

# CS 4204 Computer Graphics

---

## *2D Transformations*

*Yong Cao*  
*Virginia Tech*

***References:***

“Introduction to Computer Graphics” course notes by Doug Bowman  
Interactive Computer Graphics, Fourth Edition, Ed Angle

# Transformations

## *What are they?*

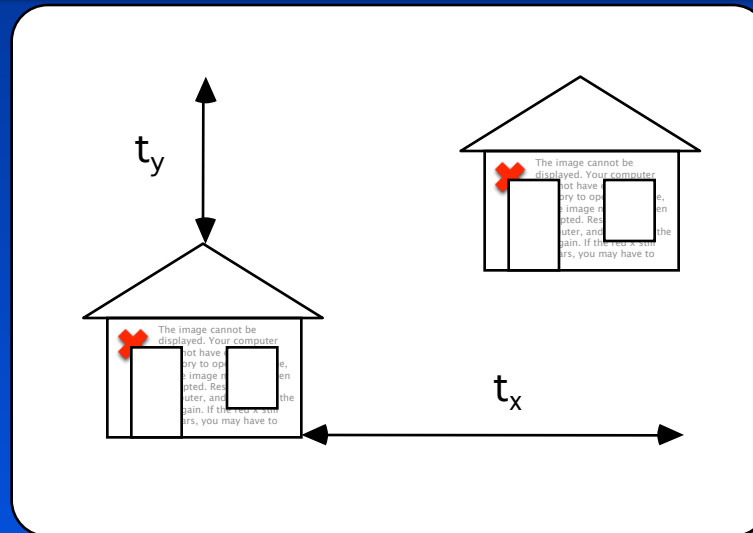
- changing something to something else via rules
- mathematics: mapping between values in a range set and domain set (function/relation)
- geometric: translate, rotate, scale, shear,...

## *Why are they important to graphics?*

- moving objects on screen / in space
- mapping from model space to world space to camera space to screen space
- specifying parent/child relationships
- ...

# Translation

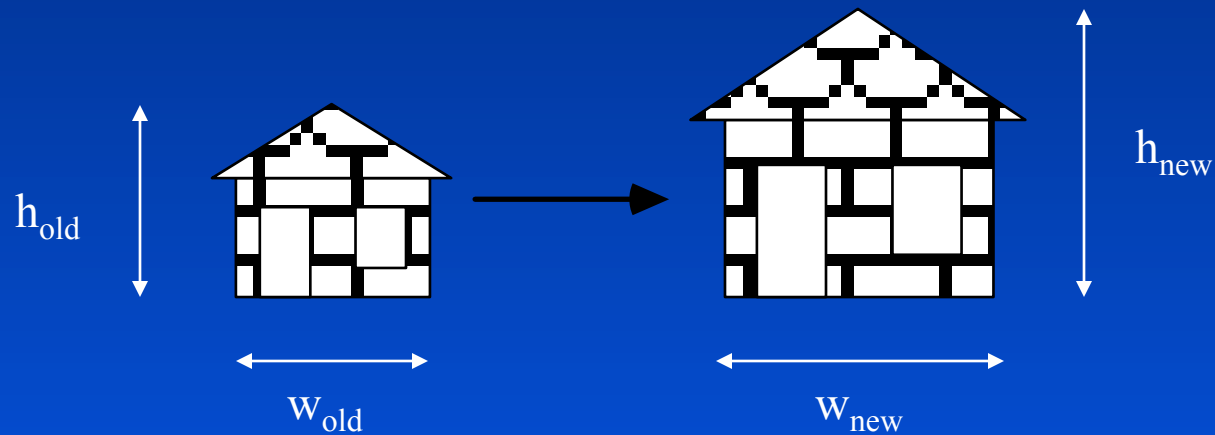
*Moving an object is called a translation. We translate a point by adding to the  $x$  and  $y$  coordinates, respectively, the amount the point should be shifted in the  $x$  and  $y$  directions. We translate an object by translating each vertex in the object.*



$$x_{\text{new}} = x_{\text{old}} + t_x; y_{\text{new}} = y_{\text{old}} + t_y$$

# Scaling

*Changing the size of an object is called a scale. We scale an object by scaling the x and y coordinates of each vertex in the object.*



