Introduction to Computer Graphics 现代计算机图形学入门

> By 闫令琪 UC Santa Barbara https://www.bilibili.com/video/BV1X7411F744

Lecture 1 : Overview of Computer Graphics

- What is CG?
- Why study CG?

  -- Applications

  -- Fundamental Intellectual

  -- Technical Challenges
- Course Topics
- Course Logistics

What is CG?

The use of computers to synthesize and manipulate visual information.

Applications:

1、Video Games -- e.g. 只狼 画面足够亮的游戏，往往质量较高(渲染中的全局光照做的比较好)

Borderlands 3 无主之地 卡通风格(怎么实现?)

2、Movies -- The Matrix(黑客帝国) Avatar(阿凡达)

特效(special effects)通过图形学实现(特效的应用并不难,平时见得少,做的粗糙也没太大关系哈哈哈哈) Rendering

3、Animations(动画) -- Zootopia(疯狂动物城)

Frozen 2(冰雪奇缘2)

4、Design -- CAD(Computer Aided Design)计算机辅助制图

Ikea(宜家) 75% of catalog is rendered imagery

5、visualization 可视化

6、Virtual Reality 虚拟现实VR:看到的东西都是电脑生成的

Augmented Reality 增强现实AR:可以看到现实的东西+电脑增加的部分

7、Digital Illustration 数码插画 -- Photoshop

8、Simulation 模拟 -- 沙尘暴、黑洞

9、GUI:Graphics User Interfaces 图形用户接口

10、Typography 字体设计: 点阵/矢量

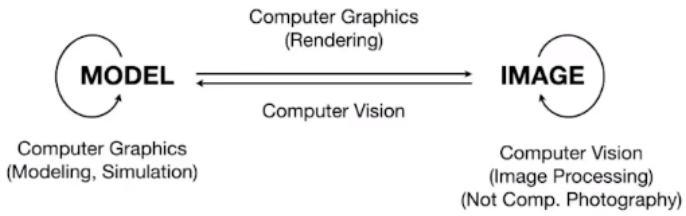
Course Topics:

- Rasterization 光栅化 OpenGL/Shader如何运作
- Curves and Meshes 曲线与曲面
- Ray Tracing 光线追踪
- Animation / Simulation 动画/模拟

CG ≠ CV Deep Learning

CV是处理现实问题，CG是重现现实世界

- Personal Understanding



Lecture 2 : Review of Linear Algebra

- 向量(矢量) Vectors

Vector Normalization $\hat{a} = \vec{a}/\|\vec{a}\|$ a-hat

Vector Addition

Vector Multipication:

Dot(scalar) Product 点乘 $\vec{a} \cdot \vec{b} = \|\vec{a}\|\|\vec{b}\| \cos\theta$

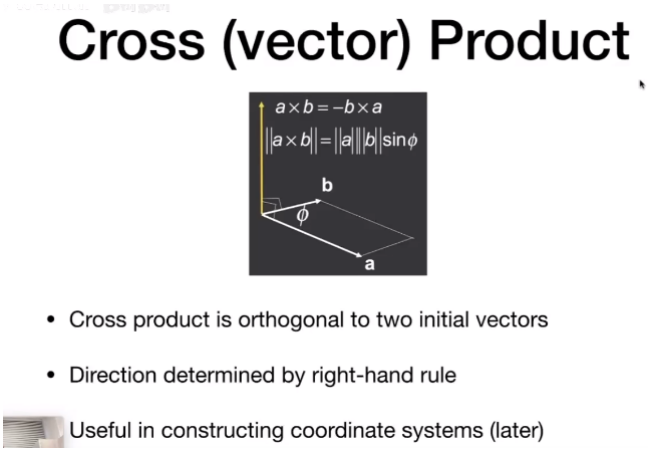
在图像中可以找到两个向量的夹角；

找一个向量在另一个向量上的投影；

b-b.perp可以measure两个向量接近的程度；

determine forward/backward；

Cross product 叉乘



Determine left/right

Determine inside/outside

- 矩阵 Matrices

Non-commutative

Associative and distributive

Lecture 3 : Transformation

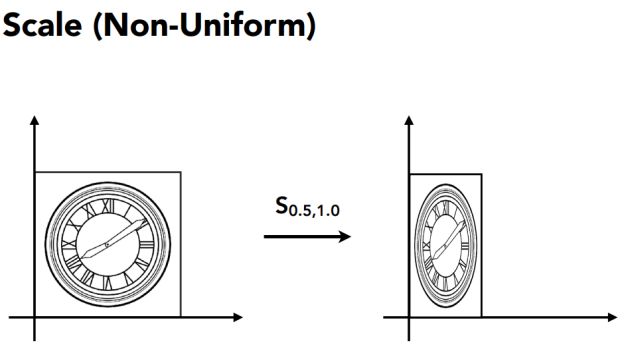
- Why study transformation
- 2D trans: rotation, scale, shear
- Homogeneous coordinates
- Composing transforms
- *3D trans

Why study trans?

-- Modeling 模型变换

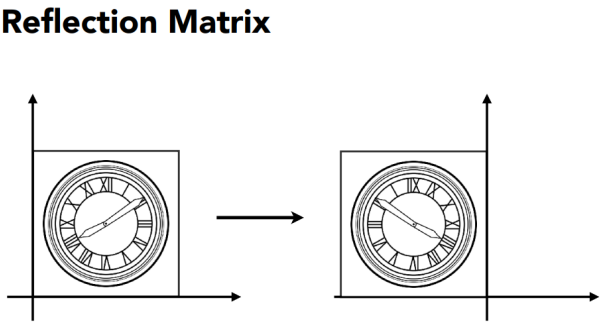
-- Viewing 视图变换

1、Scale:缩放变换



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

2、Reflection:反射



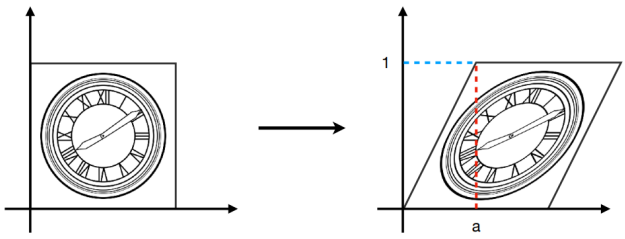
Horizontal reflection:

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

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

3、Shear:切变

Shear Matrix



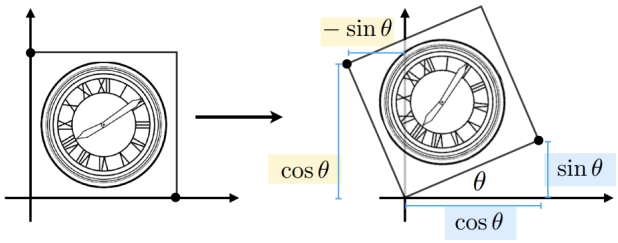
Hints:

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

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

4、Rotate:旋转

Rotation Matrix



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

以上都是线性变换:能写成 X′ = MX的形式(M是同维度的矩阵)

齐次坐标 Homogeneous coordinates

Translation cannot be represented in matrix form

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

(So, translation is NOT linear transform!)

平移(Translation)变换：不能写成X′ = MX的形式

但是!我们不希望平移成为一种special case

解决方法:引入齐次坐标(what's the cost? -- trade-off)

Solution: Homogenous Coordinates

Add a third coordinate (*w-coordinate*)

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

Matrix representation of translations

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

What if you translate a vector?

加入三维坐标:点增加一个1,向量增加一个0

向量是0,点是1的原因:

①保护向量的平移不变性,向量经过左边这个矩阵的乘法之后仍然是(x,y,0)而不是$x+t_x$;

②点是1的话, point-point=0,符合point-point=vector

- vector + vector = vector
- point - point = vector
- point + vector = point
- point + point = 这两个点的中点(根据下面的法则)

In homogeneous coordinates,

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

w只有0和1有意义，如果不是0/1，三个数都同除w

Affine Transformation(仿射变换):

使用齐次坐标之后，只需要用一个矩阵即可

Affine map = linear map + translation

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

Using homogenous coordinates:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Affine map = linear map + translation **先线性变换再平移**

齐次坐标下的其他变换：

2D Transformations

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

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

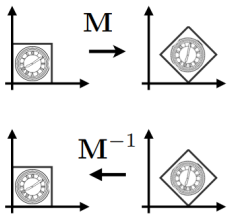
代价(cost):增加了一个维度(但是很多是0,代价不大)

Inverse Transform 逆变换:

Inverse Transform

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

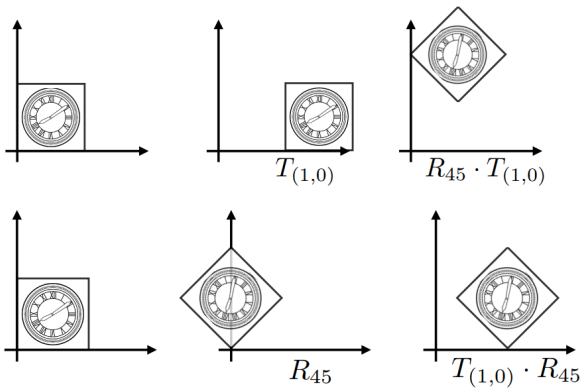
$\mathbf{M}^{-1}$ is the inverse of transform $\mathbf{M}$ in both a matrix and geometric sense



Composing Transforms 组合变换:

变换的顺序很重要(矩阵乘法无交换律)

Transform Ordering Matters!



变换的应用顺序是**从右到左**

Lecture 4 : Transformation cont.

- 3D transformations
- Viewing(观测) transformation

  -- View(视图)/ Camera transformation

  -- Projection(投影) transformaiton

    -- Orthographic(正交) projection

    -- Perspective(透视) projection

3D transformations:

原理同二维:

Use homogeneous coordinates again:

- **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)

三维中的仿射变换(同样是先线性变换再平移):

Use 4×4 matrices for affine transformations

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

三维的缩放&平移:

Scale

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

Translation

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

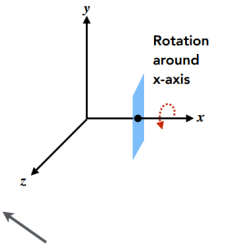
三维的旋转:

Rotation around x-, y-, or z-axis

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

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

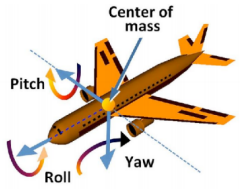
$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



任意旋转: $\mathbf{R}_{xyz}(\alpha, \beta, \gamma) = \mathbf{R}_x(\alpha)\,\mathbf{R}_y(\beta)\,\mathbf{R}_z(\gamma)$

- **Often used in flight simulators: roll, pitch, yaw**

欧拉角



罗德里格斯旋转公式:

Rodrigues' Rotation Formula

Rotation by angle α around axis $\mathbf{n}$

$$\mathbf{R}(\mathbf{n}, \alpha) = \cos(\alpha)\,\mathbf{I} + (1 - \cos(\alpha))\,\mathbf{n}\mathbf{n}^T + \sin(\alpha) \underbrace{\begin{pmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{pmatrix}}_{\mathbf{N}}$$

三维空间投影到二维平面:

View / Camera transformation:

- What is view transformation?
- Think about how to take a photo?

  Find a good place and arrange people(**model** transformation)

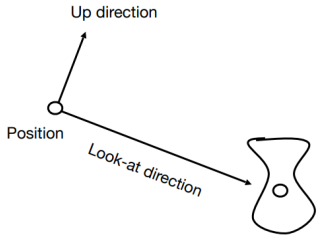
  Find a good angle to put the camera(**view** transformation)

  Cheese! (**projection** transformation)

↑MVP变换

Define the camera:

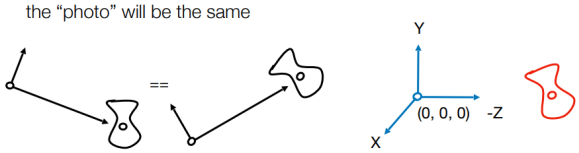
- How to perform view transformation?
- Define the camera first
  - Position $\vec{e}$
  - Look-at / gaze direction $\hat{g}$
  - Up direction $\hat{t}$
    (assuming perp. to look-at)



为了方便,把camera平移到原点,同时保持camera和物体的相对位置不变

View / Camera Transformation

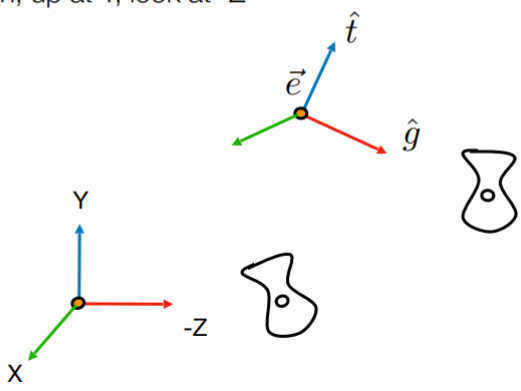
- Key observation
  - If the camera and all objects move together, the "photo" will be the same



- How about that we always transform the camera to
  - The origin, up at Y, look at -Z
  - And transform the objects along with the camera

View / Camera Transformation

- Transform the camera by M_{view}
  - So it's located at the origin, up at Y, look at -Z
- M_{view} in math?
  - Translates e to origin
  - Rotates g to -Z
  - Rotates t to Y
  - Rotates (g x t) To X
  - Difficult to write!



