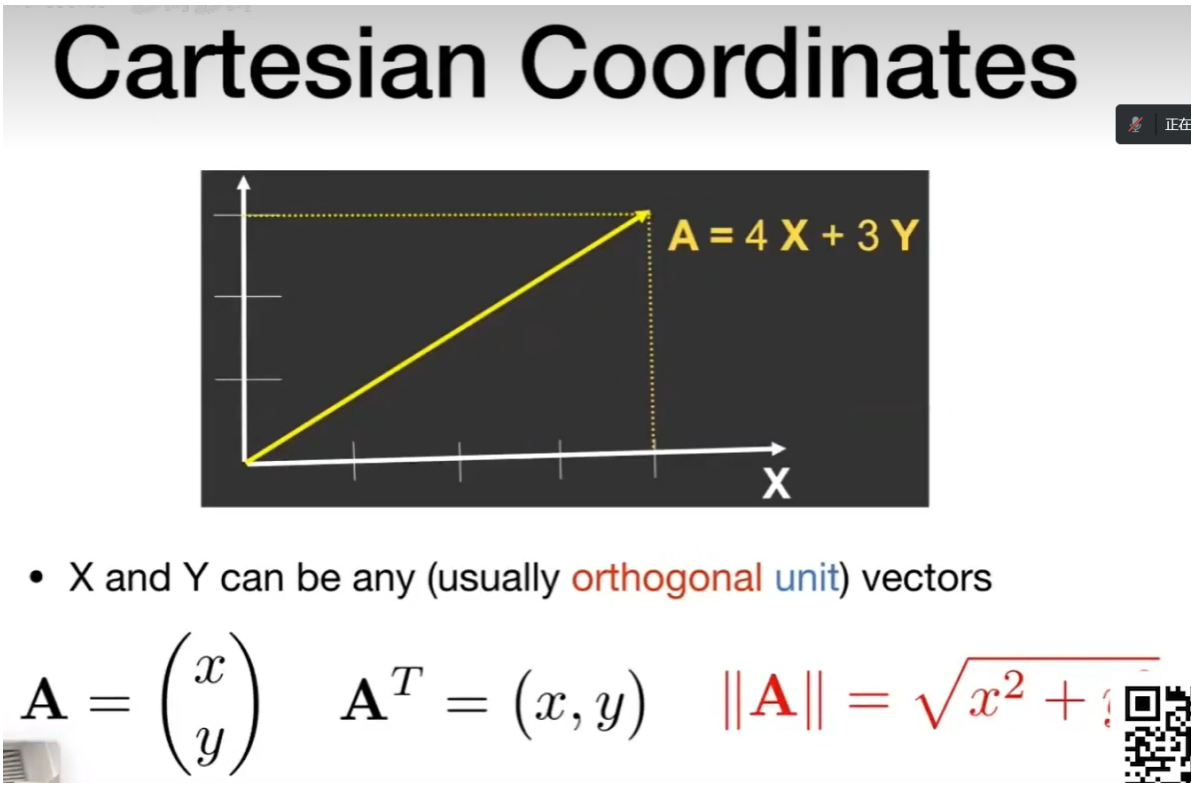
games101

第一节 overView

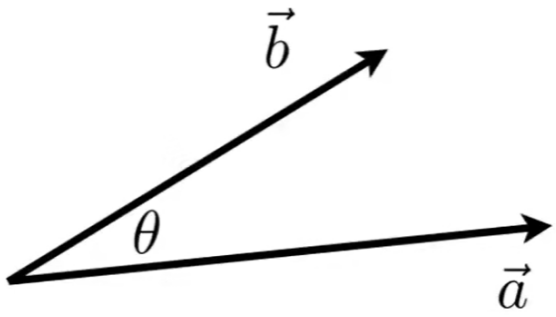
第二节 线性代数

1、vector 向量



Dot product

Dot (scalar) Product



$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \|\vec{b}\| \cos\theta$$

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|}$$

- For unit vectors

$$\cos\theta = \hat{a} \cdot \hat{b}$$

点乘可以用于求夹角余弦

- In 2D

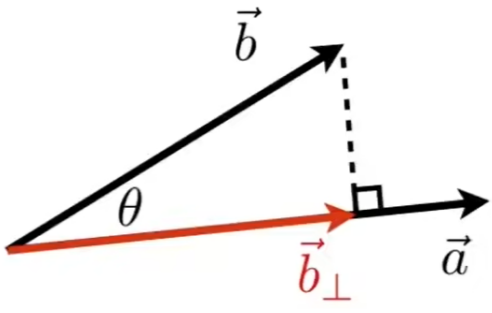
$$\vec{a} \cdot \vec{b} = \begin{pmatrix} x_a \\ y_a \end{pmatrix} \cdot \begin{pmatrix} x_b \\ y_b \end{pmatrix} = x_a x_b + y_a y_b$$

- In 3D

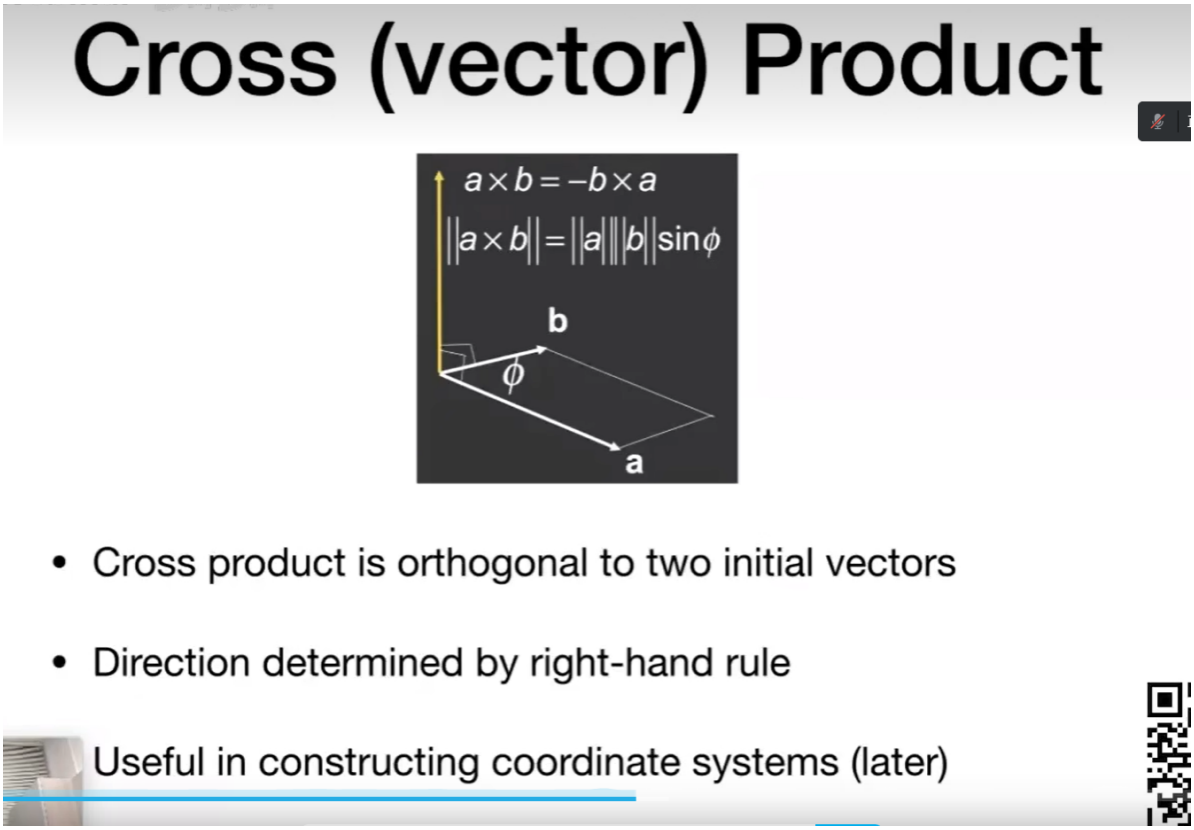
$$\vec{a} \cdot \vec{b} = \begin{pmatrix} x_a \\ y_a \\ z_a \end{pmatrix} \cdot \begin{pmatrix} x_b \\ y_b \\ z_b \end{pmatrix} = x_a x_b + y_a y_b + z_a z_b$$

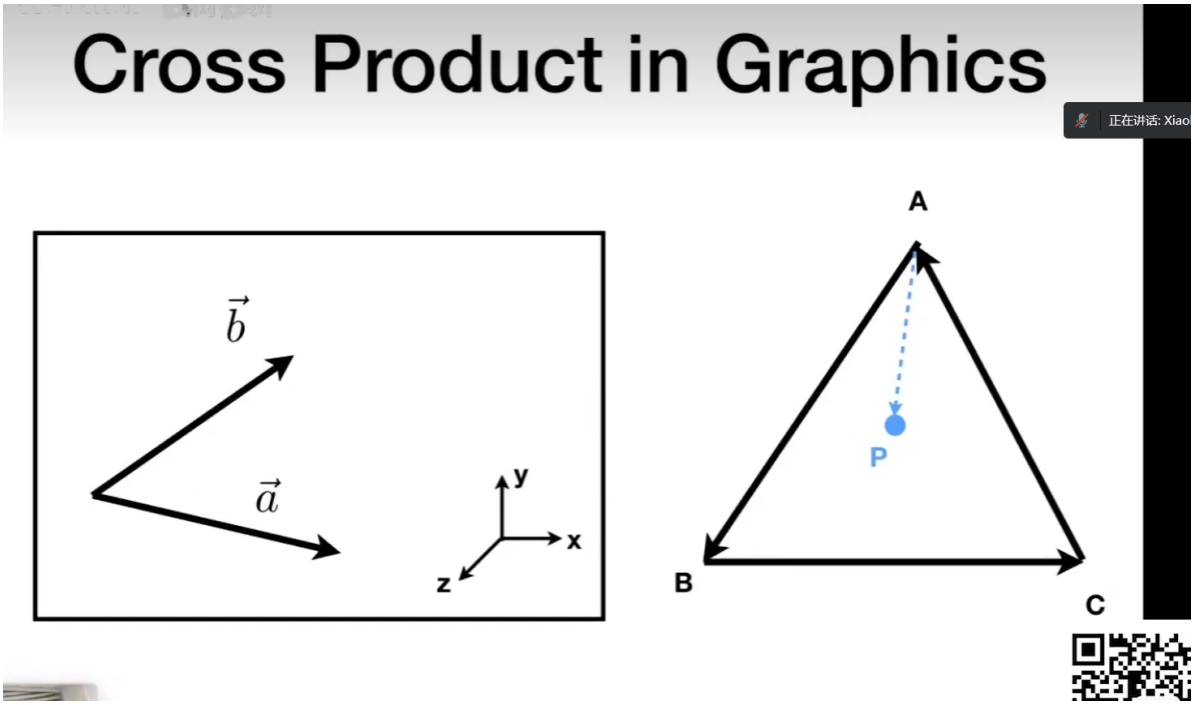
也可以用于求一个向量向另一个向量的投影

- $\vec{b}_\perp$: projection of $\vec{b}$ onto $\vec{a}$
  - $\vec{b}_\perp$ must be along $\vec{a}$ (or along $\hat{a}$)
    - $\vec{b}_\perp = k\hat{a}$
  - What's its magnitude k?
    - $k = \|\vec{b}_\perp\| = \|\vec{b}\| \cos\theta$



cross 叉乘





Matrices 矩阵

矩阵和数相乘

矩阵和矩阵相乘

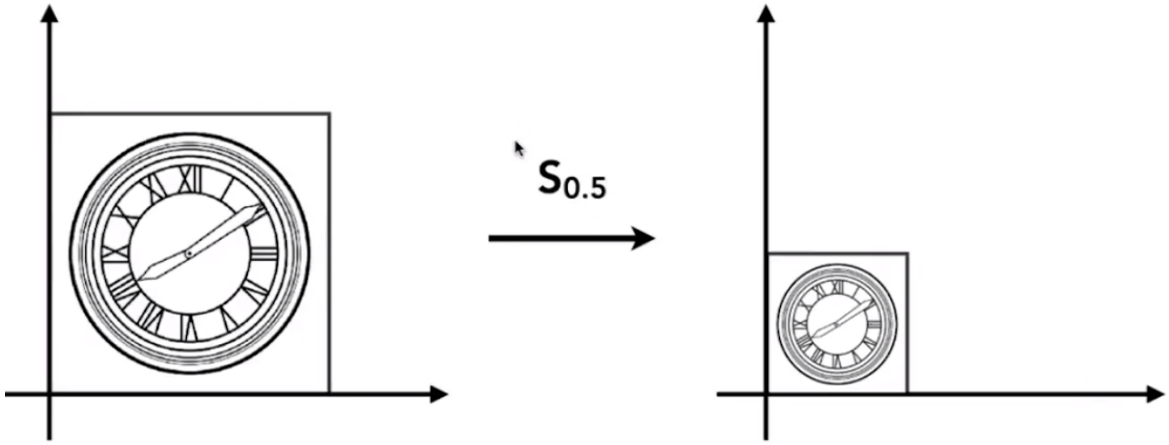
矩阵和向量相乘

- Official spoiler: 2D reflection about y-axis

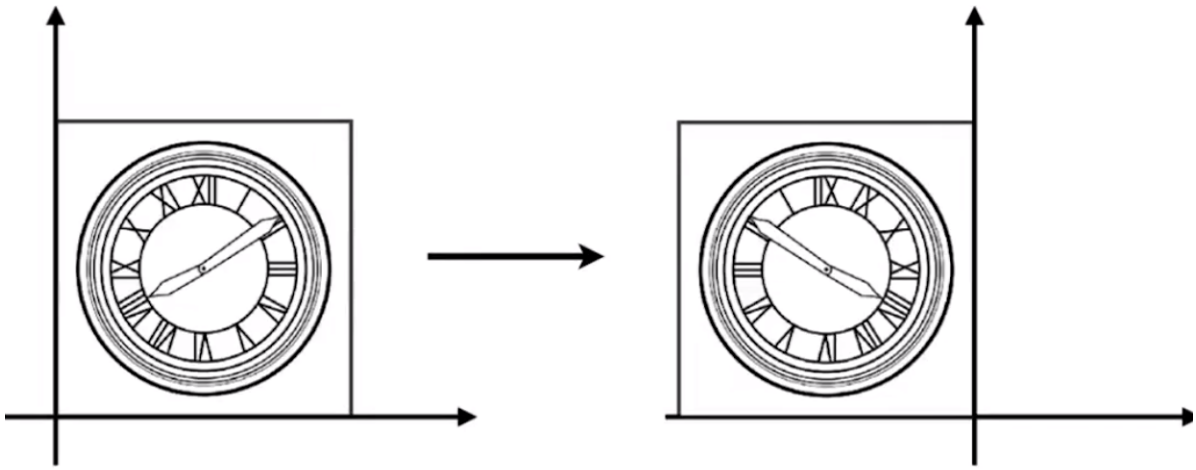
$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -x \\ y \end{pmatrix}$$

第三节 变换 transformation

scale 缩放



相对于x轴旋转



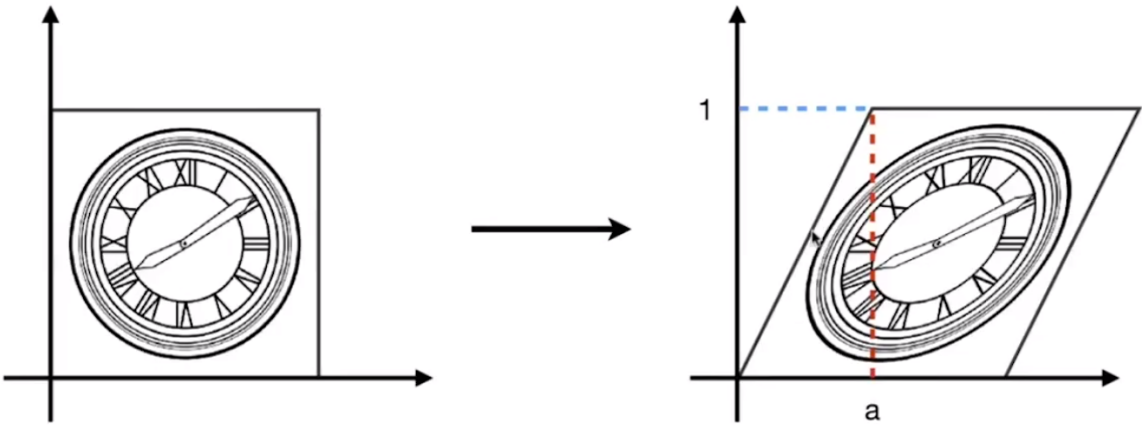
Horizontal reflection:

$$x' = -x$$
$$y' = y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

shear matrix切边

Shear Matrix



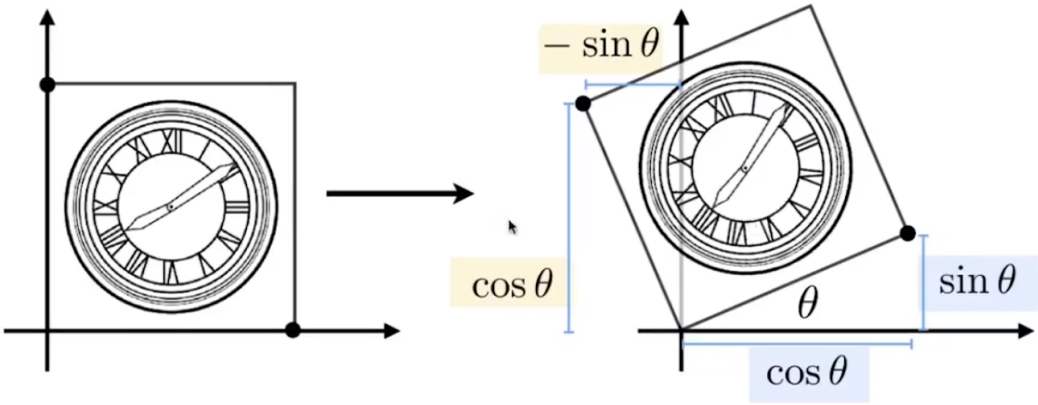
Hints:

Horizontal shift is 0 at y=0
Horizontal shift is a at y=1
Vertical shift is always 0

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

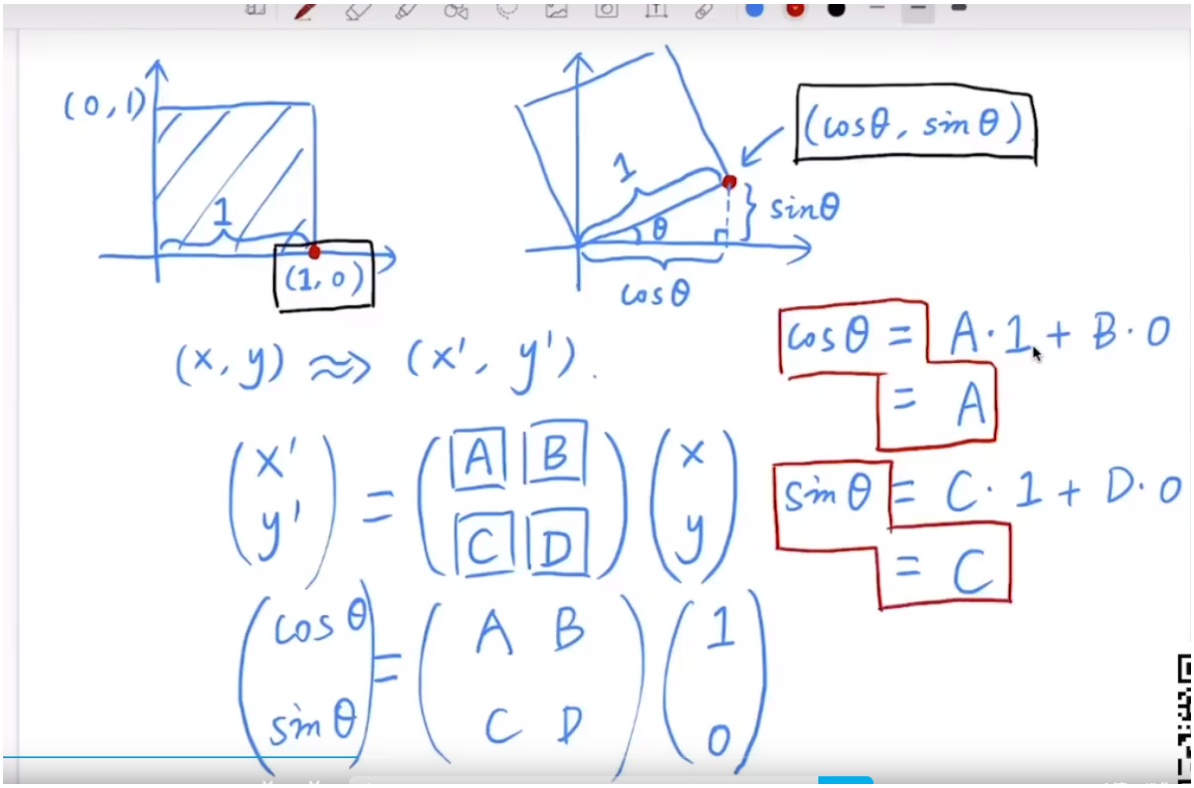
rotation旋转

Rotation Matrix



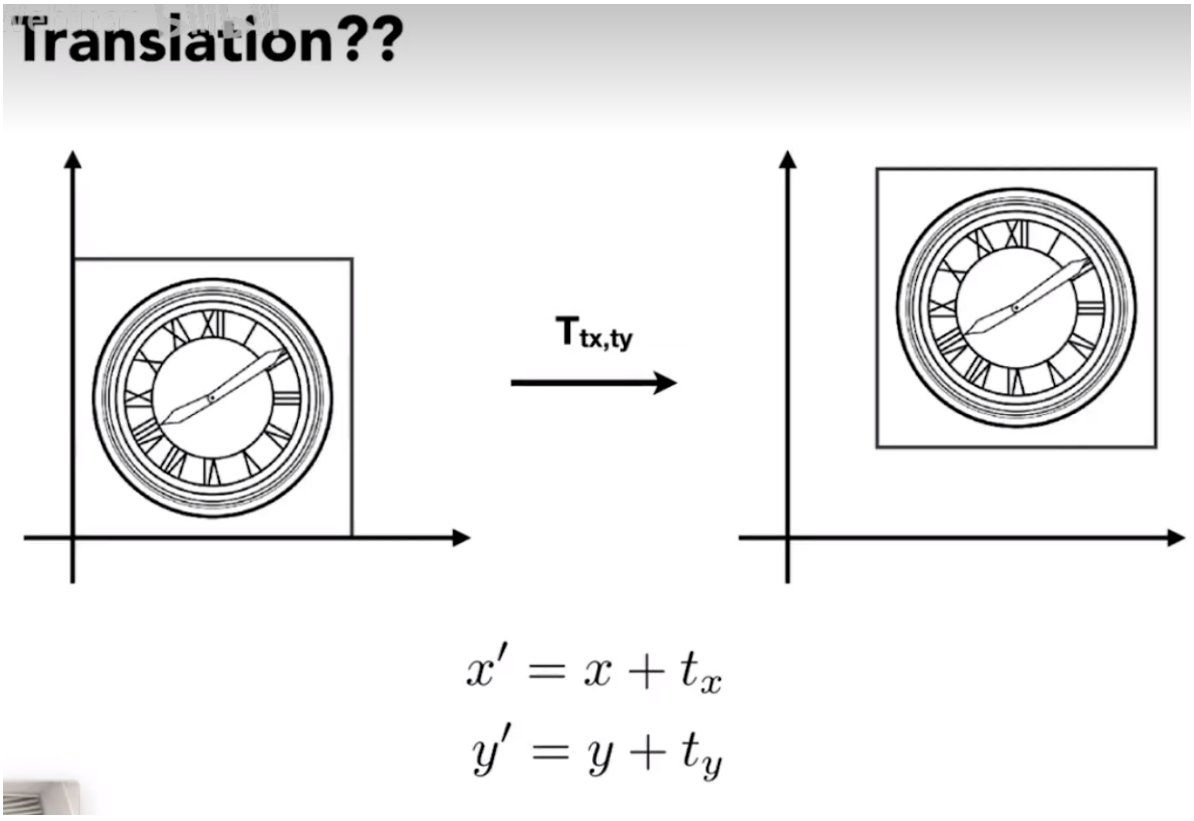
$$\mathbf{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

二维旋转的推到过程



齐次坐标

平移



$$x' = x + t_x$$
$$y' = y + t_y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

平移虽然简单，但不是线性变换，所以引入齐次坐标

引入其次坐标

Add a third coordinate (*w-coordinate*)

- 2D point = (x, y, 1)^T
- 2D vector = (x, y, 0)^T

Matrix representation of translations

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$$

Valid operation if w-coordinate of result is 1 or 0

- vector + vector = vector
- point – point = vector
- point + vector = point
- point + point = ??

