

Computer Graphics

Class 1

图形学应用之一就是进行可视化的操作

增强现实（）和虚拟现实（）之间的区别：VR 是不可见现实世界的，视野内全部为虚拟环境；AR 是可见现实的，而将一部分的虚拟物品置入了现实中。

Why Study Computer Graphics?

- Fundamental Intellectual Challenges
 - Creates and interacts with realistic virtual world
 - Requires understanding of all aspects of physical world
 - New computing methods, displays, technologies

Why Study Computer Graphics?

- Technical Challenges
 - Math of (perspective) projections, curves, surfaces
 - Physics of lighting and shading
 - Representing / operating shapes in 3D
 - Animation / simulation
 - 3D graphics software programming and hardware

- Course Topics (mainly 4 parts)

- Rasterization
- Curves and Meshes
- Ray Tracing
- Animation / Simulation

---光栅化 (OpenGL, Shader)

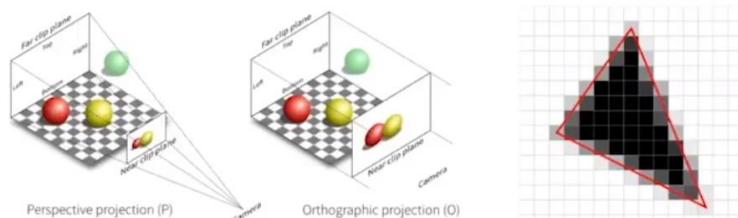
---曲线和曲面

---光追

---动画/仿真

Rasterization

- Project **geometry primitives** (3D triangles / polygons) onto the screen
- Break projected primitives into **fragments** (pixels)
- Gold standard in Video Games (Real-time Applications)

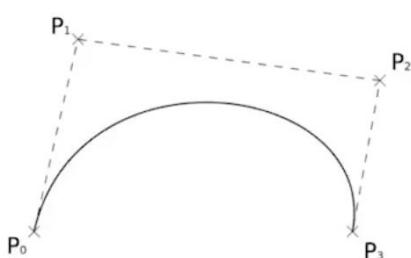


何为光栅化? - 将 3 维或者几何形体显示在屏幕上

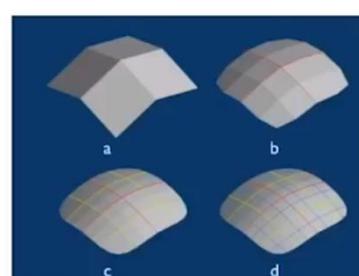
(FPS>30 称为实时, 否则成为离线)

Curves and Meshes

- How to represent geometry in Computer Graphics



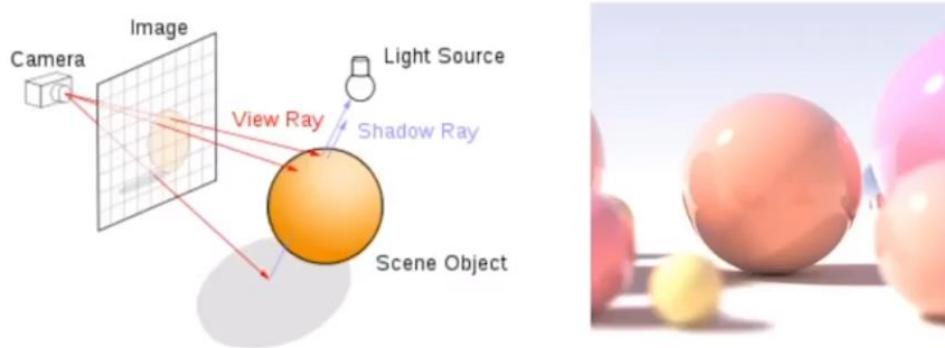
Bezier Curve



Catmull-Clark subdivision

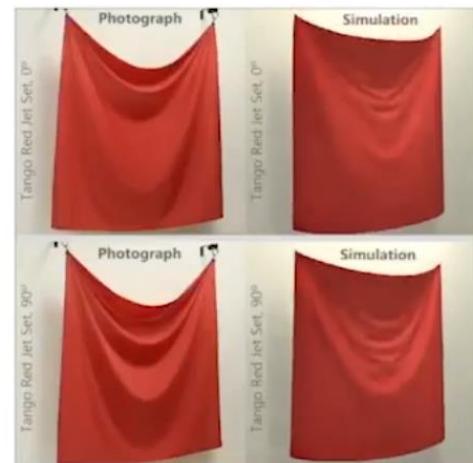
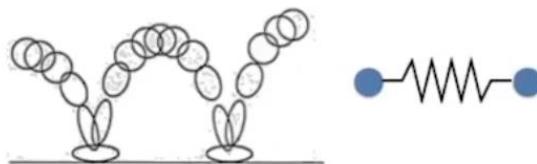
Ray Tracing

- Shoot rays from the camera through each pixel
 - Calculate **intersection** and **shading**
 - Continue to bounce the rays till they hit light sources
- Gold standard in Animations / Movies (Offline Applications)



Animation / Simulation

- Key frame Animation
- Mass-spring System



GAMES101 is NOT about

- Using OpenGL / DirectX / Vulkan
- The syntax of Shaders
- We learn Graphics,
not Graphics APIs!
- After this course,
you'll be able to learn these
by yourself (**I promise**)

~~Name~~
gluPerspective — set up a perspective projection matrix

~~C Specification~~

```
void gluPerspective(GLdouble fovy,
                    GLdouble aspect,
                    GLdouble zNear,
                    GLdouble zFar);
```

GAMES101 is NOT about

- Computer Vision / Deep Learning topics, e.g. XYZ-GAN
(where can I learn them?)



Semantic Segmentation
<https://modeldepot.io/oandrienko/icnet-for-fast-segmentation>

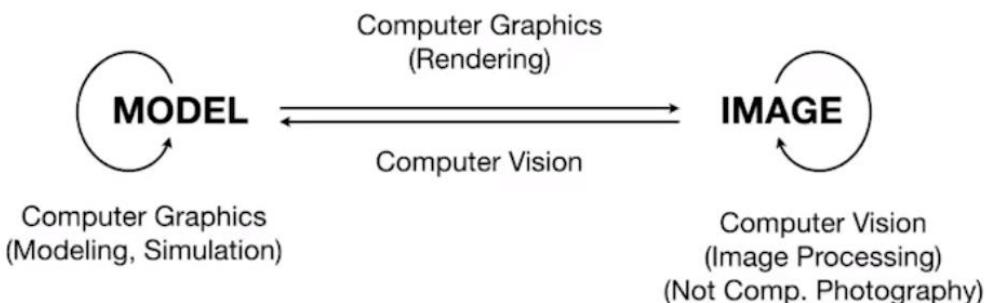


GAN 2.0: NVIDIA's
face generator (both are fake)

CV 和 CG 的区别：一切需要猜测的内容基本都是 CV

Differences?

- Personal Understanding



- No clear boundaries
- And I can't define Computer Graphics

关于两者理解：将三维的对象转换为一张图，就是 CG，（特指渲染）

将一张图内的不同对象识别出来，加以区分做出操作，如三维重建，就是 CV

环形：Model---如何去描述，去表示三维的形体；在进行一个过程的模拟之后，得到的仍然是模型

Image---运用深度学习，将一张图转变为另一张所需要的图形，如人脸的渲染

Use An IDE!

- IDE: Integrated Development Environment
- Helps you parse a entire project
 - And gives hints on syntax / usages of member functions, etc.
- Recommended IDEs
 - Visual Studio (Windows only) / Visual Studio Code (cross platform)
 - Qt Creator (personal)
- Not Recommended IDEs (for C++ programming)
 - CLion, Eclipse
 - Sublime Text, Vi / Vim, Emacs (not even IDEs)

Class 2

A Swift and Brutal Introduction to Linear Algebra!

(in fact it's relatively easy...)

VERSION: 1.0.0

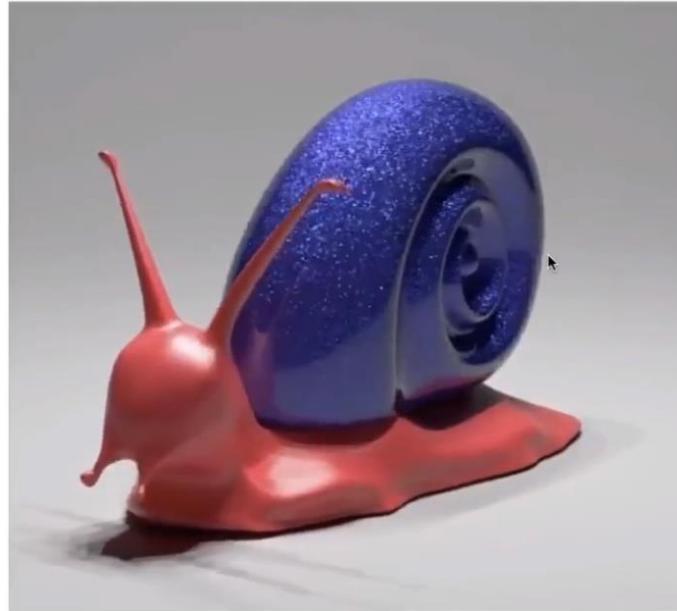
Graphics' Dependencies

- Basic mathematics
 - Linear algebra, calculus, statistics
- Basic physics
 - Optics, Mechanics
- Misc
 - Signal processing
 - Numerical analysis



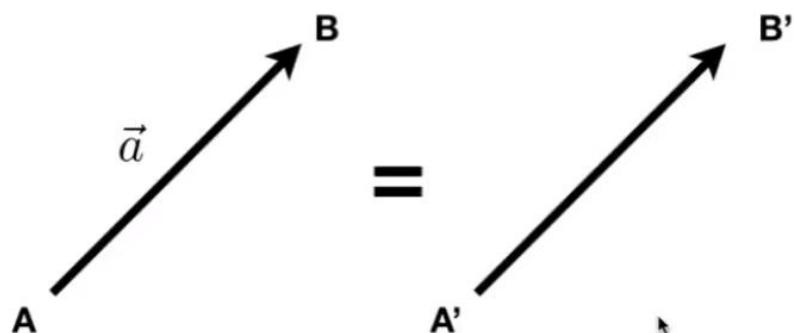
And a bit of aesthetics

An Example of Rotation



Rendering Glints on High-Resolution Normal-Mapped Specular Surfaces, Lingqi Yan, 2014

Vectors



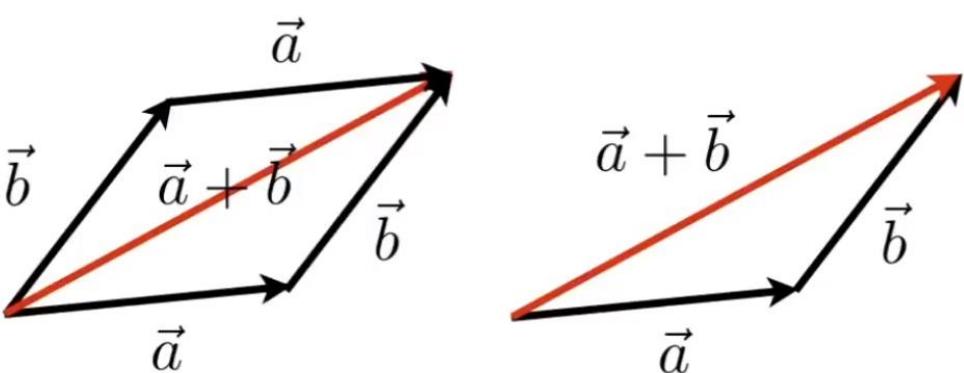
- Usually written as \vec{a} or in bold \mathbf{a}
- Or using start and end points $\vec{AB} = B - A$
- Direction and length



No absolute starting position

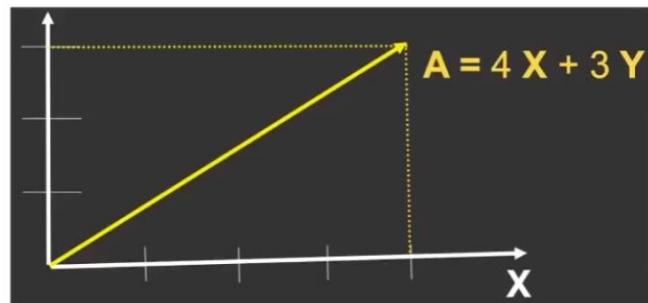
Vector Normalization

- Magnitude (length) of a vector written as $\|\vec{a}\|$
- Unit vector
 - A vector with magnitude of 1
 - Finding the unit vector of a vector (normalization): $\hat{a} = \vec{a}/\|\vec{a}\|$
 - Used to represent directions



- Geometrically: Parallelogram law & Triangle law
- Algebraically: Simply add coordinates

Cartesian Coordinates



- \mathbf{X} and \mathbf{Y} can be any (usually orthogonal unit) vectors

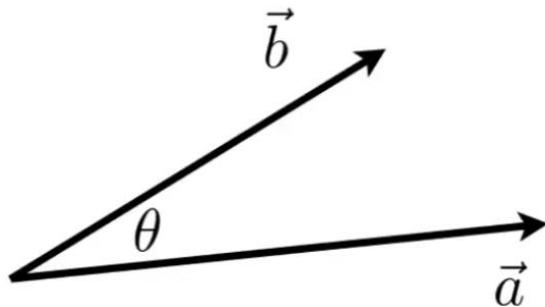
$$\mathbf{A} = \begin{pmatrix} x \\ y \end{pmatrix} \quad \mathbf{A}^T = (x, y) \quad \|\mathbf{A}\| = \sqrt{x^2 + y^2}$$

Vector Multiplication

- Dot product
- Cross product
- Orthonormal bases and coordinate frames

点乘、叉乘

Dot (scalar) Product



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

- For unit vectors



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

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

Dot (scalar) Product

- Properties

$$\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$$

$$\vec{a} \cdot (\vec{b} + \vec{c}) = \vec{a} \cdot \vec{b} + \vec{a} \cdot \vec{c}$$

$$(k\vec{a}) \cdot \vec{b} = \vec{a} \cdot (k\vec{b}) = k(\vec{a} \cdot \vec{b})$$

Dot Product in Cartesian Coordinates

- Component-wise multiplication, then adding up
 - 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