View / Camera Transformation

- M_{view} in math?
  - Let's write $M_{view} = R_{view}T_{view}$
  - Translate e to origin

$$T_{view} = \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Rotate g to -Z, t to Y, (g x t) To X
  - Consider its inverse rotation: X to (g x t), Y to t, Z to -g

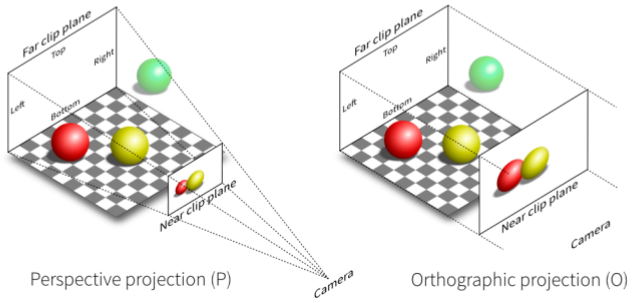
$$R_{view}^{-1} = \begin{bmatrix} x_{\hat{g}\times\hat{t}} & x_t & x_{-g} & 0 \\ y_{\hat{g}\times\hat{t}} & y_t & y_{-g} & 0 \\ z_{\hat{g}\times\hat{t}} & z_t & z_{-g} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{WHY?} \quad R_{view} = \begin{bmatrix} x_{\hat{g}\times\hat{t}} & y_{\hat{g}\times\hat{t}} & z_{\hat{g}\times\hat{t}} & 0 \\ x_t & y_t & z_t & 0 \\ x_{-g} & y_{-g} & z_{-g} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

至此,MV变换已经完成,为之后的P变换奠定基础!

Projection Transformation(投影变换):

Projection Transformation

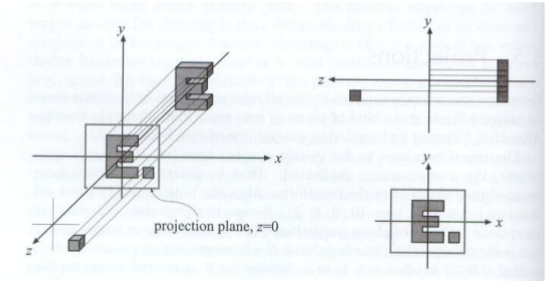
- Perspective projection vs. orthographic projection



- Perspective projection 透视投影(近大远小)
- Orthographic projection 正交投影(一叶障目) 相机无限远

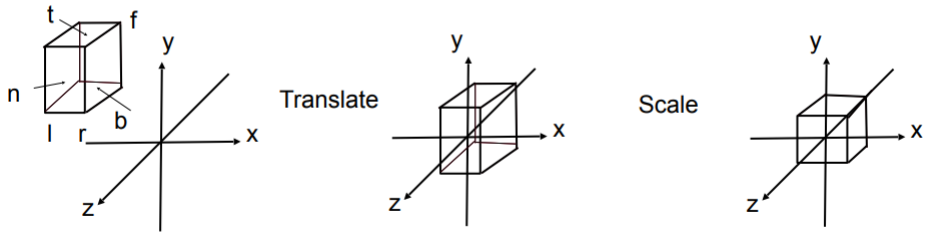
Orthographic Projection

- A simple way of understanding
  - Camera located at origin, looking at -Z, up at Y (looks familiar?)
  - Drop Z coordinate
  - Translate and scale the resulting rectangle to $[-1, 1]^2$



更加正规的方法(Bounding Box):

- In general
  - We want to map a cuboid [l, r] x [b, t] x [**f, n**] to the “canonical (正则、规范、标准)” cube $[-1, 1]^3$

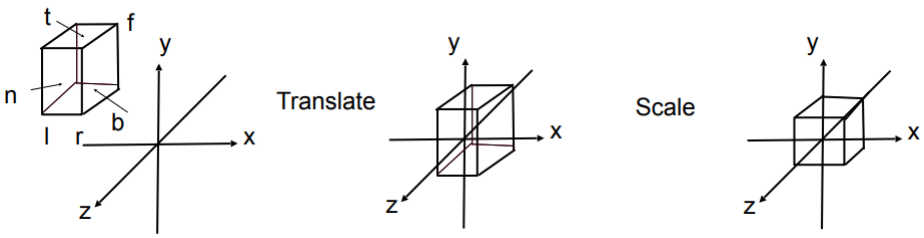


先找到视野的Bounding Box的边界

left,right; top,bottom; near,far.

- Transformation matrix?
  - Translate (**center** to origin) first, then scale (length/width/height to **2**)

$$M_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

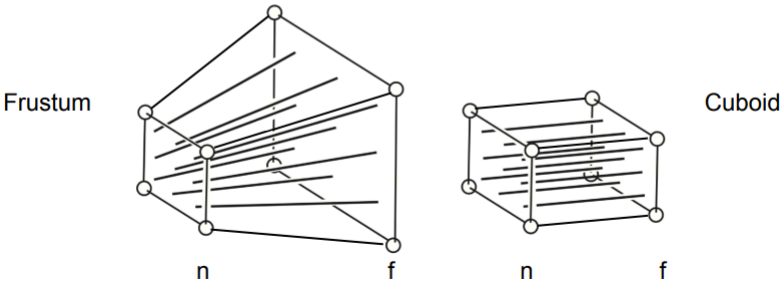


先平移到中心，再缩放到[-1,1]区间。

注意:为了保证右手系,摄像机指向-Z,所以近的物体Z坐标大,远的物体Z坐标小。

Perspective projection(透视投影):

- How to do perspective projection
  - First “squish” the frustum into a cuboid (n -> n, f -> f) ($M_{persp->ortho}$)
  - Do orthographic projection (M_{ortho}, already known!)

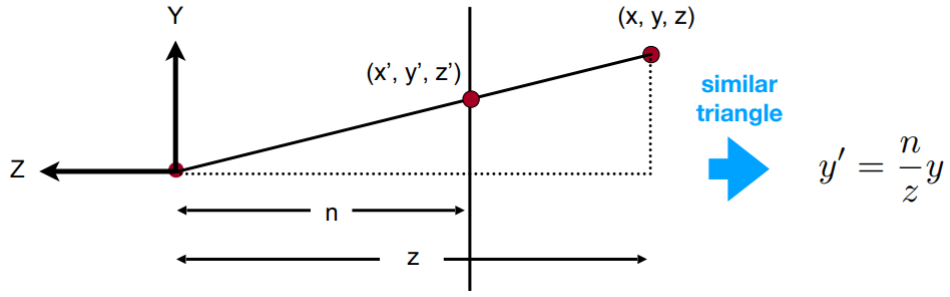


把远平面“挤”成近平面,再作正交投影。

条件:近平面、远平面、远平面中心在挤压过程中不变。

挤压的计算：相似三角形

- In order to find a transformation
  - Recall the key idea: Find the relationship between transformed points (x’, y’, z’) and the original points (x, y, z)



推导$M_{persp->ortho}$:

- So the “squish” (persp to ortho) projection does this

$$M_{persp\to ortho}^{(4\times 4)} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ \text{unknown} \\ z \end{pmatrix}$$

- Already good enough to figure out part of $M_{persp->ortho}$

$$M_{persp\to ortho} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

WHY?

推导第三行:

- Observation: the third row is responsible for z'
  - Any point on the near plane will not change
  - Any point's z on the far plane will not change

- Any point on the near plane will not change

$$M_{persp\to ortho}^{(4\times4)}\begin{pmatrix}x\\y\\z\\1\end{pmatrix}=\begin{pmatrix}nx\\ny\\\text{unknown}\\z\end{pmatrix}\xrightarrow{\text{replace z with n}}\begin{pmatrix}x\\y\\n\\1\end{pmatrix}\Rightarrow\begin{pmatrix}x\\y\\n\\1\end{pmatrix}==\begin{pmatrix}nx\\ny\\n^2\\n\end{pmatrix}$$

- So the third row must be of the form (0 0 A B)

$$\begin{pmatrix}0&0&A&B\end{pmatrix}\begin{pmatrix}x\\y\\n\\1\end{pmatrix}=n^2$$

n² has nothing to do with x and y

- What do we have now?

$$\begin{pmatrix}0&0&A&B\end{pmatrix}\begin{pmatrix}x\\y\\n\\1\end{pmatrix}=n^2\quad\Rightarrow\quad An+B=n^2$$

- Any point's z on the far plane will not change

$$\begin{pmatrix}0\\0\\f\\1\end{pmatrix}\Rightarrow\begin{pmatrix}0\\0\\f\\1\end{pmatrix}==\begin{pmatrix}0\\0\\f^2\\f\end{pmatrix}\quad\Rightarrow\quad Af+B=f^2$$

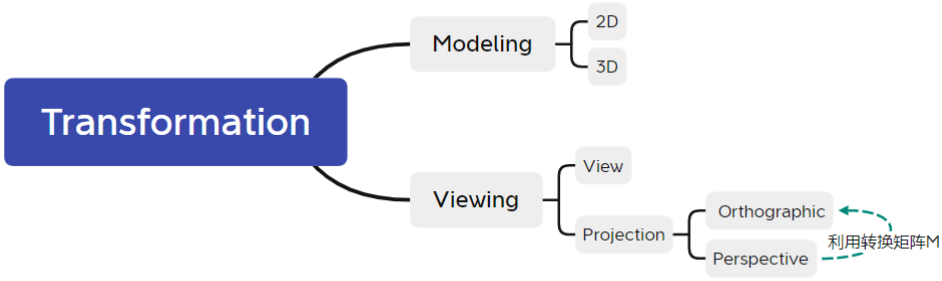
- Solve for A and B

$$\begin{aligned}An+B&=n^2\\Af+B&=f^2\end{aligned}\quad\Rightarrow\quad\begin{aligned}A&=n+f\\B&=-nf\end{aligned}$$

- Finally, every entry in $M_{persp->ortho}$ is known!

- What's next?
  - Do orthographic projection (M_{ortho}) to finish
  - $M_{persp}=M_{ortho}M_{persp\to ortho}$

Transformation总结思维导图:



Lecture 5 : Rasterization 1 (Triangles)

三角形的光栅化

- Finishing up viewing

  -- Viewport transformation

- **Rasterization**

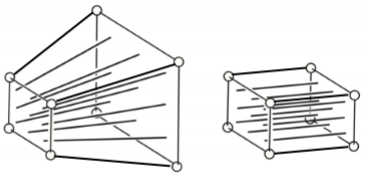
  -- Different raster displays

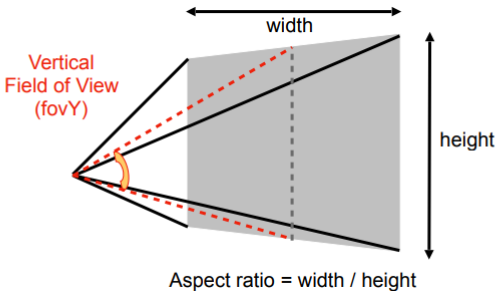
  -- Rasterizing a triangle

- Occlusions and Visibility

定义frustum(视锥)的两个概念:

- What's near plane's l, r, b, t then?
  - If explicitly specified, good
  - Sometimes people prefer: vertical **field-of-view** (fovY) and **aspect ratio** (assume symmetry i.e. l = -r, b = -t)

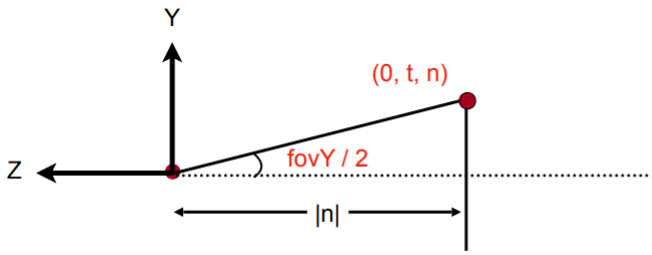




- **aspect ratio** 宽高比
- **field-of-view (fovY)** 视场角

概念的相互转化:

- How to convert from fovY and aspect to l, r, b, t?
  - Trivial



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

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

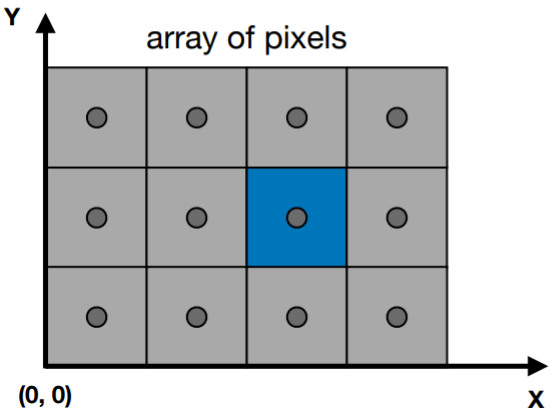
定义一个视锥,只需要定义其宽高比和视场角。

投影变换得到了[-1,1]的立方体,怎么画到屏幕上?--光栅化

What is screen?

- An array of pixels
- Size of the array: resolution 像素的多少--分辨率
- A typical kind of raster display 光栅成像设备

- 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)

Viewport transformation 视口变换:

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

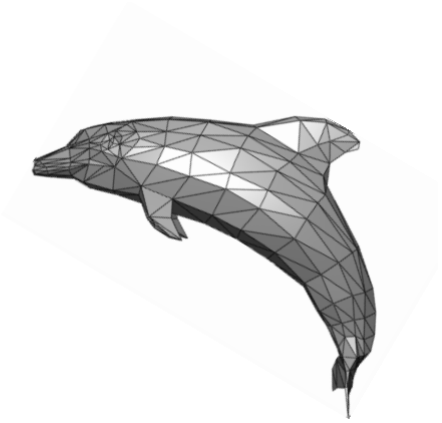
$$M_{viewport} = \begin{pmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} \\ 0 & \frac{height}{2} & 0 & \frac{height}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Next: Rasterizing Triangles into Pixels.

Triangles - Fundamental Shape Primitives

Why triangles?

- Most basic polygon
  - Break up other polygons
- Unique properties
  - Guaranteed to be planar
  - Well-defined interior
  - Well-defined method for interpolating values at vertices over triangle (barycentric interpolation)



- **A Simple Approach: Sampling(采样)**

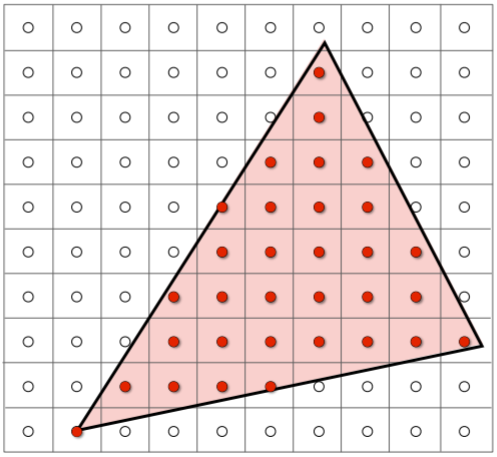
Evaluating a function at a point is sampling.