变成两个点的中点，因为如下面的式子，w会变为2故除以2变为中点。

In homogeneous coordinates,

$$\begin{pmatrix} x \\ y \\ w \end{pmatrix} \text{ is the 2D point } \begin{pmatrix} x/w \\ y/w \\ 1 \end{pmatrix}, \quad w \neq 0$$

引入其次坐标是为了让一个点可以写成一个矩阵乘以一个点的式子，而不是后面在跟上个数

在仿射变换里，第三行基本就是0 0 1，

Scale

$$\mathbf{S}(s_x, s_y) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rotation

$$\mathbf{R}(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

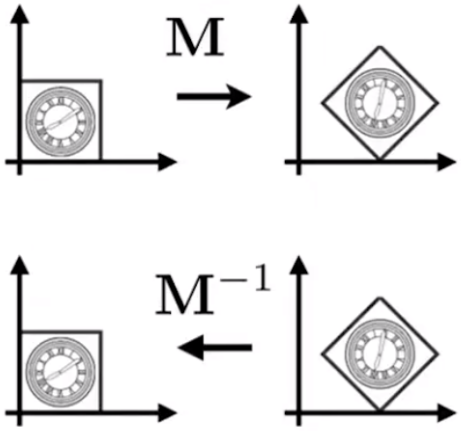
Translation

$$\mathbf{T}(t_x, t_y) = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

inverse t'ransform 逆变换

$$\mathbf{M}^{-1}$$

$\mathbf{M}^{-1}$ is the inverse of transform $\mathbf{M}$ in both a matrix and geometric sense



逆变换就是变回来的时候乘以逆矩阵

composite transform 组合变换

变化就是前面乘以一个矩阵，复杂的操作可以由一系列简单的操作合成

顺序不对，结果就可能不一样，原因是旋转这个操作默认的是绕原点（0，0）逆时针旋转x°。

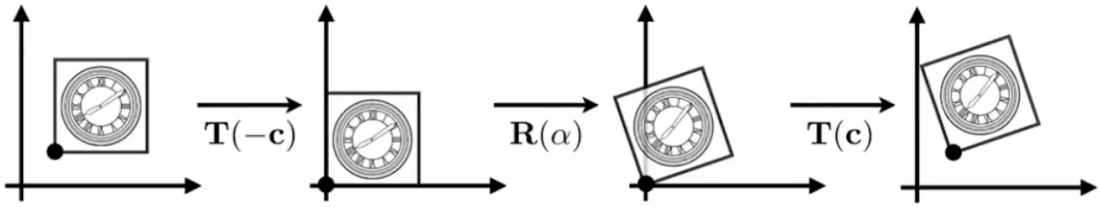
先旋转后平移写法

$$T_{(1,0)} \cdot R_{45} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 45° & -\sin 45° & 0 \\ \sin 45° & \cos 45° & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

对于不是围绕原点的旋转，可以先吧图形放到原点，旋转完后在一回来

How to rotate around a given point c?

1. Translate center to origin
2. Rotate
3. Translate back



Matrix representation?

$$\mathbf{T}(\mathbf{c}) \cdot \mathbf{R}(\alpha) \cdot \mathbf{T}(-\mathbf{c})$$

3D变换

- 3D point = (x, y, z, 1)T
- 3D vector = (x, y, z, 0)T

In general, (x, y, z, w) (w != 0) is the 3D point:

(x/w, y/w, z/w)

最后一行是0001（仿射变换）

Use 4×4 matrices for affine transformations

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

第四节 变换二 transformation Cont

旋转补充

逆向旋转： Rt ==R-1；

三维变换

View/camera transformation

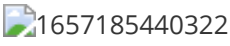
mvp　model - view - projection 相机-物体-投影

定义一个相机：

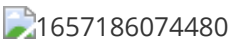
位置：position

朝向：look—at

向上：up direction



约定：相机位于原点，沿着-z看，向上方向为y



调整相机的步骤：

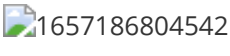
先将相机调整到（0，0，0）点，

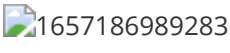
在将g到-z，t到Y。这样不好求，可以想假设从轴到这些位置的矩阵求出来，然后由于旋转矩阵其逆矩阵就是其转置，于是把求出的矩阵转置就成了要求的矩阵，在乘就行了。

相机做完这些运动后，其他物体也跟着做这些操作

projection transformation

正交投影





在正交投影里里面，把一个物体移到中心然后缩放成1 1 1 的大小，见上图步骤

透视投影 perspective projection

 1657197986453

在其次坐标里面，（1001）(2002)是同一个点

 1657198215190

可以采用先把远的的面的点往里挤，然后在做正交投影，

规定好，在挤的时候，z轴值不变，中心点的值不变，

 1657199322055

 1657199397219

 1657199500514

 1657785031679

x y 的值经过相似三角行可以推出，z的值有公式推出。

 1657199527953

最终的透视由挤压公式+正交投影得出

这里有一个问题：对于远f和近n之间的这些点，在挤压的时候他们的z数值会如何变化，首先这里已经说了不会不变，那应该就是会变远。

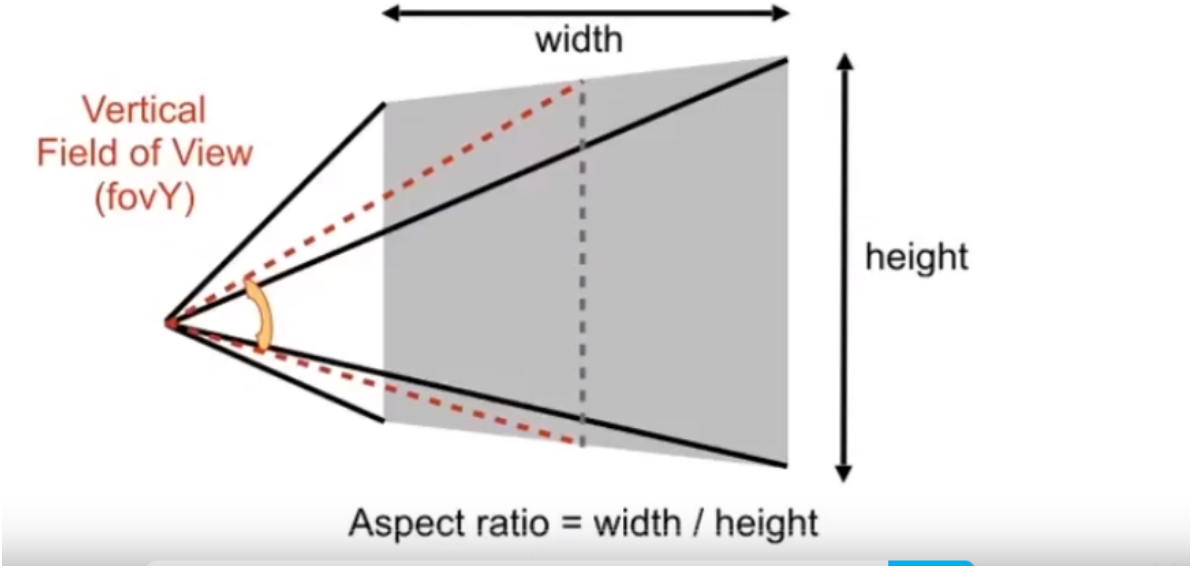
补充：在透视投影中，把斜的立方体转化为长方体

$$M = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

CSDN @beidou011

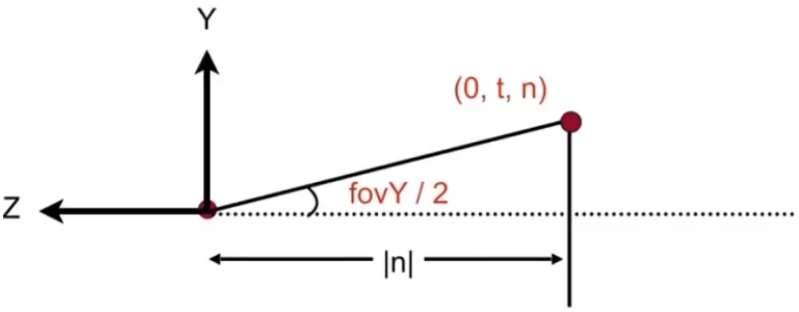
第五节、rasterization 光栅化

两个概念



长宽比：aspect ratio

垂直可视角度（类似我们生活中的广角和远角）：两红线夹角



$$\tan\frac{fovY}{2} = \frac{t}{|n|}$$

$$aspect = \frac{r}{t}$$

角度，宽高比

投影好之后，考虑要把物体画在哪？

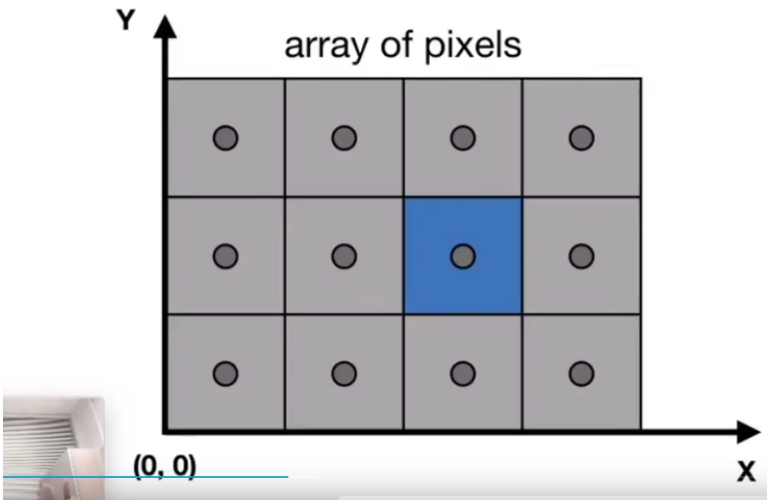
光栅化

- Raster == screen in German
  - Rasterize == drawing onto the screen

光栅化：把东西画在屏幕上的过程

Canonical Cube to Screen

- Defining the screen space
  - Slightly different from the “tiger book”



Pixels’ indices are in the form of (x, y), where both x and y are integers

Pixels’ indices are from (0, 0) to (width - 1, height - 1)

Pixel (x, y) is centered at (x + 0.5, y + 0.5)

The screen covers range (0, 0) to (width, height)

像素的定义

- Irrelevant to z
- Transform in xy plane: $[-1, 1]^2$ to [0, width] x [0, height]
- Viewport transform matrix:

$$M_{viewport} = \begin{pmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} \\ 0 & \frac{height}{2} & 0 & \frac{height}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

视口变换：暂时不考虑z轴，把-1到1变换到屏幕空间

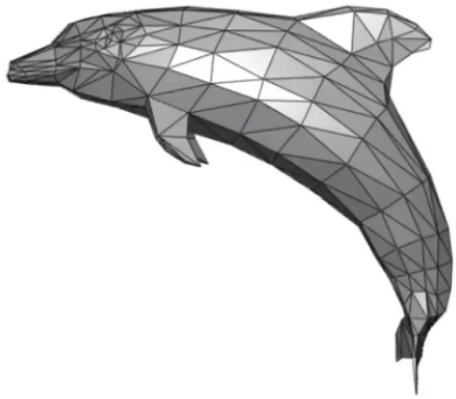
然后，打碎到一个一个的像素上，即光栅化

triangles

Triangles - Fundamental Shape Primitives

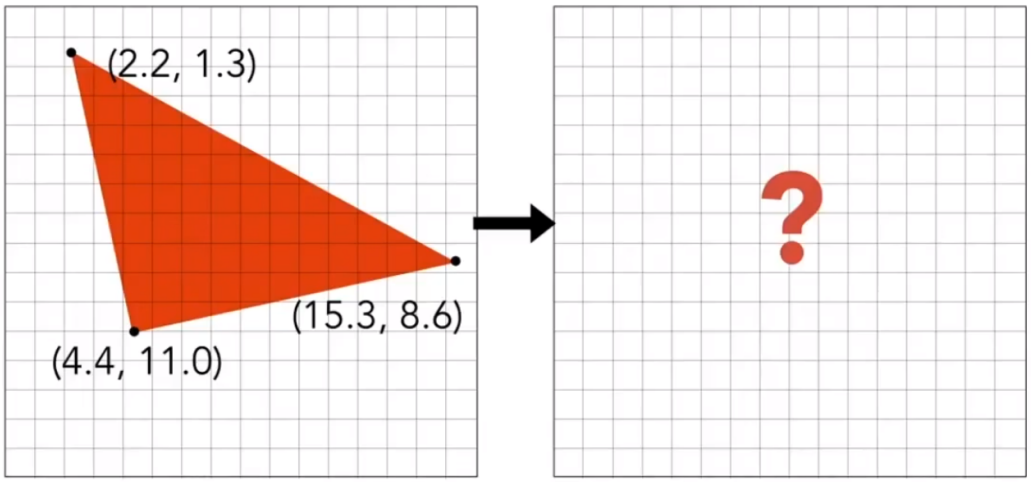
Why triangles?

- Most basic polygon
  - Break up other polygons
- Unique properties
  - Guaranteed to be planar
  - Well-defined interior



三角形是最基础的图形，是一个平面的

What Pixel Values Approximate a Triangle?



Input: position of triangle vertices projected on screen

Output: set of pixel values approximating triangle

向像素的转变：我们规定一个像素是不会变的，所以对于上图来说，只有白格和红格，不会有渐变行，故即需要算法

Sampling a Function

Evaluating a function at a point is sampling.

We can discretize a function by sampling.

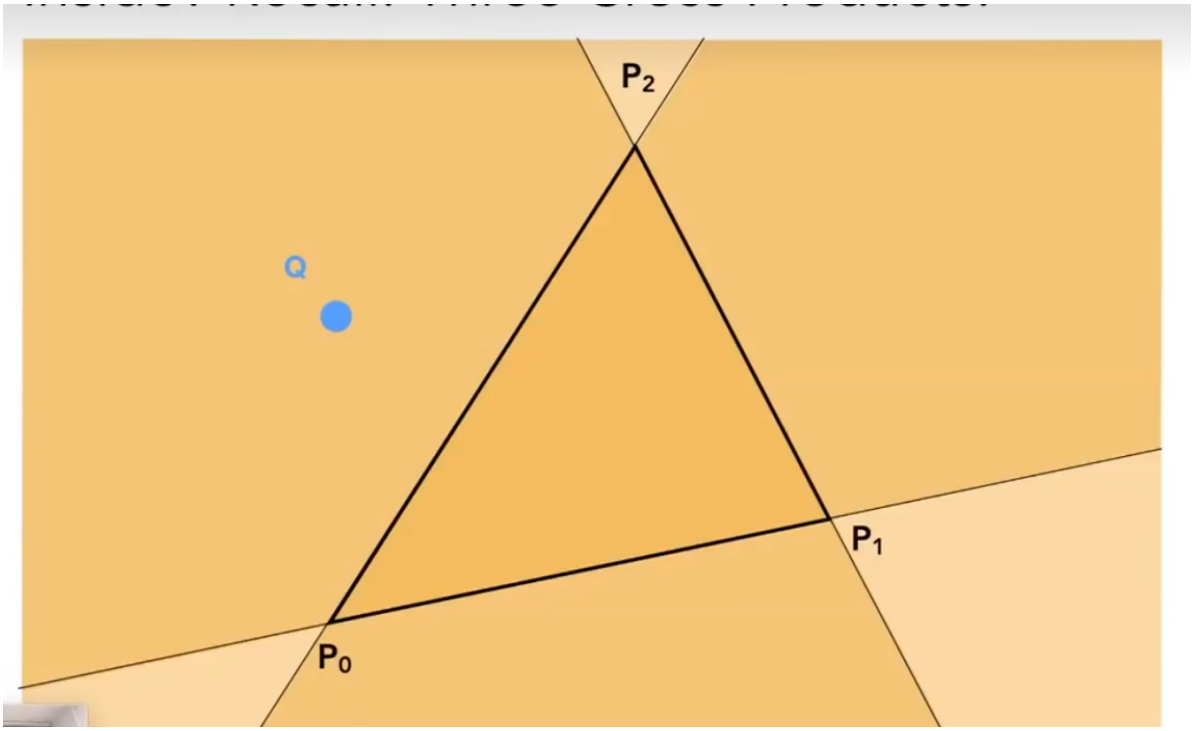
```
for (int x = 0; x < xmax; ++x)
    output[x] = f(x);
```

Sampling is a core idea in graphics.
We sample time (1D), area (2D), direction (2D), volume (3D) …

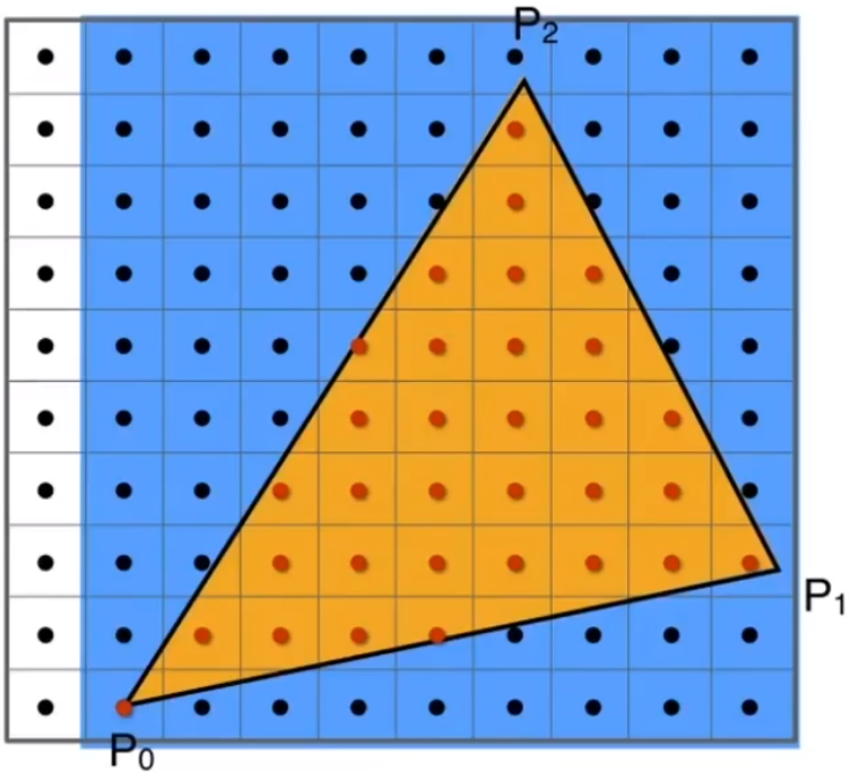
一种方法是采样

 1657288152902

对每个像素（这个是每个像素的中心点），看看是否在三角形内

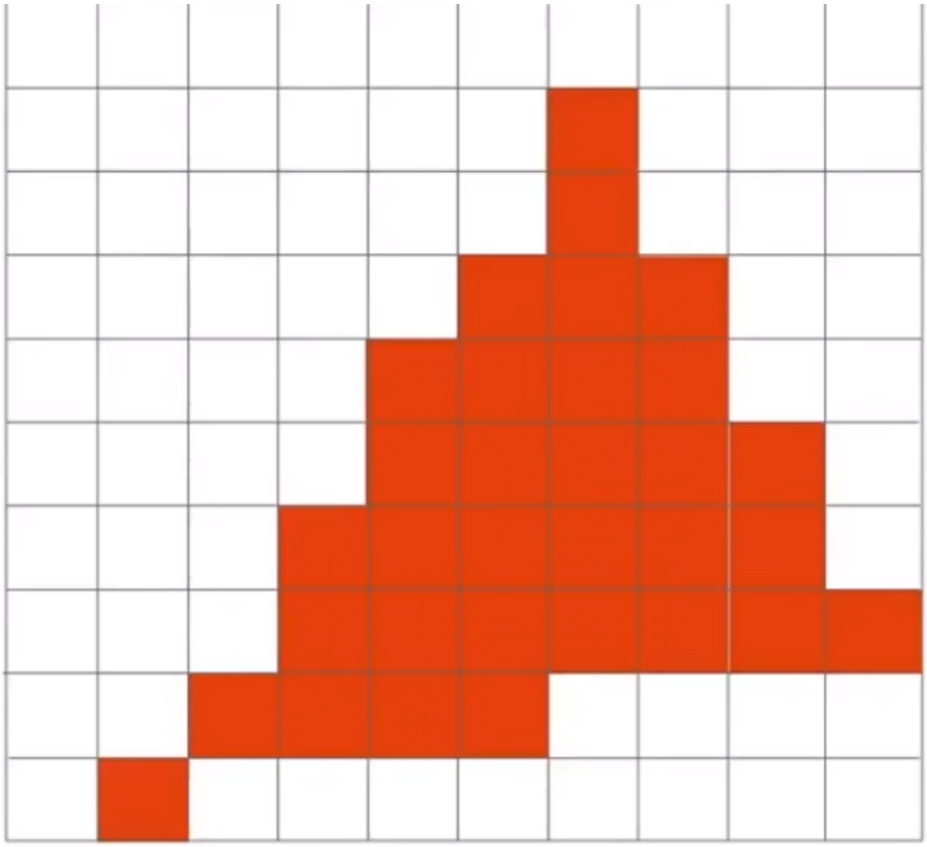


利用叉积来判断点是否在三角形内：规定一个顺序，如p0p1p2,那么只要p0p1,p1p2,p2p0分别和p0q,p1q,p2q做叉积，如果结果均为正或者负，则为在里面。



Use a Bounding Box!