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

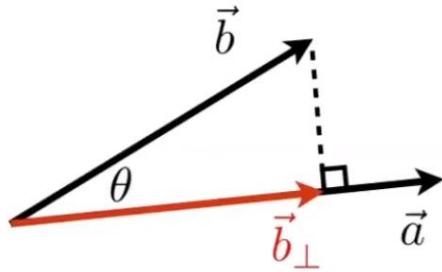
Dot Product in Graphics

- Find angle between two vectors
(e.g. cosine of angle between light source and surface)
- Finding **projection** of one vector on another

Projection—投影

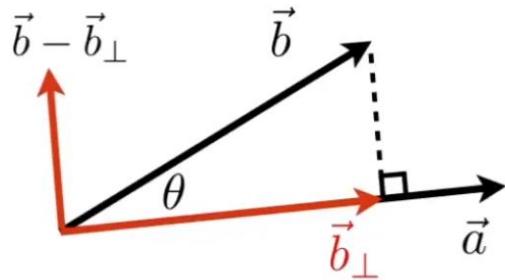
Dot Product for Projection

- \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 ?
 - $k = \|\vec{b}_\perp\| = \|\vec{b}\| \cos \theta$



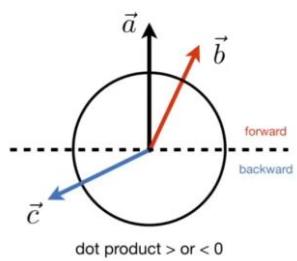
Dot Product in Graphics

- Measure how close two directions are
- Decompose a vector
- Determine forward / backward



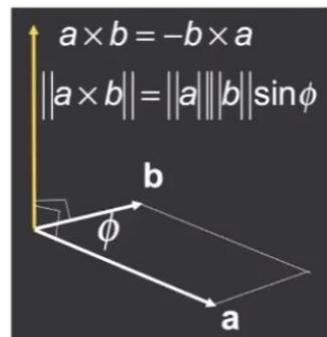
做出投影后，可以将一个向量分解为两个向量，其中一个向量平行于某个向量，另一个则垂直于某个向量。

点乘亦可以用于判断两个向量是否接近。



点乘也可以用于判断两个向量的前后关系，如图所示：我们以虚线为界，在虚线之前的，也就是于 a 做点乘结果大于 0 的，就是前方，在虚线之后的，也就是说和 a 做点乘结果小于 0 的，就是后方。也就是说，我们通过了点乘来判断向量的方向性。

Cross (vector) Product



- Cross product is orthogonal to two initial vectors
- Direction determined by right-hand rule



Useful in constructing coordinate systems (later)

Cross product: Properties

$$\vec{x} \times \vec{y} = +\vec{z}$$

$$\vec{a} \times \vec{b} = -\vec{b} \times \vec{a}$$

$$\vec{y} \times \vec{x} = -\vec{z}$$

$$\vec{a} \times \vec{a} = \vec{0}$$

$$\vec{y} \times \vec{z} = +\vec{x}$$

$$\vec{a} \times (\vec{b} + \vec{c}) = \vec{a} \times \vec{b} + \vec{a} \times \vec{c}$$

$$\vec{z} \times \vec{x} = +\vec{y}$$

$$\vec{a} \times (k\vec{b}) = k(\vec{a} \times \vec{b})$$

$$\vec{x} \times \vec{z} = -\vec{y}$$

如果 $x \times y$ 得到 z , 就称为右手坐标系

Cross Product: Cartesian Formula?

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

- Later in this lecture

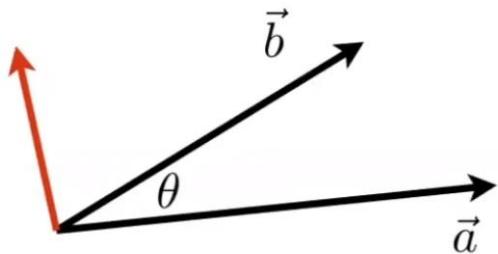
$$\vec{a} \times \vec{b} = A^* b = \begin{pmatrix} 0 & -z_a & y_a \\ z_a & 0 & -x_a \\ -y_a & x_a & 0 \end{pmatrix} \begin{pmatrix} x_b \\ y_b \\ z_b \end{pmatrix}$$

dual matrix of vector a

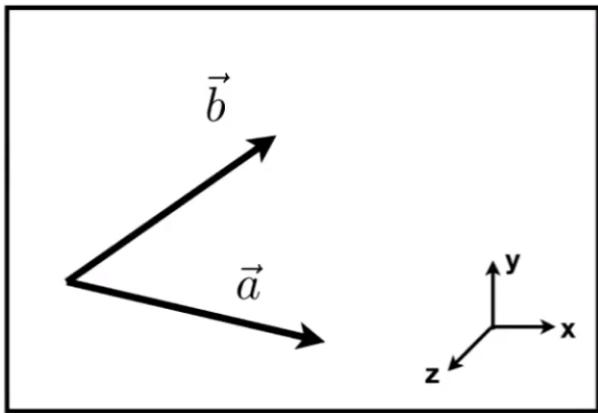


Cross Product in Graphics

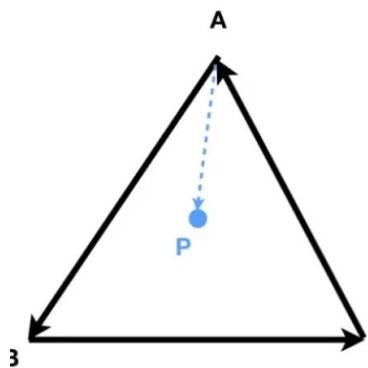
- Determine left / right
- Determine **inside / outside**



叉乘的作用：判定左和右；判定内和外



以该图为例，以 a 叉乘 b ，得到的结果是正的，就知道 b 在 a 的左侧。而以 b 叉乘 a ，得到的结果是负的，就得知 a 在 b 的右侧。



c 以该图为例， AB 和 AP 做叉乘，得到的结果是正的， P 在 AB 的左侧，再用 BC 和 BP 做叉乘，得到的结果是正的， P 在 AB 的左侧，对 CA 和 CP 做同样的操作，也可以得知 P 在 CA 的左侧，因此 P 在 ABC 的内部

Orthonormal Coordinate Frames

- Any set of 3 vectors (in 3D) that

$$\|\vec{u}\| = \|\vec{v}\| = \|\vec{w}\| = 1$$

$$\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{w} = \vec{u} \cdot \vec{w} = 0$$

$$\vec{w} = \vec{u} \times \vec{v} \quad (\text{right-handed})$$

$$\vec{p} = (\vec{p} \cdot \vec{u})\vec{u} + (\vec{p} \cdot \vec{v})\vec{v} + (\vec{p} \cdot \vec{w})\vec{w}$$

(projection)

Matrices

- Magical 2D arrays that haunt in every CS course
- In Graphics, pervasively used to represent **transformations**
 - Translation, rotation, shear, scale
(more details in the next lecture)

矩阵

Matrix-Matrix Multiplication

- Properties
 - **Non-commutative**
(AB and BA are different in general)
 - Associative and distributive
 - $(AB)C = A(BC)$
 - $A(B+C) = AB + AC$
 - $(A+B)C = AC + BC$

Matrix-Vector Multiplication

- Treat vector as a column matrix ($m \times 1$)
- Key for transforming points (next lecture)

- Official spoiler: 2D reflection about y-axis

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

利用左边的矩阵，将原来的二维图形进行关于 y 轴的对称操作，使得 x 符号改变

Transpose of a Matrix

- Switch rows and columns ($ij \rightarrow ji$)

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

- Property

$$(AB)^T = B^T A^T$$

- Dot product?

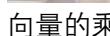
$$\vec{a} \cdot \vec{b} = \vec{a}^T \vec{b}$$

$$= \begin{pmatrix} x_a & y_a & z_a \end{pmatrix} \begin{pmatrix} x_b \\ y_b \\ z_b \end{pmatrix} = (x_a x_b + y_a y_b + z_a z_b)$$

- Cross product?

$$\vec{a} \times \vec{b} = A^* b = \begin{pmatrix} 0 & -z_a & y_a \\ z_a & 0 & -x_a \\ -y_a & x_a & 0 \end{pmatrix} \begin{pmatrix} x_b \\ y_b \\ z_b \end{pmatrix}$$

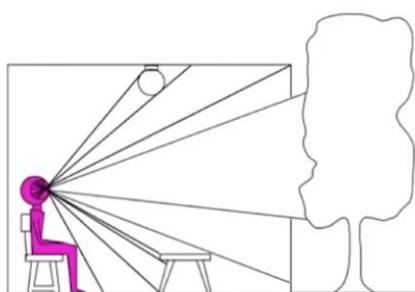
dual matrix of vector a



向量的乘积也可以写成矩阵的形式

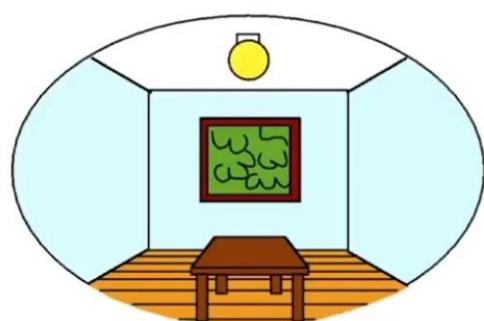
Class 3

3D world



Point of observation

2D image



Figures © Stephen E. Palmer, 2002

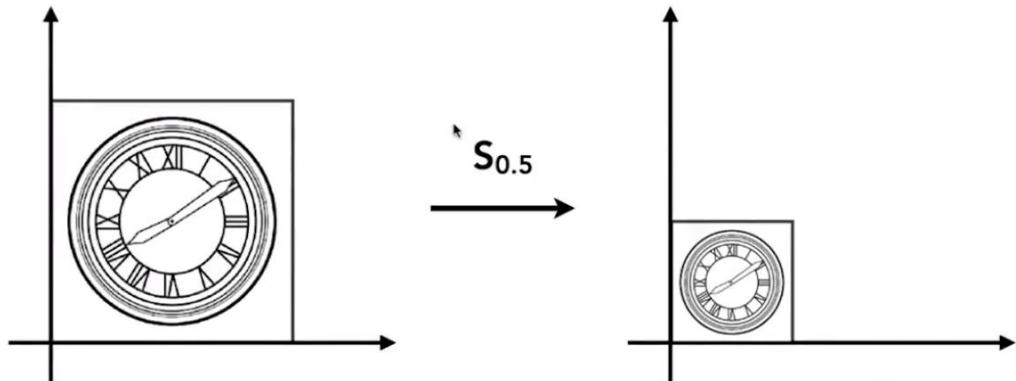


Viewing: (3D to 2D) projection

三维到二维的投影也是一种变换

Transformation

Scale



缩放变换

$$x' = sx$$

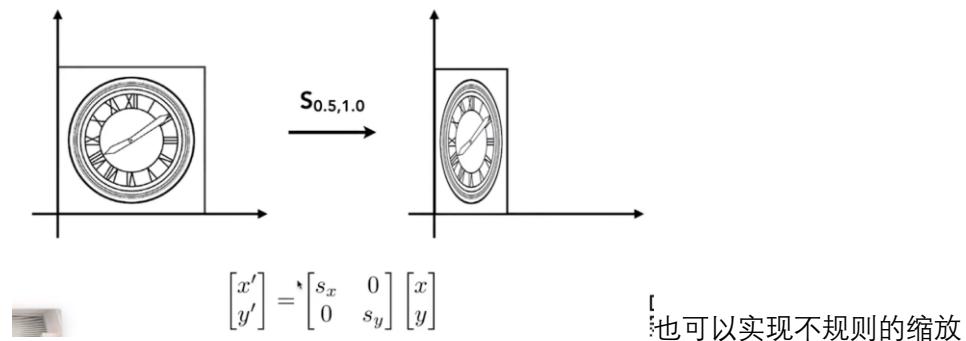
$$y' = sy$$

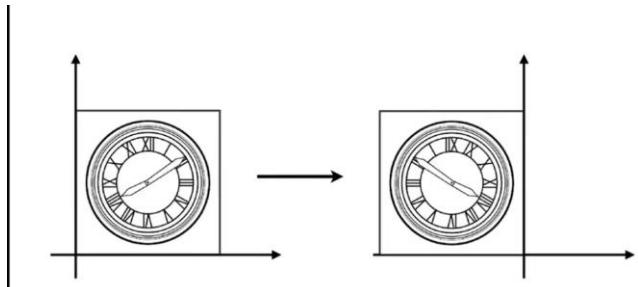
我们设原来的 x 坐标和 y 坐标为 x, y , 而经过变换后的坐标为 x' 和 y' 。

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

写为矩阵形式为:

Scale (Non-Uniform)





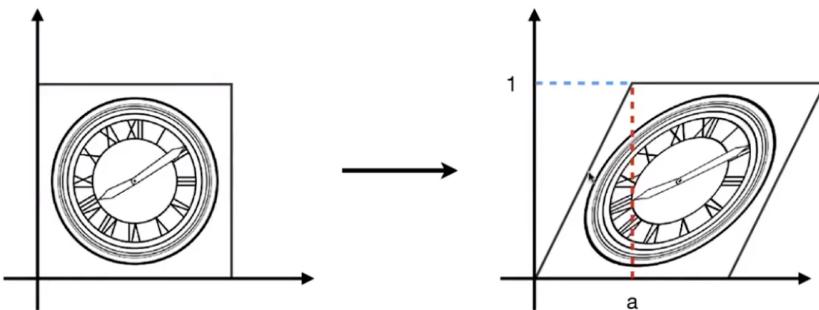
Horizontal reflection:

$$\begin{aligned}x' &= -x \\y' &= y\end{aligned}$$

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

进行图形关于 y 轴的对称操作

Shear Matrix



Hints:



Horizontal shift is 0 at $y=0$

Horizontal shift is a at $y=1$

Vertical shift is always 0

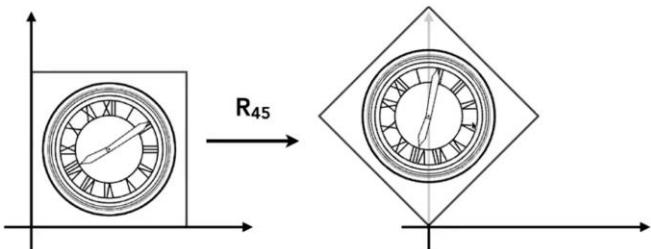
$$\begin{bmatrix}x' \\ y'\end{bmatrix} = \begin{bmatrix}1 & a \\ 0 & 1\end{bmatrix} \begin{bmatrix}x \\ y\end{bmatrix}$$