$$S_x = W_{new} / W_{old}$$

$$S_y = h_{new} / h_{old}$$

$$X_{new} = S_x X_{old}$$

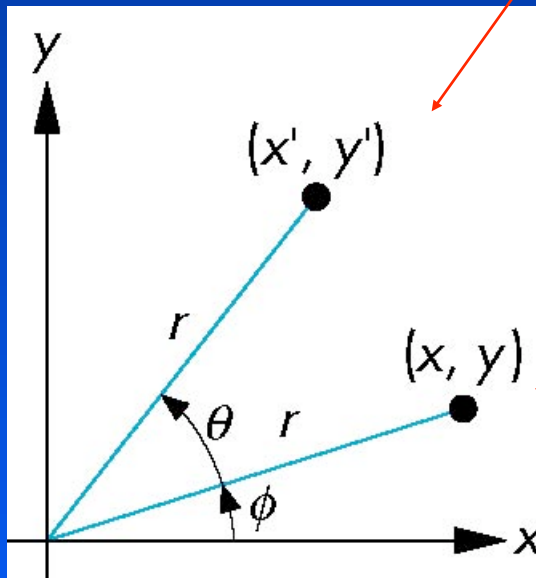
$$Y_{new} = S_y Y_{old}$$



# Rotation about the origin

**Consider rotation about the origin by  $q$  degrees**

- radius stays the same, angle increases by  $q$



$$x' = r \cos (\phi + \theta)$$

$$y' = r \sin (\phi + \theta)$$

$$\begin{aligned} x' &= x \cos \theta - y \sin \theta \\ y' &= x \sin \theta + y \cos \theta \end{aligned}$$

$$\begin{aligned} x &= r \cos \phi \\ y &= r \sin \phi \end{aligned}$$

# Rotation about the origin (cont.)

*From the double angle formulas:*

$$\sin (A + B) = \sin A \cos B + \cos A \sin B$$

$$\cos (A+B) = \cos A \cos B - \sin A \sin B$$

# Transformations as matrices

## Scale:

$$x_{new} = s_x x_{old}$$

$$y_{new} = s_y y_{old}$$

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

## Rotation:

$$x_{new} = x_{old} \cos \theta - y_{old} \sin \theta$$

$$y_{new} = x_{old} \sin \theta + y_{old} \cos \theta$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \end{bmatrix}$$

## Translation:

$$x_{new} = x_{old} + t_x$$

$$y_{new} = y_{old} + t_y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \end{bmatrix}$$

# Homogeneous Coordinates

*In order to represent a translation as a matrix multiplication operation we use 3 x 3 matrices and pad our points to become 3 x 1 matrices. This coordinate system (using three values to represent a 2D point) is called homogeneous coordinates.*

$$P_{(x,y)} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$R_{\theta} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$S_{x,y} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$T_{x,y} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

# Composite Transformations

*Suppose we wished to perform multiple transformations on a point:*

$$P_2 = T_{3,1} P_1$$

$$P_3 = S_{2,2} P_2$$

$$P_4 = R_{30} P_3$$

---

$$M = R_{30} S_{2,2} T_{3,1}$$

$$P_4 = M P_1$$

Remember:

- Matrix multiplication is associative, not commutative!
- Transform matrices must be **pre-multiplied**
- The first transformation you want to perform will be at the far right, just before the point