优化：找出这个三角形的大致边界，在边界外面的不用进行光栅化（找出包围盒，BoundingBox）



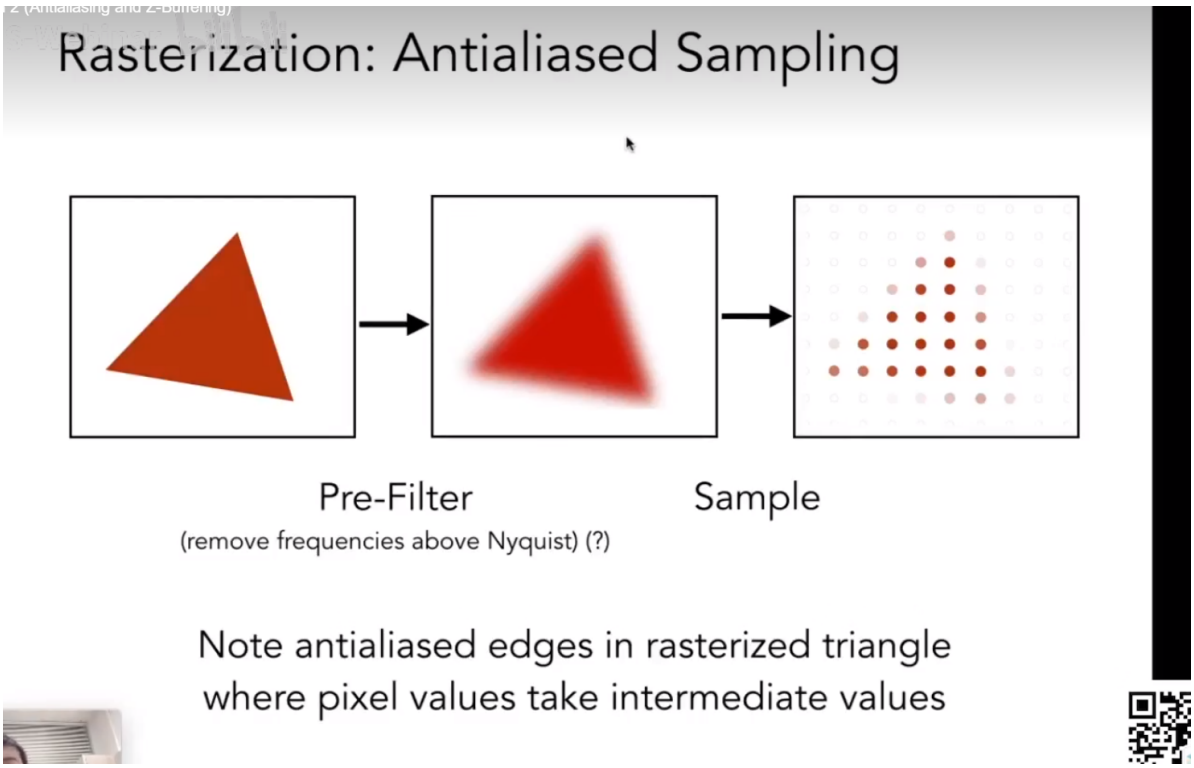
Jaggies!

一个问题：锯齿，jaggies：就是在三角形区域内光栅化然后在返回操作时，发现不是三角形了。也称走样

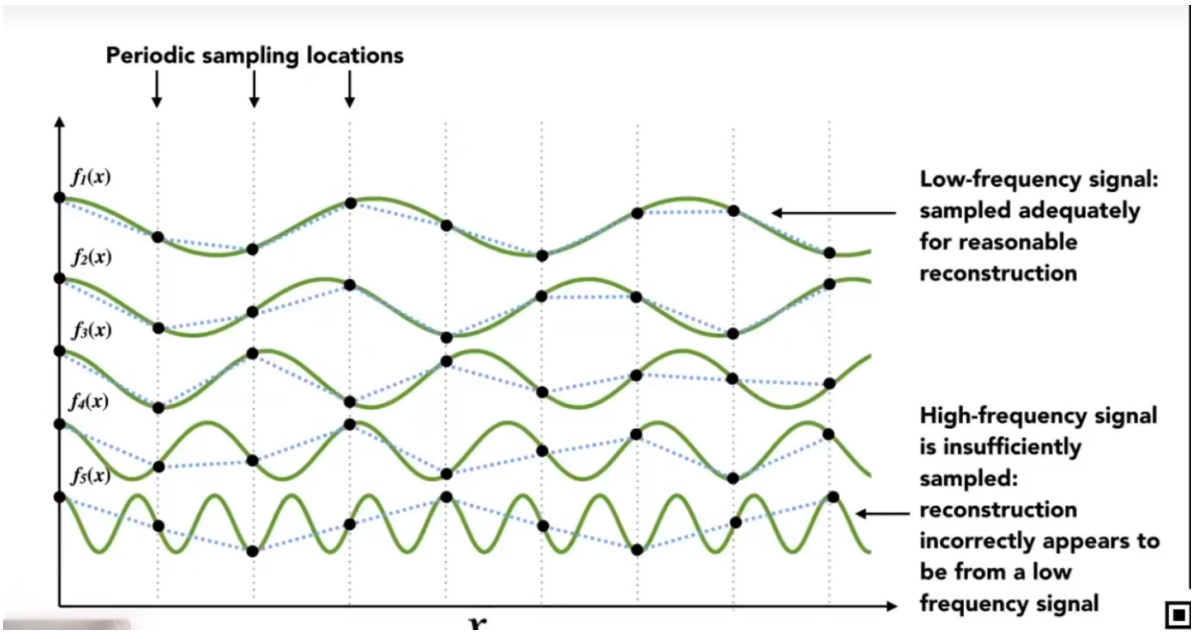
第六节、Rasterization2 （Antialasing，Z-Buffering）反走样和深度缓冲

分析走样

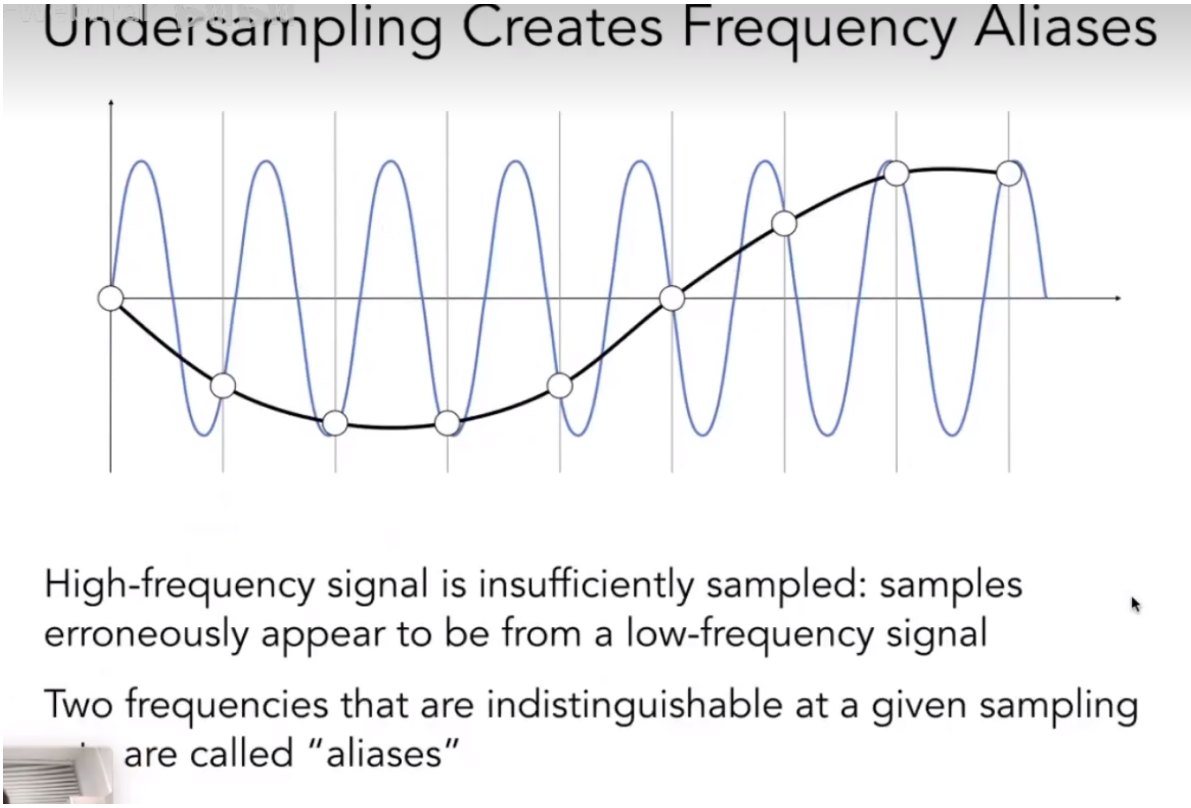
artifacts 图形学中的一切瑕疵都简称于此



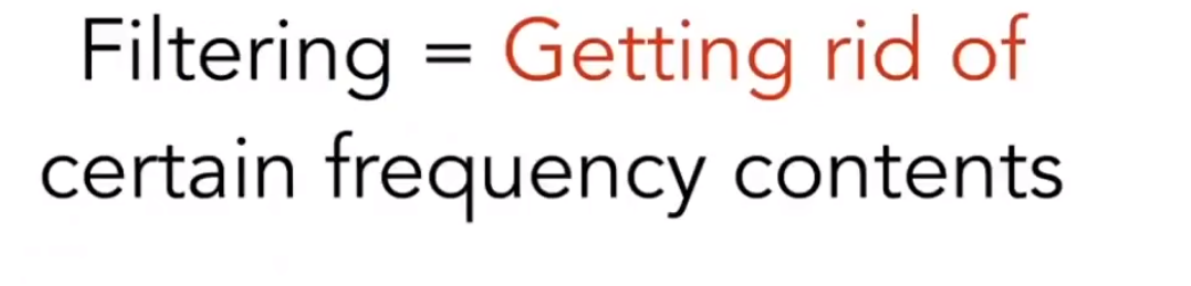
模糊（滤波）处理：可以抗锯齿，反走样。 先对图像模糊处理，再采样。



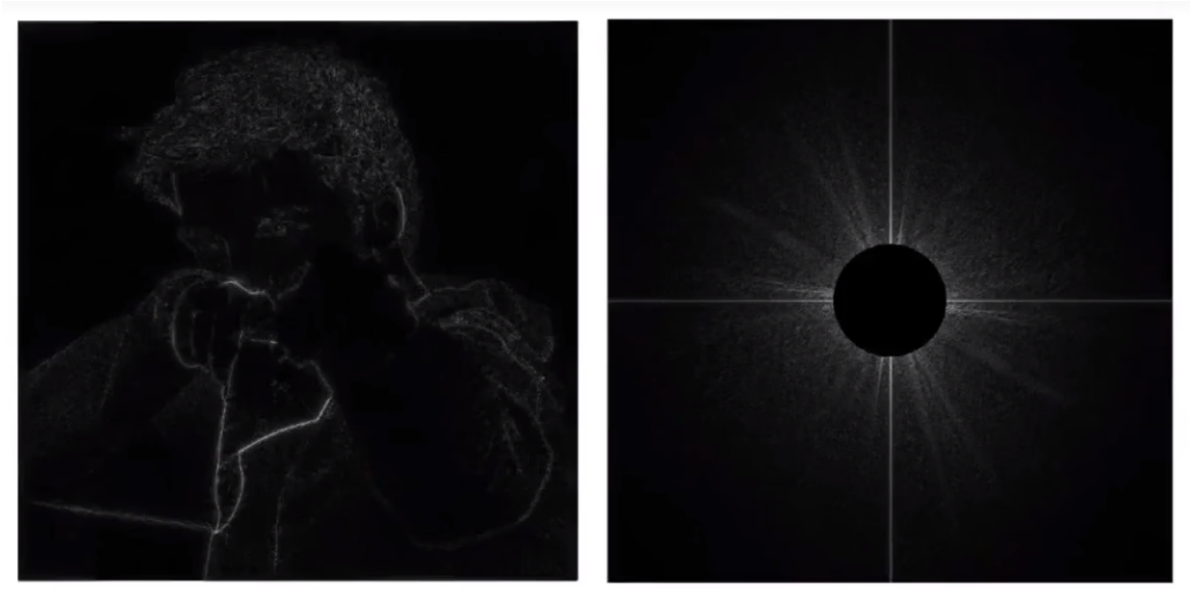
用上图函数频率来解释频率问题：上图中，点表示采样，而函数一道五的函数频率是逐渐变快，会使得采样出的结果很差。



上图说明，两个完全不同的图像，可能采样出一样的结果



Filtering滤波　去掉一系列频率



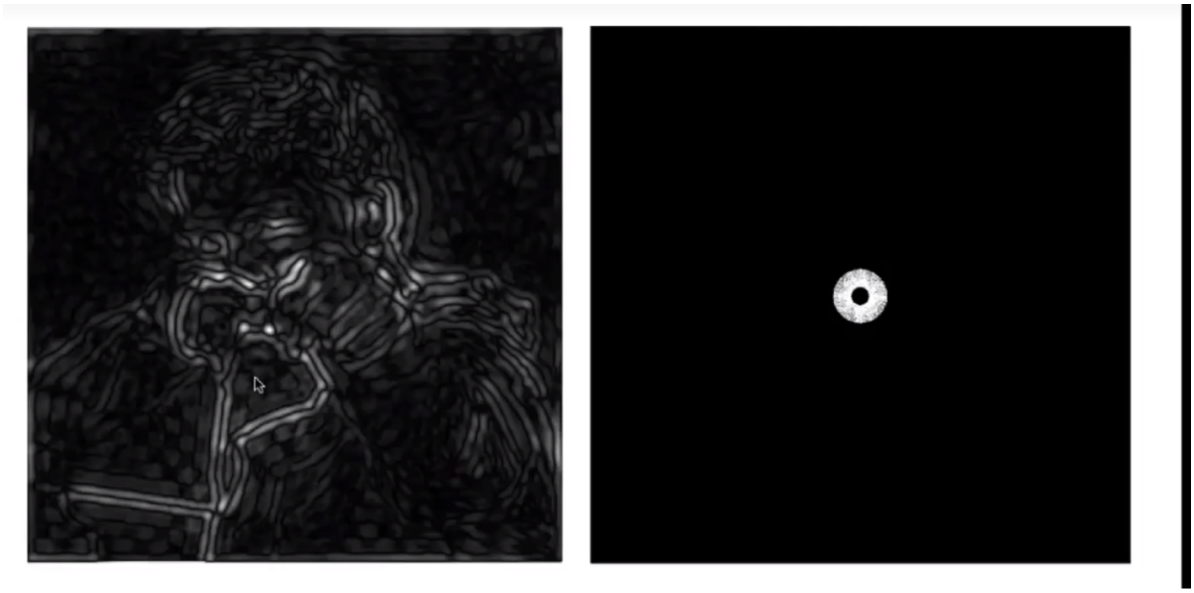
High-pass filter

高通滤波（把图中的低频低频信号给挡住，留下高频的）

一些边界是高频信息：即两边变化非常大。

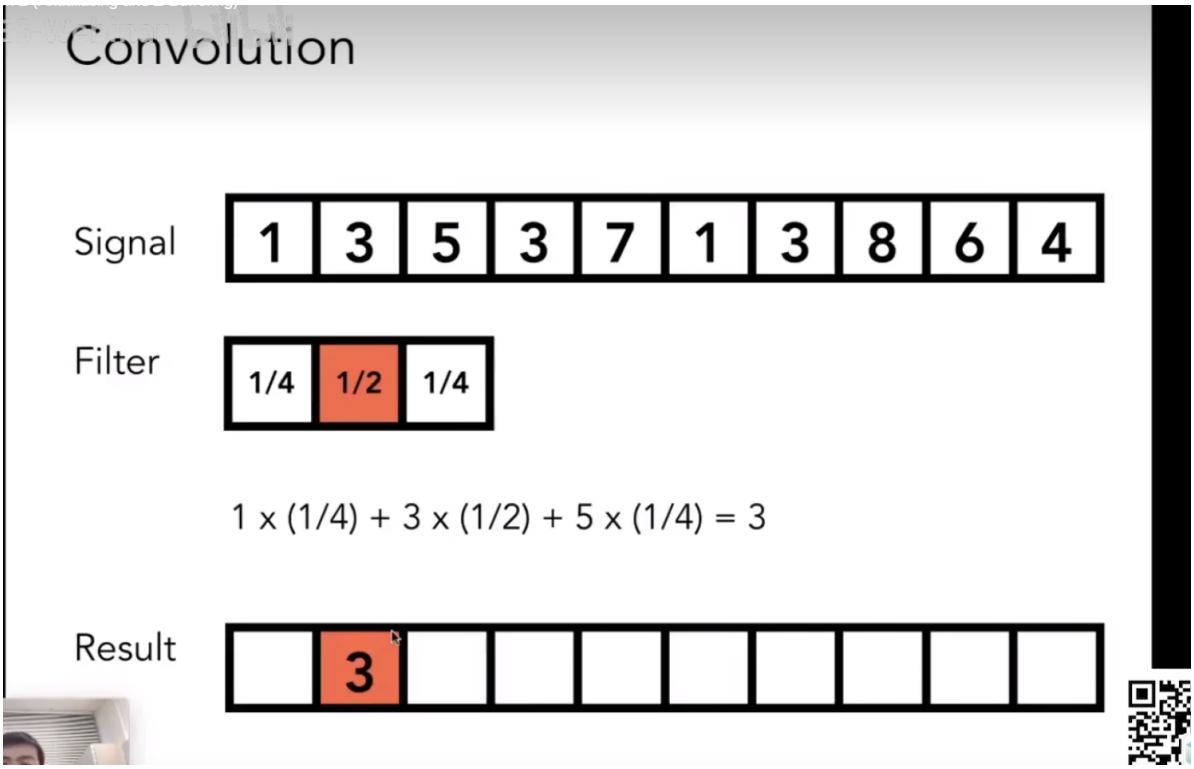
Low-pass filter

低通滤波：把高频信息抹掉



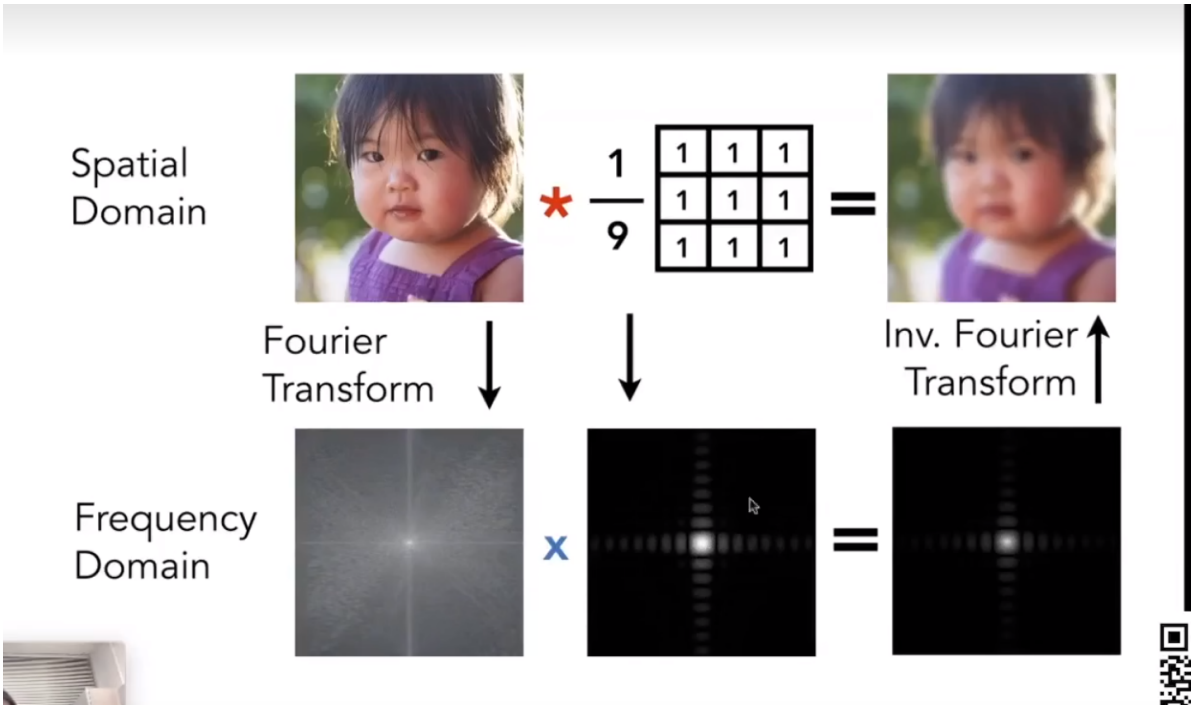
Filtering = Convolution
(= Averaging)