切变操作----在图

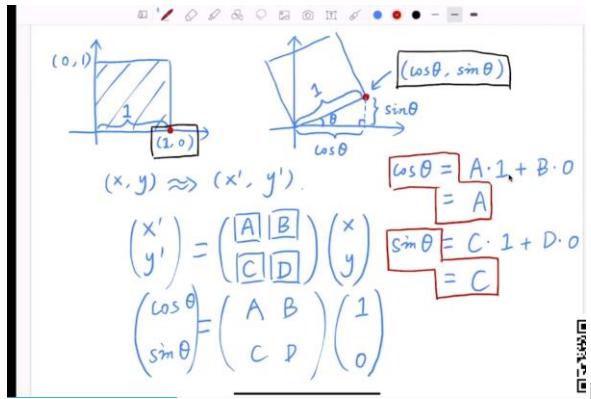
中可以观察到, y 并没有发生任何改变, 但 x 发生了改变, 在 $y=1$ 时, $x=a$; 在 $y=0.5$ 时, $x=0.5a$, 因此新的 x 坐标就是 $x+ay$; 以这样的操作做变换, 原来的 $(1, 1)$ 点就变为了 $(1+0, 5, 1)$, 即为 $(1.5, 1)$ 。寻找变换矩阵最重要就是找到变换前后的坐标之间的关系。

Rotate (about the origin $(0, 0)$, CCW by default)

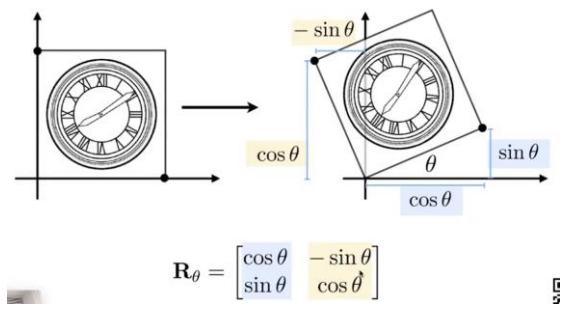


默认围绕 $(0, 0)$ 旋转, 默认方向

为逆时针



由于图形上的所有点都满足变换矩阵，那特殊的某个点肯定也满足，在这里我们取 $(1, 0)$ ，并且我们已知旋转的角度 θ ，通过三角对应关系就可以得到变换前后的两个点之间的关系。我们在通过特殊点 $(0, 1)$ 来推导矩阵的剩余元素 B, D



Linear Transforms = Matrices

(of the same dimension)

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

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

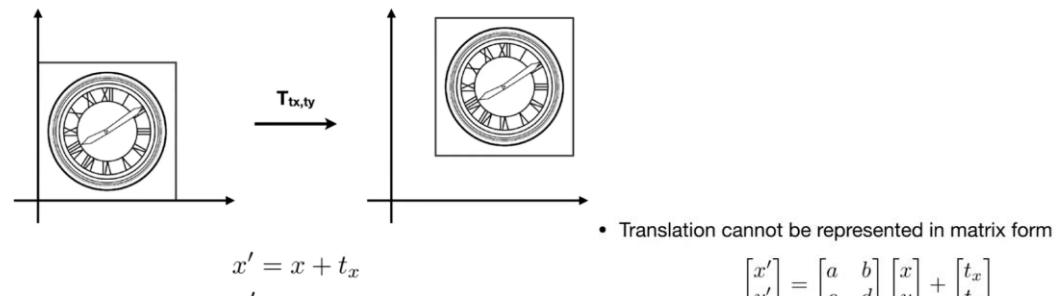
$$\mathbf{x}' = \mathbf{M} \mathbf{x}$$

通过输入坐标，乘某个矩

阵可以得到输出坐标，称这种变换为线性变换 (上图红色文字---相同维度的矩阵)

引入齐次坐标的原因---平移变换

Translation??



我们可以看到，需要完成平移变换就需要在原坐标的基础上+ t 的变量，而单纯的2维变换矩阵只能是 x 和 y 之间的组合，无法增加新的变量，因为在其后加上一个T矩阵，来完成平

(So, translation is NOT linear transform!)

移操作。

因此平移操作并

非线性变换。

- But we don't want translation to be a special case
- Is there a unified way to represent all transformations? (and what's the cost?)

人们试图找到一种包括平移、旋转等等操作的线性变换。因此引入齐次坐标。

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

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

将二维的点的坐标增加一个维度，记为 $(x,y,1)$ ，将二维的向量增加一个维度，记为 $(x,y,0)$ 。看下面的矩阵操作，我们能够看到，改写后的点乘以一个矩阵，就可以实现平移的效果，因此，添加了齐次向量，就使得平移操作变为了线性变换。

那为何将2维向量添加的维度记为0？因为---向量平移之后，仍然是这个向量，因此，在经过上图中的矩阵变换后，我们仍然想要得到 $(x,y,0)^T$ 的结果，

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

如图所示

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

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

这种操作，还有一个好处----就是上面的操作—在增加了维数之后，仍然是可以实现的。

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 肯定不为 0，否则她就是一个点了。而且----由于矩阵的特性，我们将每个元素都除以 w，整个矩阵其实没有发生变化，但是他却变成了标准的点的形式了。**一个点+另一个点，在齐次坐标的表示下，就是两个点的中点**

Affine Transformations

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:

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

仿射变换

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

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

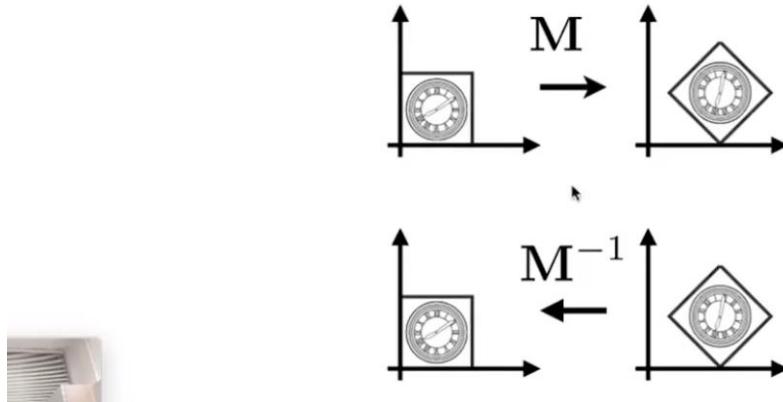


在仿射变换下，齐次向量的线性变换矩阵的最后一行是 0, 0, 1

Inverse Transform

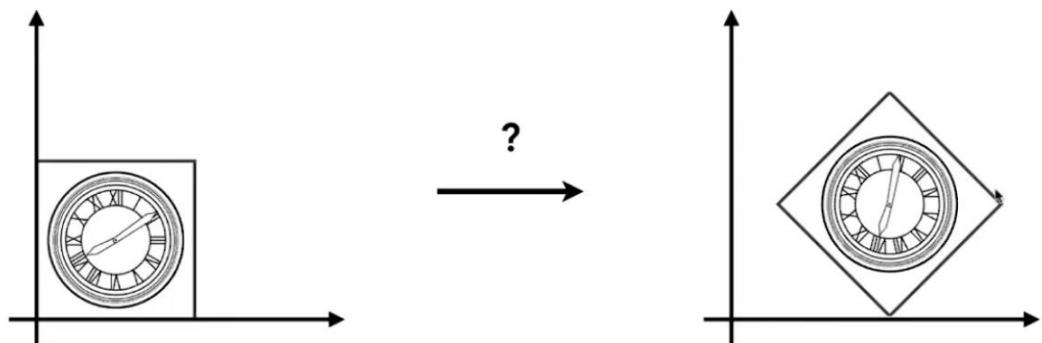
$$\mathbf{M}^{-1}$$

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

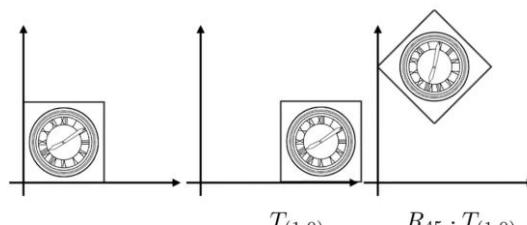


逆变换

Composite Transform

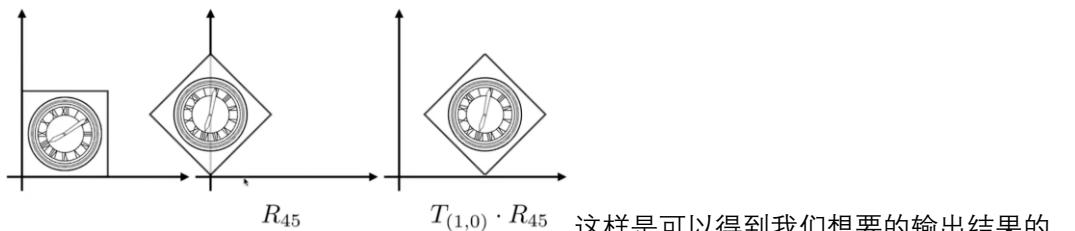


Translate Then Rotate?



是否是先平移, 在旋转呢?
得到的结果是不正确的, 需要注意的是, 我们所做的旋转-----是绕着原点所做的, 不是围绕 (1, 0) 点, 当然结果是不正确的。、

Rotate Then Translate



这样是可以得到我们想要的输出结果的。
因此---我们可以知道, 做的变换的顺序是很重要的, 而这一切的原因-----就是矩阵没有交換律

Transform Ordering Matters!

Matrix multiplication is not commutative

$$R_{45} \cdot T_{(1,0)} \neq T_{(1,0)} \cdot R_{45}$$

Note that matrices are applied right to left:

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

矩阵的变换---是从右到左的，因此，我们要完成先旋转，再平移的操作，就应该先将 R_{45} 于点做乘法，再将其结果与 $T_{(1,0)}$ 做乘法

Composing Transforms

Sequence of affine transforms A_1, A_2, A_3, \dots

- Compose by matrix multiplication
 - Very important for performance!

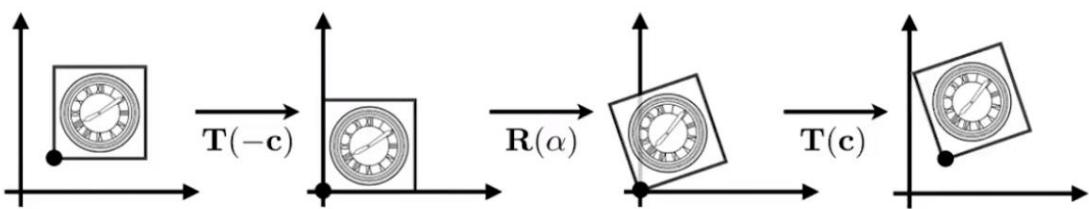
$$A_n(\dots A_2(A_1(\mathbf{x}))) = \underbrace{\mathbf{A}_n \cdots \mathbf{A}_2 \cdot \mathbf{A}_1}_{\text{Pre-multiply } n \text{ matrices}} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Pre-multiply n matrices to obtain a single matrix representing combined transform

Decomposing Complex Transforms

How to rotate around a given point c ?

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



Matrix representation?



$$T(c) \cdot R(\alpha) \cdot T(-c)$$

在这个例子中，我们想要实现图形绕着 c 点而非原点旋转，首先我们将所有点移动 $-c$ ，再将图形进行旋转操作 $R(\alpha)$ ，再对其进行 $T(c)$ 操作，就可以实现目标效果

3D Transforms

3D Transformations

Use homogeneous coordinates again:

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

类比于二维空间做变换

In general, (x, y, z, w) ($w \neq 0$) is the 3D point:

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

Use 4x4 matrices for affine transformations

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

在三维空间中，仿射变换对应的齐次坐标形式的变换矩阵的最后一行仍然为(0, 0, 0, 1)。
三维空间中的点以齐次坐标的形式做变换，实际的三维坐标仍然是先线性变换，再平移。也要明确齐次坐标变换矩阵，是可以分解为线性变换和平移的

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

$$R_{-\theta} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = R_\theta^T$$

$$R_{-\theta} = R_\theta^{-1} \text{ (by definition)}$$

上式中， R_θ 为二维空间内旋转一个 θ 角度的二阶矩阵，而第二个矩阵为旋转 $-\theta$ 的矩阵，由于 \cos 为偶函数，因此不变， \sin 为奇函数，取负值。观察可得，第二个矩阵就是第一个矩阵的转置。由定义可知，旋转 $-\theta$ 角就是旋转 θ 角的逆，就有一个结论，在旋转的操作中，旋转 θ 角矩阵的逆矩阵就是他的转置，而这种矩阵---逆矩阵就是他的转置---称为正交矩阵

Class 4

- 3D transformations
 - Viewing (观测) transformation
 - View (视图) / Camera transformation
 - Projection (投影) transformation
 - Orthographic (正交) projection
 - Perspective (透视) projection
-

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

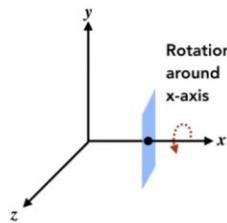
$$\mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

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

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

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



如左图所示, 图形绕着 x 轴旋转,

则他的 y,z 在发生变化, 而 x 不变, 反映到矩阵上, 就是里面的 3*3 的线性变换矩阵对应的 x 轴变换的变量为 1

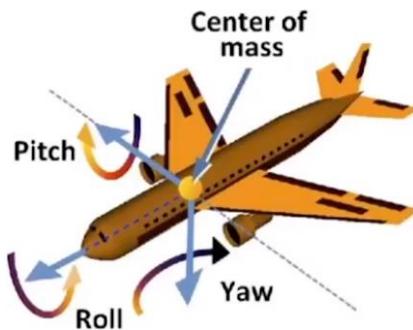
上图的三个矩阵中, Ry 明显不同, 这是因为-----x 叉乘 y 得到 z, y 叉乘 z 得到 x, 但是 y 却是 z 叉乘 x 得到的, 顺序发生了变化, 因此-----Ry 的矩阵明显区别于 Rx 和 Rz, 角度是反的。

3D Rotations

Compose any 3D rotation from $\mathbf{R}_x, \mathbf{R}_y, \mathbf{R}_z$?