We can discretize a function by sampling. 离散化一个函数

```
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) …

Sample If Each Pixel Center Is Inside Triangle



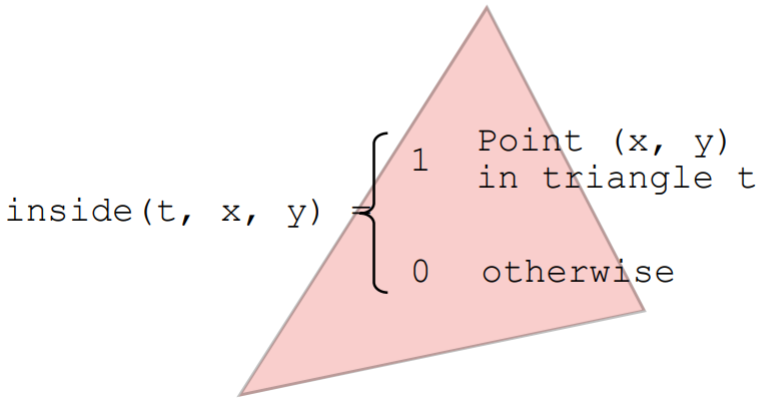
对三角形采样:判断像素中心是否在三角形内

Rasterization = Sampling A 2D Indicator Function

```
for (int x = 0; x < xmax; ++x)
  for (int y = 0; y < ymax; ++y)
    image[x][y] = inside(tri,
                         x + 0.5,
                         y + 0.5);
```

Define Binary Function: `inside(tri, x, y)`

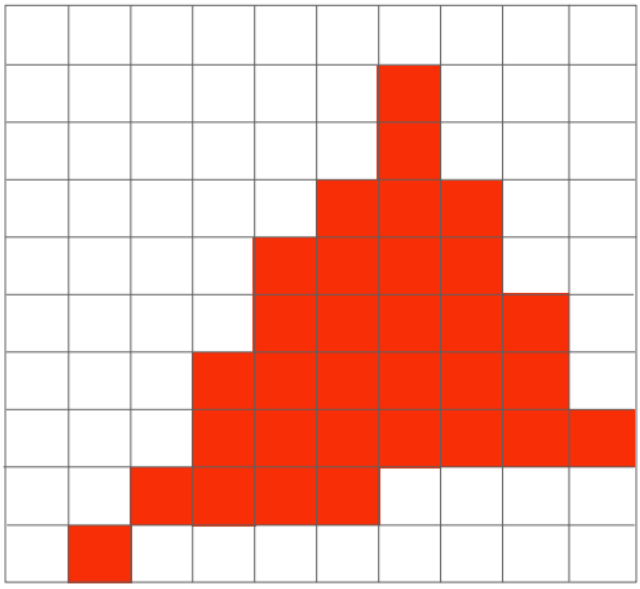
x, y: not necessarily integers



如何判断这个点是否在三角形内?--向量的叉乘

(Edge cases -- 边界自定义,本课不做处理)

What's Wrong With This Picture?



Jaggies!

问题:Aliasing(Jaggies) 走样(锯齿) -- 信号的采样率不够高

Lecture 6 : Rasterization 2 (Antialiasing and Z-Buffering)

- **Antialiasing 反走样(抗锯齿)**
- Z-Buffering 深度缓冲

Today:

- **Antialiasing**

  -- Sampling theory

  -- Antialiasing in practice

- Visibility / occlusion

  -- Z-buffering

Artifacts(瑕疵) due to sampling - “Aliasing”

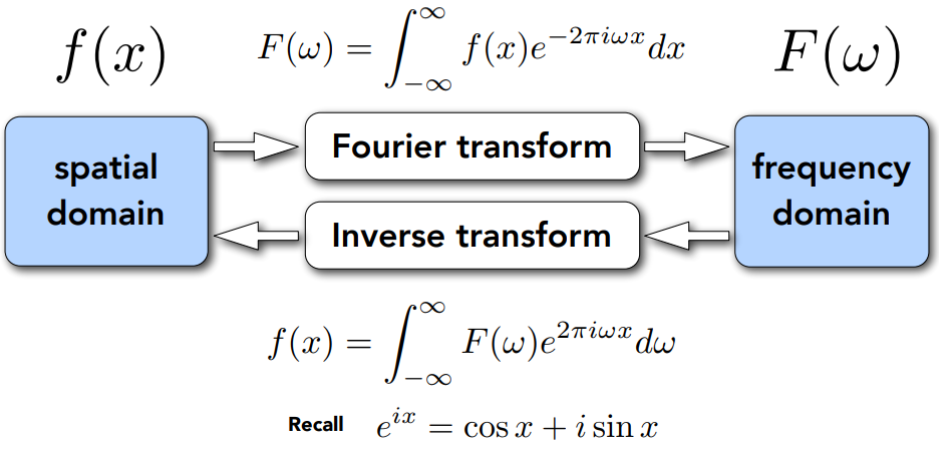
- Jaggies - 锯齿
- Moire - 摩尔纹
- Wagon wheel effect - 车轮效应
- ……

本质:信号变化速度过快,而采样的频率太低

先对信号进行模糊(blurring),滤波(pre-filter),再采样,可以减少锯齿化。(不能先采样再滤波)

Frequency Domain频域

Fourier Transform Decomposes A Signal Into Frequencies



Recall $e^{ix} = \cos x + i \sin x$

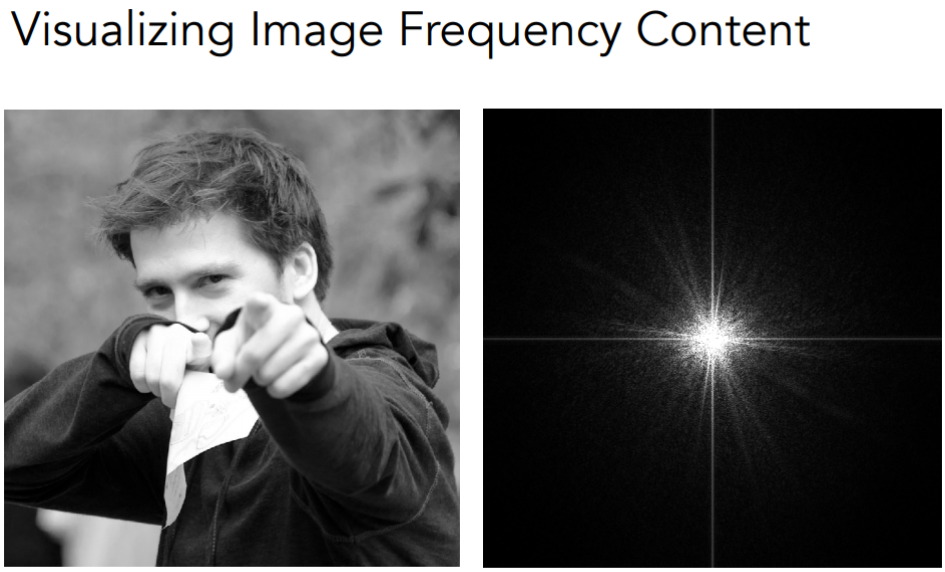
把一个函数分解为不同频率的段，并显示出来。

滤波:

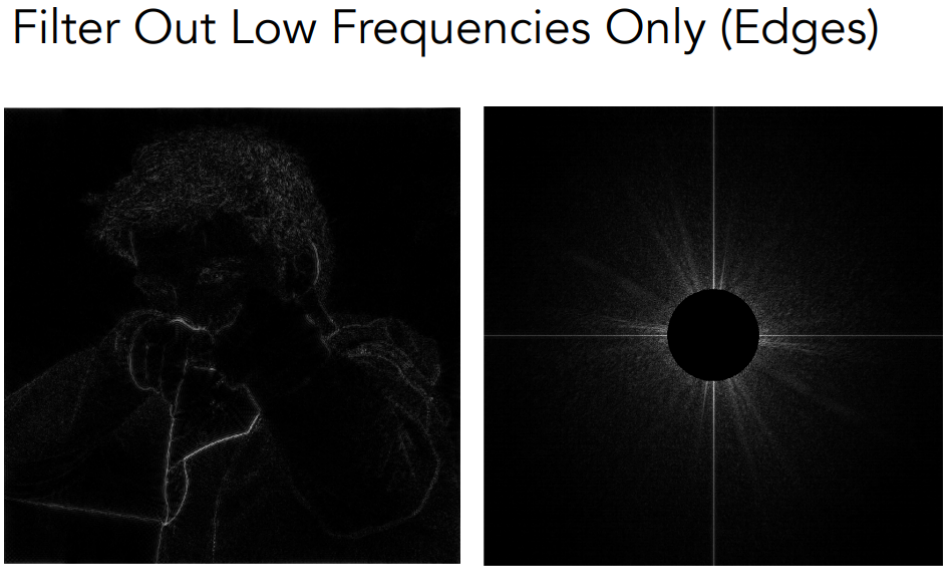
Filtering = getting rid of certain frequency contents 去掉一些特定的频率

傅里叶变换: 时域->频域

①正常图片:



②高通滤波:



High-pass filter

③低通滤波:

Filter Out High Frequencies (Blur)

Low-pass filter

卷积定理:时域卷积=频域乘积

Convolution Theorem

Convolution in the spatial domain is equal to multiplication in the frequency domain, and vice versa

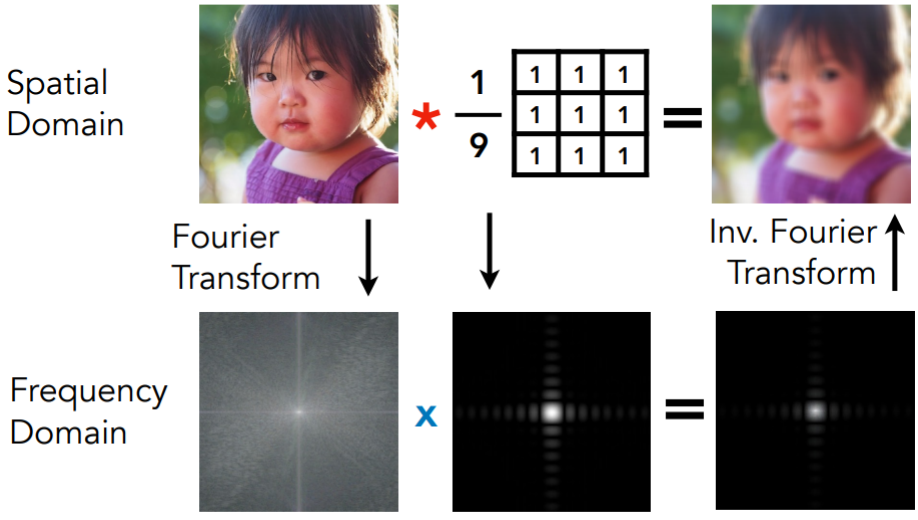
Option 1:

- Filter by convolution in the spatial domain

Option 2:

- Transform to frequency domain (Fourier transform)
- Multiply by Fourier transform of convolution kernel
- Transform back to spatial domain (inverse Fourier)

Convolution Theorem

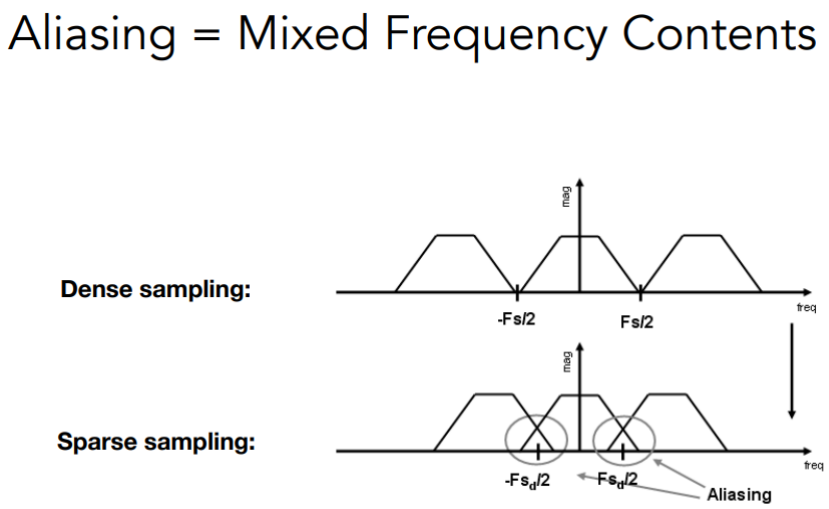


卷积核(滤波器)越大,频率越低,图像越模糊。

Sampling = Repeating Frequency Contents

采样=重复频率内容

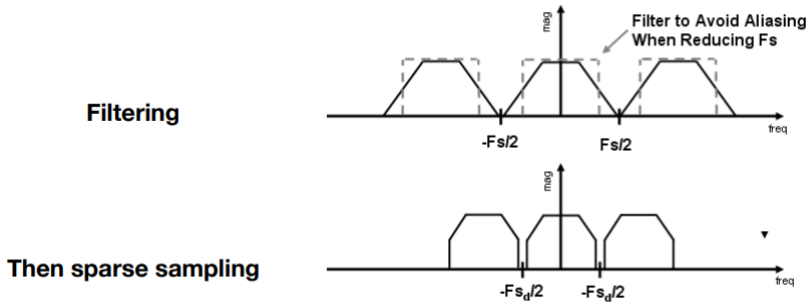
Aliasing = Mixed Frequency Contents



Antialiasing 反走样:

1、*Increase sampling rate 换个显示器

2、先做模糊(低通滤波，过滤高频信息)再做采样



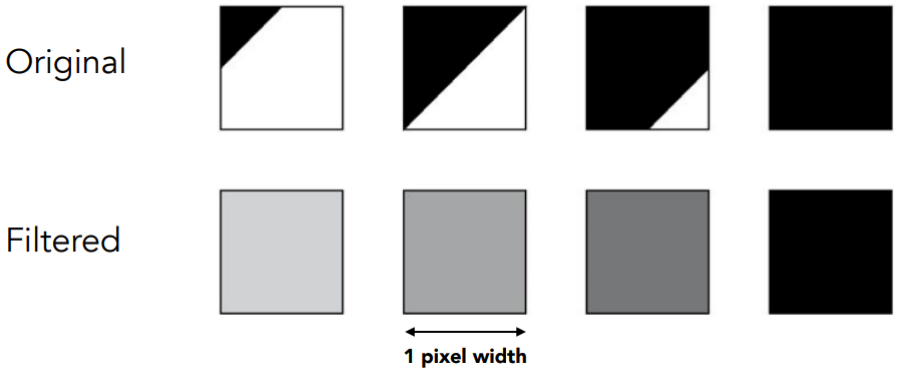
减小频谱的覆盖范围

怎么变模糊?