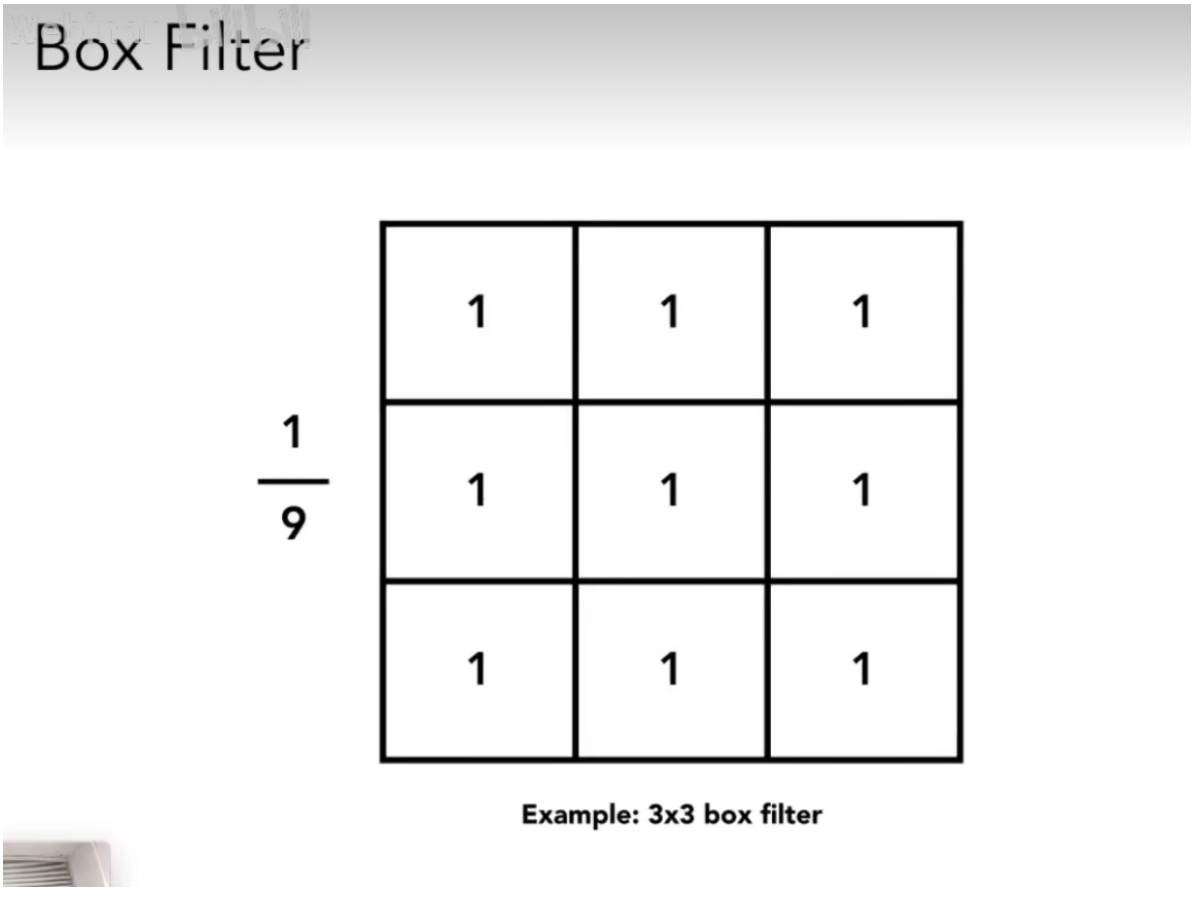
滤波又等于 卷积（convolution）



卷积操作

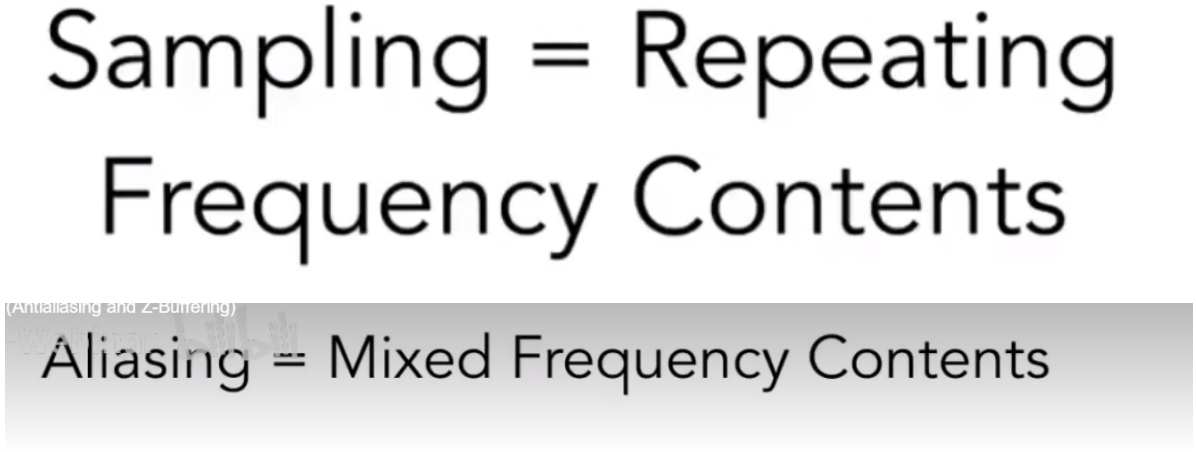
频率上卷积，时域上乘积

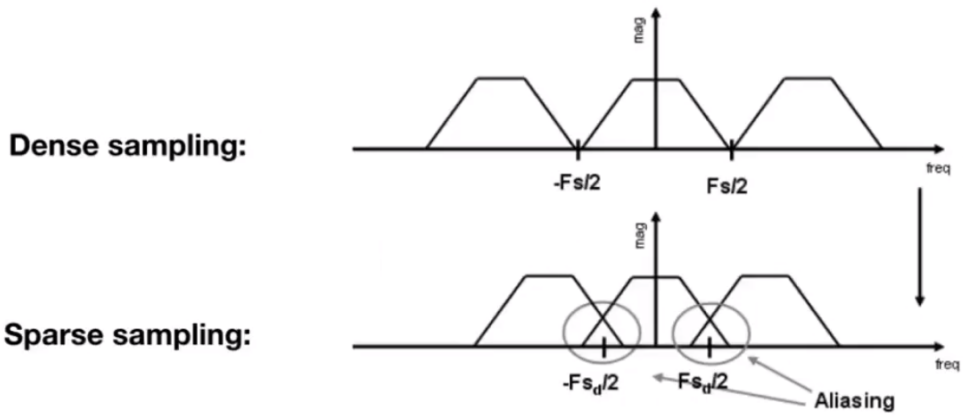




滤波器　box folter 低通滤波器

推想：box越大，图像越模糊

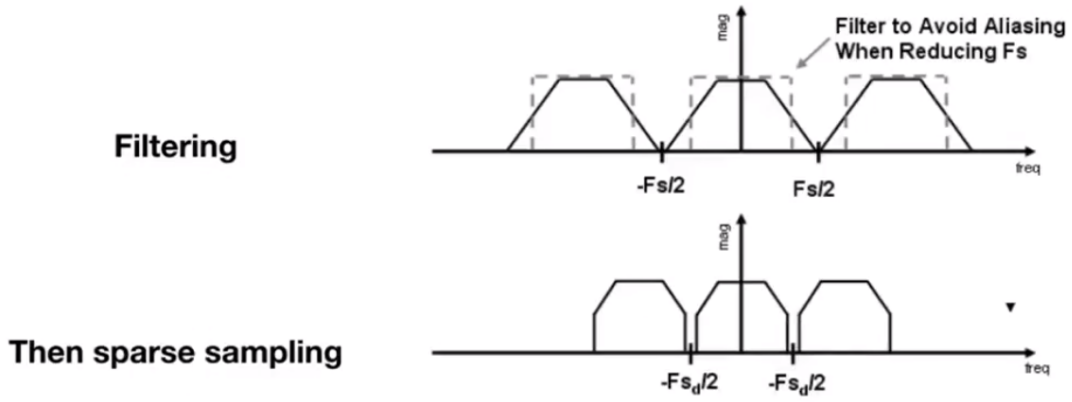




从频普分析走样。

反走样

先模糊，在采样，即先把高频拿掉



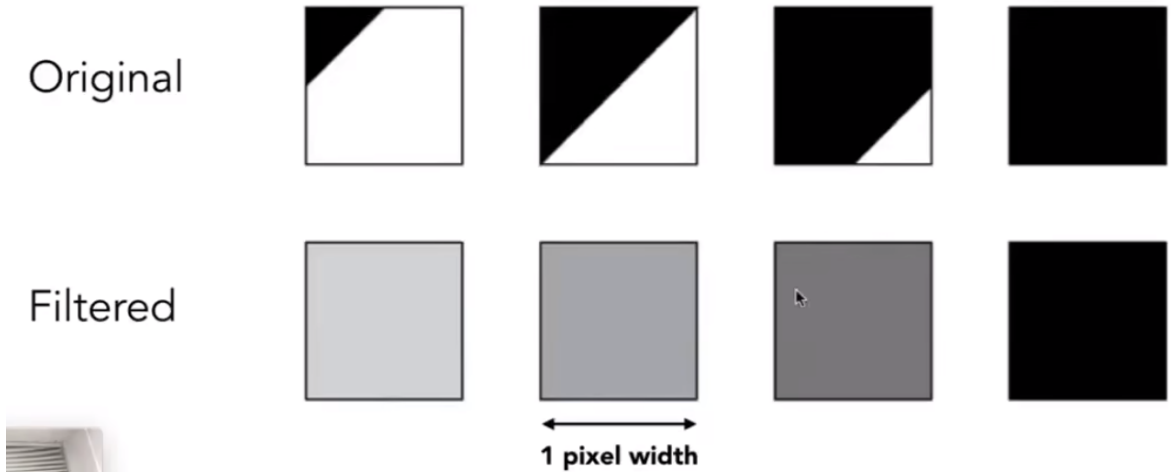
上图的在频谱上，把一些高频去掉，这样发生冲突就小了。

具体操作：用一定大小的滤波器，进行卷积。

Solution:

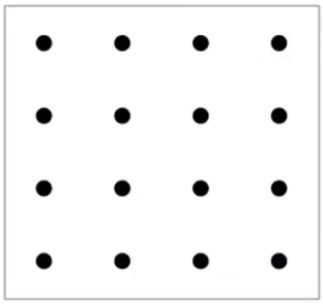
- **Convolve** f(x,y) by a 1-pixel box-blur
  - Recall: convolving = filtering = averaging
- **Then sample** at every pixel's center

In rasterizing one triangle, the average value inside a pixel area of f(x,y) = inside(triangle,x,y) is equal to the area of the pixel covered by the triangle.



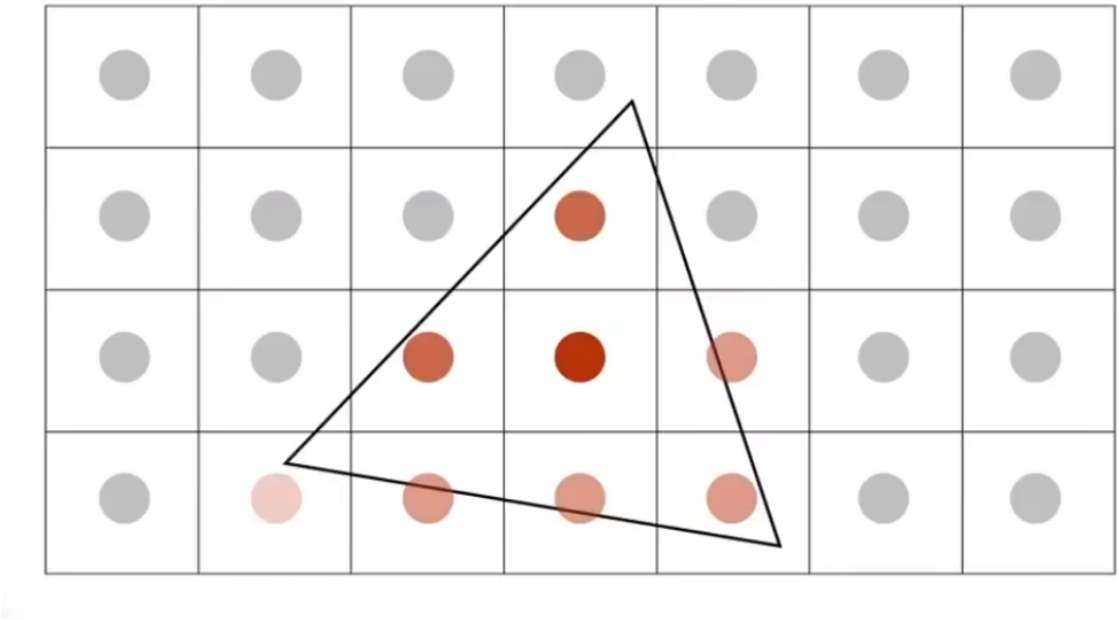
MSAA放法

Approximate the effect of the 1-pixel box filter by sampling multiple locations within a pixel and averaging their values:

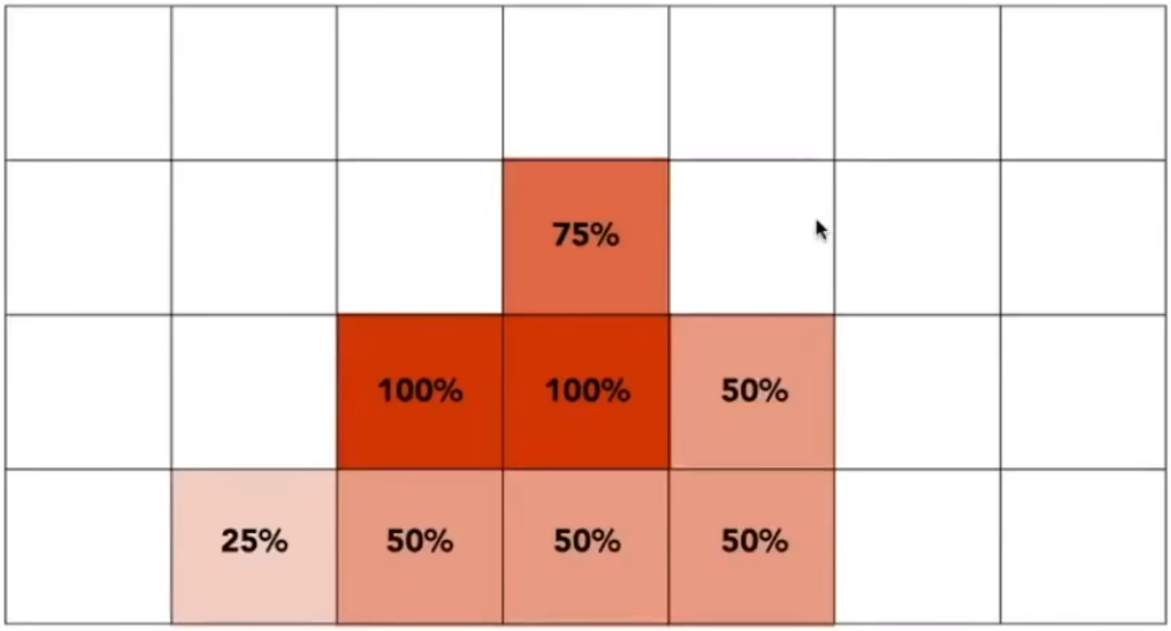


4x4 supersampling

把一个像素在分成更多小像素



求在每个像素里的覆盖率



说明：MSAA不是通过提高覆盖率，采样率来提高的。

采用MSAA方法确实起到抗锯齿的效果，但是增加了计算量（课中讲道的是虚拟出均匀的点来计算覆盖率，但是实际上会采用一些特殊图案来计算覆盖率。）

FXAA方法：先得到有锯齿的图案，然后在把边界找到，把锯齿换掉。

TAA：会沿用一定的上一帧的图。

第七章、iiiumination shading graphics pipeline

光照，着色，图形管线

前要

Today

- Visibility / occlusion
  - Z-buffering
- Shading
  - Illumination & Shading
  - Graphics Pipeline

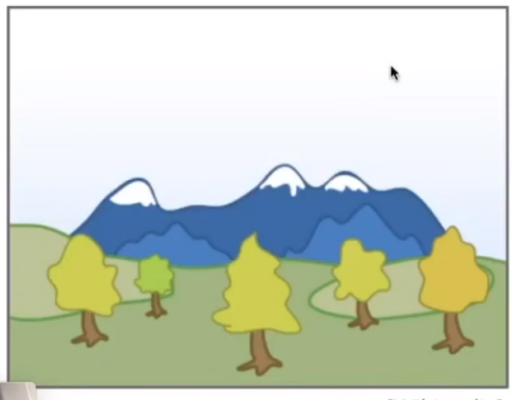
painter algorthm

umination, Shading and Graphics Pipeline)

Painter's Algorithm

Inspired by how painters paint

Paint from back to front, overwrite in the framebuffer

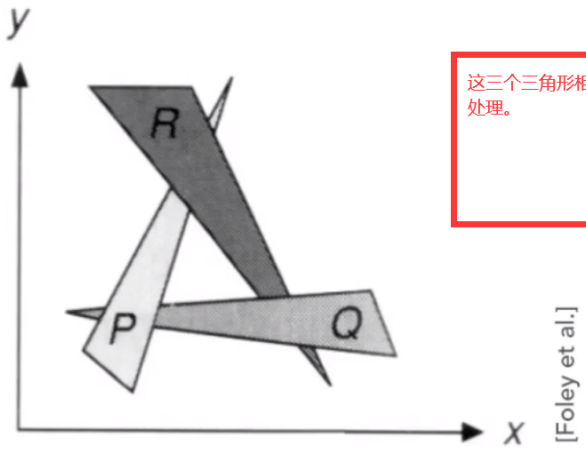


[Wikipedia]

由远到近做光栅化处理，解决遮挡问题

Requires sorting in depth (O(n log n) for n triangles)

Can have unresolvable depth order



这三个三角形相互覆盖，画家算法不好处理。

画家算法在解决上述问题是不好，故引入深度缓存

Z-Buffer 深度缓存

This is the algorithm that eventually won.

Idea:

- Store current min. z-value for each sample (pixel)
- Needs an additional buffer for depth values
  - frame buffer stores color values
  - depth buffer (z-buffer) stores depth

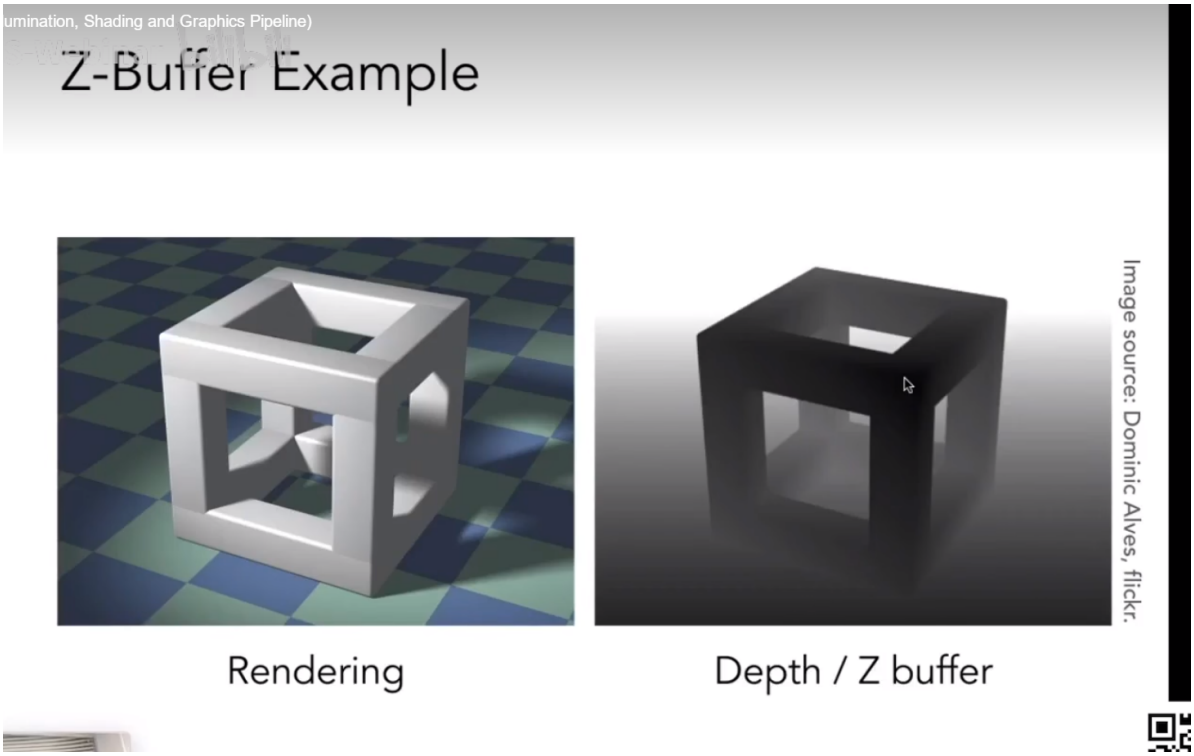
IMPORTANT: For simplicity we suppose
z is always positive
(smaller z -> closer, larger z -> further)