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

- So-called *Euler angles*
- Often used in flight simulators: roll, pitch, yaw



Rodrigues' Rotation Formula

Rotation by angle α around axis \mathbf{n}

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

How to prove this magic formula?

Check out the supplementary material on the course website!

罗德里格斯旋转公式，定义了一个旋转轴 n 和一个旋转角度 α

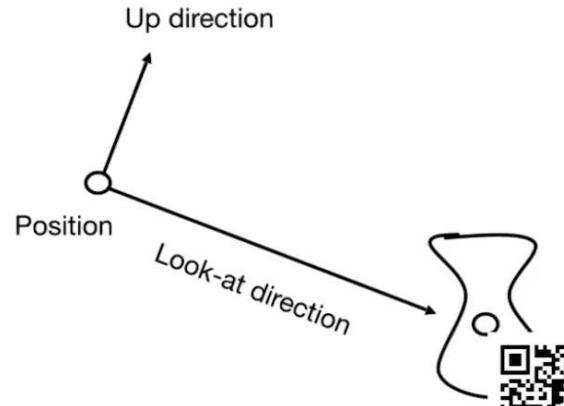
View / Camera Transformation

- 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)

- How to perform view transformation?

- Define the camera first

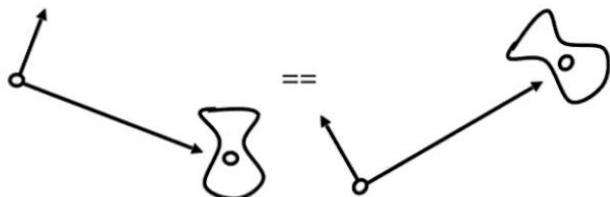
- Position \vec{e}
- Look-at / gaze direction \hat{g}
- Up direction \hat{t}
(assuming perp. to look-at)



视图变换需要定义一个相机，首先我们定义它的位置： e ; 在定义他面向的方向： g ; 在定义一个向上的方向，来确定其位置

- 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



通常摆放相机的方式====将其放在原点，并且看向-z 方向；由于相对运动的存在，我们只要保证物体和相机一起发生移动，那得到的效果肯定是一致的

- Transform the camera by M_{view}

- So it's located at the origin, up at Y, look at -Z

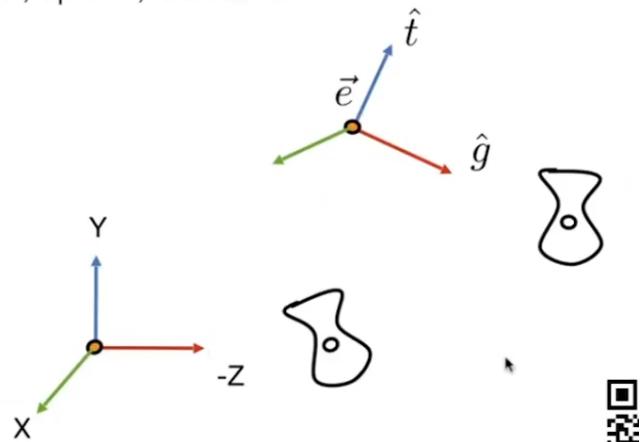
- M_{view} in math?

- Translates e to origin
- Rotates g to -Z
- Rotates t to Y
- Rotates $(g \times t)$ To X
- Difficult to write!



Bottom Left

Top Right



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, $(g \times t)$ To X

- Consider its inverse rotation: X to $(g \times t)$, Y to t, Z to -g

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



将相机移动的矩阵====先做平移，在做旋转====

由于将相机坐标移动到世界坐标不容易，因此我们反过来将世界坐标移动到相机坐标上

来，也就是做它的逆操作。由于旋转矩阵是正交矩阵，因此它的逆就是它的转置，即上图中的右边矩阵。

所谓向上方向，就是指固定的垂直于相机的一个方向，在相机发生转动时，它能够表示它的旋转

Projection

CONTINUE ➞ ➞ ➞

Projection Transformation

- Projection in Computer Graphics

- 3D to 2D
- Orthographic projection
- Perspective projection

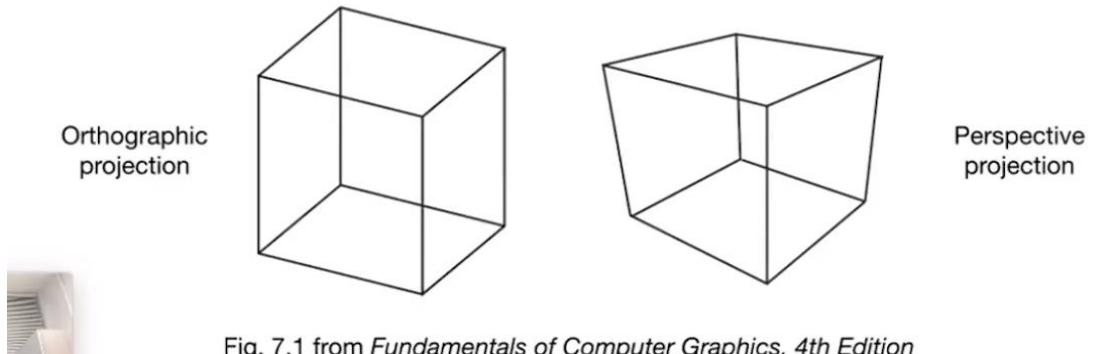
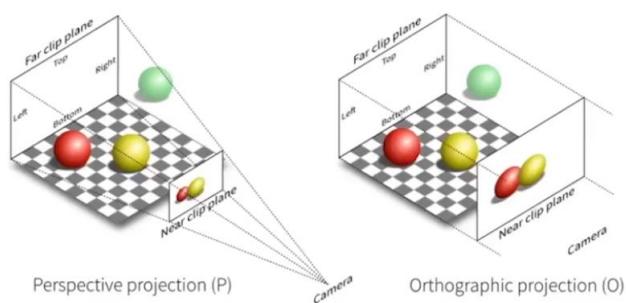


Fig. 7.1 from *Fundamentals of Computer Graphics, 4th Edition*

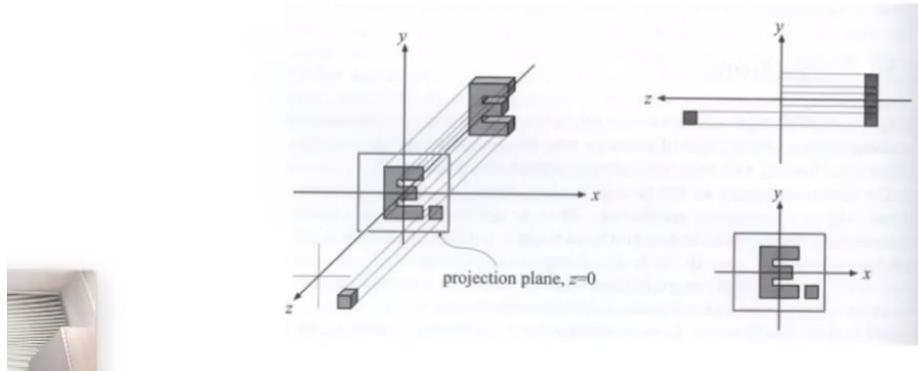
如上图，第一张为正交投影，第二张为透视投影；第一张投影中，原来物体上是平行的线投影之后仍然是平行的，而第二张投影中则不然，原本平行的线会相交

- Perspective projection vs. orthographic projection



- A simple way of understanding

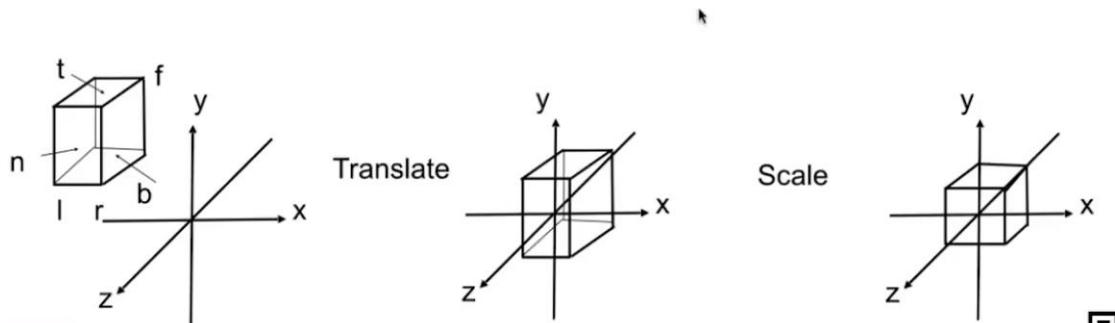
- Camera located at origin, looking at -Z, up at Y (looks familiar?)
- Drop Z coordinate
- Translate and scale the resulting rectangle to $[-1, 1]^2$



举一例，将相机放在原点，方向朝相-z 轴方向，其中两个物体====物体 E 和一方块分别在 z 的负半轴和正半轴区域内，他们两个的前后当然不同，但将 z 轴丢弃，那他们在 xy0 平面上，就会如第三张图所示，看不出远近关系，

- In general

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

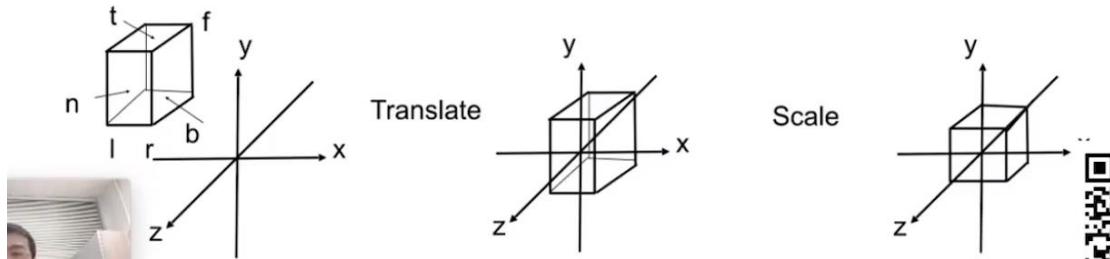


通常来说，我们的做法是这样的，将一个长方体转变为一个标准的立方体。首先将立方体的中心移动到世界的原点，然后再对其进行变换，得到期望的立方体。

- Transformation matrix?

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

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



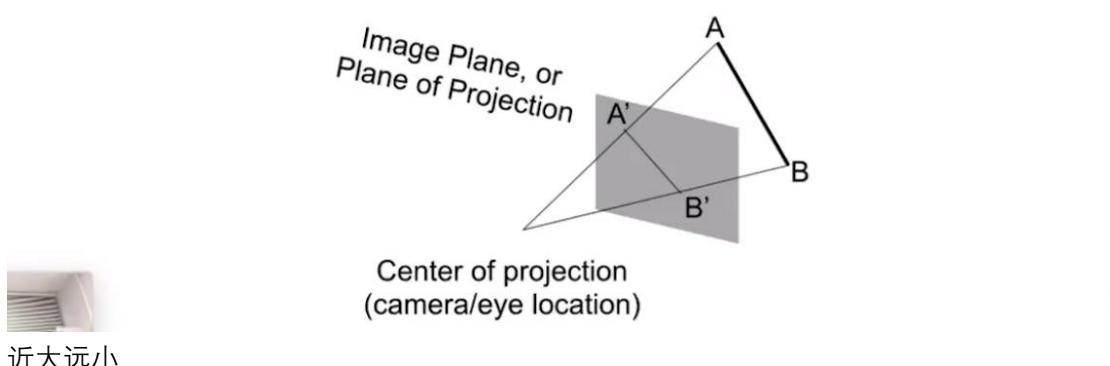
- Caveat

- Looking at / along -Z is making near and far not intuitive ($n > f$)
- FYI: that's why OpenGL (a Graphics API) uses left hand coords.

由于相机的方向为-Z，所以n（近）实际上是大于f（远）的，而这也是openGL等IDE采用左手系的原因

Perspective Projection

- Most common in Computer Graphics, art, visual system
- Further objects are smaller
- Parallel lines not parallel; converge to single point



Perspective Projection

- Before we move on
- Recall: property of homogeneous coordinates
 - $(x, y, z, 1), (kx, ky, kz, k \neq 0)$, $(xz, yz, z^2, z \neq 0)$ all represent the same point (x, y, z) in 3D
 - e.g. $(1, 0, 0, 1)$ and $(2, 0, 0, 2)$ both represent $(1, 0, 0)$
- Simple, but useful
- How to do perspective projection
 - First “squish” the frustum into a cuboid ($n \rightarrow n, f \rightarrow f$) ($M_{\text{persp} \rightarrow \text{ortho}}$)
 - Do orthographic projection (M_{ortho} , already known!)