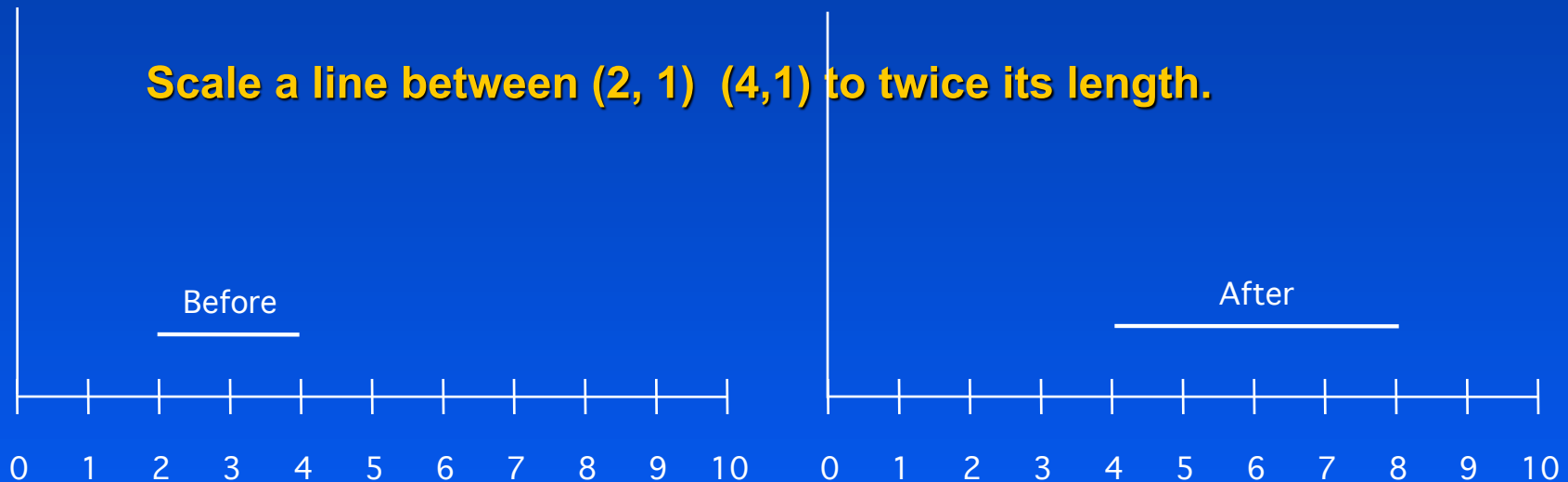
# Composite Transformations - Scaling

*Given our three basic transformations we can create other transformations.*

*Scaling with a fixed point*

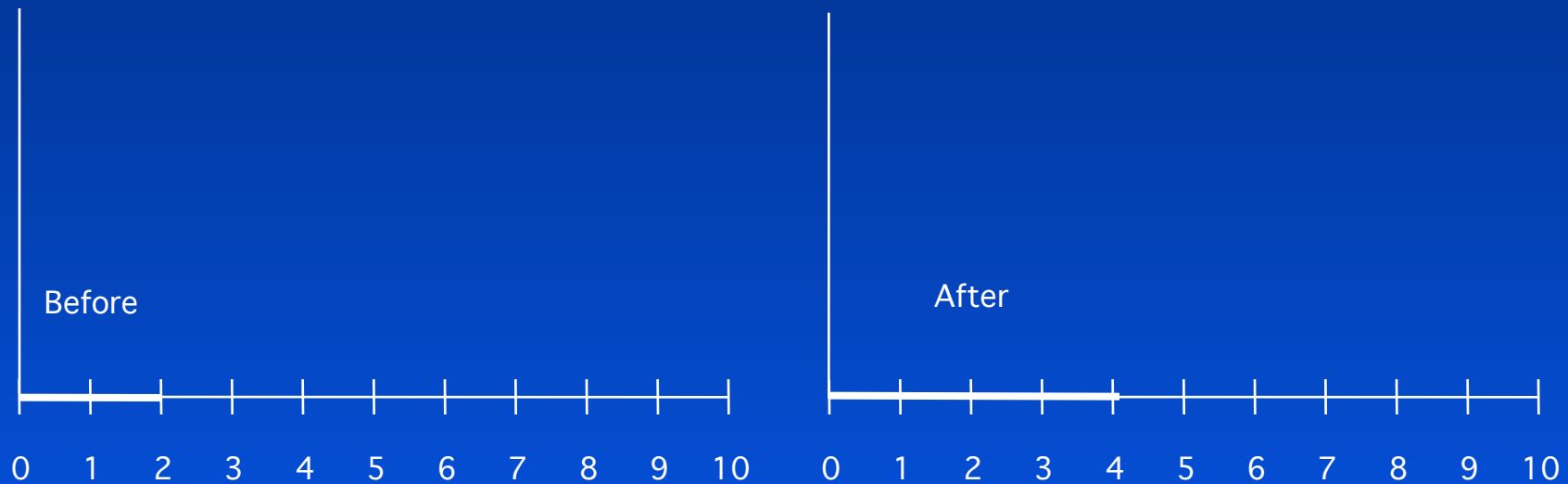
*A problem with the scale transformation is that it also moves the object being scaled.*