寻找一个一定大小的低通滤波器进行卷积操作

Antialiasing by Computing Average Pixel Value

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.



怎么算出一个像素被覆盖的面积?

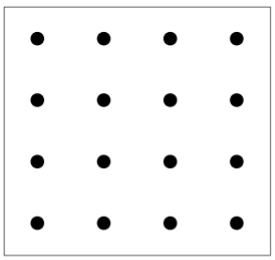
Antialiasing By Supersampling(MSAA)

MSAA = Multi Sample Antialiasing

是反走样的近似，不能完全解决走样

Supersampling

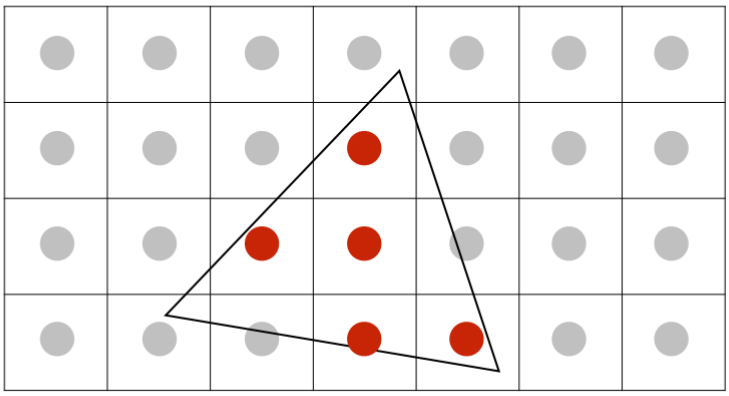
Approximate the effect of the 1-pixel box filter by sampling multiple locations within a pixel and averaging their values:



4x4 supersampling

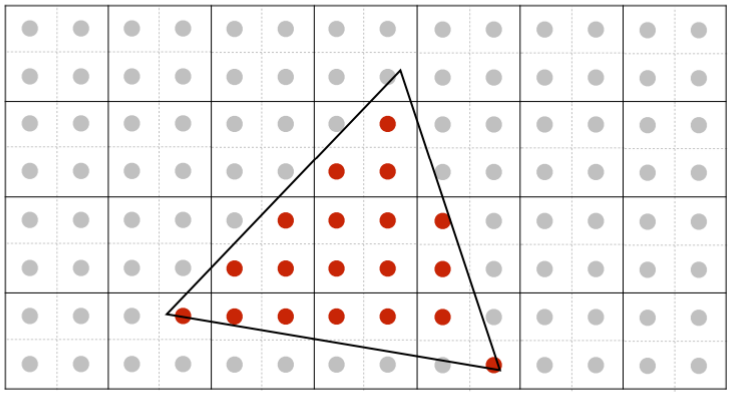
把一个像素再分成小像素，判断是否在三角形内

Point Sampling: One Sample Per Pixel



Supersampling: Step 1

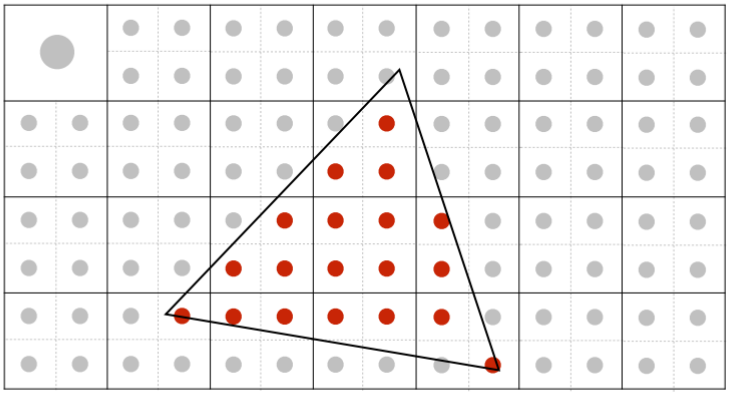
Take NxN samples in each pixel.



2x2 supersampling

Supersampling: Step 2

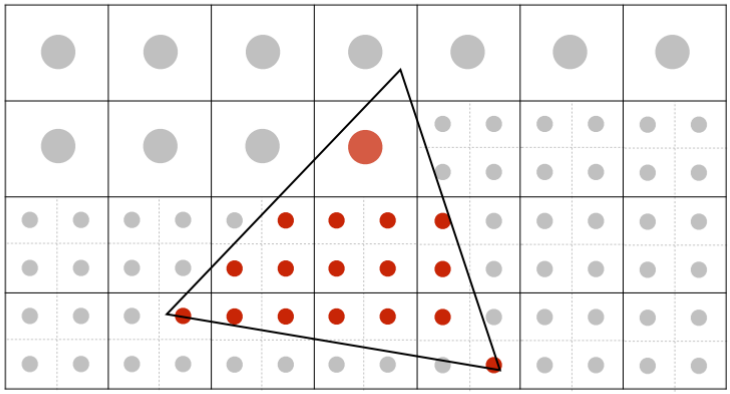
Average the NxN samples “inside” each pixel.



Averaging down

Supersampling: Step 2

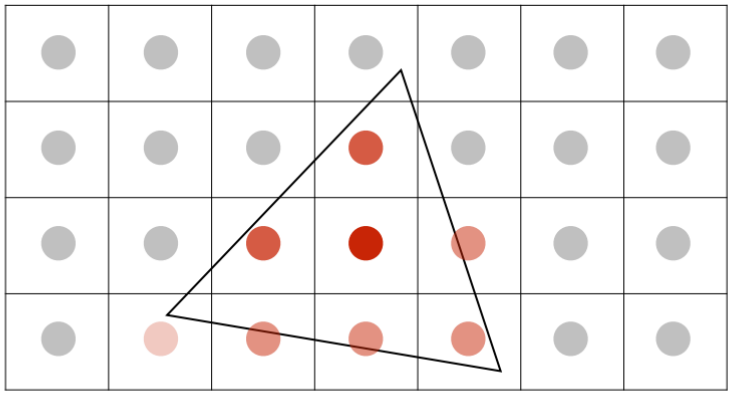
Average the NxN samples “inside” each pixel.



Averaging down

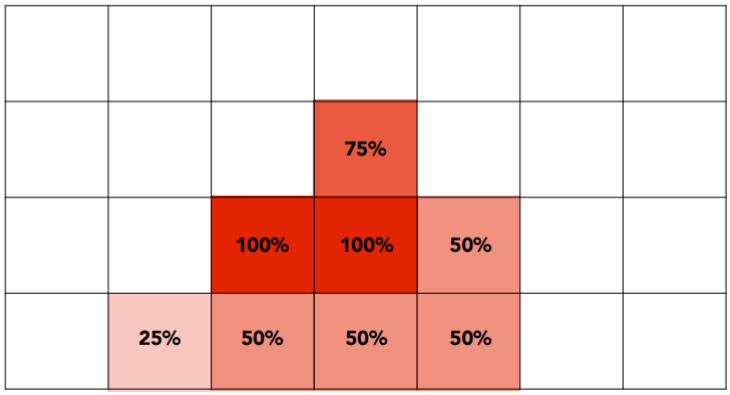
Supersampling: Step 2

Average the NxN samples “inside” each pixel.



Supersampling: Result

This is the corresponding signal emitted by the display



- 至此，模糊操作做完了，再进行采样即可(还是这个结果)
- MSAA进行的是模糊操作(采样操作已经隐含在里面了)
- 并没有提高采样率，只是对三角形的覆盖面积进行近似

但是!No free lunch! What's the cost of MSAA?

- 放置了更多的点，增加了计算量。

Milestones(personal idea):

- FXAA(Fast Approximate AA) 快速近似抗锯齿(后期图像处理抗锯齿，速度很快，效果也好)
- TAA(Temporal AA) 复用上一帧的图像(同一像素的不同位置)，即将没有MSAA的图像分布在时间上

Super resolution / super samling:

超分辨率≠抗锯齿，但本质相似

- From low resolution to high resolution
- Essentially still "not enough samples" problem 图片虽然分辨率高，但是采样不足
- DLSS(Deep Learning Super Sampling) 图片拉大后信息缺失，用深度学习进行"猜"

Lecture 7 : Shading 1 (Illumination, Shading and Graphics Pipeline)

- Visibility / occlusion

  -- Z-buffering 深度缓冲

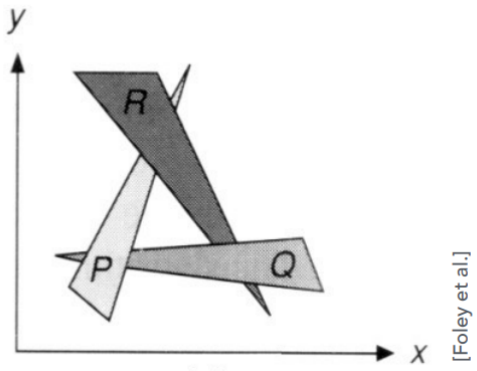
- Shading

  -- Illumination & Shading

  -- Graphics Pipeline

Painter's Algorithm 画家算法:先画远处的，进行光栅化，再画近处的进行覆盖。

特殊情况:存在互相遮挡的情况，无法定义深度关系，也无法使用画家算法。



由此，引入Z-Buffer.

Z-Buffer

This is the algorithm that eventually won.

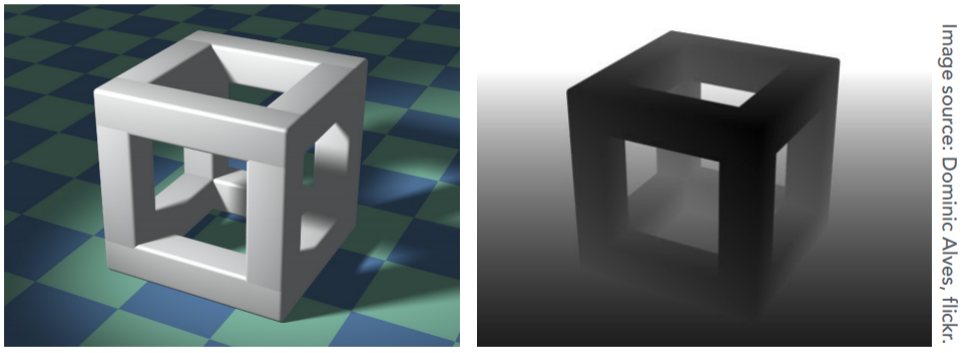
Idea:

- 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)

Z小则近，大则远。与Transform里面的不同(为了简化)

Z-Buffer Example



Rendering　　Depth / Z buffer

同步生成两幅图 color(FrameBuffer) & depth(Z-Buffer)

Z-Buffer Algorithm

Initialize depth buffer to ∞

During rasterization:

```
for (each triangle T)
    for (each sample (x,y,z) in T)
        if (z < zbuffer[x,y])                  // closest sample so far
            framebuffer[x,y] = rgb;            // update color
            zbuffer[x,y] = z;                  // update depth
        else
            ;                          // do nothing, this sample is occluded
```

注意:Z-Buffer 无法处理透明物体

Shading 着色:

Shading:The process of applying a material to an object.

(不同材质:木头、铅球等不同材质与光源有不同的互动)

A Simple Shading Model(**Blinn-Phong** Reflectance Model)

- 高光
- 漫反射
- 环境光

Perceptual Observations

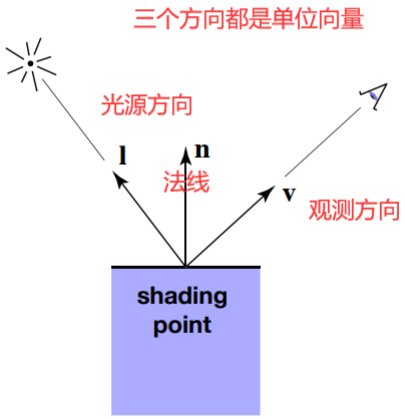
Photo credit: Jessica Andrews, flickr

Shading is Local

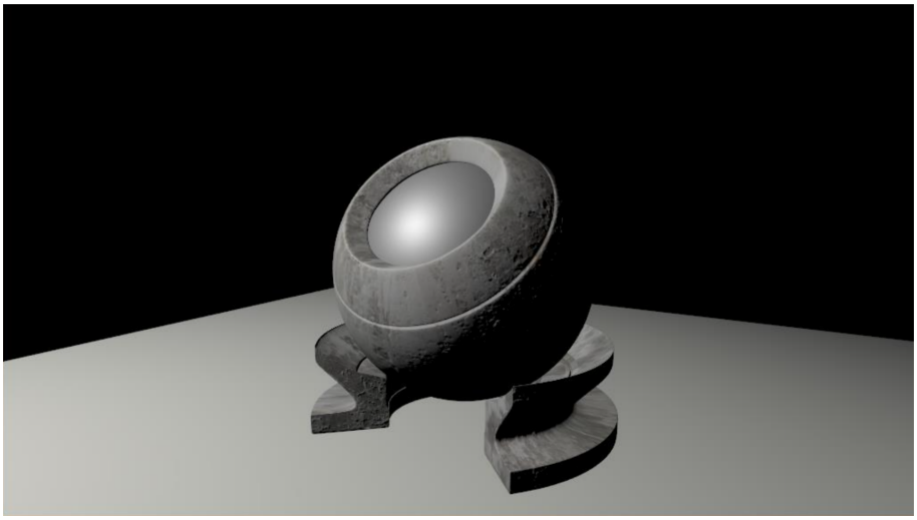
Compute light reflected toward camera at a specific shading point

Inputs:

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



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

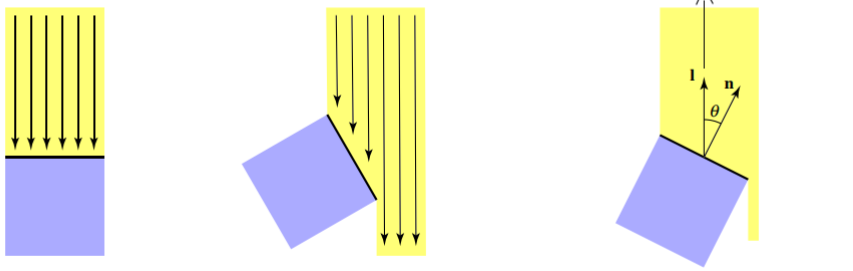


只考虑Shading point 自己，不考虑其他情况，即只着色，但没有阴影。 shading ≠ shadow!

Lambert's cosine law :

Diffuse Reflection

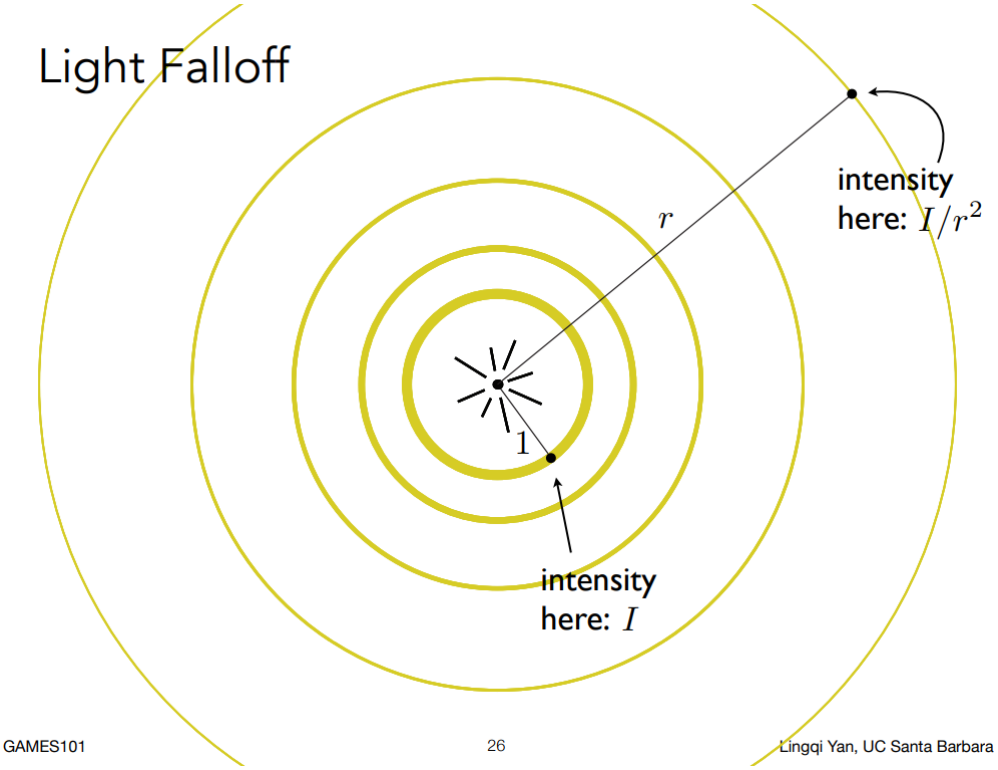
- But how much light (energy) is received?
  - Lambert's cosine law



Top face of cube receives a certain amount of light

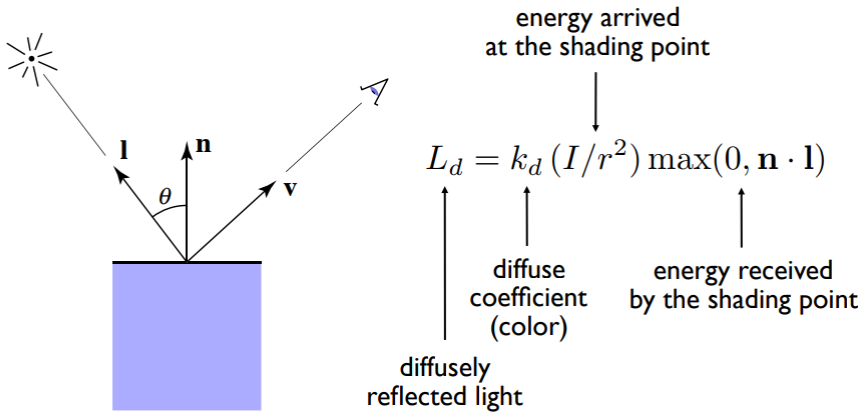
Top face of 60° rotated cube intercepts half the light

In general, light per unit area is proportional to cos θ = l • n



Lambertian (Diffuse) Shading

Shading independent of view direction



漫反射和观测方向没有关系(因此公式里没有v)

L_d:漫反射的光强

k_d:漫反射系数(Shading point 的颜色，决定吸收了哪些光，黑色为0(全吸收了)，白色为1)

Lecture 8 : Shading 2 (Shading, Pipeline and Texture Mapping)

- Blinn-Phong reflectance model

  -- Specular and ambient terms

- Shading frequencies
- Graphics pipeline

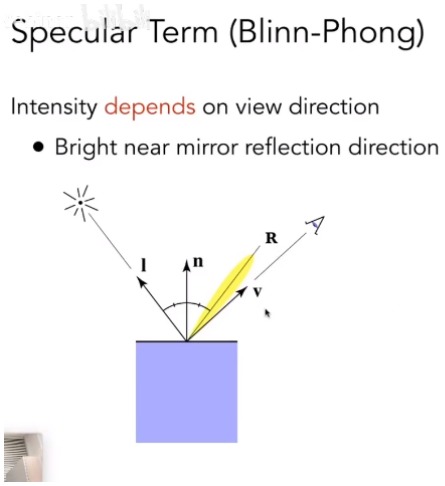
Blinn-Phong reflectance model:

1. Specular
2. Diffuse // 上节课讲过，今天讲另外两个
3. Ambient

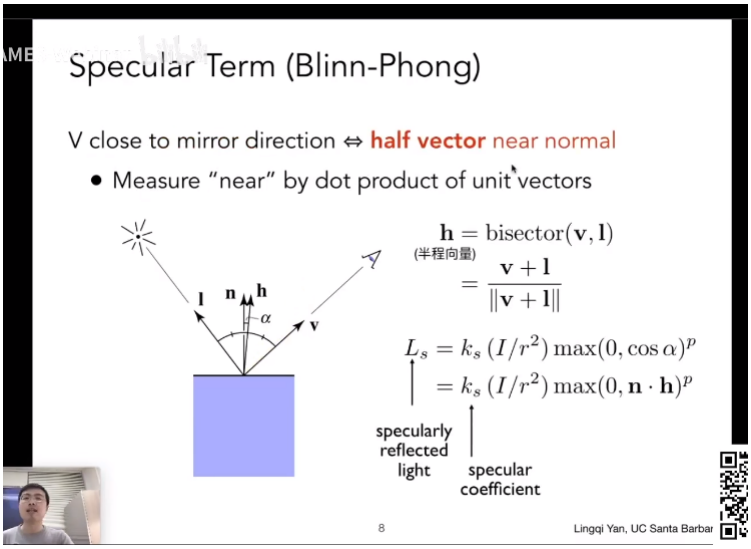
Specular terms:

物体表面比较光滑，光线接近镜面反射

观察方向和镜面反射方向接近(v和R足够接近)



引入半程向量(bisector)h:



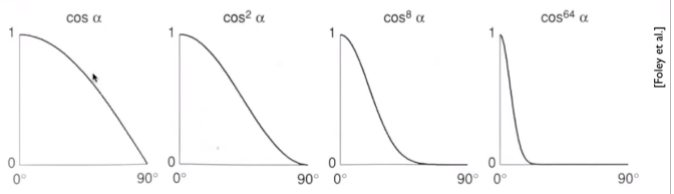
v和R接近 就是 n和h接近

- Phong模型：用v和R的接近程度
- Blinn-Phong模型：用n和h的接近程度(改进)，半程向量比较好算

指数p:减少容忍度，用来控制高光的大小，通常取100~200

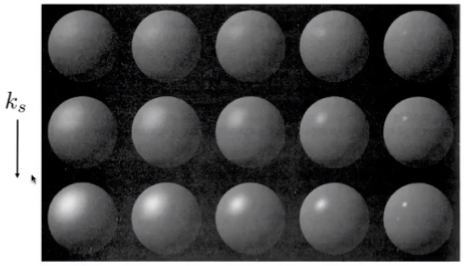
Cosine Power Plots

Increasing p narrows the reflection lobe