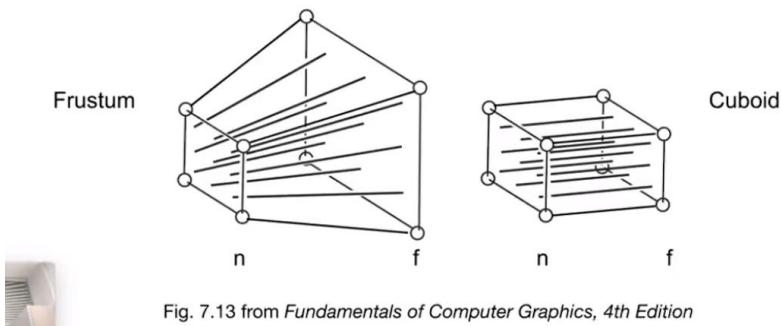
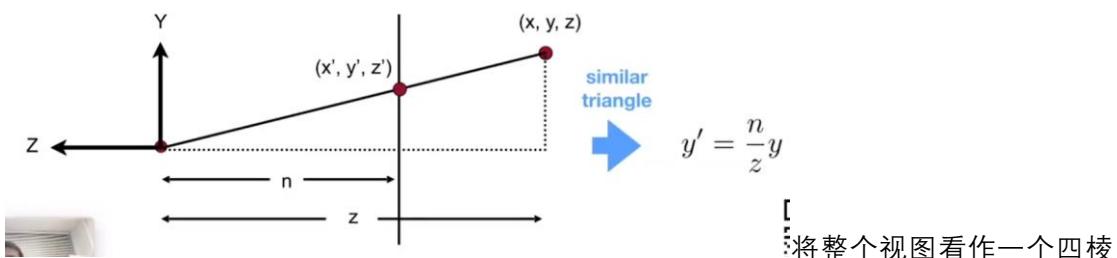


Fig. 7.13 from *Fundamentals of Computer Graphics, 4th Edition*

- 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)



将整个视图看作一个四棱锥，其中近的视图就是前一个切面，而远的视图就是后一个切面，然后再对他做正交视图的变换。下图做出具体的解释====将 (x,y,z) 压缩到 (x,z,y') 的位置，我们能够根据相似三角形的原理来得到两个 y 之间的相对关系，就是 n (近) / z (远) 的比例关系

- In order to find a transformation

- Find the relationship between transformed points (x' , y' , z') and the original points (x , y , z)

$$y' = \frac{n}{z}y \quad x' = \frac{n}{z}x \text{ (similar to } y')$$

- In homogeneous coordinates,



$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} nx/z \\ ny/z \\ \text{unknown} \\ 1 \end{pmatrix} \stackrel{\text{mult. by } z}{=} \begin{pmatrix} nx \\ ny \\ \text{still unknown} \\ z \end{pmatrix}$$

相同的， x 与 x' 也有这样的对应关系；

假如我们对一个点做这样的挤压操作，无论如何，我们都可以得到第二个这种形式，其中 $x'=nx/z, y'=ny/z$ ，而 z' 未知，对于一个点，我们同时乘一个值仍然是这个点，因此我们对每个坐标都乘上 z ，得到第三个形式，而 $z*z'$ 仍然是未知的

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

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

- Already good enough to figure out part of $M_{\text{persp} \rightarrow \text{ortho}}$



$$M_{\text{persp} \rightarrow \text{ortho}} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad \text{WHY?}$$

我们在原来的形式左边乘一个 4 阶矩阵，得到右边的形式，而根据这些并不是完全已知的条件，我们也能写出一个并不完整的四阶矩阵，这是容易做到的

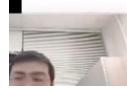
- Observation: the third row is responsible for z'
 - Any point on the near plane will not change
 - Any point's z on the far plane will not change

而仅仅这些条件，我们就能得到关于 z 的一些信息====任何在近的平面上的点都没有发生改变（即第一个切面的点不变，仍然是== (x,y,n) 在近平面上 $z'=n$)); 而远平面上的点的 z 坐标也没有发生变化，

- Any point on the near plane will not change

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

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



$$(0 \ 0 \ A \ B) \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} = n^2 \quad \text{n}^2 \text{ has nothing to do with } x \text{ and } y$$



我们再来做

这样的假设====将未知的 z 坐标设为 n ，我们对每个坐标都乘 n ，这对点而言并没有发生改变，但是再通过一个 4 阶方阵做这样的操作时，我们可以知道该矩阵的第三行必然是这样的形式==== $(0,0,A,B)$ ，第一，二个元素必然是 0，因为变换后的 z 不含 x 和 y ，而无法确定第三第四个元素的值，有可能 $A=n, B=0$ ，也有可能 $A=0, B=n^2$ ，因此我们无法确定这两个元素的值。而由于近平面的点不发生变化，因此经过处理后的 z 坐标应该和原来的 z 坐标相同，就可以得到一个等式。

- What do we have now?

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

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



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

我们还可以得到另一个结论====远平面的中心点，并不会因为压缩而发生变化，因此我们取这个点 $(0,0,f,1)^T$ ，做我们刚才做的变换==将每一项都乘上 z，这个点没有发生变化，这样的话，就可以得到两个点，一个特殊的中点，一个普通的近平面点

■ ■ ■

- 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 $M_{persp \rightarrow ortho}$ is known!

- What's next?

- Do orthographic projection (M_{ortho}) to finish
- $M_{persp} = M_{ortho}M_{persp \rightarrow ortho}$

通过这两个式子，我们能够得到 A 和 B 的值，直到这一步=====整个 4 阶矩阵的元素我们就都知道了。然后我们在对经过了这个压缩变换的物体进行正交投影---- M_{ortho} ,这样就完成了整个的透视投影的操作。

一个问题=====在两个平面中任取一个平面内的点，在远平面进行压缩操作时，它的 z 是如

何变化？变大变小或是不变？

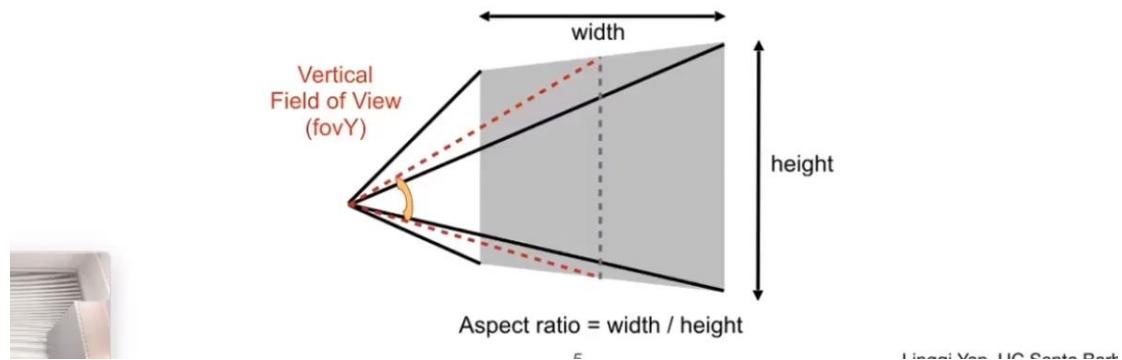
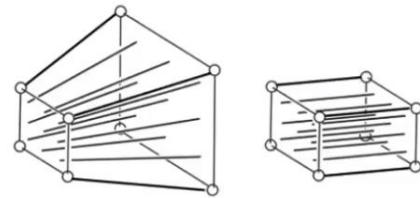
Class 5

Week 5: 3D

Perspective Projection

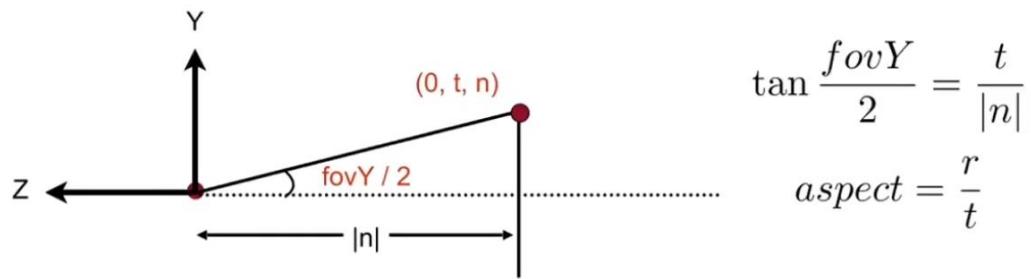
- What's near plane's l, r, b, t then?

- If explicitly specified, good
- Sometimes people prefer:
vertical **field-of-view** (fovY) and
aspect ratio
(assume symmetry i.e. l = -r, b = -t)



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

- Trivial



从侧面观察整个相机，假如可视角度为 Y，则这个三角形的一个锐角为 Y/2。
如果我们要定义一个可视角度，只需要定义一个可视角度，一个宽高比，则其他参数都可以通过它们来得到

- **Model transformation** (placing objects)
- **View transformation** (placing camera)
- **Projection transformation**
 - Orthographic projection (cuboid to “canonical” cube $[-1, 1]^3$)
 - Perspective projection (frustum to “canonical” cube)

得到的投影我们要将其放置在屏幕上，那关于屏幕我们给出定义----

Canonical Cube to Screen

- What is a screen?
 - An array of pixels
 - Size of the array: resolution
 - A typical kind of raster display

一个二维的数组，用像素来表示每一个数组的元素，我们通常用分辨率来表示这个数组的大小，而且屏幕是一个典型的光栅成像设备。

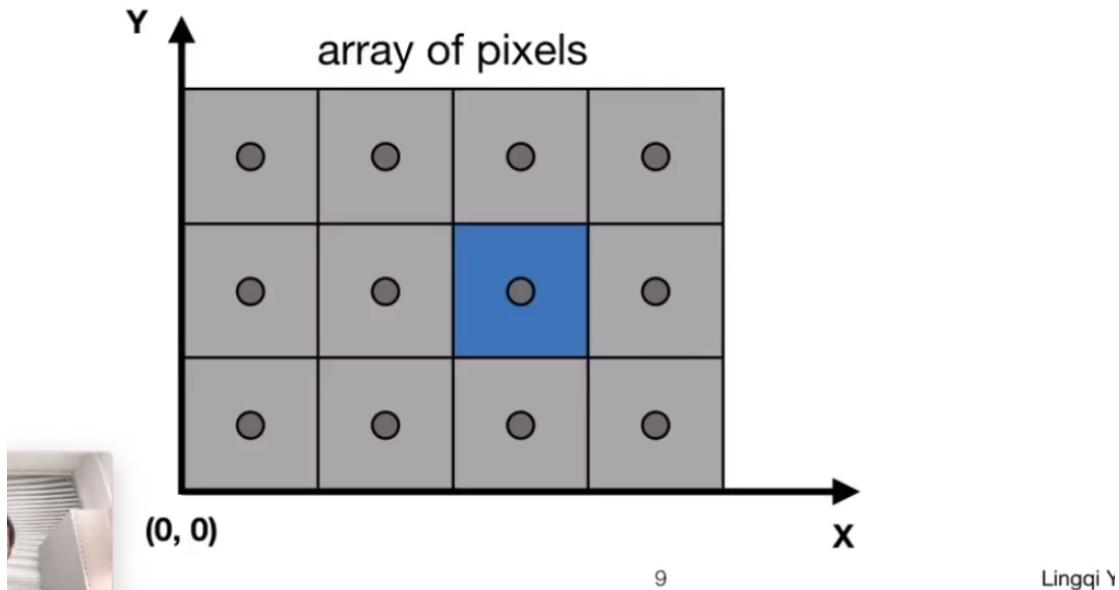
那什么是光栅---一个德语的词汇。

所谓光栅化，就是将光栅画在屏幕上的过程

- Pixel (FYI, short for “picture element”)
 - For now: A pixel is a little square with uniform color
 - Color is a mixture of (red, green, blue)

在 CG 中，我们理解像素为一个一个的方块，而每个方块的颜色是不变的，也就是说，每个像素只有一个颜色。而每个像素的颜色，都可以用 RGB 三个参数来表征。而颜色的强度则可以用灰度来表示，分为 256 个等级

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



关于屏幕空间的定义：将屏幕的左下角定义为原点

我们也可用这种方式来定义像素的位置，例如上图中的蓝色像素，定义为 $(2, 1)$ ，

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

Pixels' indices are from
 $(0, 0)$ to $(\text{width} - 1, \text{height} - 1)$

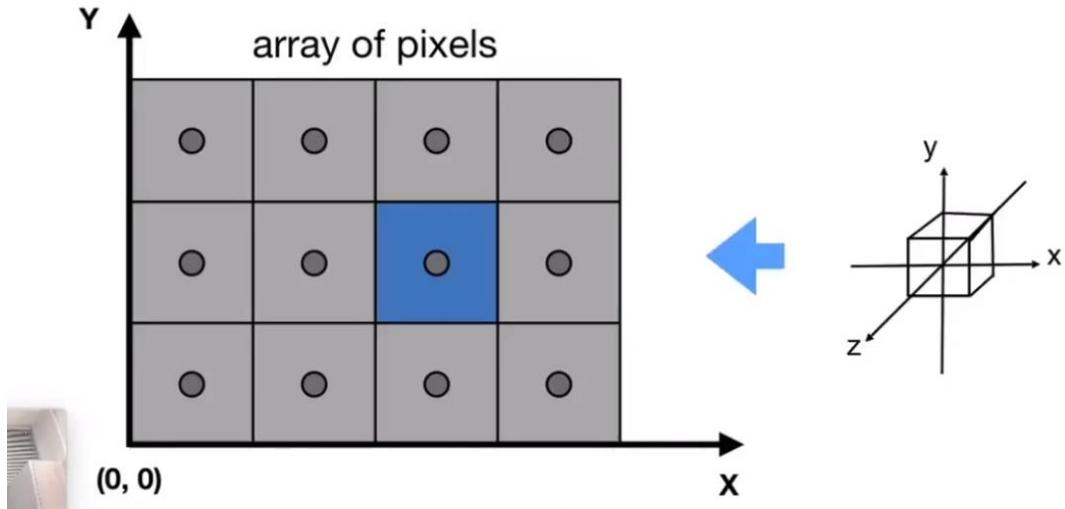
同时，我们认为---像素的坐标是 $(0,0)$ 到 $(\text{宽}-1, \text{长}-1)$

Pixel (x, y) is centered at
 $(x + 0.5, y + 0.5)$ 实际上，像素的坐标在 $(x+0.5,y+0.5)$

The screen covers range
 $(0, 0)$ to $(\text{width}, \text{height})$

因此，我们定义屏幕覆盖的范围就是 $(0,0)$ 到 $(\text{宽}, \text{长})$

- Irrelevant to z
- Transform in xy plane: $[-1, 1]^2$ to $[0, \text{width}] \times [0, \text{height}]$



- Irrelevant to z
- Transform in xy plane: $[-1, 1]^2$ to $[0, \text{width}] \times [0, \text{height}]$
- Viewport transform matrix:

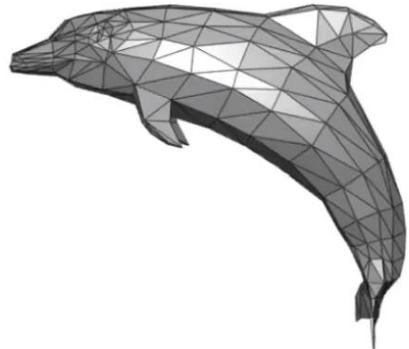
$$M_{viewport} = \begin{pmatrix} \frac{\text{width}}{2} & 0 & 0 & \frac{\text{width}}{2} \\ 0 & \frac{\text{height}}{2} & 0 & \frac{\text{height}}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

先忽略 z 轴，将原来的 x, y 压缩成 width 和 height

Why triangles?

