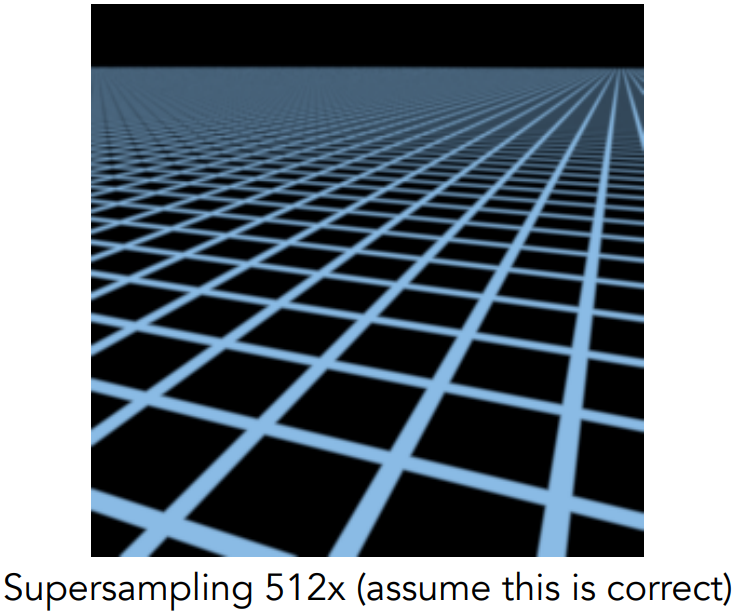
那么，知道了屏幕空间中一个像素，如何得知在纹理空间中该查询多大的范围呢？见下图：



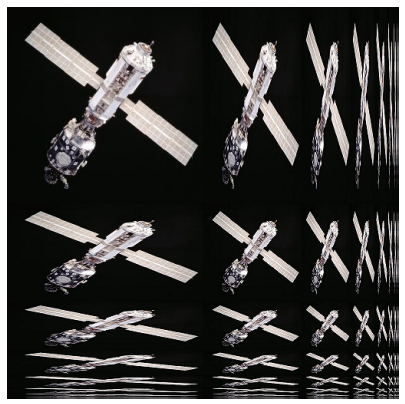
在Mipmap算法中，我们计算像素点在纹理上该查询多大范围，实际上就是计算该在mip hierarchy中第几层查询。但mip hierarchy只有整数层，要是我们计算出的D是1.5该怎么办？我们就要使用三线性插值(trilinear interpolation)：



Mipmap算法有一个很大的优点是它基本不消耗算力，而且它只消耗比原来的纹理多1/3的存储空间，但是它有个很大的缺点就是它查询的范围只能是一个方形，对于横向或纵向的长条形它效果不好，更别说斜着的长条形或更加奇奇怪怪的形状。这通常是由于Mipmap计算出来的查询范围比本来的查询范围大所引起的。效果见下图：



为了适应存在很多长条形查询范围的情况，我们在mipmap的基础上引入了“各向异性过滤(anisotropic filter)”，得到Ripmap，它能够适应横向或纵向的长条形查询范围。为什么叫各向异性？我们一般的mipmap算法查询的是方形，它假设长宽相同即是查询范围各个方向的性质是相同的，所以一般的mipmap算法是“各向同性”的。ripmap相较于mipmap有一个缺点就是它会消耗相比于原始纹理3倍的存储空间。至于拿到屏幕空间中一个像素，该如何得知查询ripmap中哪一层map，这和mipmap差不多，不细讲。anisotropic filter示意图如下，你看能到，对角线上就是Mipmap：



但ripmap只解决了长条形查询范围的问题，针对斜着的条形或更奇怪的形状，有EWA filter(Eliptical Weighted Average Filter)，跟ripmap一样，这也是一种各向异性过滤的算法。

隐式表达和显式表达：

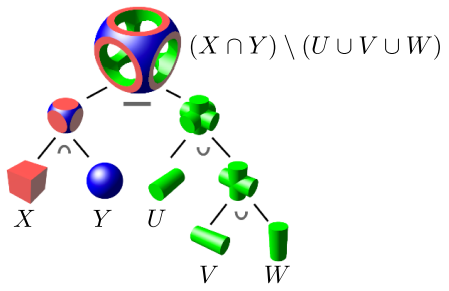
对几何物体的表示形式分为隐式表达和现式表达。

隐式表达例如x^3+y^3+z^3=1，表达了一个球体，你或许没办法很快想象出隐式表达的物体的形状，但是你能够很快计算出一个点是否在该物体的表面上。

显式表达例如你直接用CAD建模出一个物体的3D模型。也有特殊情况：你定义两个变量uv，遍历uv的定义域，就能通过f(u,v)=((2+cosu)\*cosv, (2+cosu)\*sinv, sinu)计算出对应3D模型每个点的3D坐标。这种公式形式的显式表达和隐式表达有什么区别呢？隐式表达的输入是3D坐标，显式表达的输出是3D坐标。

构造性实体几何：

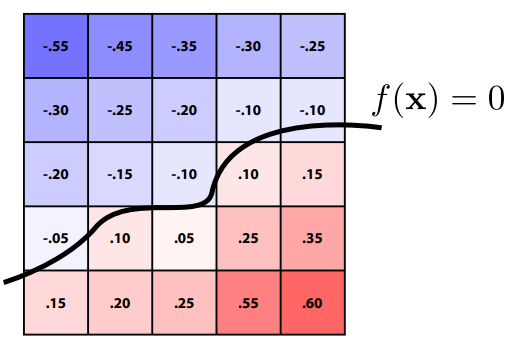
Constructive Solid Geometry.



DF:

Distance Function，距离函数，也是种对几何物体的隐式表达。

在离散的屏幕空间中，我们可以“水平集法Level Set Method”来表示SDF：



当然，Level Set的应用不仅限于表示SDF，Level Set是一种表示和跟踪接口（如流体的表面）的数值技术，其中接口被表示为一个函数的特定等值线或等值面。Level Set还可以表示地理上的等高线，可以给定CT扫描出来的3D空间中各点的密度，然后让你求出人体表面在哪里。