Specular Term (Blinn-Phong)

Blinn-Phong $L_s = k_s\,(I/r^2)\max(0, \mathbf{n}\cdot\mathbf{h})^p$



Note: showing
Ld + Ls together

10

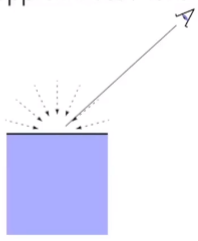
Lingqi Yan, UC Santa Barbar

Ambient Term:

Ambient Term

Shading that does not depend on anything

- Add constant color to account for disregarded illumination and fill in black shadows
- This is approximate / fake!



$L_a = k_a\,I_a$

ambient coefficient

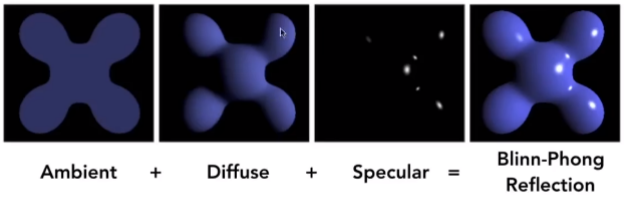
reflected ambient light

11

Lingqi Yan, UC Santa Barbar

环境光：与光源方向、观测方向、法线都无关，是一个常数(某种颜色)。保证没有一个地方完全是黑的

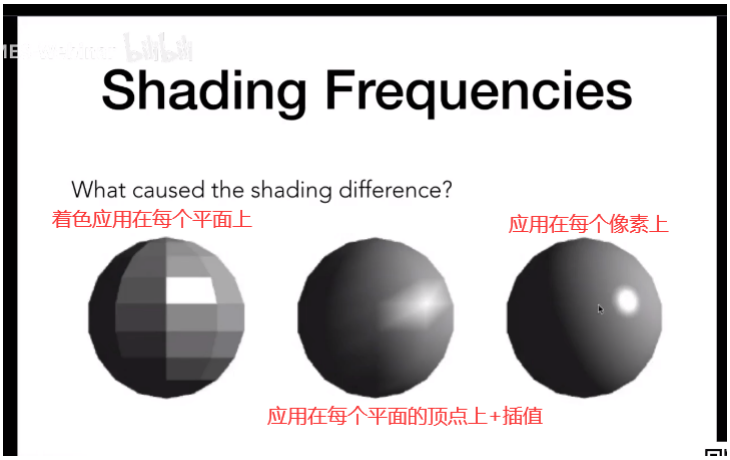
Blinn-Phong Reflection Model



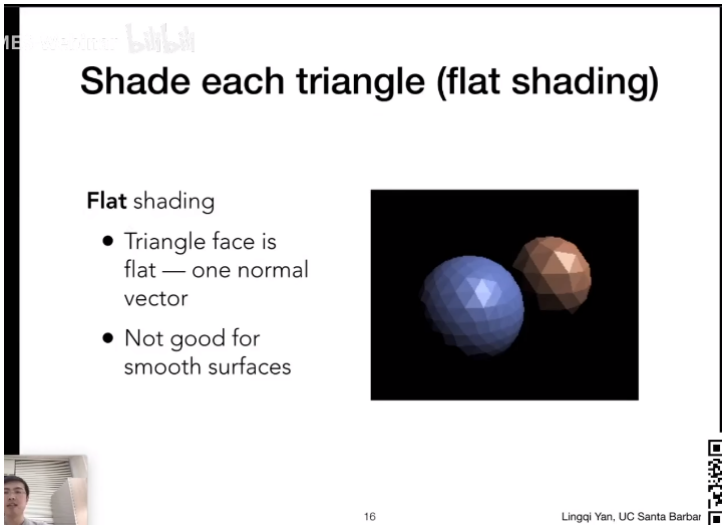
$$L = L_a + L_d + L_s$$
$$= k_a\, I_a + k_d\, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s\, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

着色过程完成!

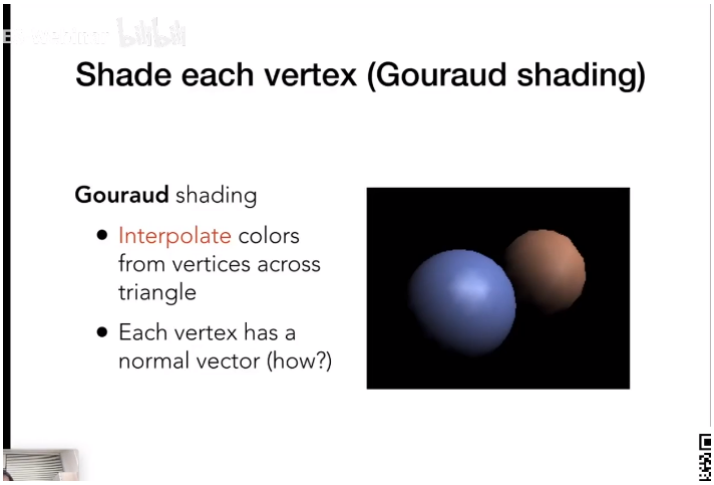
Shading Frequencies:



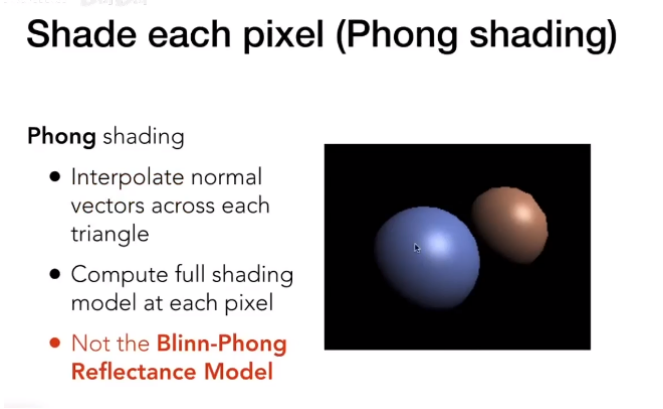
1、Flat shading 逐三角形

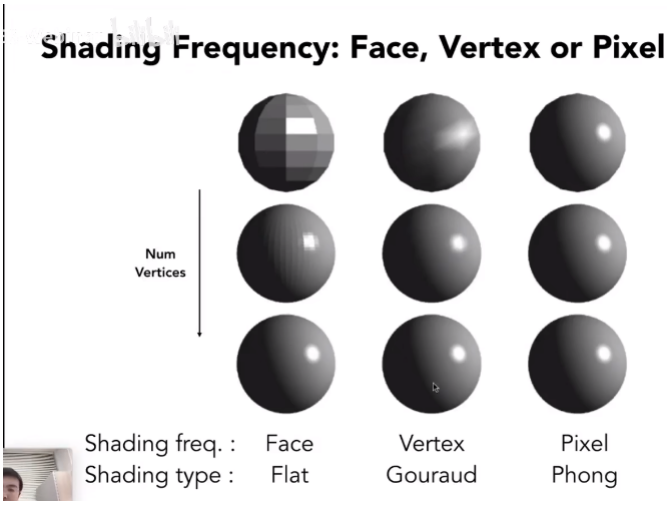


2、Gouraud shading 高洛德 逐顶点

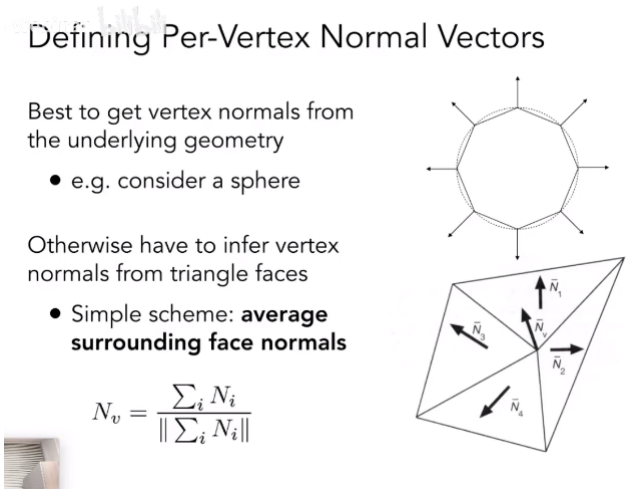


3、Phong shading 逐像素





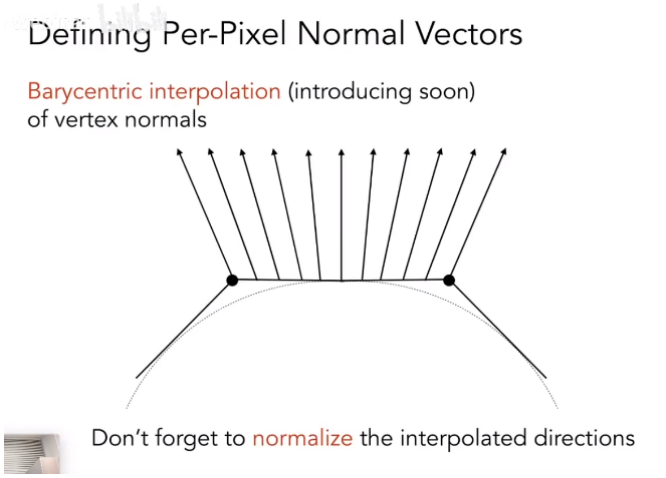
定义逐顶点的法线：



与这个点相关联的三角形的法线的简单平均/加权平均。

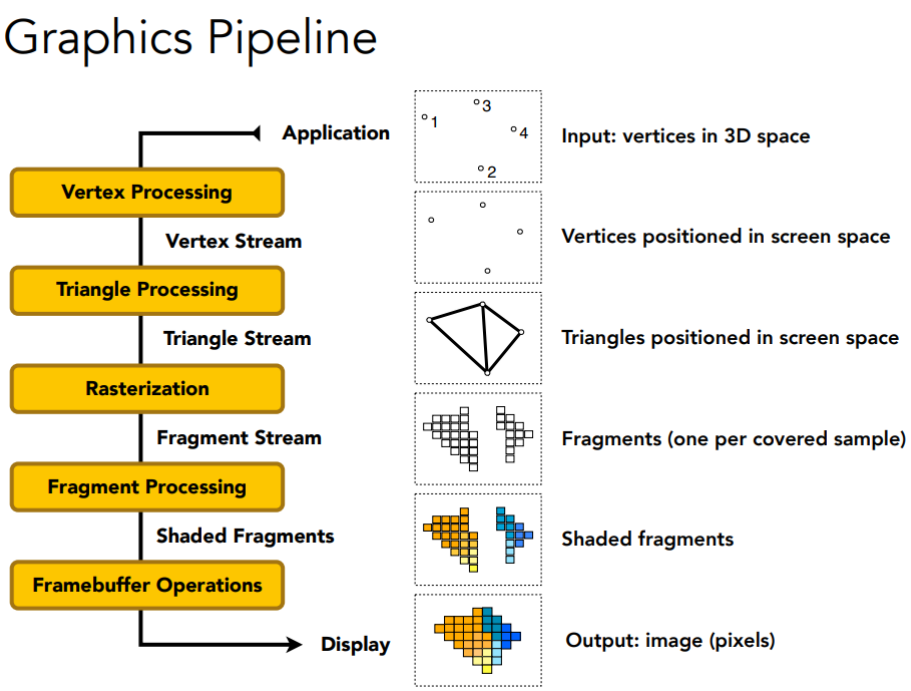
加权平均要更准确

定义逐像素的法线：



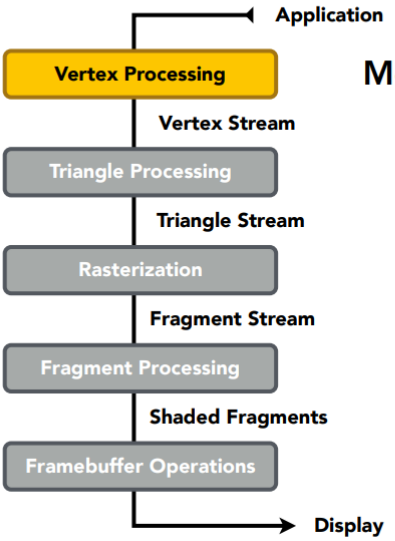
重心插值

Graphics(Real-time Rendering) Pipeline:

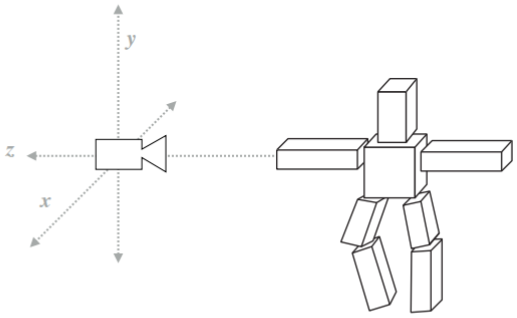


图形管线(一系列的操作):三维场景→二维图片

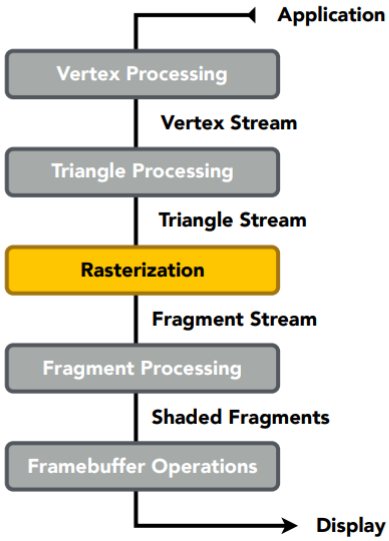
Graphics Pipeline



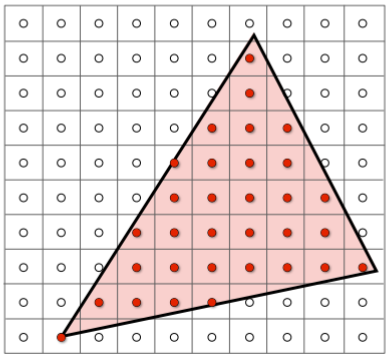
Model, View, Projection transforms



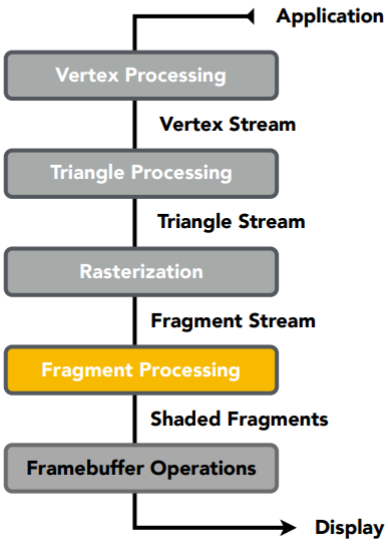
Graphics Pipeline



Sampling triangle coverage

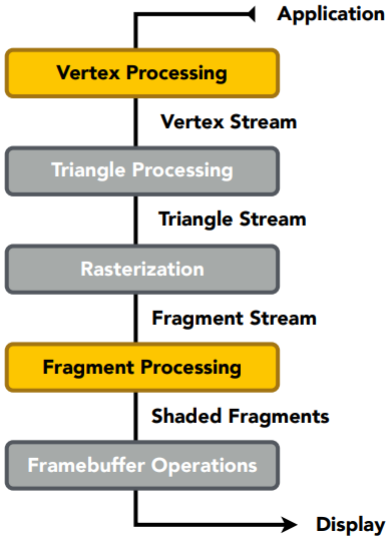


Rasterization Pipeline

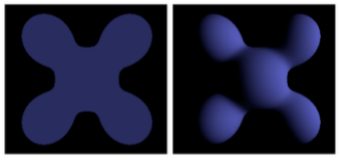


Z-Buffer Visibility Tests

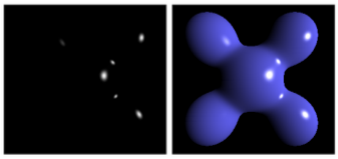
Graphics Pipeline



Shading

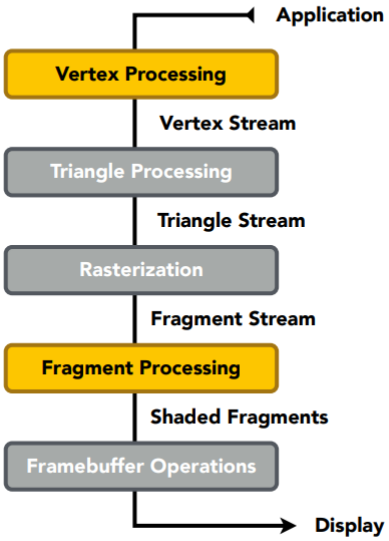


Ambient + Diffuse



+ Specular = Blinn-Phong Reflectance Model

Graphics Pipeline



Texture mapping
(introducing soon)

Shader着色器(OpenGL):

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.**

Shader Programs

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

Example GLSL fragment shader program