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)



光栅化中最重要的一个部分就是判断三角形和像素中心点之间的位置关系，在这里我们采用采样的方法。

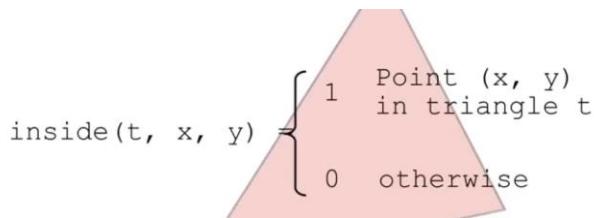
那么什么叫做采样呢？就是给定一个连续的函数，根据不同的变量值，来得到函数的值。也就是说，采样就是将一个连续函数离散化的过程

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);
```

我们可以定义一个函数来确定三角形和像素中心的关系：

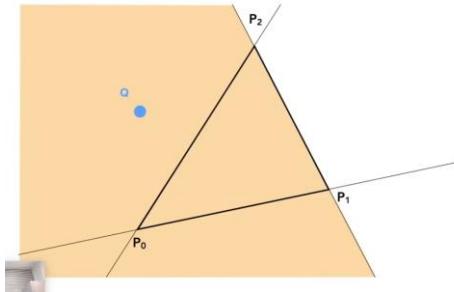