**Scale a line between  $(2, 1)$   $(4, 1)$  to twice its length.**



# Composite Transforms - Scaling (cont.)

*If we scale a line between  $(0,0)$  &  $(2,0)$  to twice its length, the left-hand endpoint does not move.*



$(0,0)$  is known as a **fixed point** for the basic scaling transformation.  
We can use composite transformations to create a scale transformation with different fixed points.

# Fixed Point Scaling

*Scale by 2 with fixed point = (2,1)*

*Translate the point (2,1) to the origin*

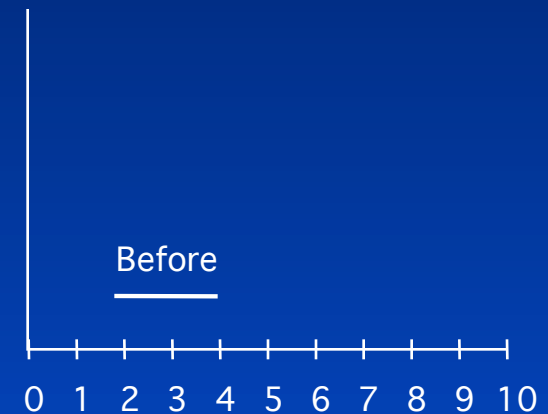
*Scale by 2*

*Translate origin to point (2,1)*

$$\begin{matrix} \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} & = & \begin{bmatrix} 2 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ T_{2,1} & S_{2,1} & T_{-2,-1} & & C \end{matrix}$$

$$\begin{matrix} \begin{bmatrix} 2 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} & = & \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} \\ C & & & \end{matrix}$$

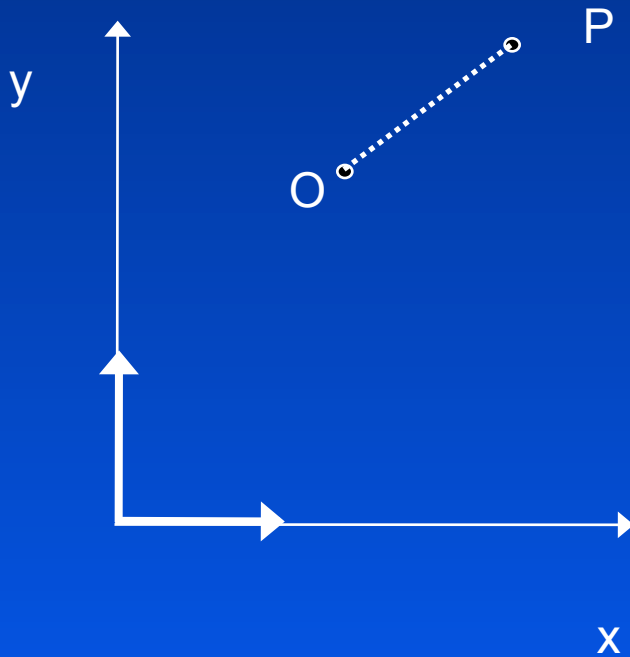
$$\begin{matrix} \begin{bmatrix} 2 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 4 \\ 1 \\ 1 \end{bmatrix} & = & \begin{bmatrix} 6 \\ 1 \\ 1 \end{bmatrix} \\ C & & & \end{matrix}$$