```
uniform sampler2D myTexture;      // program parameter
uniform vec3 lightDir;            // program parameter
varying vec2 uv;                  // per fragment value (interp. by rasterizer)
varying vec3 norm;                // per fragment value (interp. by rasterizer)

void diffuseShader()
{
  vec3 kd;
  kd = texture2d(myTexture, uv);                  // material color from texture
  kd *= clamp(dot(–lightDir, norm), 0.0, 1.0);    // Lambertian shading model
  gl_FragColor = vec4(kd, 1.0);                   // output fragment color
}
```

Shader体验网站:https://www.shadertoy.com/view/ld3Gz2

现代图形学发展:

Goal: Highly Complex 3D Scenes in Realtime

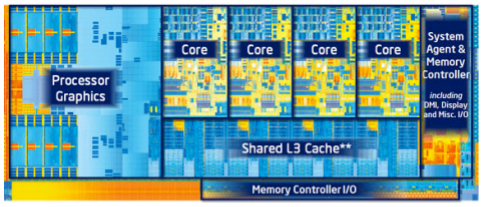
Unreal Engine Kite Demo (Epic Games 2015)

UE4 快速渲染

Graphics Pipeline Implementation: GPUs

Specialized processors for executing graphics pipeline computations

Discrete GPU Card
(NVIDIA GeForce Titan X)

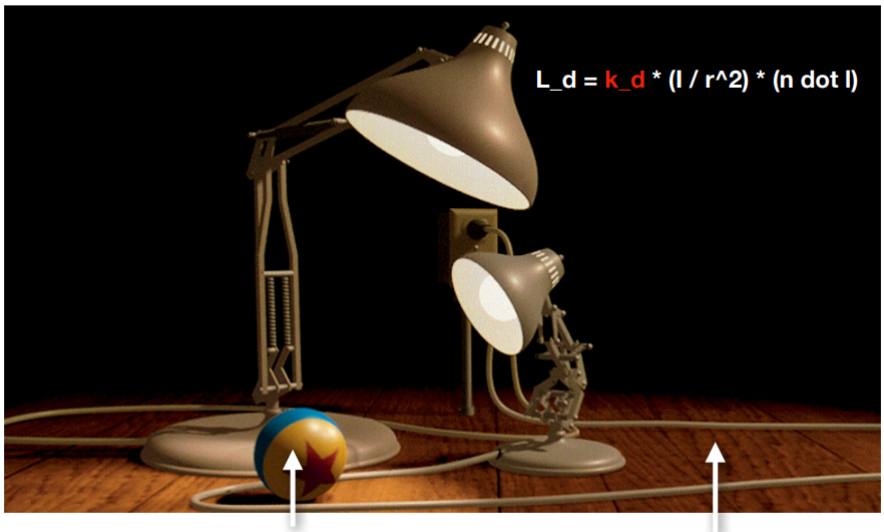


Integrated GPU:
(Part of Intel CPU die)

GPU快速发展

Texture Mapping:纹理映射

Different Colors at Different Places?



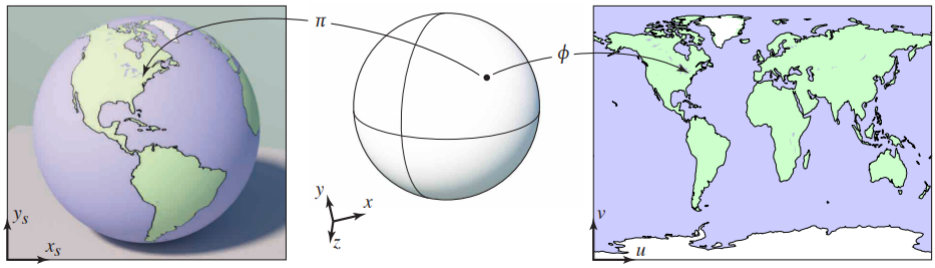
Pattern on ball

Wood grain on floor

Surfaces are 2D

Surface lives in 3D world space

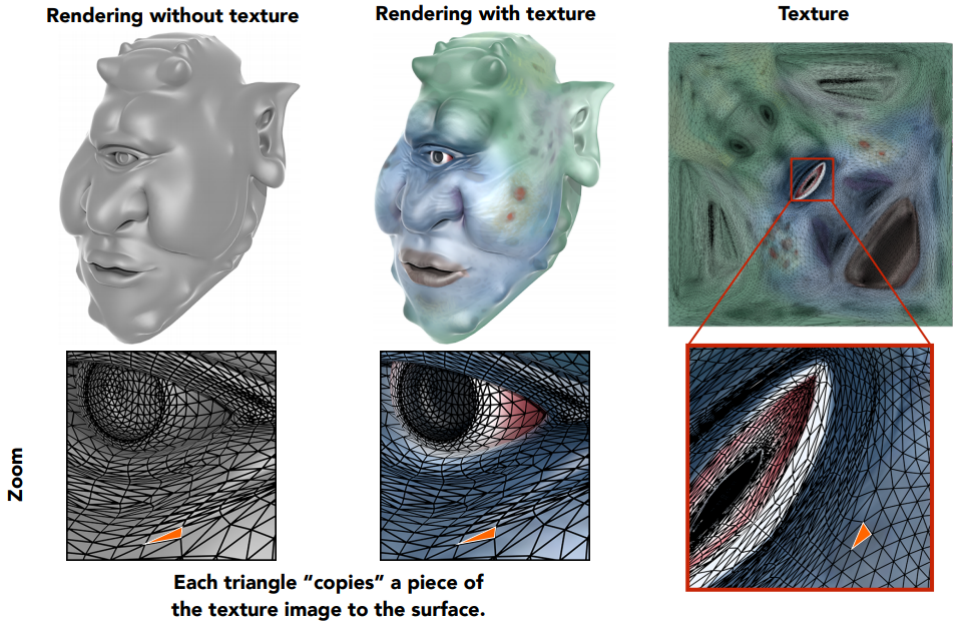
Every 3D surface point also has a place where it goes in the 2D image (**texture**).



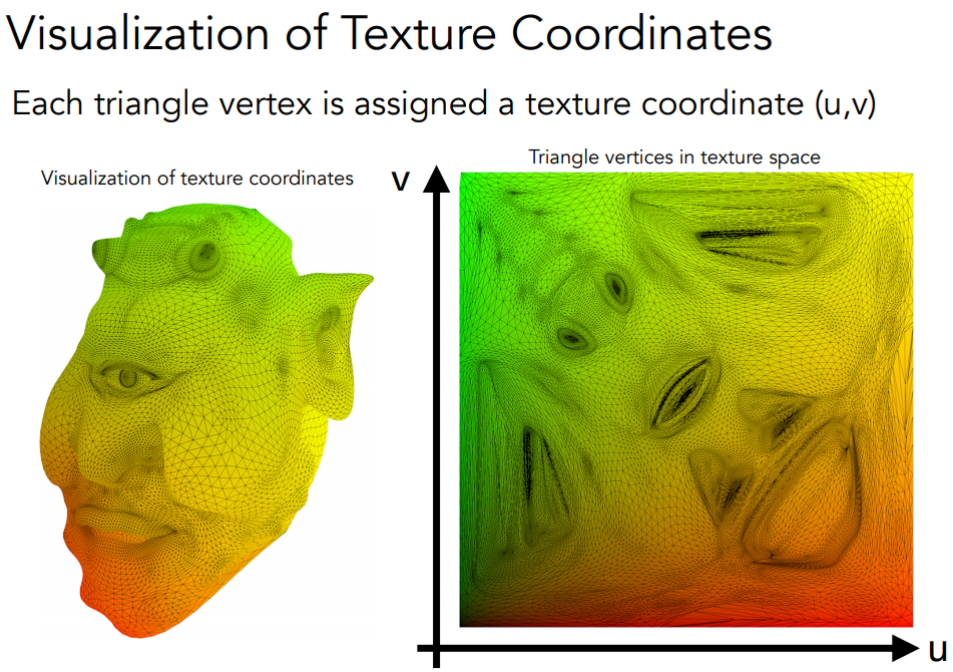
Screen space　　World space　　Texture space

定义：三维物体的表面是二维的，相当于贴图(一张图贴到三维模型上去)

Texture Applied to Surface



纹理坐标：



规定u,v∈[0,1]，三角形每个顶点都对应一个(u,v)

无缝衔接纹理的绘制(难度很大)，复用时看不出缝隙

Lecture 9 : Shading 3 (Texture Mapping cont.)

- Barycentric coordinates 重心坐标(插值)
- Texture queries
- Applications of textures

Interpolation Across Triangles:Bartcentric Coordinates

1、为什么要做插值?

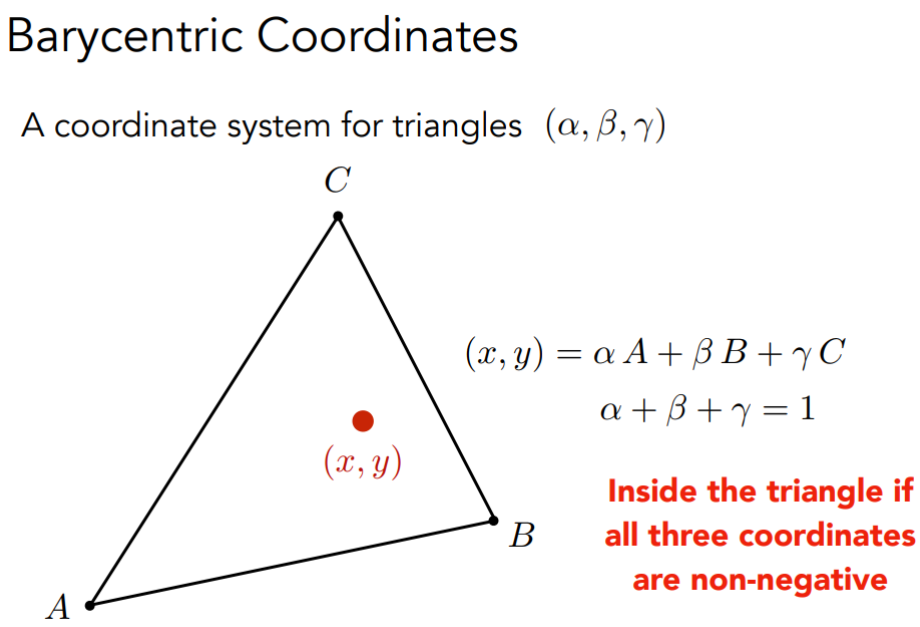
- 利用三角形顶点的值，得到内部的值(平滑过渡)

2、插值要得到什么?

- 纹理坐标，颜色，法线.....

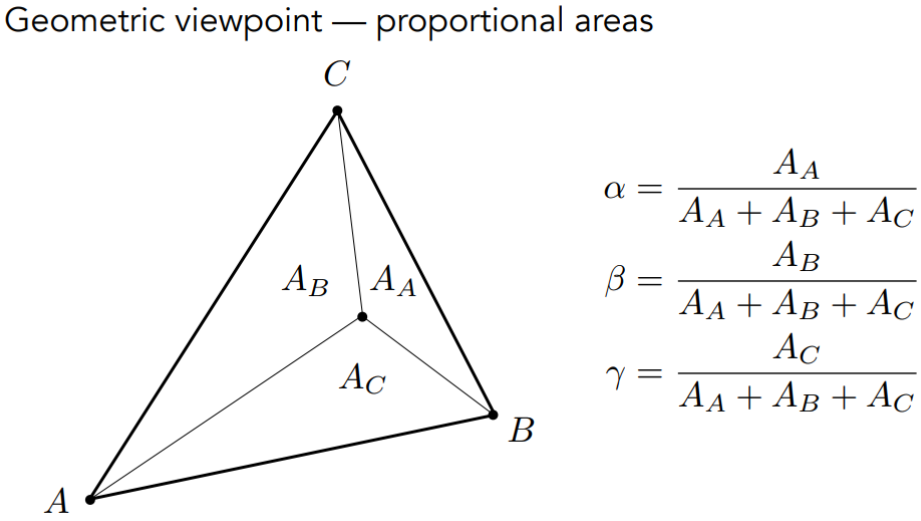
3、怎么做插值?

- 重心坐标

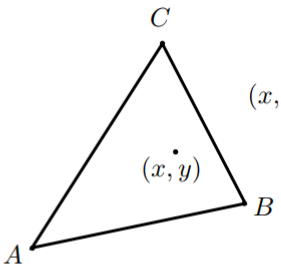


(α,β,γ)均为非负

可以通过面积计算:



Barycentric Coordinates: Formulas



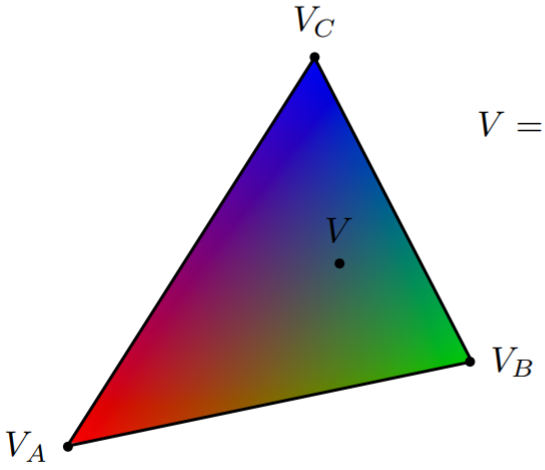
$$(x, y) = \alpha A + \beta B + \gamma C$$
$$\alpha + \beta + \gamma = 1$$

$$\alpha = \frac{-(x - x_B)(y_C - y_B) + (y - y_B)(x_C - x_B)}{-(x_A - x_B)(y_C - y_B) + (y_A - y_B)(x_C - x_B)}$$
$$\beta = \frac{-(x - x_C)(y_A - y_C) + (y - y_C)(x_A - x_C)}{-(x_B - x_C)(y_A - y_C) + (y_B - y_C)(x_A - x_C)}$$
$$\gamma = 1 - \alpha - \beta$$

利用重心坐标进行插值计算：

