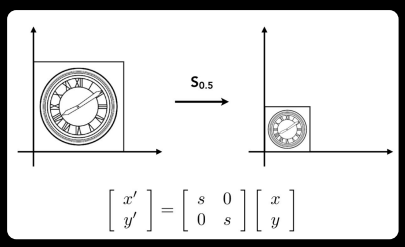
Modeling Transformation

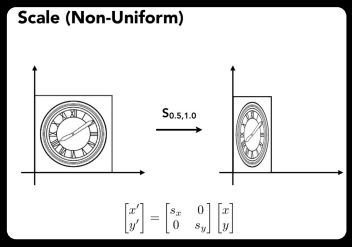
Viewing Transformation

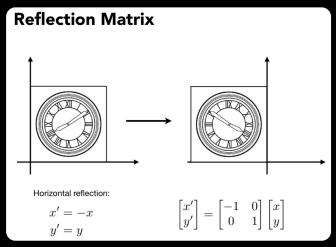
# 2D Transformation

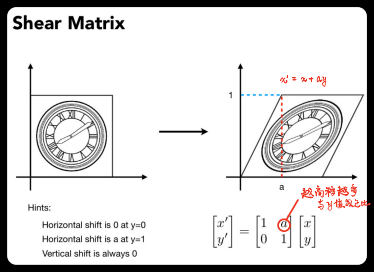
## Linear Transformation

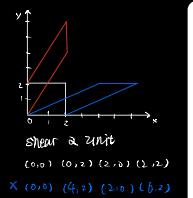
x ⇒ −x

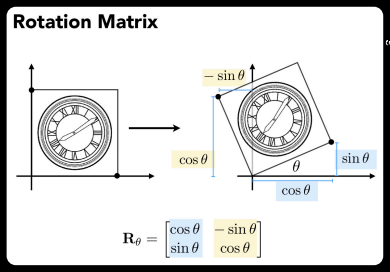


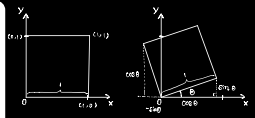



Δ 旋转永远是以原点为中心

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

$R_\theta^T = R_\theta^{-1}$ <by definition> 转置 = 逆 → 正交矩阵 Orthogonal Matrix

$R_\theta \times R_\theta^{-1} = I \quad R_\theta^T \Rightarrow R_{\theta\,ij} = R_{\theta\,ji}$

## Linear Transformation Conclusion

### Linear Transforms = Matrices

(of the same dimension)

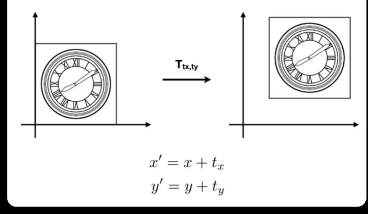
$$x' = a\,x + b\,y$$
$$y' = c\,x + d\,y$$

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

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

## Homogeneous Coordinates 齐次坐标

### Translation??




### Why 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!)

- 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?)



### Solution: Homogenous Coordinates

Add a third coordinate (*w-coordinate*)

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

通过增加一个坐标来使非特殊变换得以操作

↳ 向量具有平移不变性

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?

### 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:

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

(transpose)

Always 0 e.g. a 2D translation



仿射变换

two matrices

one matrix

### 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

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

### Homogenous Coordinates

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

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

In homogenous coordinates,

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

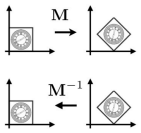
画家中点

Inverse Transformation

## Inverse Transform

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

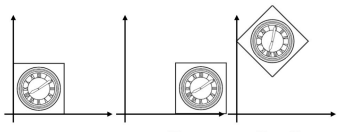
$\mathbf{M}^{-1}$ is the inverse of transform $\mathbf{M}$ in both a matrix and geometric sense




# Composing Transform

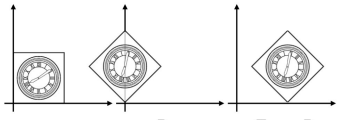
ORDER MATTERS !

## Translate Then Rotate?






## Rotate Then Translate




## 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:

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

TRANSLATION ROTATE

## Composing Transforms

Sequence of affine transforms $A_1$, $A_2$, $A_3$, ...

- Compose by matrix multiplication
  - Very important for performance!

$$A_n(\dots A_2(A_1(\mathbf{x}))) = \mathbf{A}_n \cdots \mathbf{A}_2 \cdot \mathbf{A}_1 \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

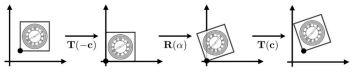
Pre-multiply *n* matrices to obtain a single matrix representing combined transform

# Decomposing Transformation

## Decomposing Complex Transforms

How to rotate around a given point c?