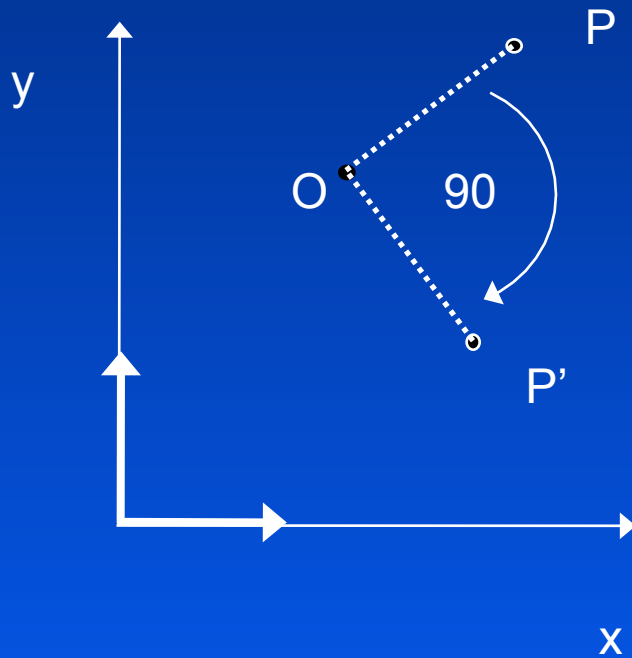


# Example of 2D transformation

*Rotate around an arbitrary point O:*

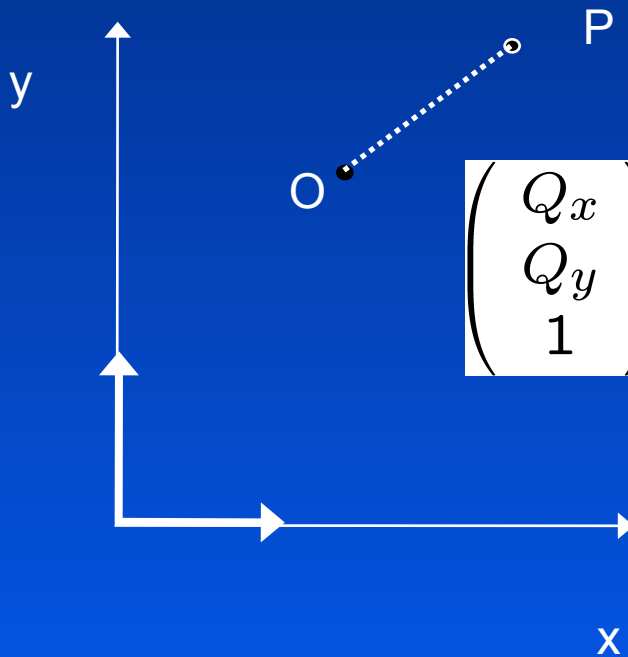


# Rotate around an arbitrary point



# Rotate around an arbitrary point

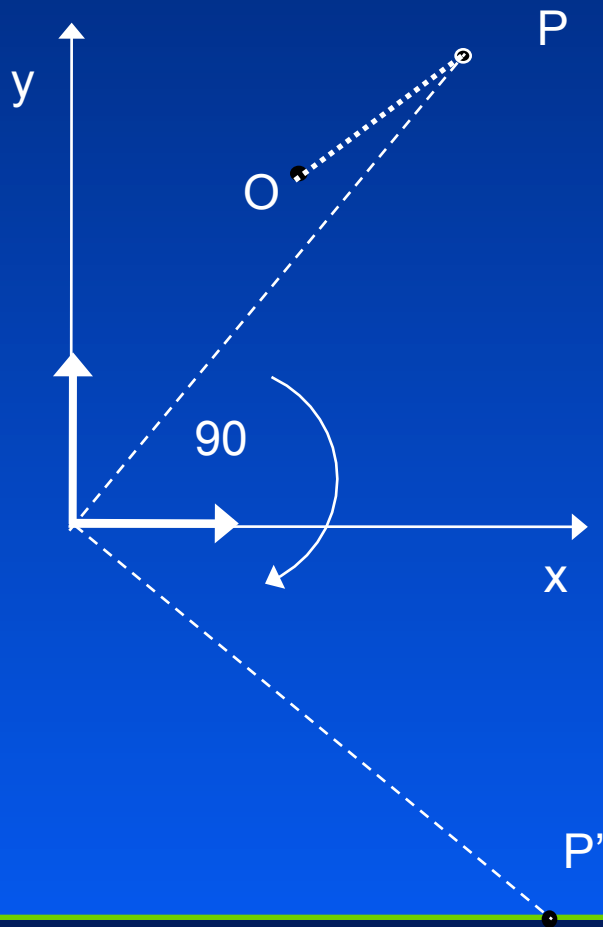
*We know how to rotate around the origin*