```
for (int x = 0; x < xmax; ++x)
    for (int y = 0; y < ymax; ++y)
        image[x][y] = inside(tri,
            x + 0.5,
            y + 0.5);
```

用这段代码来判断关系

那如何来判断中心点是否在三角形内？

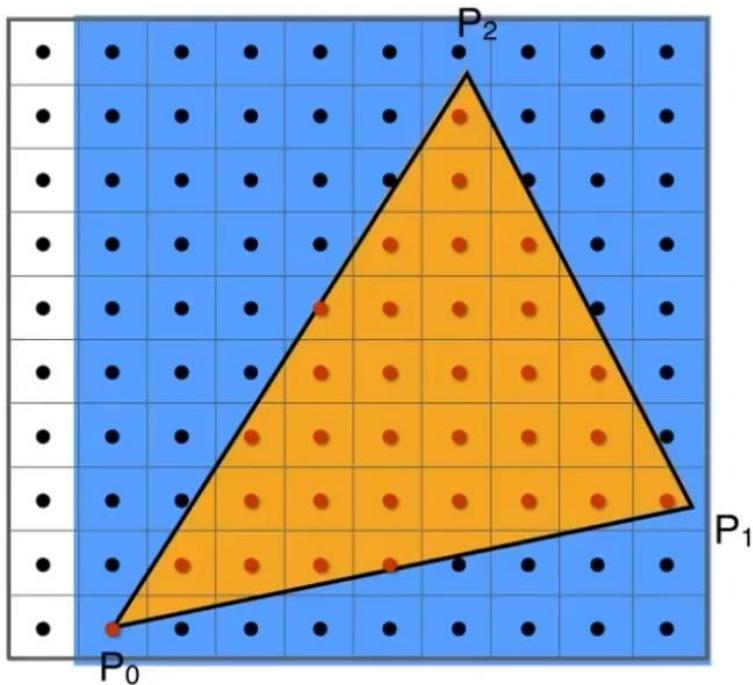
inside? Recall: Three Cross Products!



以该图为例，矢量 P_1P_2 和矢量 P_1Q 做叉乘，得到的

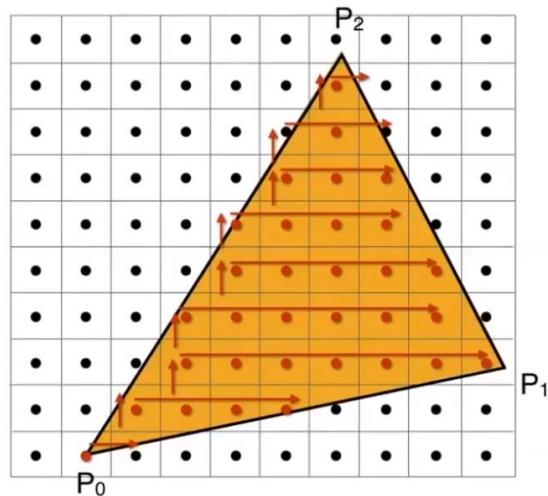
矢量是向屏幕外的，那我们就知道了 P_1Q 在 P_1P_2 的左侧。同样的，我们在用 P_0P_1 和 P_0Q 做叉乘，也得到 P_0Q 在 P_0P_1 的左侧； P_2P_0 和 P_2Q 做叉乘，得到 P_2Q 在 P_2P_0 的右侧，综上，我们得知， Q 在所有矢量的左侧，因此 Q 不在三角形内

而在三角形边界上的像素中心点，我们要不不做处理，要不就做特殊处理。在本科目中，我们对其不做处理



Use a Bounding Box!

是否要检查屏幕上的所有像素？很明显的，不在三角形覆盖区域内的像素，我们不需要做光栅化，因此我们只需要扫描图中蓝色的区域，这个区域我们称为包围盒，另外也有其他方式来减少光栅化的像素数量，但其实实现起来并不简单



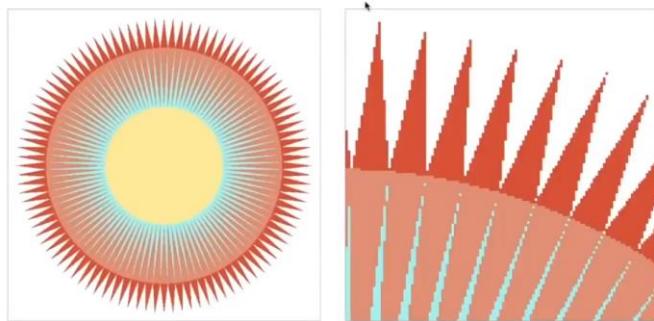
suitable for thin and rotated triangles

如果我们要光栅化的三角形

很细长，并且摆放的并不竖直，那他要占据的包围盒其实并不小，这个时候我们采用上图的这种扫描方式，能够更快速的光栅化。也就是说，光栅化的方式也是有各种使用场合的。

Class 6

Sampling Artifacts (Errors / Mistakes / Inaccuracies) in Computer Graphics



 This is also an example of "aliasing" – a sampling error 「边缘的走样」



Skip odd rows and columns

摩尔纹：将奇数行奇数列去掉，整个图形变

小，但再将他放大会原来的尺寸，就会出现上图的现象

Artifacts due to sampling - "Aliasing"

- Jaggies – sampling in space
- Moire – undersampling images
- Wagon wheel effect – sampling in time
- [Many more] ...

Behind the Aliasing Artifacts

- Signals are **changing too fast** (high frequency),
but **sampled too slowly**

造成这些 Artifacts

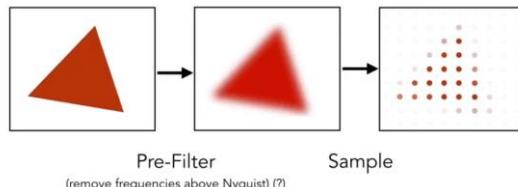
的原因，其实都可以归结于====信号（函数）的变化太快，以至于采样跟不上他的速度

Antialiasing Idea: Blurring (Pre-Filtering) Before Sampling

反走样的方法：在采样之前，先

做一个模糊处理

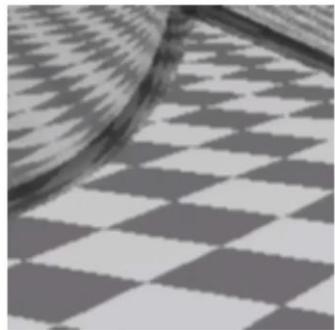
Rasterization: Antialiased Sampling



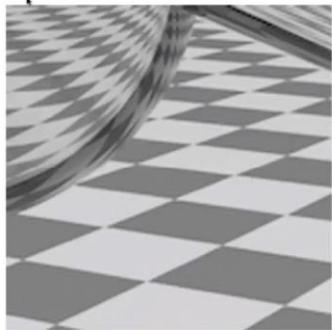
Note antialiased edges in rasterized triangle
where pixel values take intermediate values

如该图，在三角形光栅化前先做一个模糊化

（滤波）的处理，将三角形变为一个模糊的三角形，将靠近模糊边界的像素的颜色变为浅红色，就可以起到一定的反走样（抗锯齿）的作用。这一过程的顺序不可颠倒，例子如下



'Sample then filter, WRONG!'



(Filter then sample)

1. Why undersampling introduces aliasing?
2. Why pre-filtering then sampling can do antialiasing?

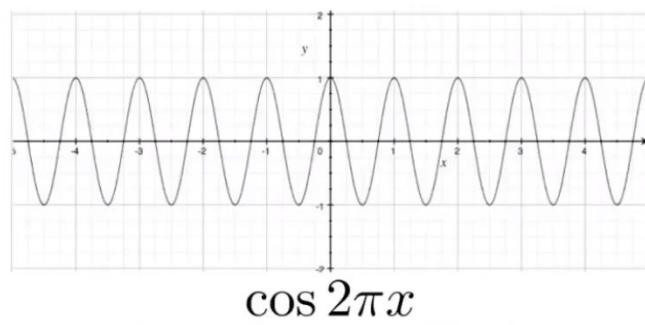
Let's dig into fundamental reasons

And look at how to implement antialiased rasterization

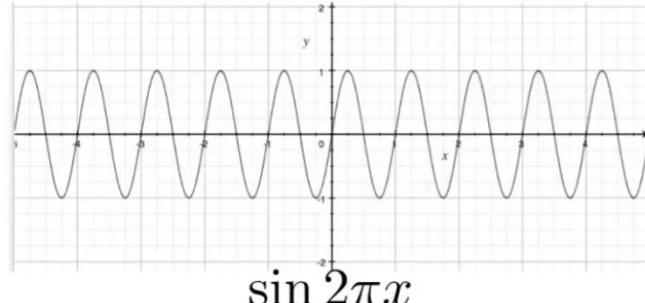
为什么采样速度相对较低会导致走样?

为什么先滤波，再采样可以反走样?

频域

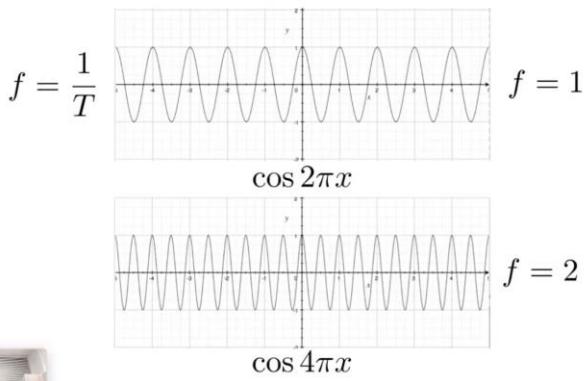


$\cos 2\pi x$



$\sin 2\pi x$

Frequencies $\cos 2\pi f x$

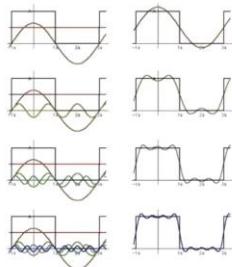


如果将变量前的系数考虑在内，我们就能

得到不同频率的波形，如上图所示

Fourier Transform

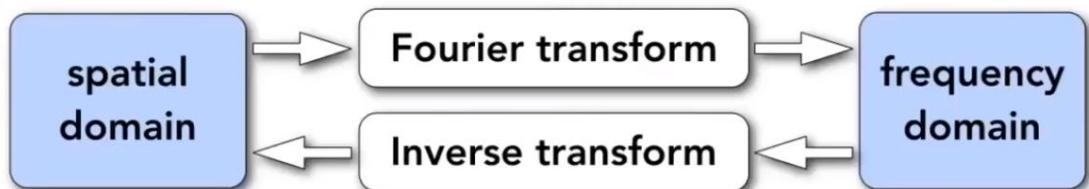
Represent a function as a weighted sum of sines and cosines



$f(x) = \frac{A}{2} + \frac{2A \cos(t\omega)}{\pi} - \frac{2A \cos(3t\omega)}{3\pi} + \frac{2A \cos(5t\omega)}{5\pi} - \frac{2A \cos(7t\omega)}{7\pi} + \dots$ 这里需要用到傅里叶级数展开，所谓傅里叶级数展开，就是可以将所有周期函数变为余弦函数的线性组合再加一个常数项

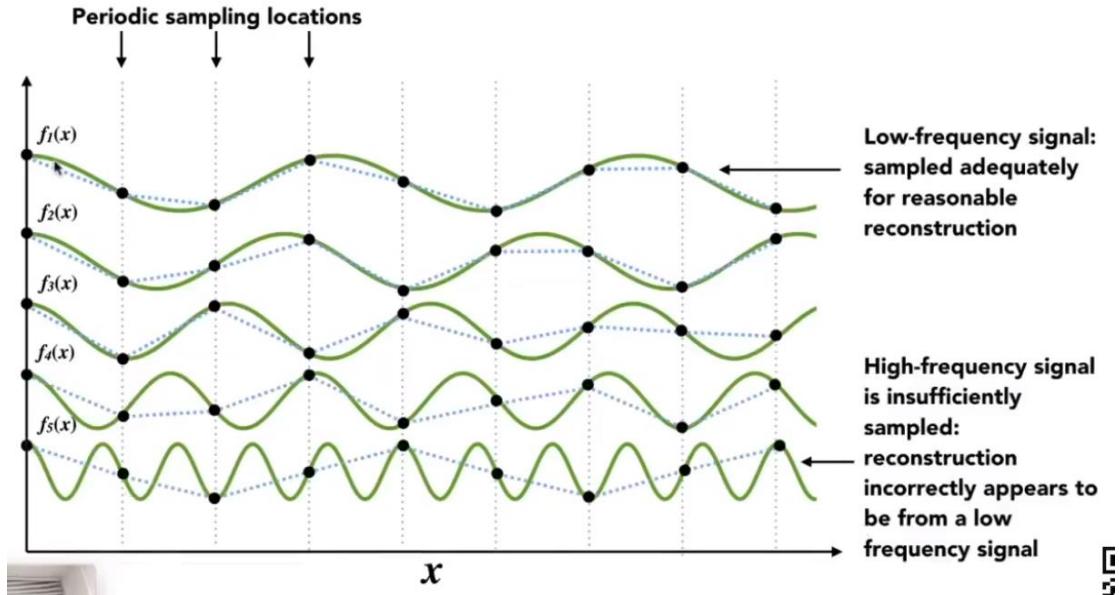
Fourier Transform Decomposes A Signal Into Frequencies

$$f(x) \quad F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \omega x} dx \quad F(\omega)$$



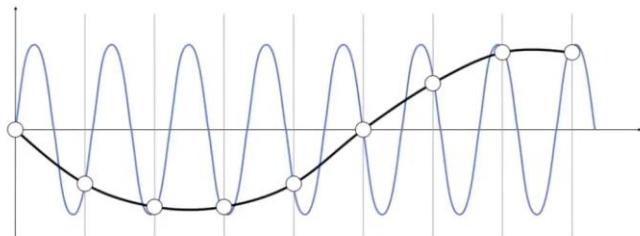
$$f(x) = \int_{-\infty}^{\infty} F(\omega) e^{2\pi i \omega x} d\omega$$

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



在函数的频率较低时，我们设定的采样点还能够显示函数的形状，但当函数的频率提高，还采用原来的采样点，就不能够显示函数的真实变化了。

Undersampling Creates Frequency Aliases



High-frequency signal is insufficiently sampled: samples erroneously appear to be from a low-frequency signal

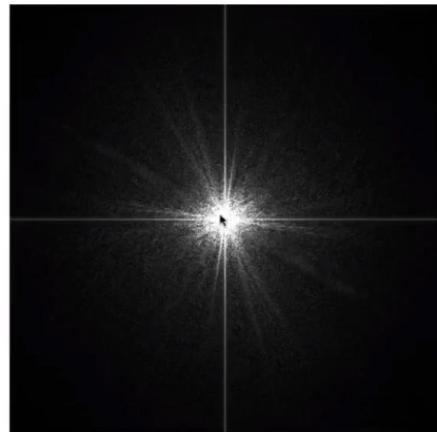
再如此例，我们可以看到，采

样得到的函数和实际的函数相差特别大

Two frequencies that are indistinguishable at a given sampling rate are called “aliases”

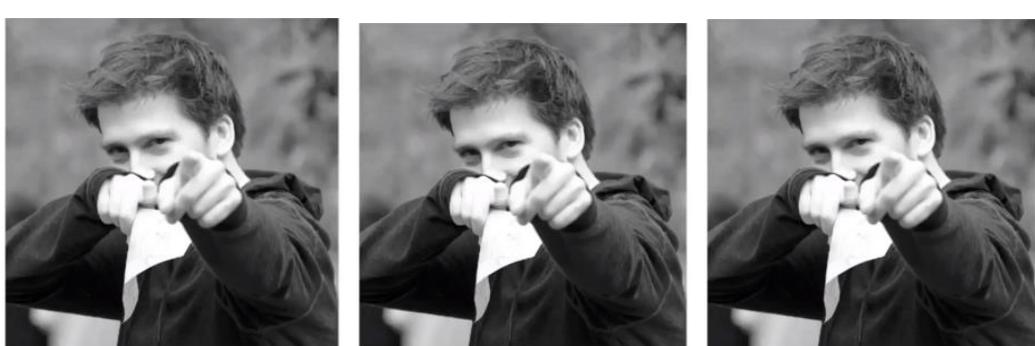
用同样的两种采样方法，来采样频率不同的两个函数，得到的结果完全相同，无法区分两个函数，就称为走样。

Filtering = Getting rid of
certain frequency contents



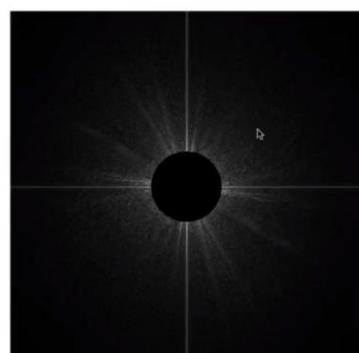
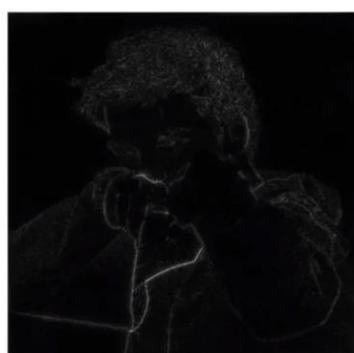
我们利用傅里叶变换，将图 1 从时域（并非时间上的概念，我们将空间内得图形也定义在时域内）变换为频域，得到图 2。我们应该该如何理解频域呢？在图二中，中心得频率低，外部的频率高，也就是说：频率从中心向外部慢慢降低，我们也可以知道，在该图中，中心的亮度高，说明低频信号多，而高频信号则较少，外部亮度低。很多图片都有这样的性质，即低频信息多，高频信息少。

如图 1 所示，很少有图片是周期显示的，那我们就这么认为这张图片====他在图片的左右边界又重复了这张图片，如同下面的效果。



我们可以看到，在图片的左右边界相接时，由于图片不是连续的，会产生很大的信号变化，产生极其高的高频，这就是频域图中两条类似于坐标的线的来由，因此我们在分析频域时，一般忽略这两条线

Filter Out Low Frequencies Only (Edges)



High-pass filter

我们利用滤波技术，将频域图

中的低频信号滤除掉，再利用逆傅里叶变换，得到时域内的图像==此时我们能够看到，图像的边界被保留了下来，也就是说，我们能够得到原来图像的轮廓==这种滤波，称为高通滤波，意指只剩下高频信号，使其通过滤波器。

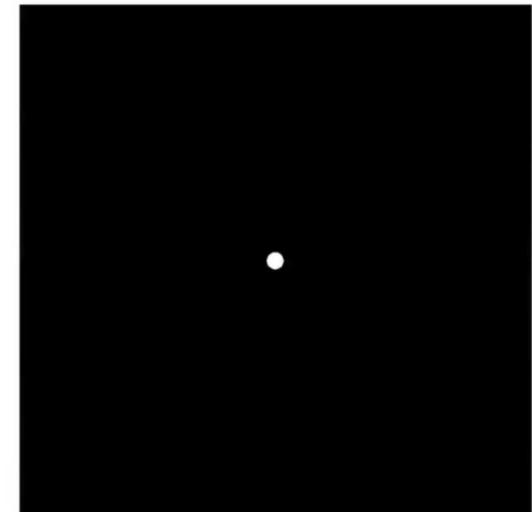
那在数学上如何理解边界呢？我们这么认为==图像发生了剧烈变化的地方，就是边界，举



一例：