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




Matrix representation?

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

# 3D Transformation

## 3D Transformations

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)

Homogeneous Coordinate in 3D

## 3D Transformations

Use 4×4 matrices for affine transformations

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

What's the order?
Linear Transform first or Translation first?

① ②

# Scale and Translation

## 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

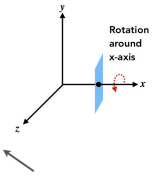
$$\mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

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

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

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



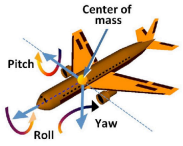

Anything strange about $R_y$?

Z × x ⇒ y
not counter-clockwisely, x × z so flip x and z (T)

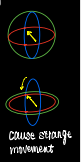
## Compose any 3D rotation from $\mathbf{R}_x, \mathbf{R}_y, \mathbf{R}_z$?

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

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




gimble - lock



cause strange movement

## Rodrigues' Rotation Formula

Rodrigues

Rotation by angle α around axis n  ∝ y n z

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

### How to prove this magic formula?

Check out the supplementary material on the course website!

# Viewing Transformation

## View / camera transformation

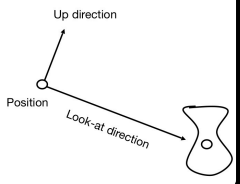
- What is view transformation?
- Think about how to take a photo
  - Find a good place and arrange people (**model** transformation)
  - Find a good "angle" to put the camera (**view** transformation)
  - Cheese! (**projection** transformation)  消去像素变成二维图像

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

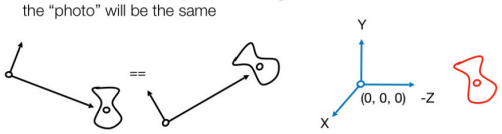
MVP
渲染过程

## View / Camera 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)




- 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

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

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

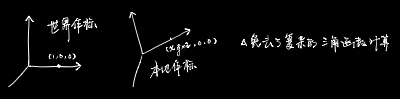
  - Rotate g to -Z, t to Y, (g x t) To X (gaze, up)
  - Consider its inverse rotation: X to (g x t), Y to t, Z to -g

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

WHY?

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

△ 旋转矩阵的逆就是它的转置 <正交矩阵的性质>
inverse transpose




△ 免去了复杂的三维画板计算

将摄像机从环境变换至原点位置
视图变换

- Summary
  - Transform objects together with the camera
  - Until camera's at the origin, up at Y, look at -Z
- Also known as ModelView Transformation
- But why do we need this?
  - For projection transformation!

# Projection Transformation

- Projection in Computer Graphics
  - 3D to 2D
  - Orthographic projection
  - Perspective projection

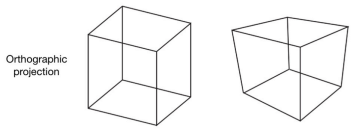
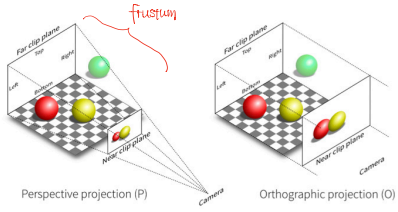
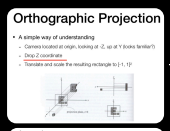



Fig. 7.1 from *Fundamentals of Computer Graphics, 4th Edition*

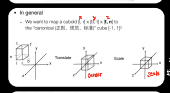
- Perspective projection vs. orthographic projection



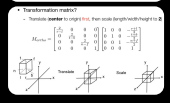

Projection
- Orthographic
- Perspective

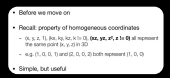


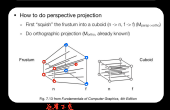

方便计算




看向 -Z
远小近大

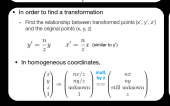


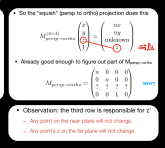





相似三角




为不变点方面提条件推算

Premise ① NEAR: $\begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ n^2 \\ n \end{pmatrix}$ the 3rd row must be (0 0 A B)