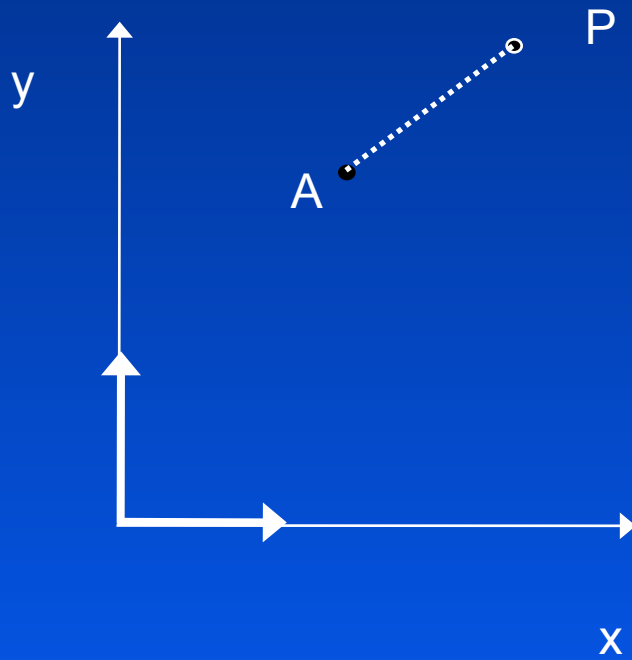
$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