frame buffer:保存最终的成品

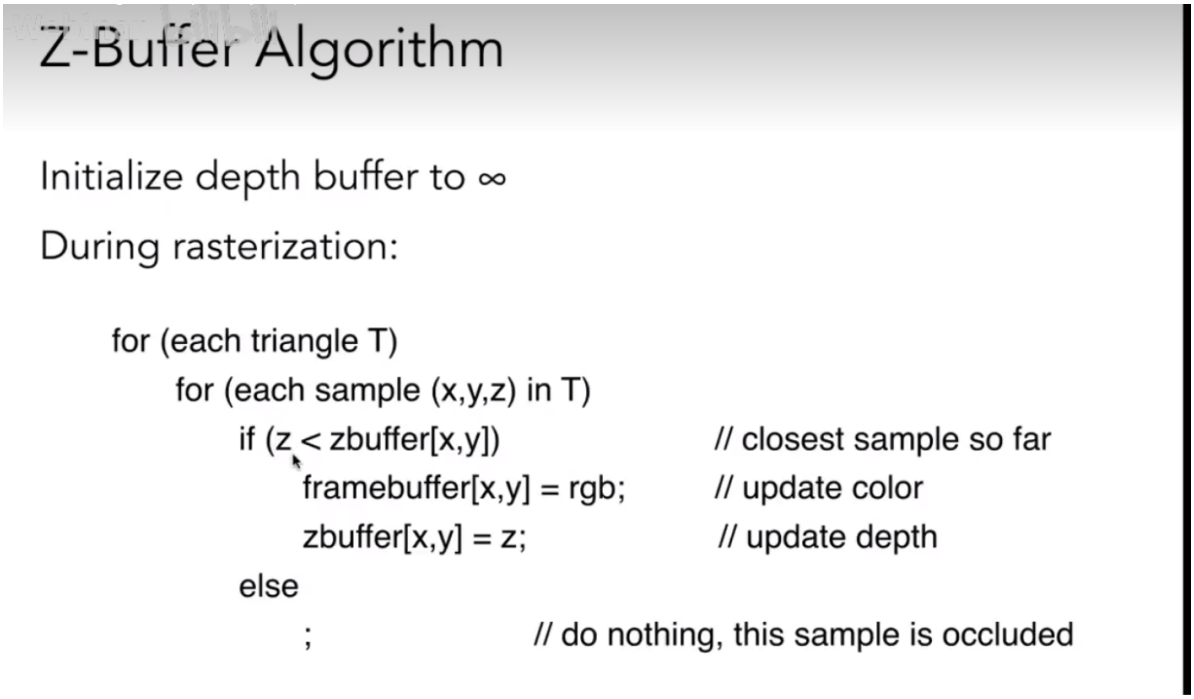
depth buffer：保存每个像素看的的最浅图

向-z看去，这里理解为 z为深度

例子



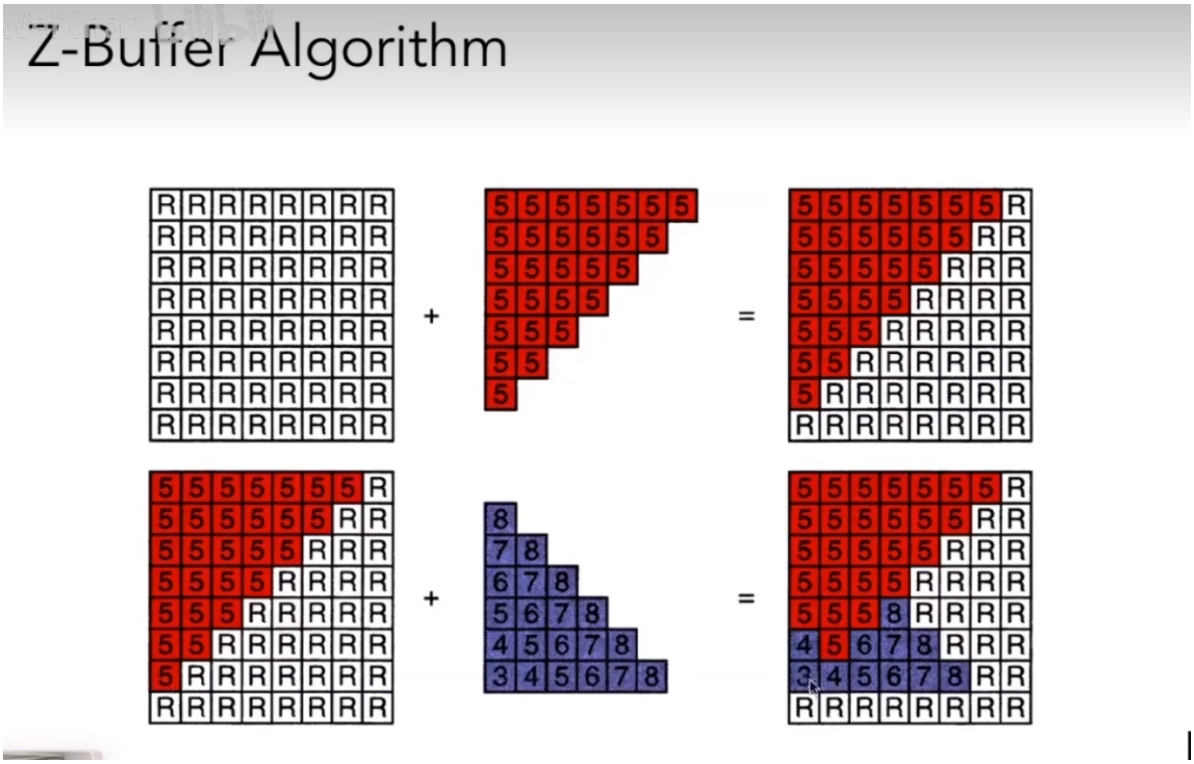
算法演示



初始默认每个三角形（一个图认为是由许多三角形组成）为无限远，

对于每个三角形，找到他覆盖的 像素，判断这个三角行和之前像素记录的那个距离更近。

具体实例



算法分析

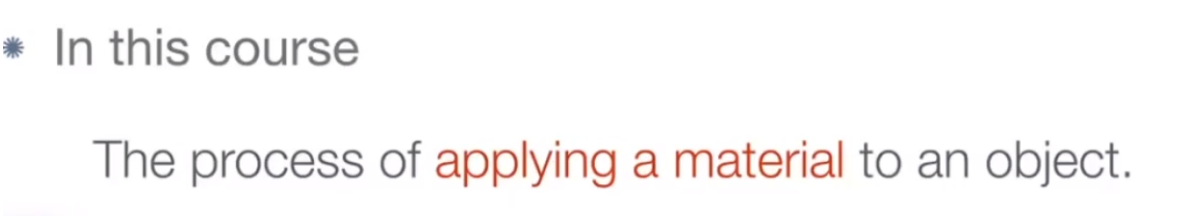
时间复杂度为O(n),注意这里只是对每个三角形处理，并没有排序

和MSAA结合

MSAA是每个像素有多个采样点，如果二者结合，那就是对每个采样点分析而不是每个像素了。

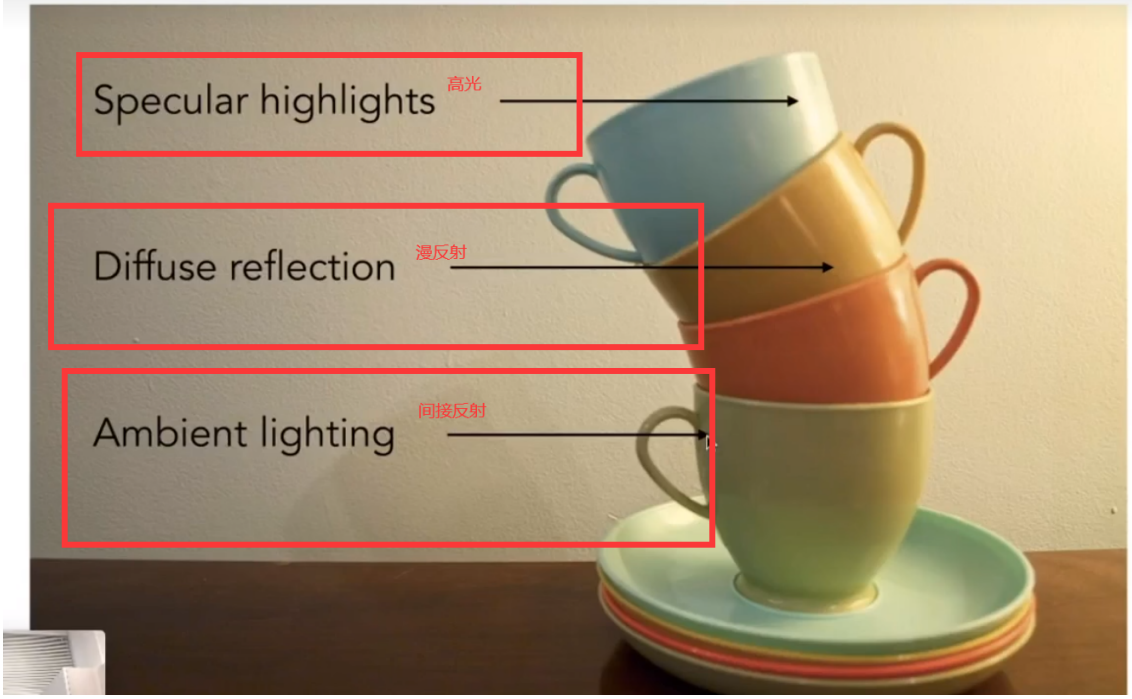
Z-Buffer处理不了透明物体

shading



对不同物体应用不同材质----着色 shading

Perceptual Observations

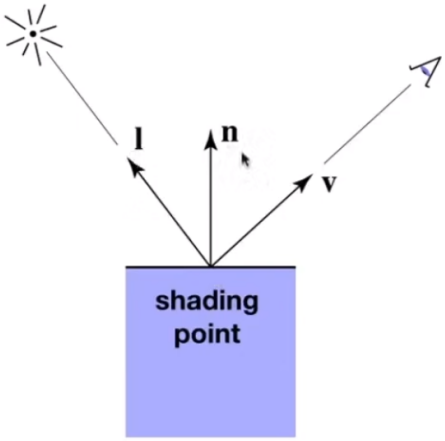


Shading is Local

Compute light reflected toward camera at a specific shading point

Inputs:

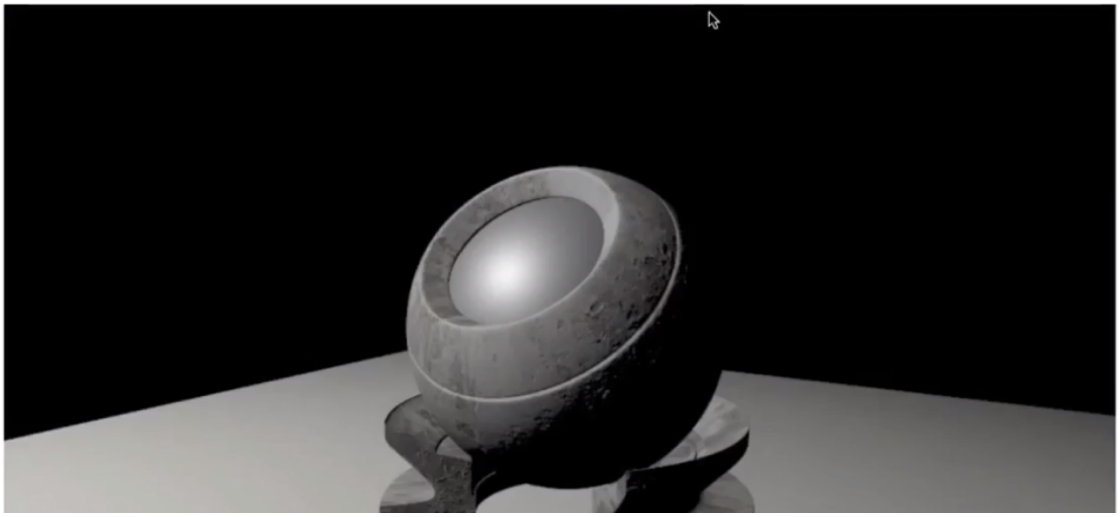
- Viewer direction, v
- Surface normal, n
- Light direction, l (for each of many lights)
- Surface parameters (color, shininess, …)



规定：法线，着色点，相机方向，光照方向

Shading is Local

No shadows will be generated! (**shading ≠ shadow**)



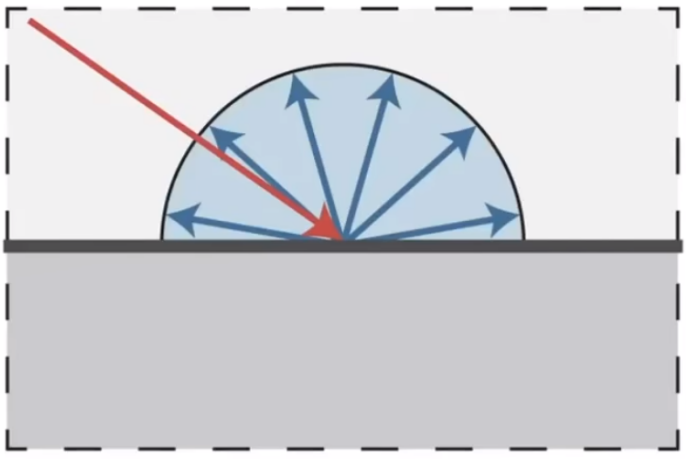
着色是只考虑局部，只考虑光线照向，会有明暗变化，不考虑其他物体，不考虑物体的遮挡而形成阴影。

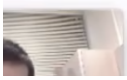
shading！=shadow（着色！=阴影）

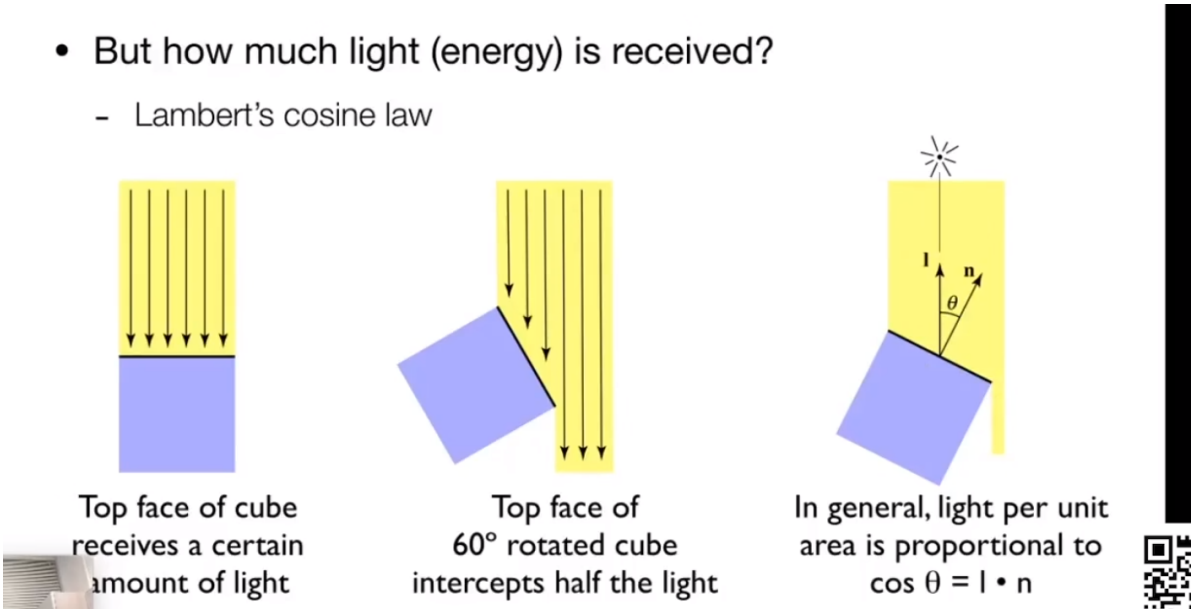
漫反射 diffuse reflection

Diffuse Reflection

- Light is scattered uniformly in all directions
  - Surface color is the same for all viewing directions