图中标识的区域中，上部为模糊的背景，下部为衣料，这里就有剧烈的变化，因此我们将发生变化的地方，视为边界。按照我们的思路，发生了剧烈变换的地方，就是高频信号的产生处，因此，我们使用高通滤波时，保留下来的高频信号，使得图像的边界得以保留。

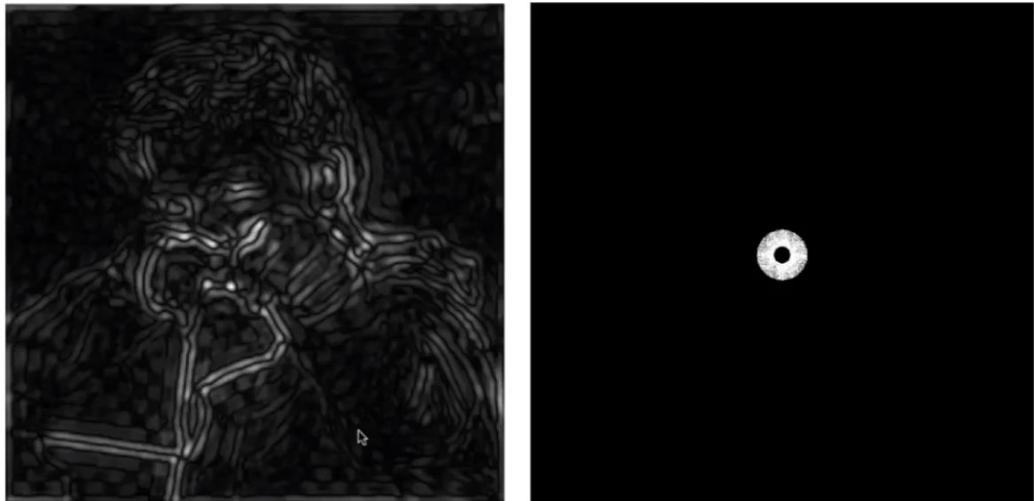
Filter Out High Frequencies (Blur)



Low-pass filter

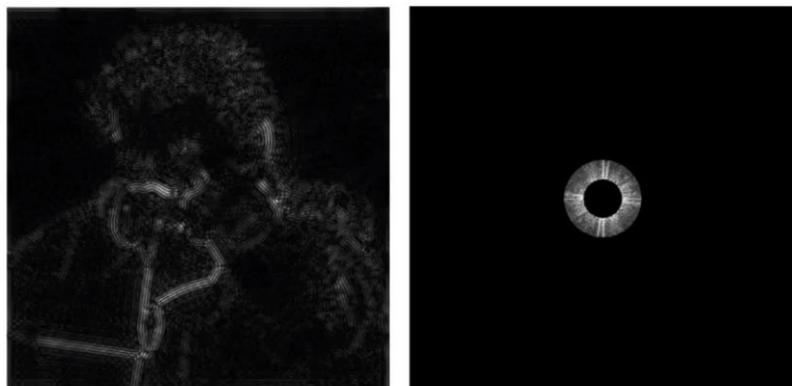
我们将频域内的高频信号滤除掉，保留中心的低频信号，就是所谓的低通滤波，使得图像的边界被抹除，整个图像变得模糊。

Filter Out Low and High Frequencies



我们保留某个不特殊的频段的信息，可以提取到一些不是很明显的边界特征

Filter Out Low and High Frequencies



提取的更高频一些，可以得到更明显的边界特征。这些内容隶属于数字图像处理课程的内容。

Filtering = Convolution
(= Averaging)

Convolution

Signal

1	3	5	3	7	1	3	8	6	4
---	---	---	---	---	---	---	---	---	---

Filter

1/4	1/2	1/4
-----	-----	-----

$$1 \times (1/4) + 3 \times (1/2) + 5 \times (1/4) = 3$$

Result

	3							
--	---	--	--	--	--	--	--	--

| 卷积操作:

举一例，我们用卷积操作来处理一个一维数组，滤波器如图所示，三个格子内有值，我们将信号和滤波器对应的位置做点乘，得到的值将其置于滤波器的中间位置，如图所示。且滤波器每次移动一格，将新得到的结果继续写入。

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

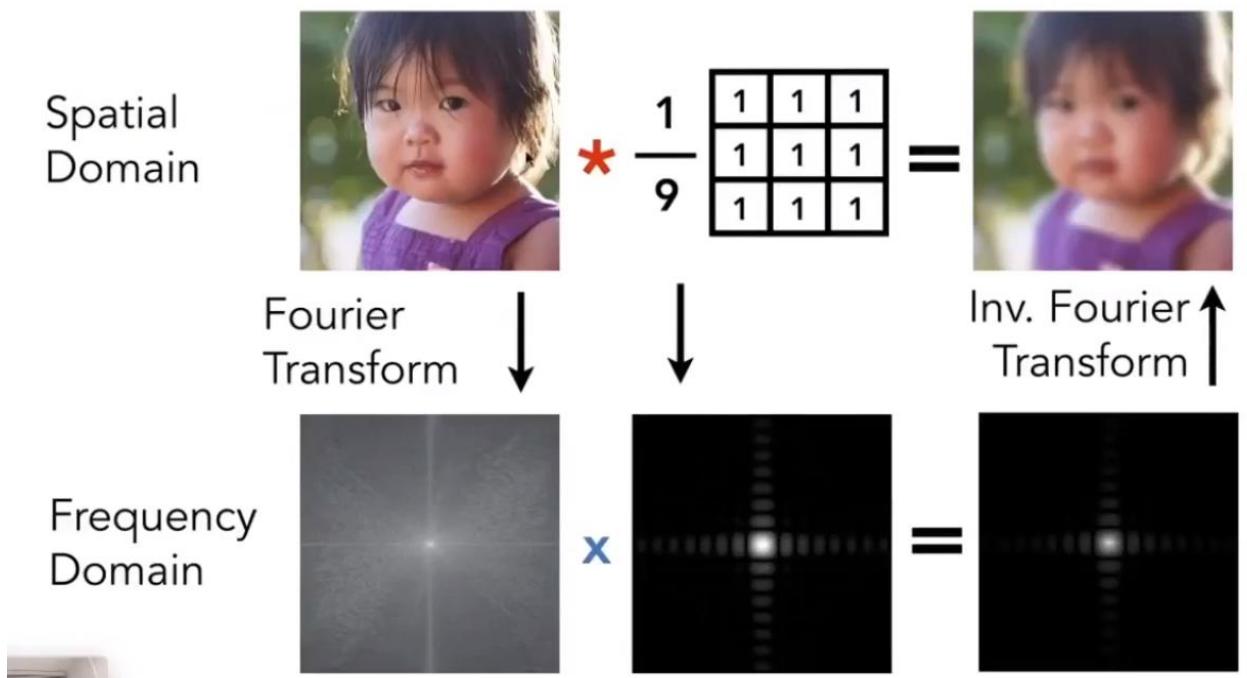
Option 1:

- 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)

几个关于卷积的结论：我们在时域内的卷积操作，在频域内就是乘积操作



我们对时域图像做卷积操作，就是将每个像素周围 3×3 取平均，写回原来的像素位置，得到了模糊的图像。

而如果我们对频域图像做变换，就是将其频域图像乘上一个卷积和，得到另一个频域图像，再对其做逆傅里叶变换，就可以得到模糊的图像。实际上我们就是做了类似于低通滤波的操作，得到了模糊化的图像。

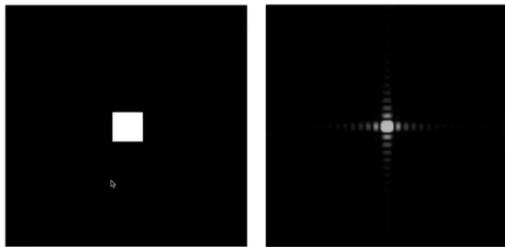
$$\frac{1}{9} \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

Example: 3×3 box filter

下面我们来观察这个滤波器，滤波器需要乘一个 $1/9$ ，从而保证中间的像素点的颜色不发生改变，否则中间的像素点就是周围九个像素点的和了，使得亮度极大的提高



这是时域上的一个小盒子对应的频域上的图像，如果我们对时域图像放大缩小，会对频域图像产生什么样的影响呢？



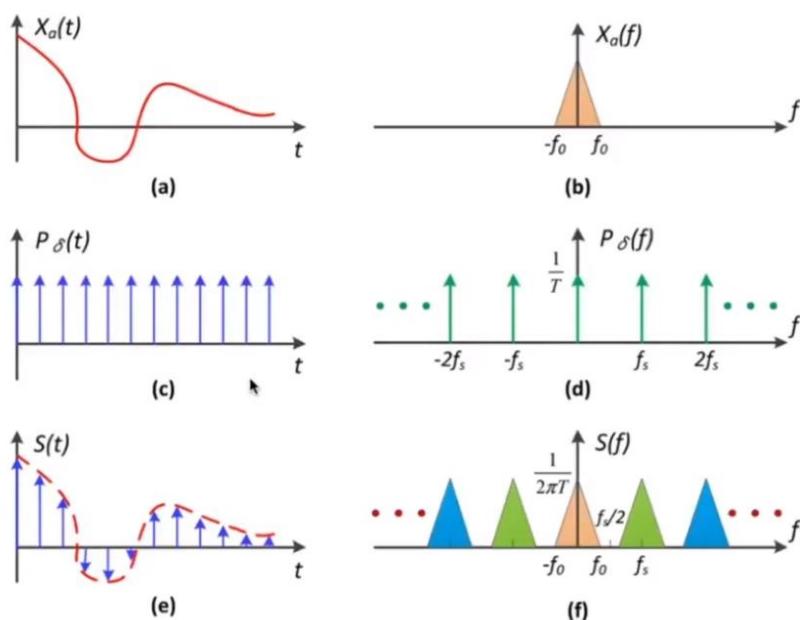
我们可以看到，时域内的图像变大，反而频域的图像缩小了。我们可以这么理解：用更大的盒子来滤波，那得到的平均值会更小，反映到频域内就是低通滤波的更厉害，滤掉的高频信号更多，那需要乘的那个滤波器就更小了，如图二所示，得到的时域图像就更模糊。

而如果时域的图像更小，我们考虑一个极端的情况，这个盒子比一个像素还要小，那他滤波的作用就不存在了，那在频域内的图像会很大。

总的来说，低通滤波越厉害，盒子越大，图像越模糊。而低通滤波作用越小，盒子就越小，图像就模糊的轻。

Sampling = Repeating Frequency Contents

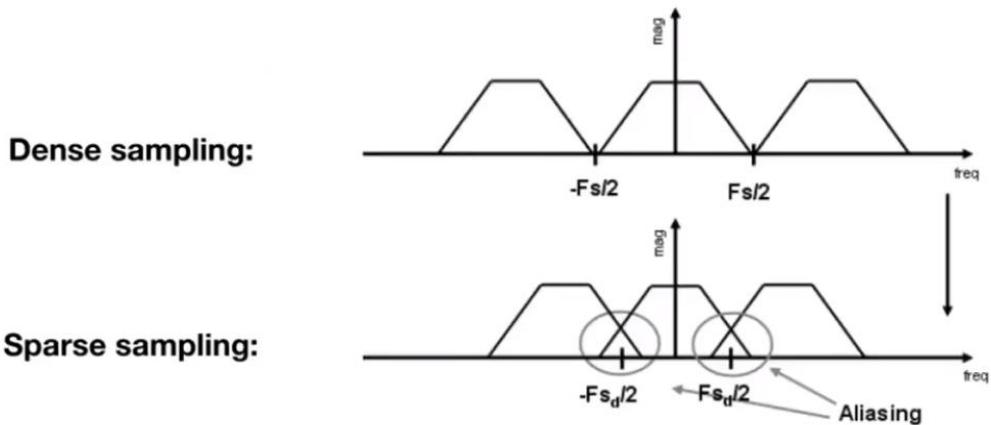
采样：重复频率的内容



上图分别为时域和频域采样的展示：

其中，左边为时域的操作， a 为我们要采样的对象，而 c 为冲击函数，用它来做采样的操作，将两者乘起来，可以得到 e 的结果，也就是离散化的 a 。这个过程就叫做采样。

另一边，为频域的操作，我们知道，在时域内的乘积就等于频域内的卷积。 B 为 a 在频域内的形式，而 d 为冲击函数的频域形式，两者做卷积操作，得到了 f 的结果。我们可以看到，采样其实就是将原来的频域内的函数复制了许多份，因此，我们略去中间的过程，可以得到一个结论====所谓采样，就是不断地重复原始信号的频谱。



在正常情况下，我们得到的采样信号间隔应该是恰好的，如上图 1 所示，而在采样间隔太大时，反映到频域就是信号之间的间隔太小（时域和频域的性质有很多相反的地方），使得信号混合在一起，就是走样

How Can We Reduce Aliasing Error?

Option 1: Increase sampling rate

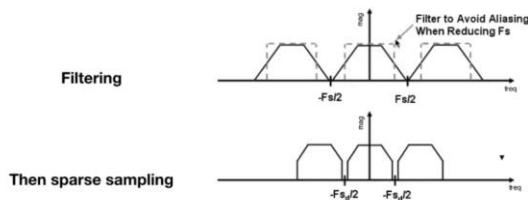
- Essentially increasing the distance between replicas in the Fourier domain
- Higher resolution displays, sensors, framebuffers...
- But: costly & may need very high resolution

反走样的方法：(1) 提高采样率

Option 2: Antialiasing

- Making Fourier contents “narrower” before repeating
- i.e. **Filtering out high frequencies before sampling**

(2) 低通滤波模糊后再采样

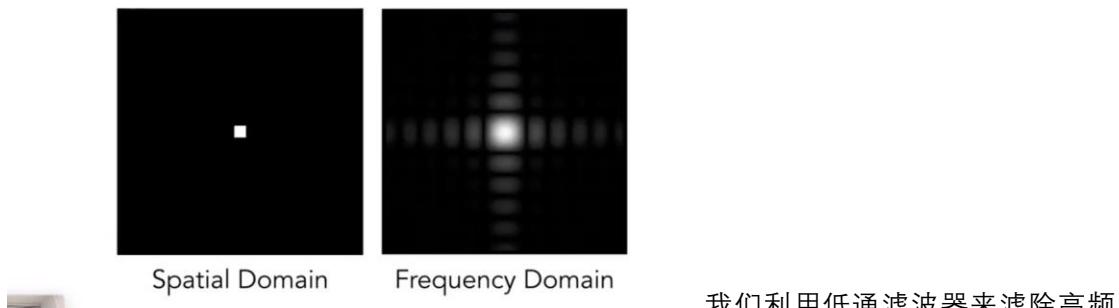


我们对其原理做出解释：首先低通滤波，如

图一，就可以将频谱中高频的部分去掉，我们再按照原来的采样频率做采样，由于频谱的缩小，使得频谱混合的现象不再出现，就起到了反走样的目的。

那么，这个滤波的操作具体是怎样实施的？

A 1 pixel-width box filter (low pass, blurring)



我们利用低通滤波器来滤除高频

信号，从而模糊图形的边界，而这个低通滤波应该如何选取？我们就选用一个像素大小的低通滤波器，来与原图形做卷积操作。

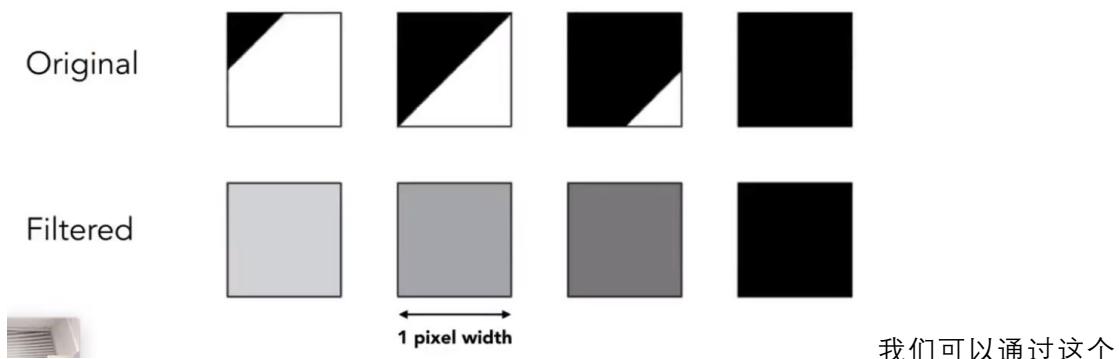
Solution:

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

我们用一个像素的低通滤波器来滤波原来的函数，我们所说的滤波、模糊、平均，其实都是一个概念。

然后采样每个像素中心。

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

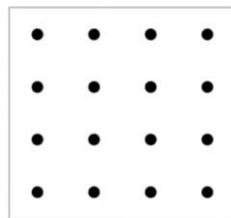


我们可以通过这个

对比来直观的得到滤波的效果，三角形覆盖的面积越大，则滤波后的颜色更深，滤波后一个像素的值就等于三角形所覆盖的区域。

Antialiasing By Supersampling (MSAA)

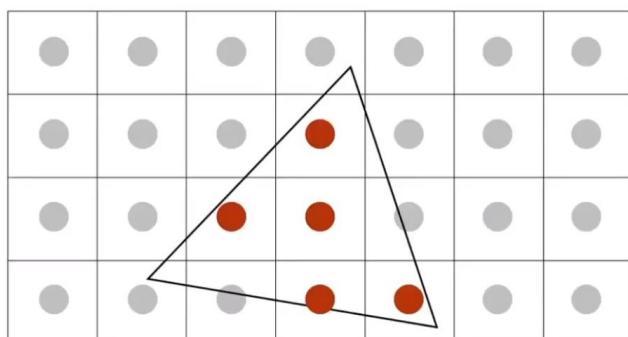
Approximate the effect of the 1-pixel box filter by sampling multiple locations within a pixel and averaging their values:



4x4 supersampling

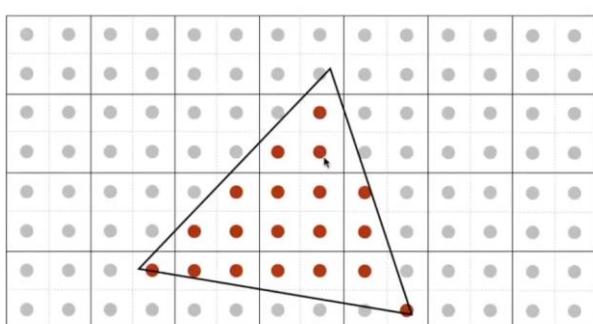
实际上，我们判断一个

像素内三角形覆盖的范围是不容易的，因此，我们采用近似的方法来判断其范围。这里使用了超采样技术====将一个像素再划分为更小的像素，上图中将一个像素又划分为了一个 4×4 的像素点阵。我们判断三角形覆盖的这些小像素的数量，从而近似的判断三角形的覆盖范围。下面对整个过程做一下分析



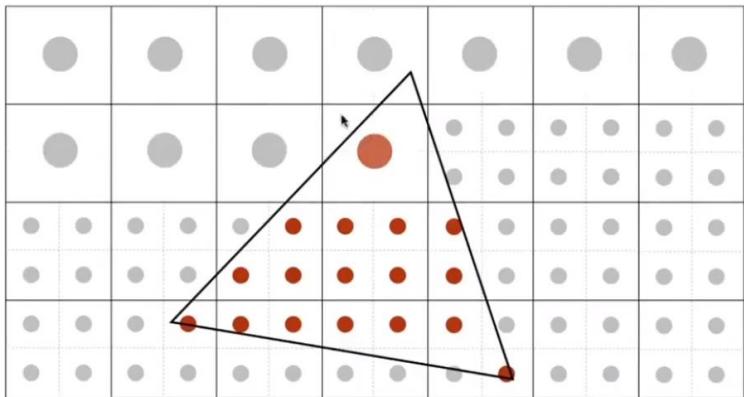
这是没有采用超采样技术时的采样

结果。

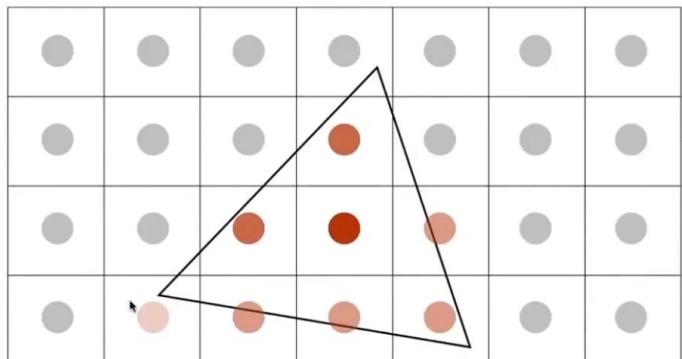


2x2 supersampling

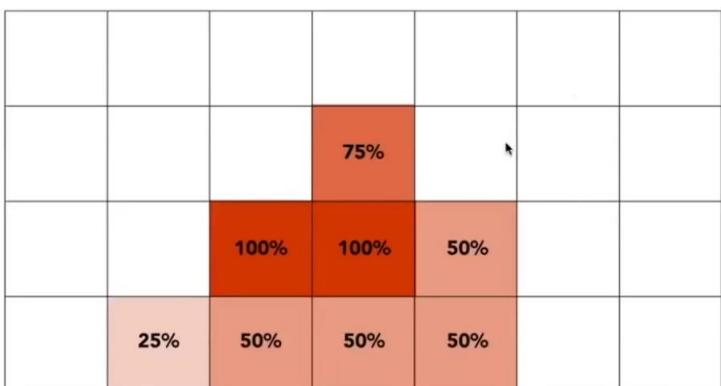
采用超采样技术后的采样结果。



由于三个像素被覆盖, 我们认为原来的一个大像素的 75% 被三角形所覆盖。



最后得到的覆盖结果就如图所示。



这个时候我们完成了模糊操作这一步。接下来在做采样操作, 由于模糊操作已经完成了, 所以采样的过程很简单, 每个像素的颜色是一致的, 只需要采集中心点的颜色即可。

Milestones (personal idea)

- FXAA (Fast Approximate AA)
- TAA (Temporal AA)

其他的两种广泛应用的抗锯齿

方法

Super resolution / super sampling

- From low resolution to high resolution
- Essentially still “not enough samples” problem



- DLSS (Deep Learning Super Sampling)

超分辨率技术, 高分辨率的

图采样不够, 使得分辨率降低, 想要恢复高分辨率的图片, 可以通过深度学习的方法来猜测原来的图形, 来补充原来的图形从而提高分辨率