Using Barycentric Coordinates

Linearly interpolate values at vertices



$$V = \alpha V_A + \beta V_B + \gamma V_C$$

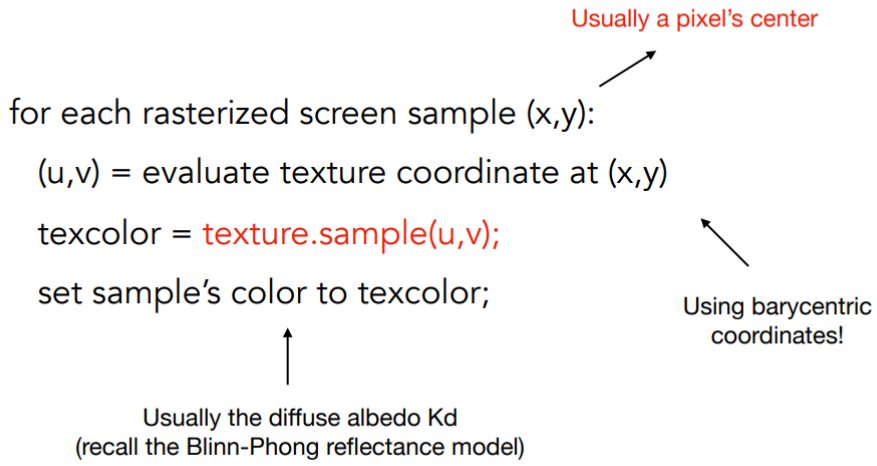
V_A, V_B, V_C can be positions, texture coordinates, color, normal, depth, material attributes…

However, barycentric coordinates are not invariant under projection!

注意：投影之后重心坐标会发生变化!

Applying Textures:

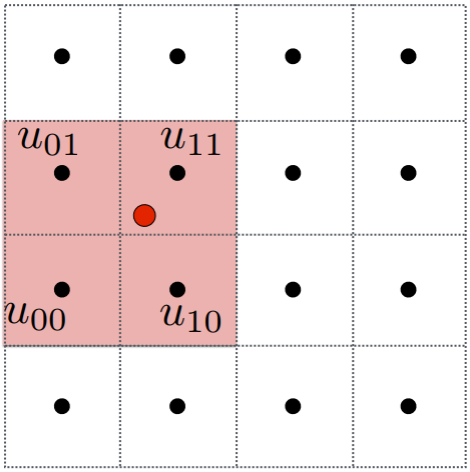
Simple Texture Mapping: Diffuse Color



问题:Texture Magnification(What if the texture is too small?)

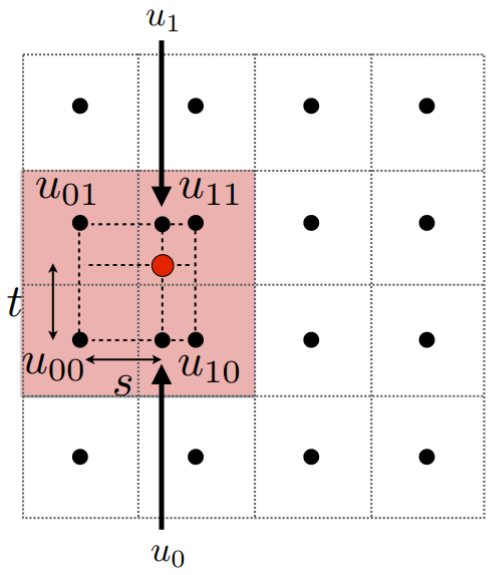
双线性插值：找临近的4个点

Bilinear Interpolation



Take 4 nearest sample locations, with texture values as labeled.

Bilinear Interpolation



Linear interpolation (1D)

$$\text{lerp}(x, v_0, v_1) = v_0 + x(v_1 - v_0)$$

Two helper lerps

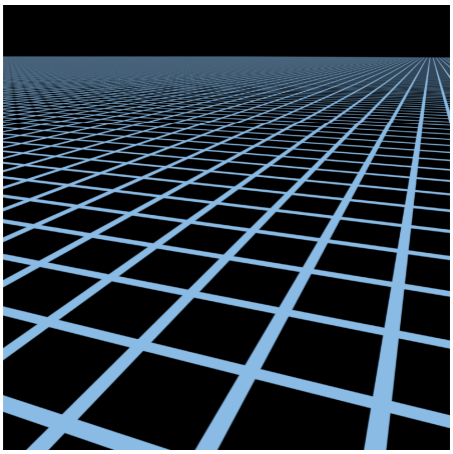
$$u_0 = \text{lerp}(s, u_{00}, u_{10})$$
$$u_1 = \text{lerp}(s, u_{01}, u_{11})$$

Final vertical lerp, to get result:

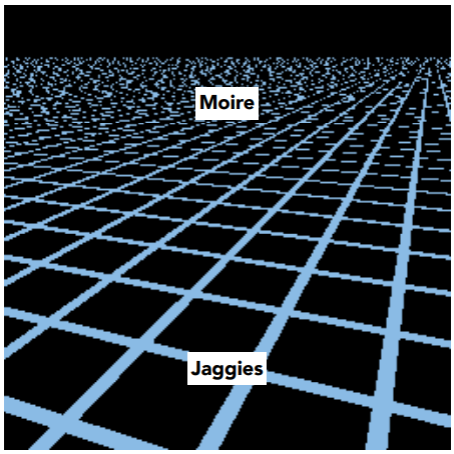
$$f(x, y) = \text{lerp}(t, u_0, u_1)$$

What if the texture is too big?

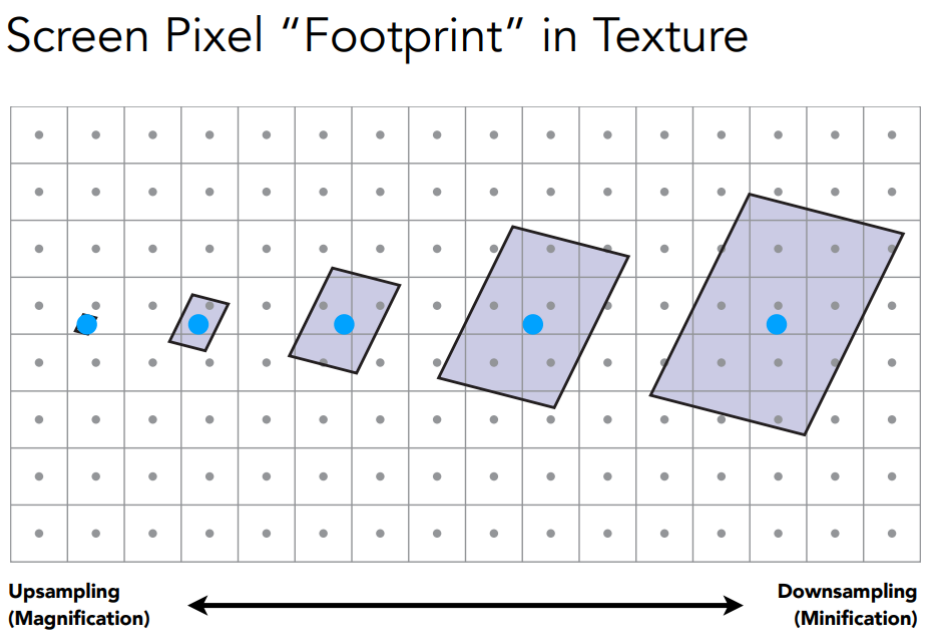
Point Sampling Textures — Problem



Reference

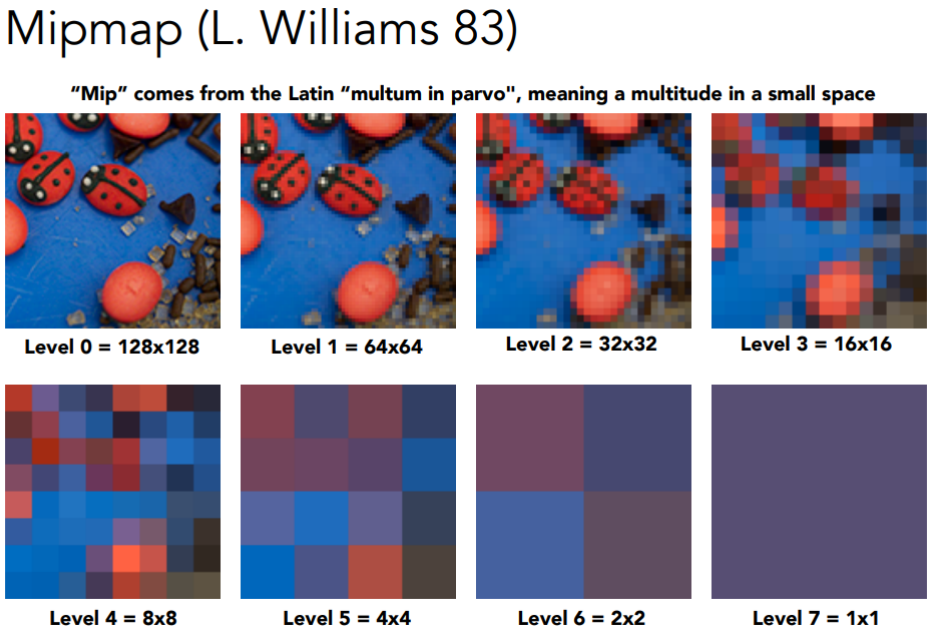


Point sampled

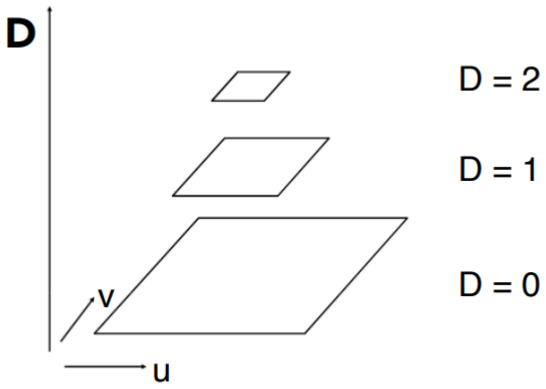


Mipmap (Allowing(fast, approx., square) range queries)

近似的方形范围查询



Mipmap (L. Williams 83)



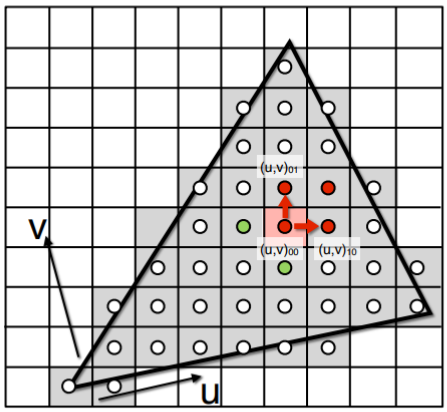
"Mip hierarchy"
level = D
What is the storage overhead of a mipmap?

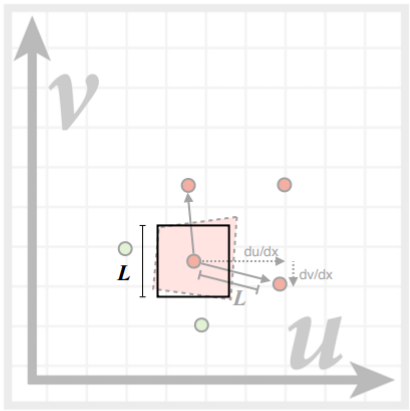
内存多花销了1/3

1 --> 4/3

计算在Mipmap的第几层:

Computing Mipmap Level D





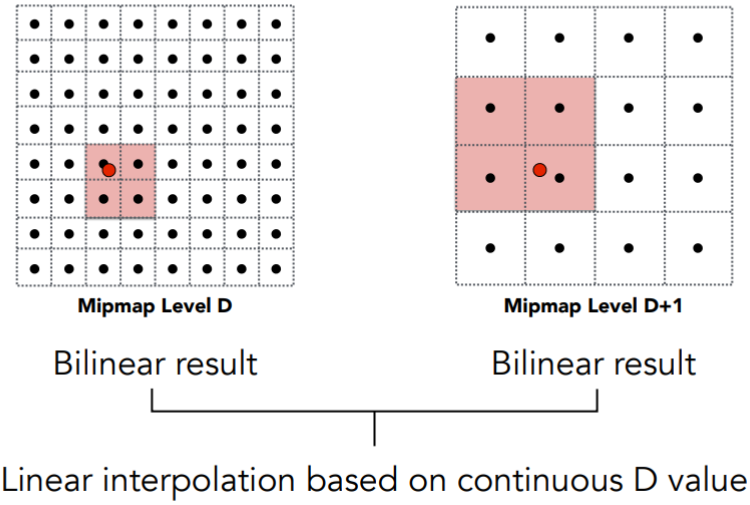
$$D = \log_2 L \qquad L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}\right)$$

离得近，细节充足，在低层查询；

离得远，细节模糊，在高层查询。

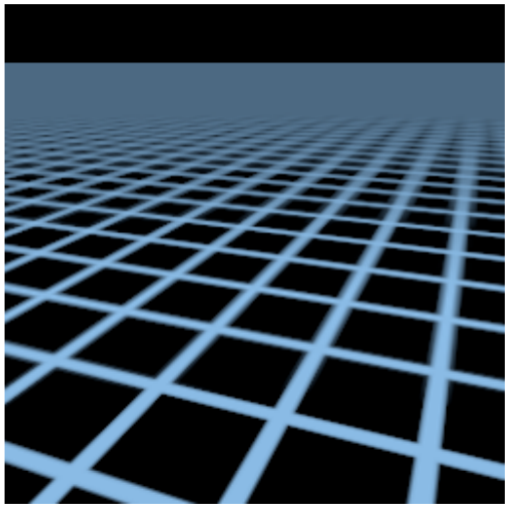
层与层之间可以用三线性插值：

Trilinear Interpolation



Mipmap Limitations

Overblur
Why?

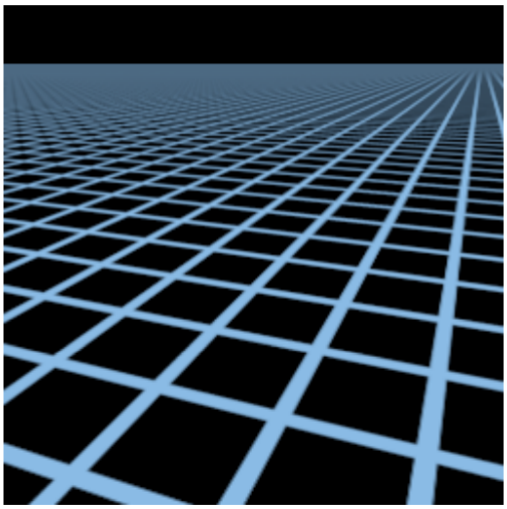


Mipmap trilinear sampling

Mipmap在远处出现了Overblur(Mipmap只能处理正方形，无法处理特殊形状)

Better:各向异性过滤(可以处理矩形，但不能处理特殊形状)

Anisotropic Filtering



Better than Mipmap!

Anisotropic Filtering

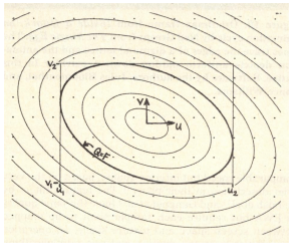
Ripmaps and summed area tables

- Can look up axis-aligned rectangular zones
- Diagonal footprints still a problem

EWA filtering

- Use multiple lookups
- Weighted average
- Mipmap hierarchy still helps
- Can handle irregular footprints

Wikipedia



Greene & Heckbert '86