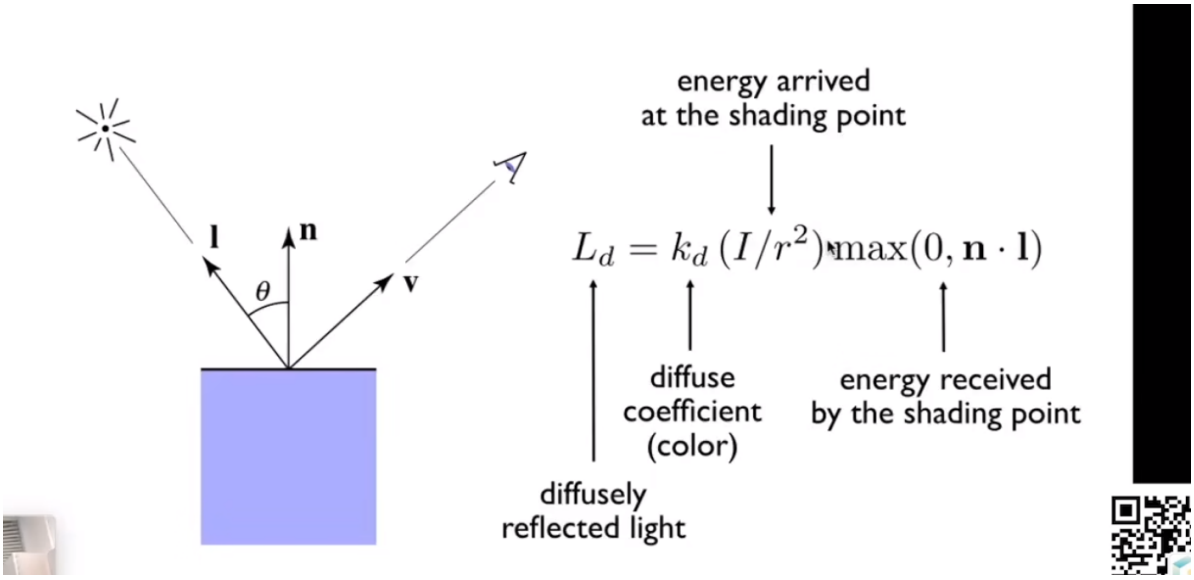






通过法线和入射光线的夹角来看接受的能量->亮度

漫反射能量公式



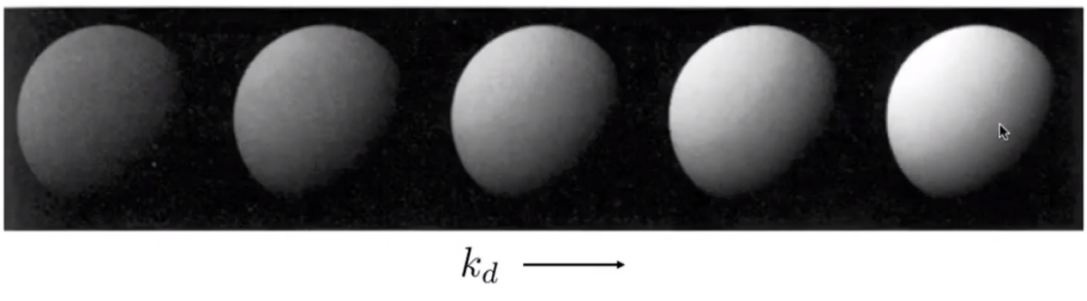
I：单位面积光源的能量，

r: 着色点到光源的距离

max（0,n . l）：向量n点乘l指着色点和光源夹角的cos值。（取0是当cos值为负数时，即从下面打到物体表面时）

Kd：表示物体对光的吸收情况，（1表示白色，什么也不吸收。0表示黑色，吸收所有）

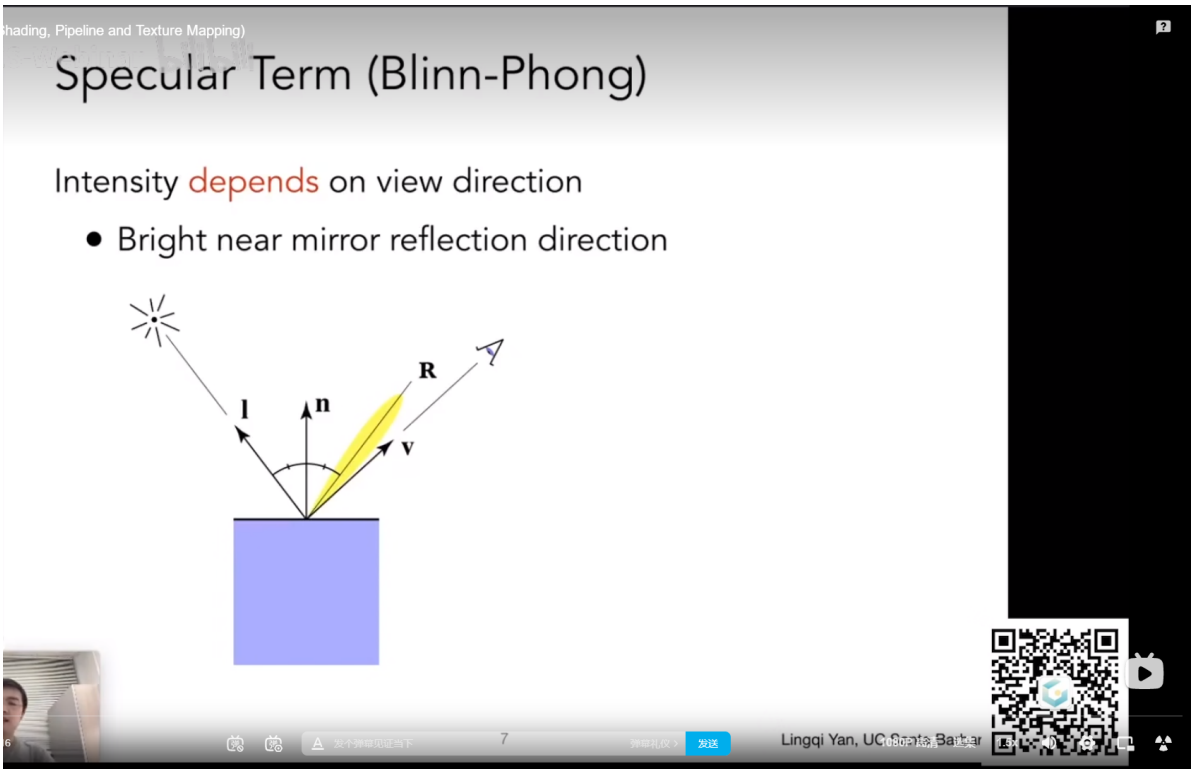
漫反射跟观察的方向无关。从哪一个骄傲都看都一样。



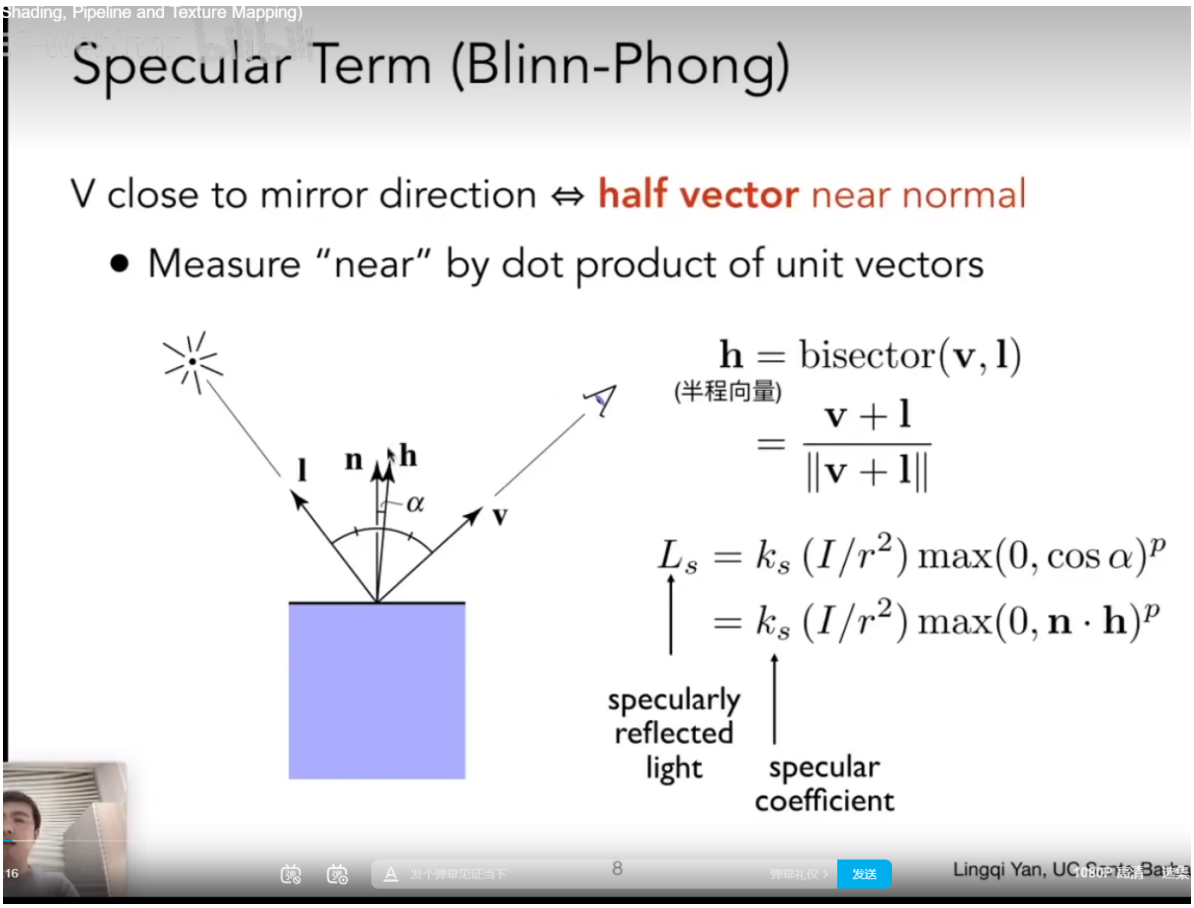
第八节、shading2

图形管线，纹理映射

BLion Phong 模型 计算高光



高光接近镜面反射的方向



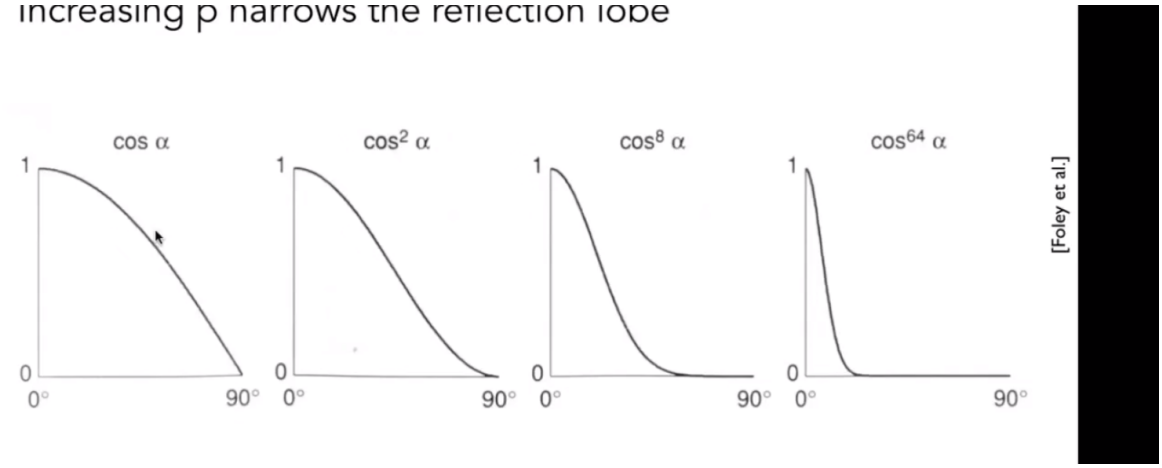
h：半程向量：就是入射向量加上观察向量，然后除以自身大小。

h向量和法线很接近（说明v和r接近就约等于h和n接近）

这里这个max（0,n*h）的 nh 的原理是，nh均为单位向量，二者相乘越接近1，说明二者越向近，说明你的观察方向很进阶高光。

ks通常认为是白色的系数

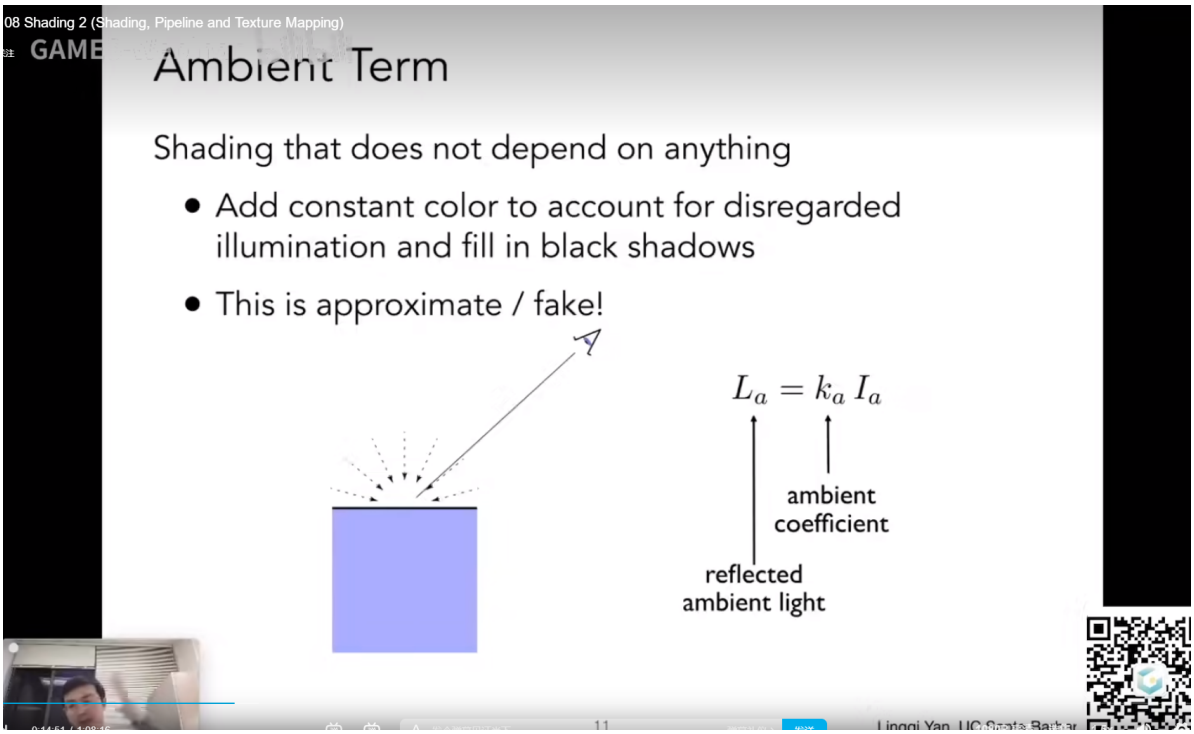
这里这个高光模型没有考虑 观察者到点的距离，只考虑了光源到点的距离。



上述公式的有一个指数p，是为了处理cosx值，即使不是位于高光是，也有较大的光照值。

故加上一个指数p， 通常为100-200.

ambient 环境光照



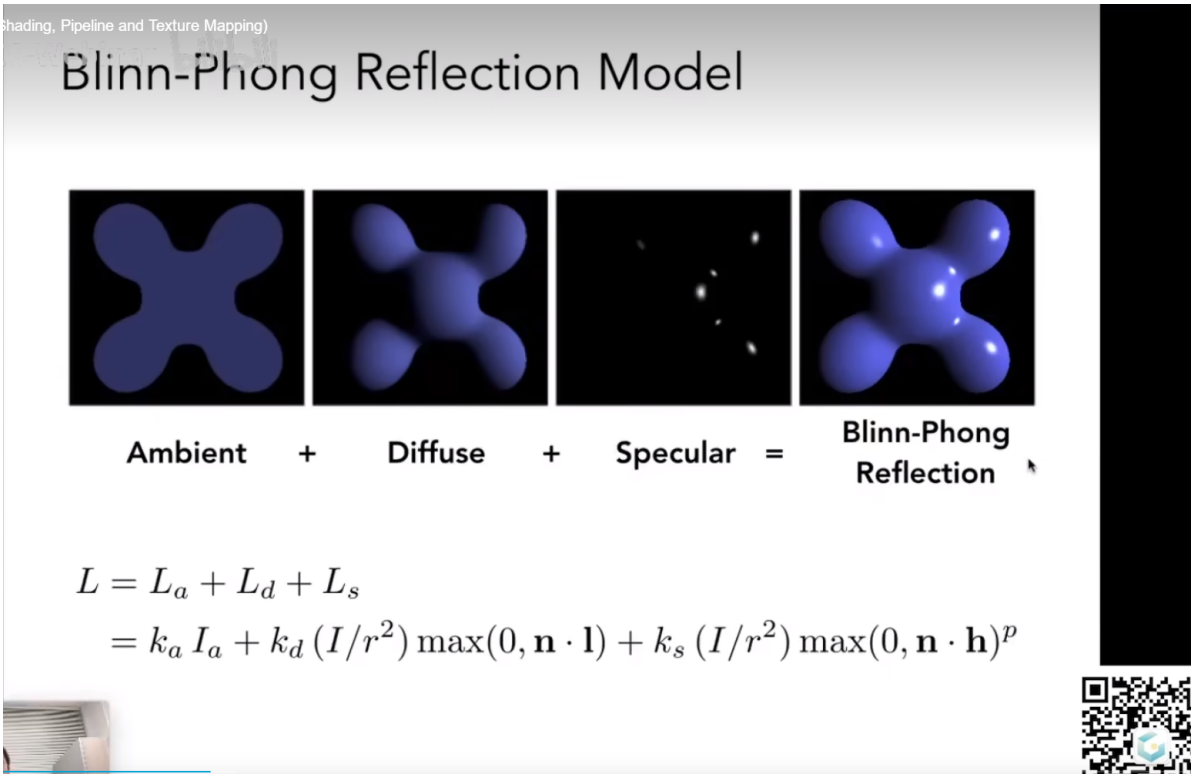
La =Kala

Ia表示环境光，这里同意认为环境光是相同的（光反射来反射去，十分复杂）

Ka 系数

特点：环境光不在乎角度 ，在哪个角度看都一样。-》常熟-》一样的颜色

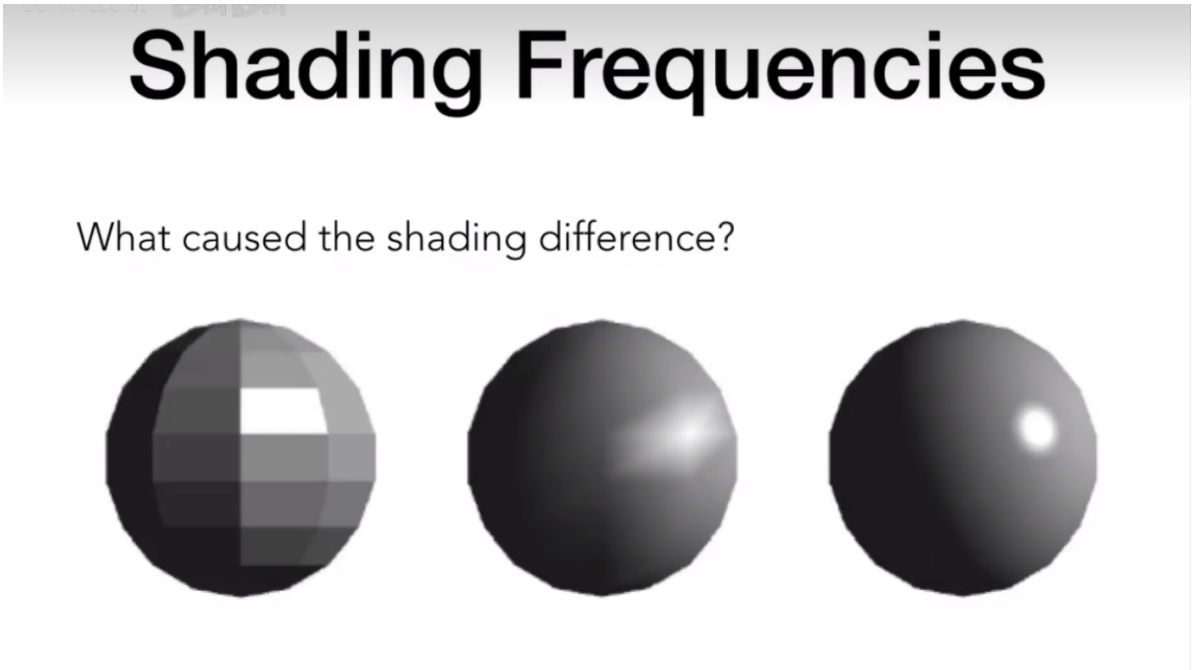
Blion Phong 反射模型



环境光 + 漫反射 + 高光

那么着色操作就是对每个点来一次上述公式

shading frequencies 着色频率



1、flat shading

每个三角行设为一个面进行说色，看起来就跟一个一个三角形拼起来。

2、Gouraud shading

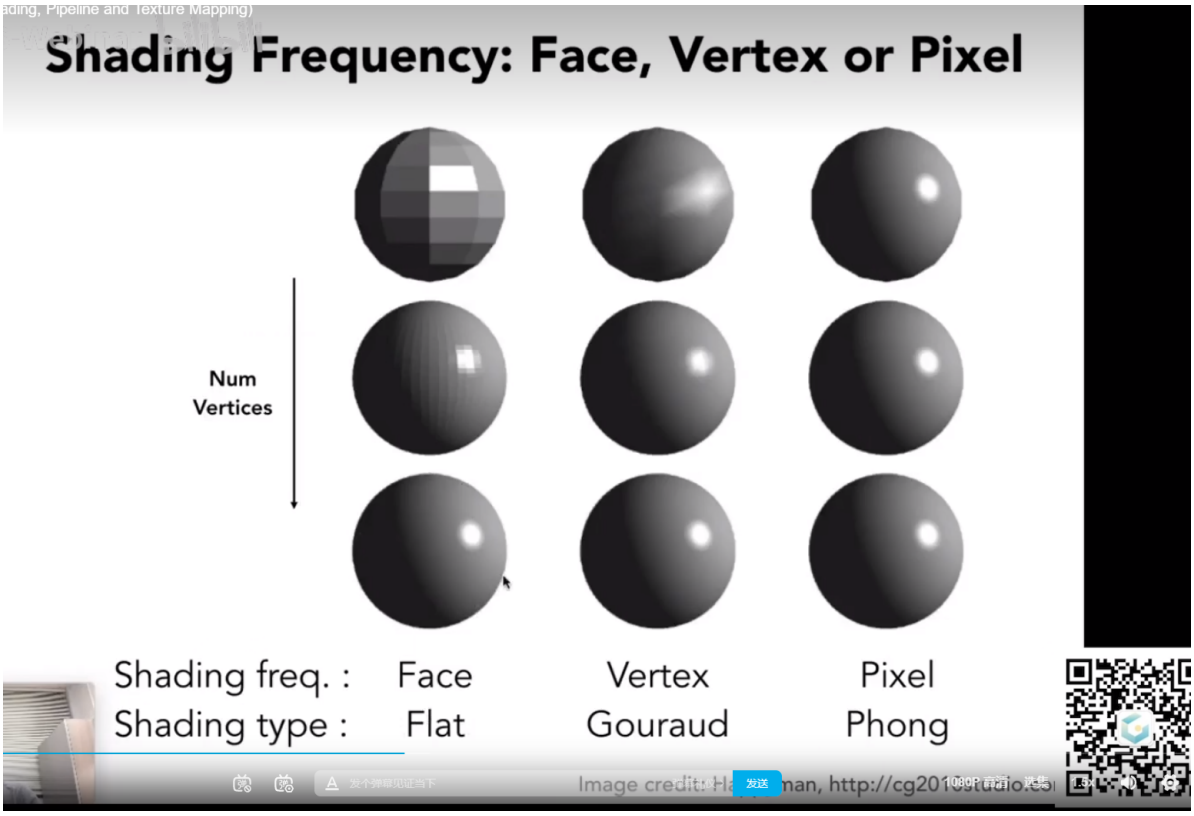
对于每个三角形，求出每个顶点的法线，对每个顶点着色。三角形内部的区域用插值的方法得出。