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

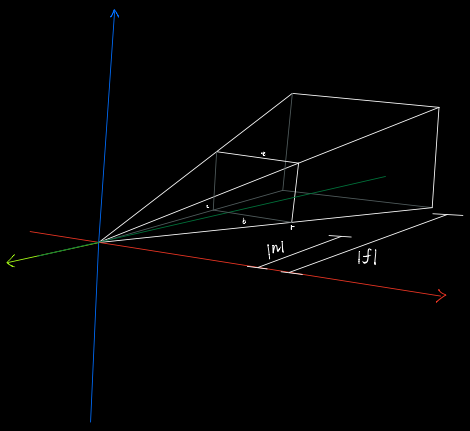
Premise ② CENTER OF FAR: $\begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ f^2 \\ f \end{pmatrix}$

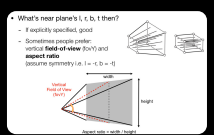
Therefore $(0\ 0\ A\ B)\begin{pmatrix} x \\ y \\ f \\ 1 \end{pmatrix} = Af + B = f^2$

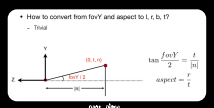
$$\begin{cases} An + B = n^2 \\ Af + B = f^2 \end{cases} \Rightarrow \begin{cases} A = n + f \\ B = -nf \end{cases}$

☆ n, f 均为坐标，有正负

$$M_{persp\to ortho} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

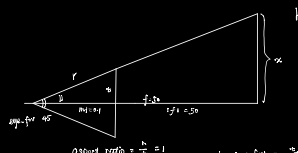


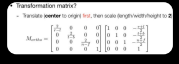

实际中

可视角度 + 视锥比 即可推出所有参数

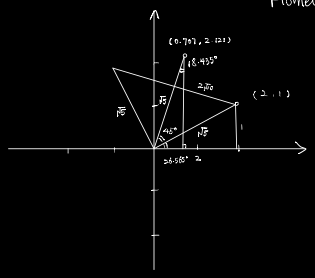


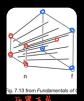

$\tan(\frac{\alpha}{2}) = \frac{t}{|n|} = \frac{\alpha}{|f|}$

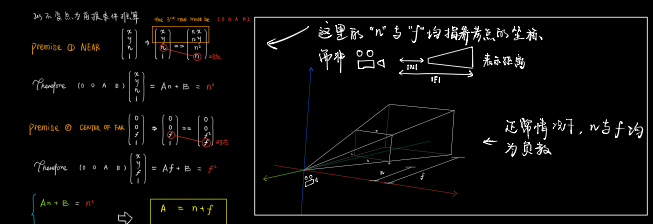
$\tan(\frac{\text{aspect}\cdot\alpha}{2}) =$



# Homework 0 Note



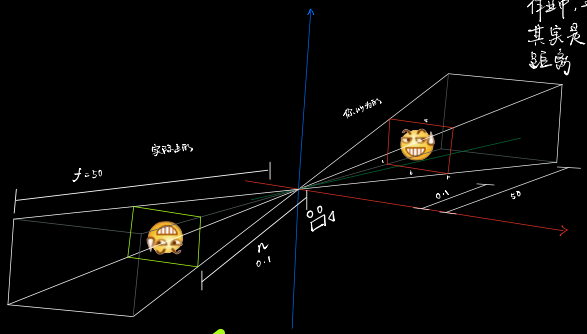






```
while (key != 27) {
    r.clear(rst::Buffers::Color | rst::Buffers::Depth);

    r.set_model(get_model_matrix(axis, angle));
    r.set_view(get_view_matrix(eye_pos));
    r.set_projection(get_projection_matrix(45, 1, 0.1, 50));
```

作业中，老师给的 "n" 和 "f" 其实是 OO 与近远平面之距离




与课上情况不同

因此，直接套用公式就会出现颠倒

解决方法

①

```
Eigen::Matrix4f perspective = Eigen::Matrix4f::Identity();
perspective << zNear,    0,         0,              0,
               0,        zNear,     0,              0,
               0,        0,         zNear + zFar,   -(zNear * zFar),
               0,        0,         1,              0;
                                    改为-1
```

改写 perspective to ortho 矩阵形值
以达到手动模符号的效果

②

```
while (key != 27) {
    r.clear(rst::Buffers::Color | rst::Buffers::Depth);

    r.set_model(get_model_matrix(axis, angle));
    r.set_view(get_view_matrix(eye_pos));
    r.set_projection(get_projection_matrix(45, 1, -0.1, -50));
```

改为负数输入

修改输入以使程序与课程情况一致

```
// calculate the sizes of the frustum
eye_fov         = (eye_fov / 180) * MY_PI;
float yTop      =  abs(zNear) * tan(eye_fov / 2);
float yBottom   = -yTop;
float xRight    =  yTop * aspect_ratio;
float xLeft     = -xRight;
```

注意，若修改了输入，此处要加绝对值