3、Phong shading

对每个像素着色

注意：Blion-Model是一种着色模型， Phong-shading是一种着色频率

三者区别



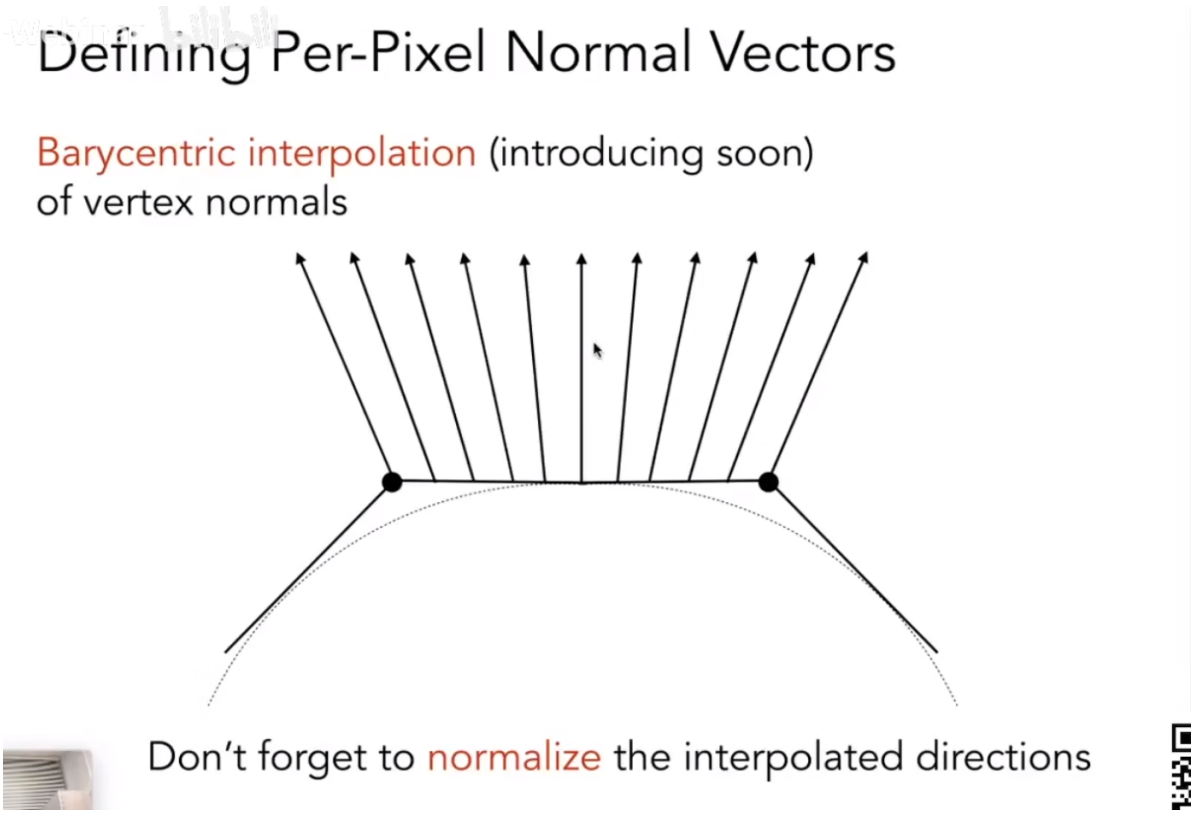
不能说flat shading就一定不好， 如果三角形的面积足够小，也十分精细

Gouraud shading 中求每点的法线的方法

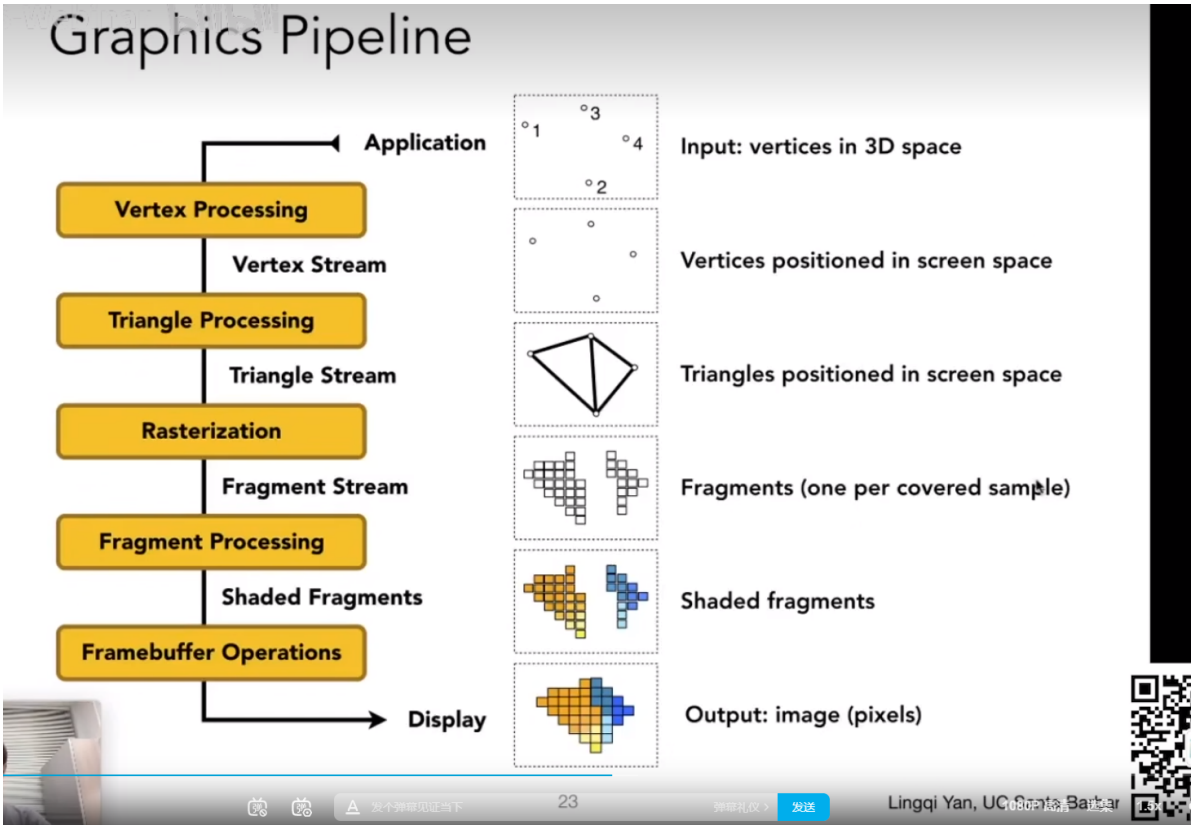
1、把点所关联的面的法线求一个平均或者加权平均

确定phong中每个像素的法线

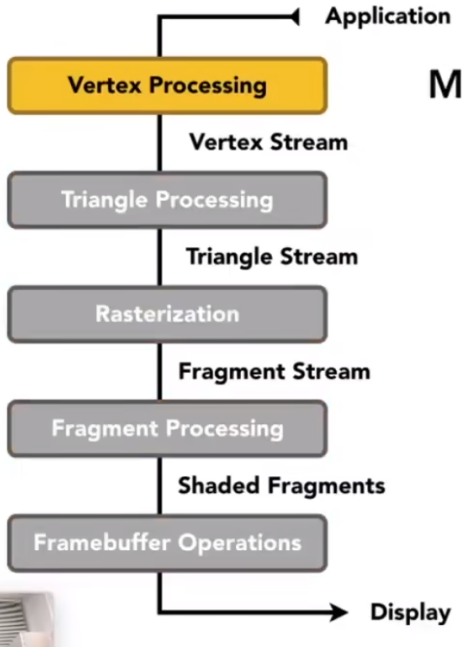
两个点的法向量之间来平滑过度



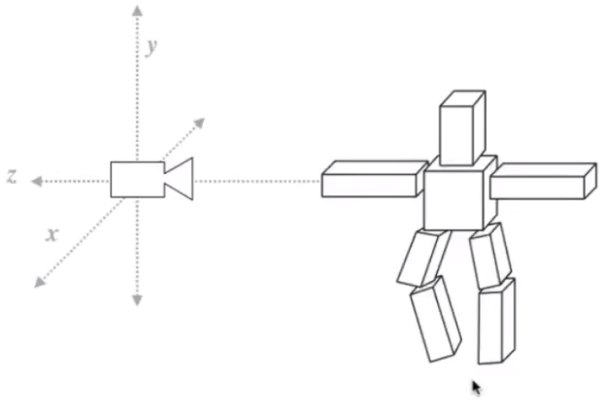
渲染管线 graphics pipeline



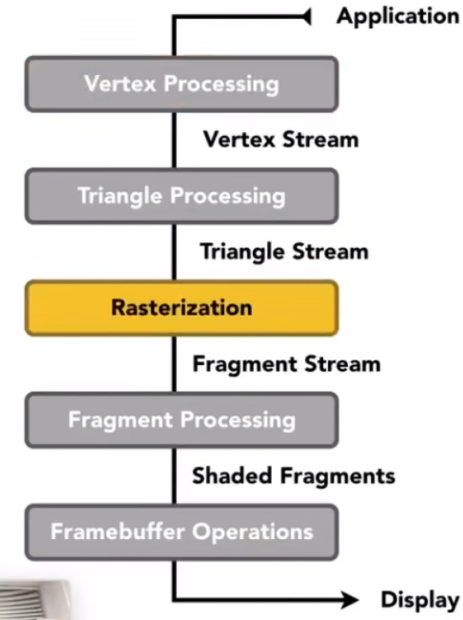
Graphics Pipeline



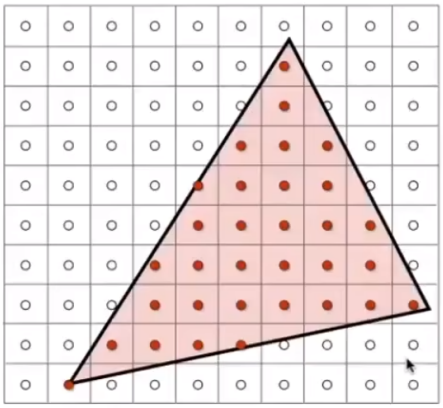
Model, View, Projection transforms

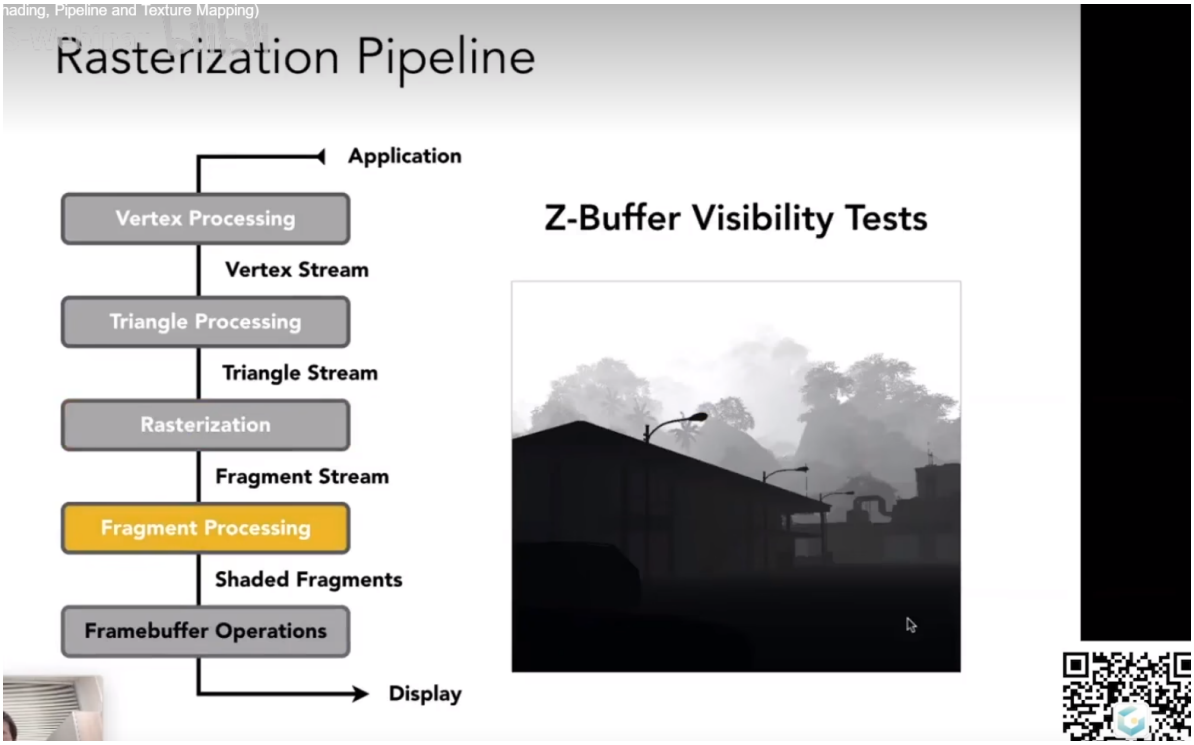


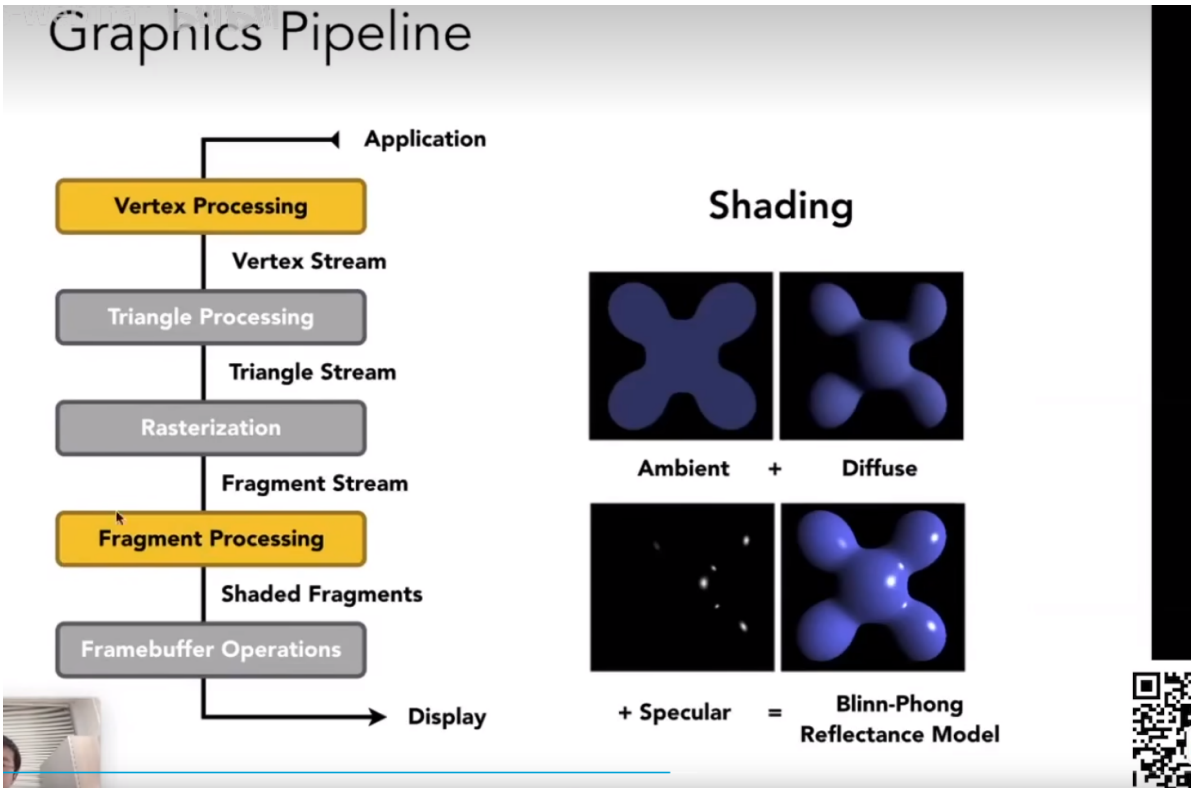
Graphics Pipeline



Sampling triangle coverage




hading, Pipeline and Texture Mapping)
Rasterization Pipeline
Application
Vertex Processing
Vertex Stream
Triangle Processing
Triangle Stream
Rasterization
Fragment Stream
Fragment Processing
Shaded Fragments
Framebuffer Operations
Display
Z-Buffer Visibility Tests


Graphics Pipeline
Application
Vertex Processing
Vertex Stream
Triangle Processing
Triangle Stream
Rasterization
Fragment Stream
Fragment Processing
Shaded Fragments
Framebuffer Operations
Display
Shading
Ambient
+
Diffuse
+ Specular
=
Blinn-Phong
Reflectance Model

shader编程

Shader Programs

- Program vertex and fragment processing stages
- Describe operation on a single vertex (or fragment)

Example GLSL fragment shader program

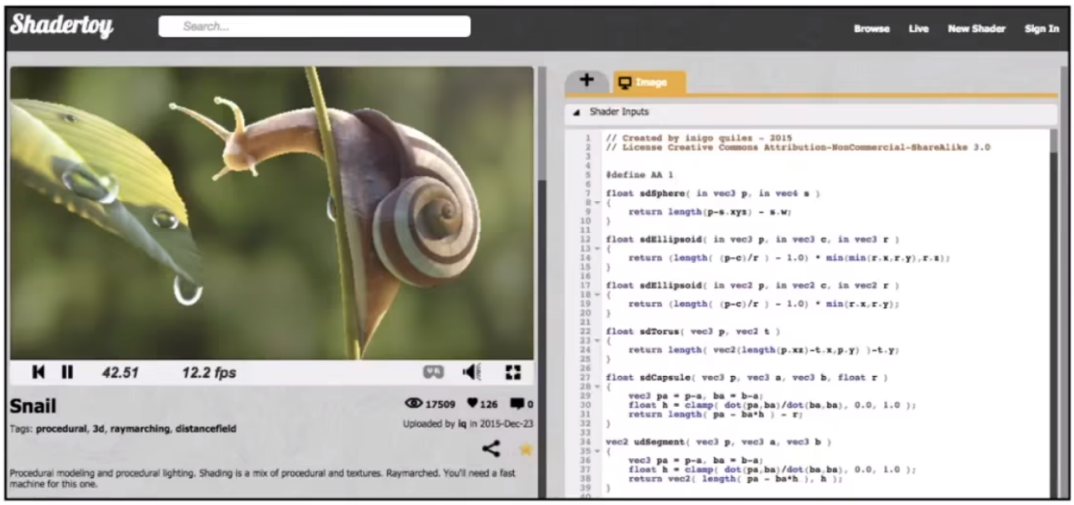
```
uniform sampler2D myTexture;
uniform vec3 lightDir;
varying vec2 uv;
varying vec3 norm;

void diffuseShader()
{
  vec3 kd;
  kd = texture2d(myTexture, uv);
  kd *= clamp(dot(–lightDir, norm), 0.0, 1.0);
  gl_FragColor = vec4(kd, 1.0);
}
```

- Shader function executes once per fragment.
- Outputs color of surface at the current fragment's screen sample position.
- This shader performs a texture lookup to obtain the surface's material color at this point, then performs a diffuse lighting calculation.

包括顶点着色和像素着色

可以直接写着色的网站

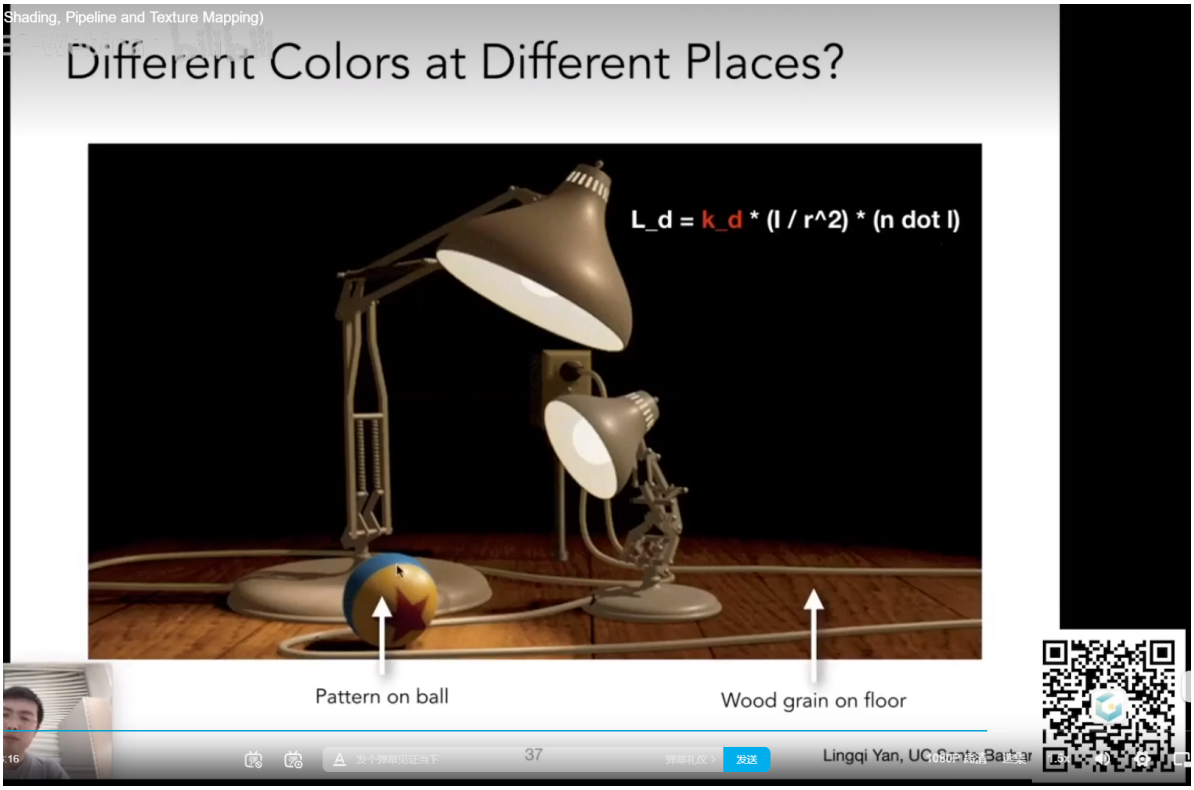
Snail Shader Program



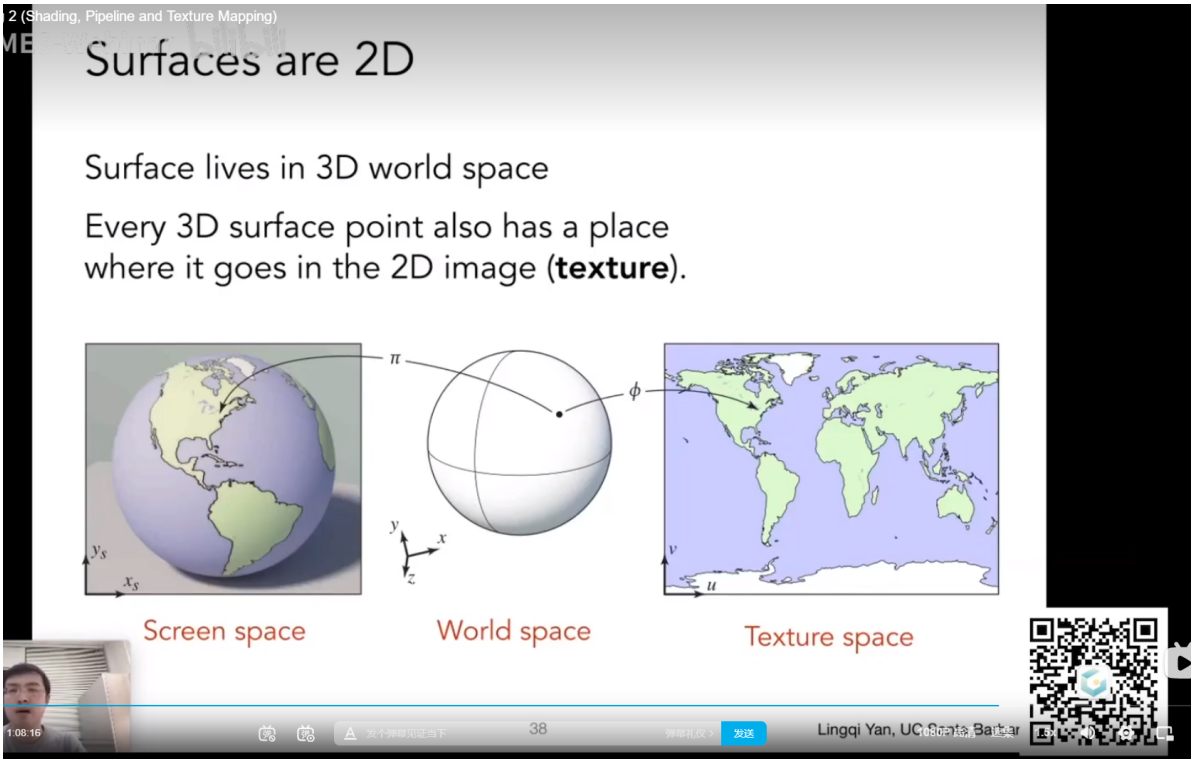
Inigo Quilez

Procedurally modeled, 800 line shader.
http://shadertoy.com/view/ld3Gz2

texture mapping 纹理映射



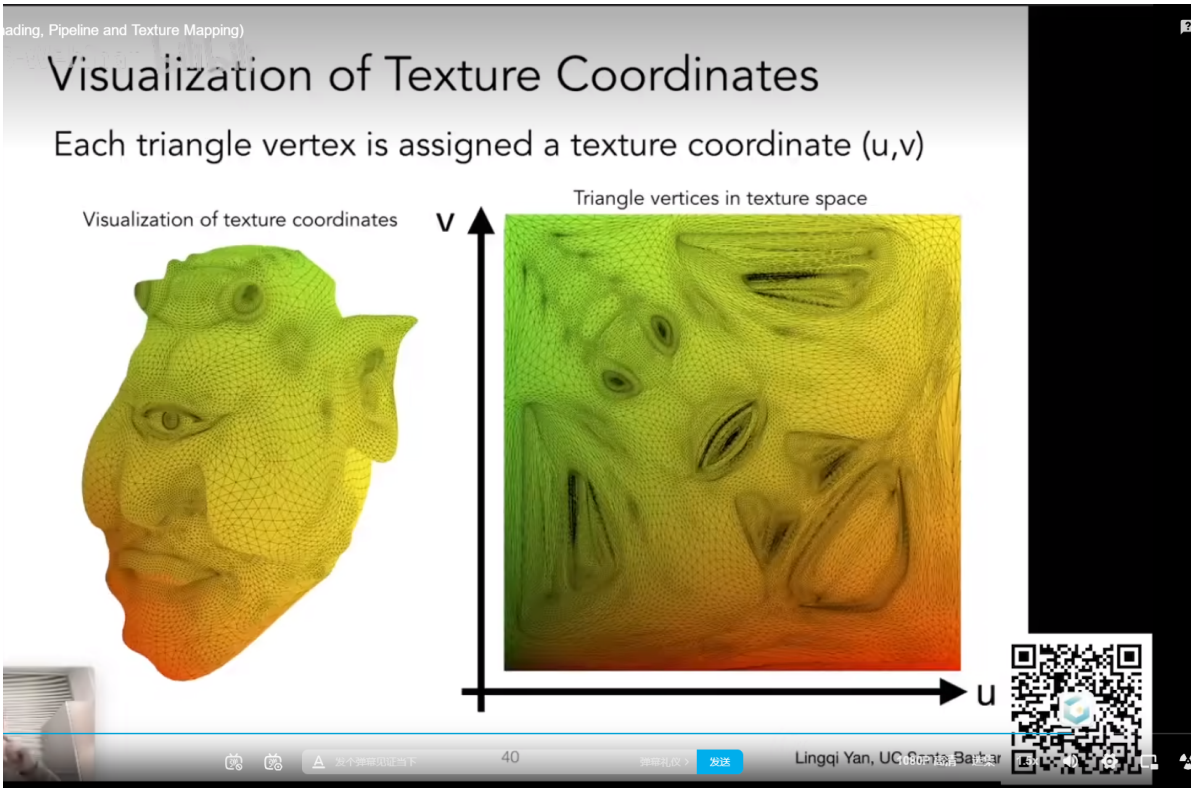
找的每个点的一个属性，这个属性可以决定漫反射系数，是物体有不同的颜色等等。



一个三维物体可以展开成一个平面，这个平面就是纹理

三维物体和这张图的一一对应关系成为纹理映射

纹理坐标系



纹理包含许多三角形

一般纹理坐标系用 u v 表示

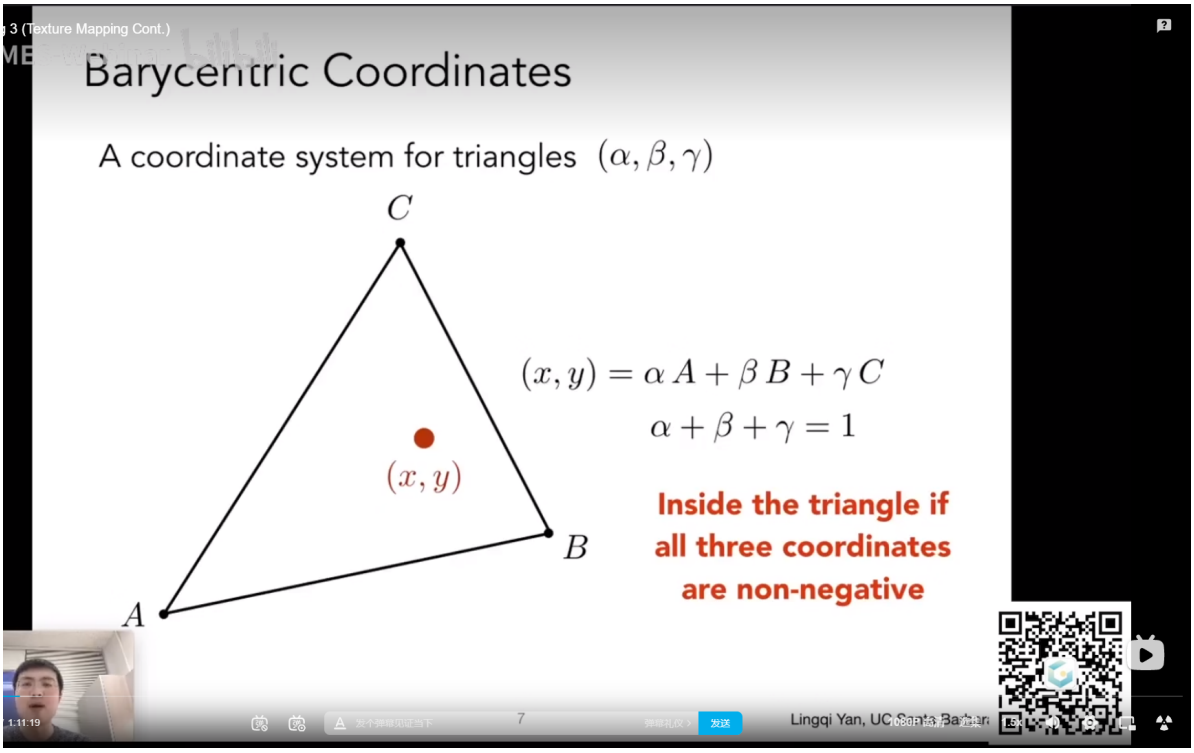
通常认为 u v 属于 0-1

可以理解纹理是 在着色时每个点都有自己的样式

第9节、Shading3（纹理应用）

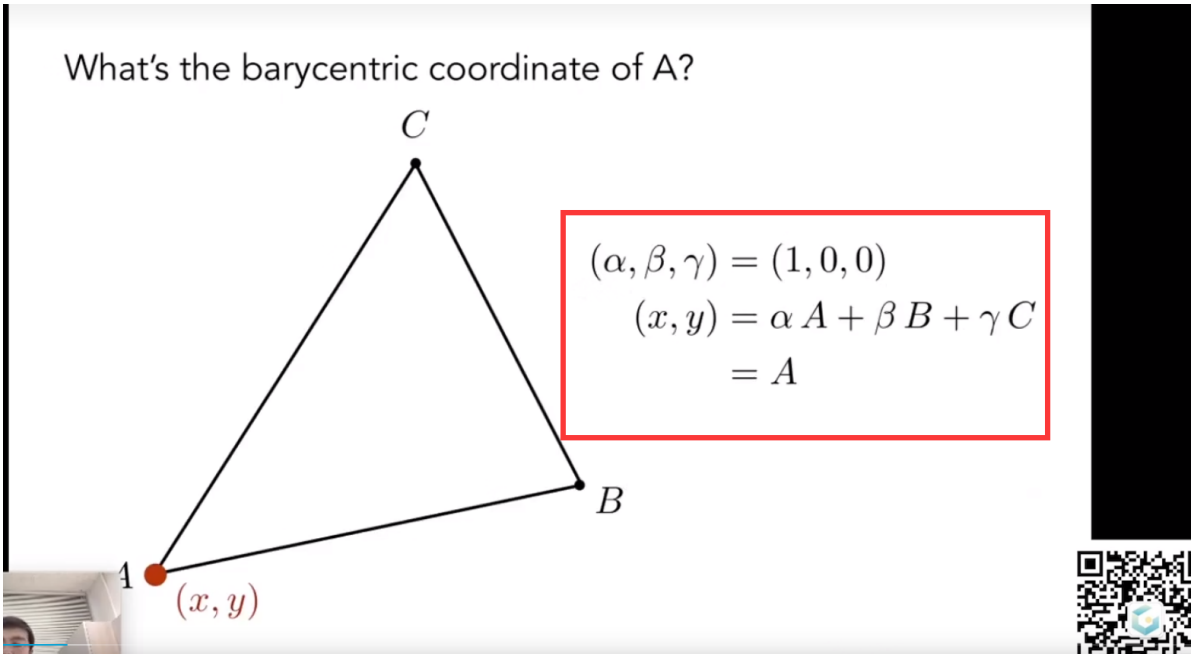
重心坐标 barycentric coordinates

是为了求三角形内部的差值

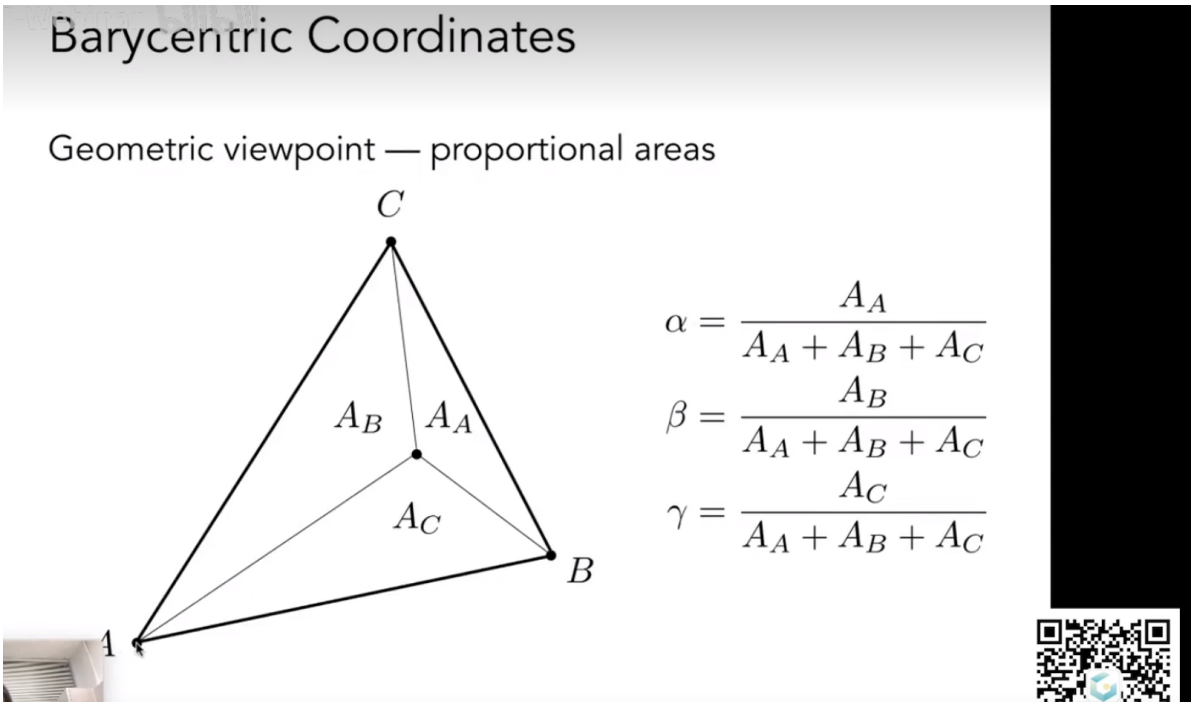


1、认为三角形内部的点可以用三个顶点来表示， 且系数和为1 α + β + γ =1 ，且系数均>0

三个顶点的坐标

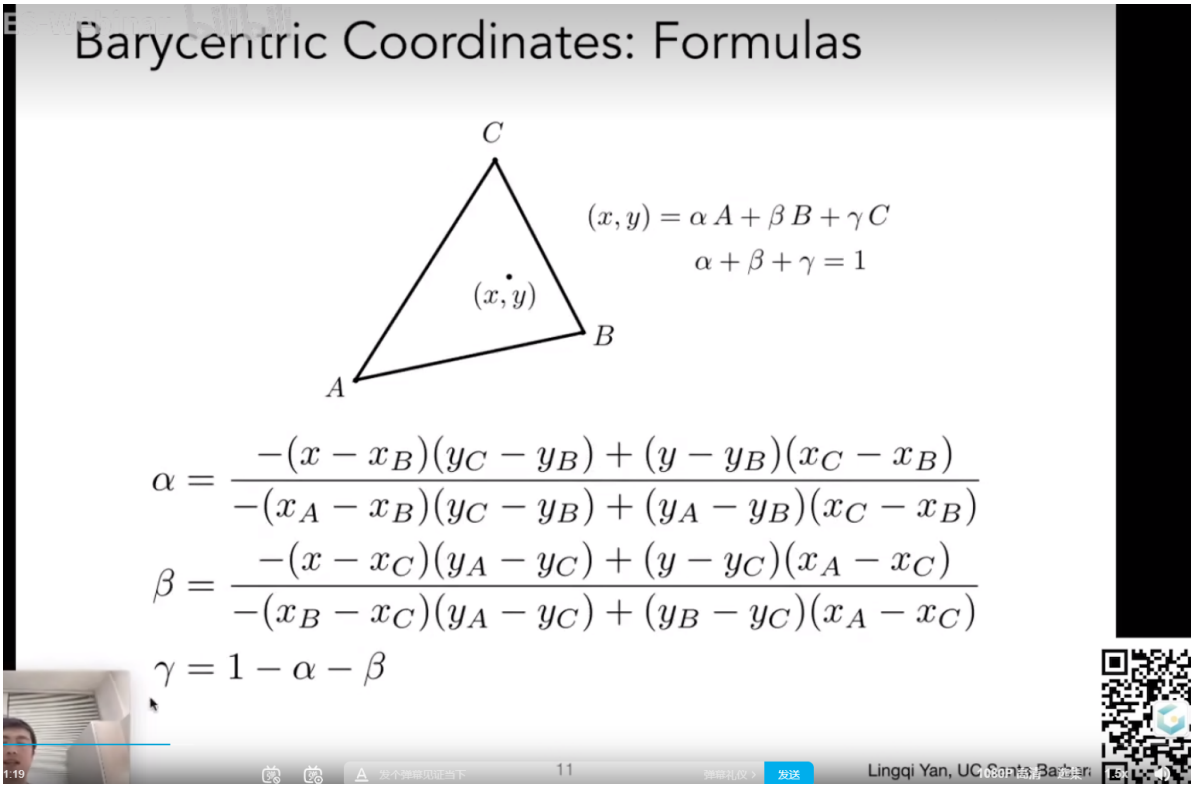


求内部点



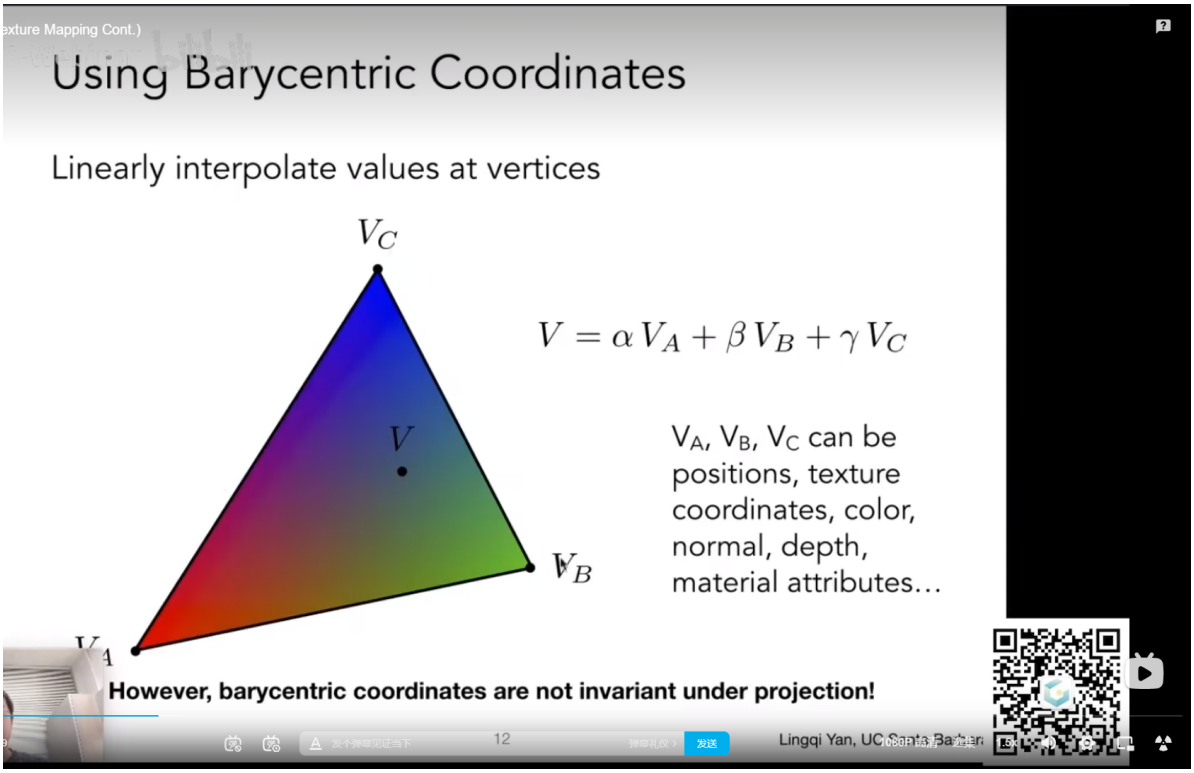
利用三角形的面积比。

三角的重心坐标为（1/3，1/3，1/3）



计算的另一个公式

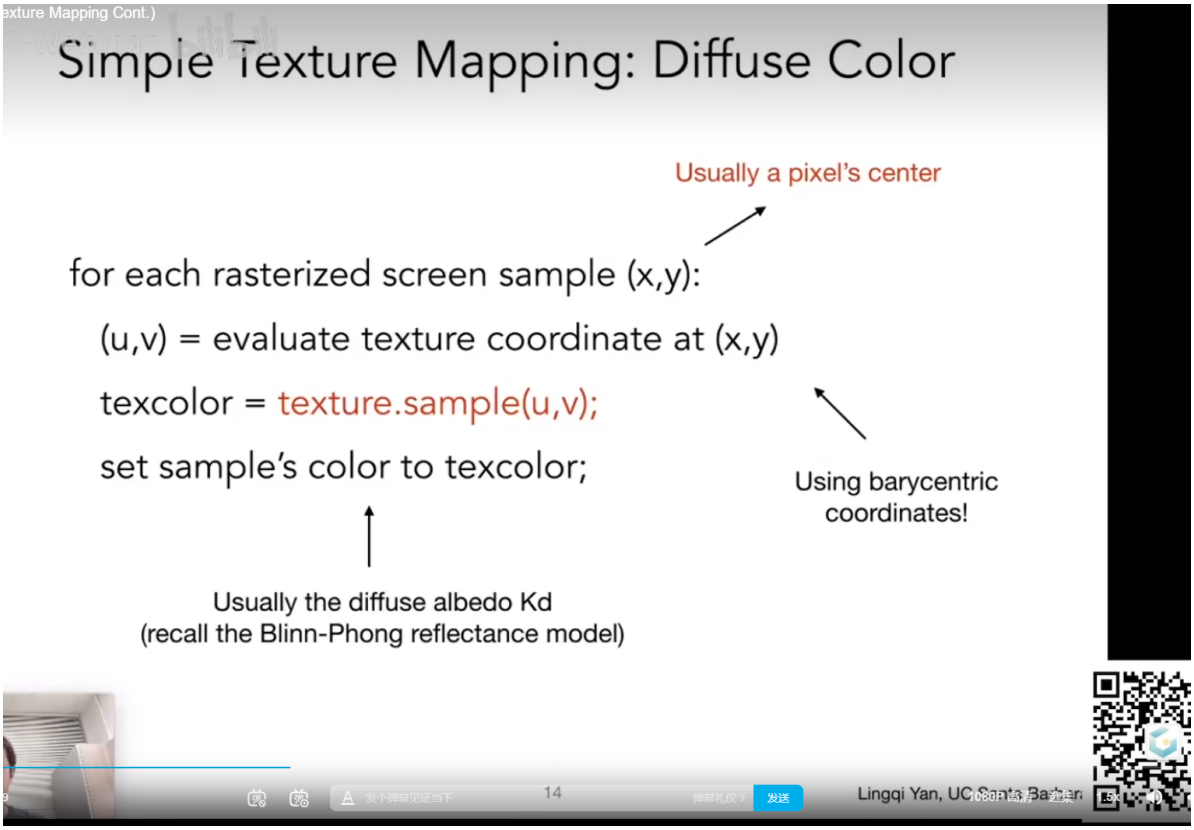
利用重心坐标来求插值（插值的属性：颜色等等）



注意：三角形在投影到平面上后，中心坐标可鞲会变化， 所以要在三巍峨空间中计算好后，在投影

应用纹理 apply texture

Applying Textures



过程：对于一个采样点（一个像素或这个MSAA后的点），计算这个位置插值出来的坐标或者uv，在纹理上查询uv

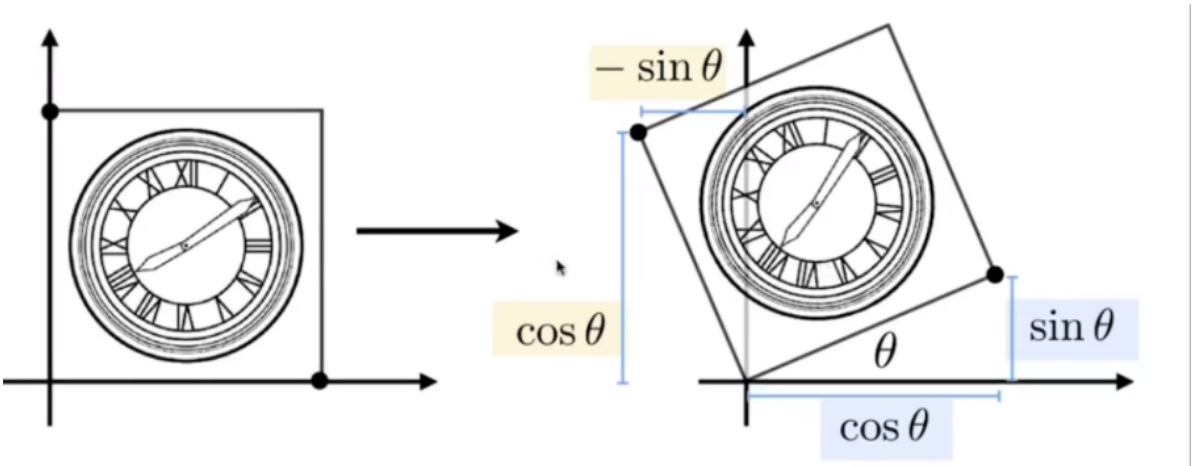
但是这样做会出现一些问题

作业讲解

作业一

知识大概是透视投影，变换

1、求一个旋转，利用旋转公式



$$\mathbf{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

2、求由透视投影 ->正则立方体上

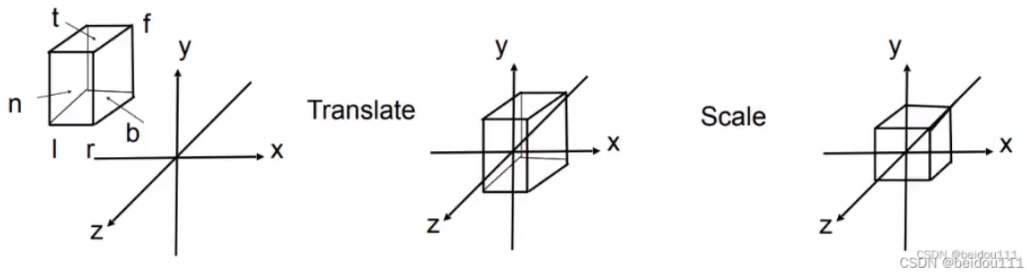
- 由透视 ->正交：

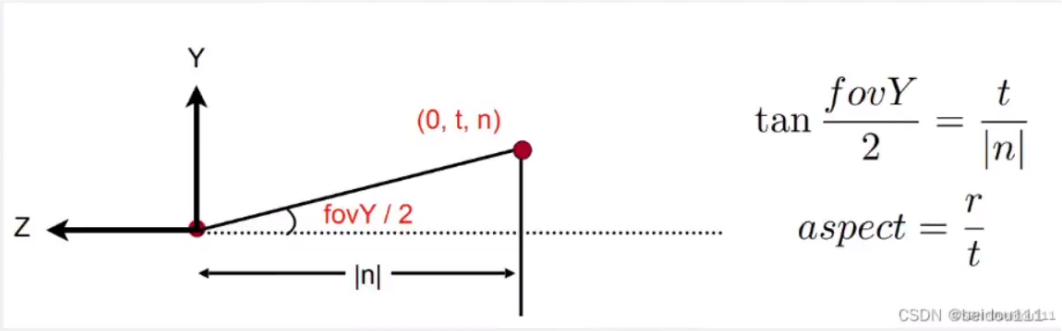
$$M = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

CSDN @beidou111

n：z轴进的　f:z轴远的

- 由正交->正则（-1 1 ）

> **2.2 将正交投影转化到正则立方体**
>
> $$M_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
>
> 
>
>
> 注意利用视锥角和宽高比求出参数



作业二

内容是光栅化

1、判断一个点是否在三角形内，利用三角形三个点和点的叉乘

注意几点：

- eigen库不支持二维的叉乘，要手写（x1y2- x2y1）或者三维，把z值设为0.

2、找到一个三角形的bounding box，判断box里面的每个点是否在三角形内。如果在，则执行深度缓冲，就是用一个更近的替换更远的。