# Rotate around an arbitrary point

*...but that is not what we want to do!*

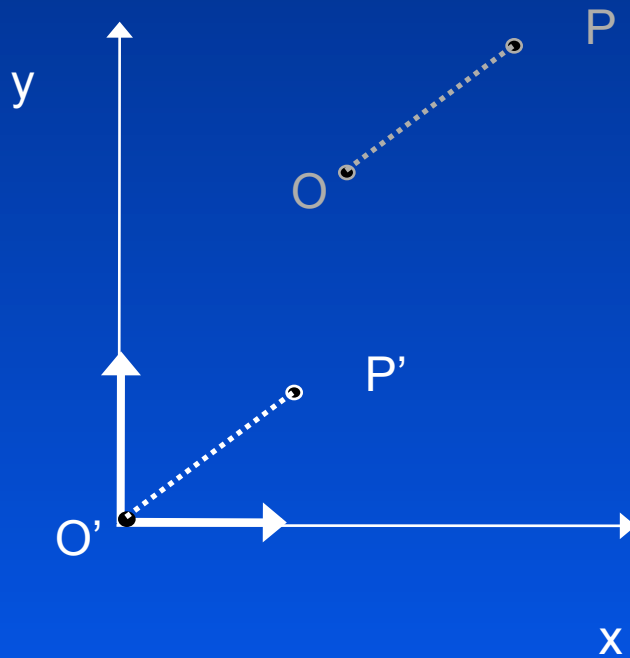


# So what do we do?



# Transform it to a known case

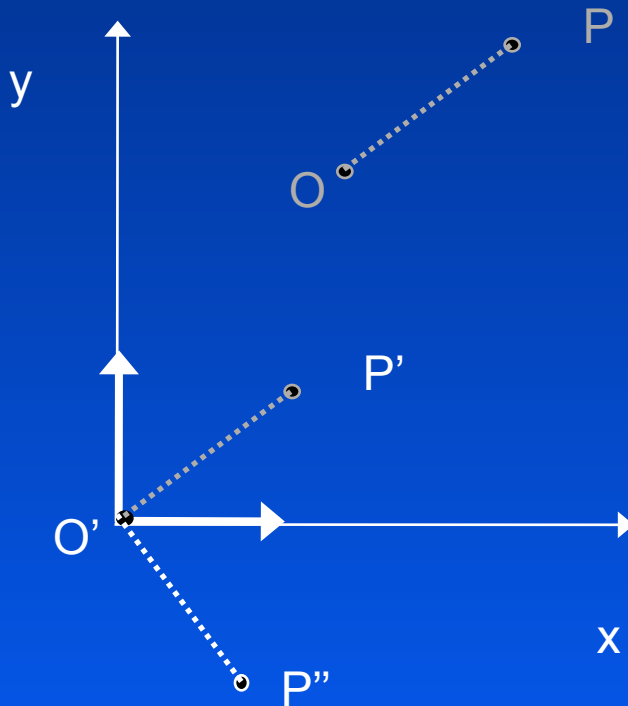
*Translate(-Ox,-Oy)*



# Second step: Rotation

*Translate(-Ox,-Oy)*

*Rotate(-90)*

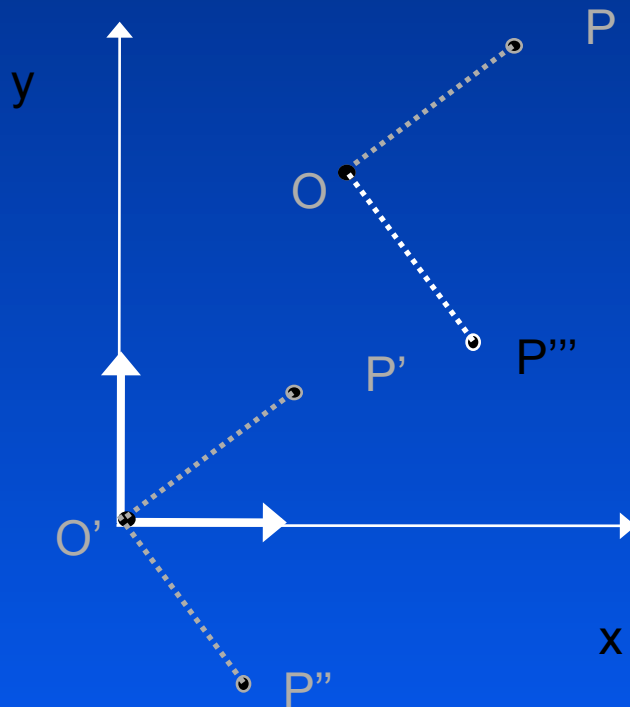


# Final: Put everything back

*Translate(-Ox,-Oy)*

*Rotate(90)*

*Translate(Ox,Oy)*

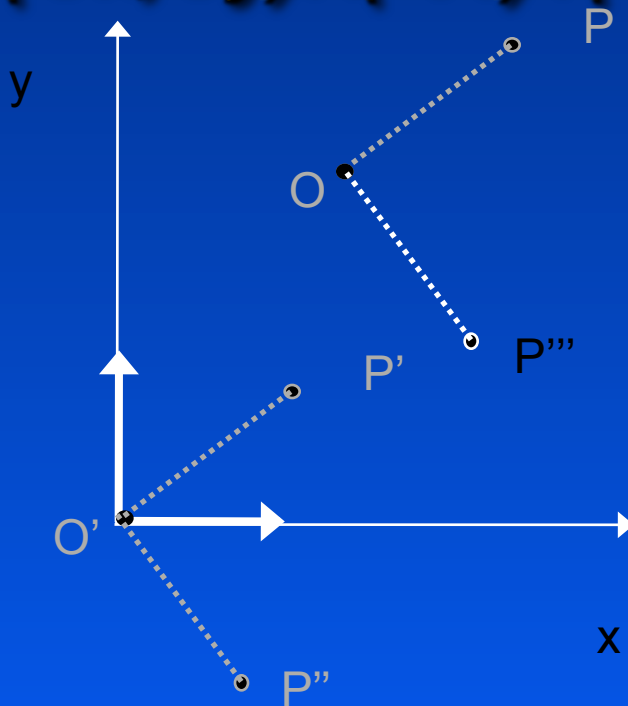




# Rotation about arbitrary point

**IMPORTANT!: Order**

$$M = T(Ox, Oy)R(-90)T(-Ox, -Oy)$$



# Rotation about arbitrary point

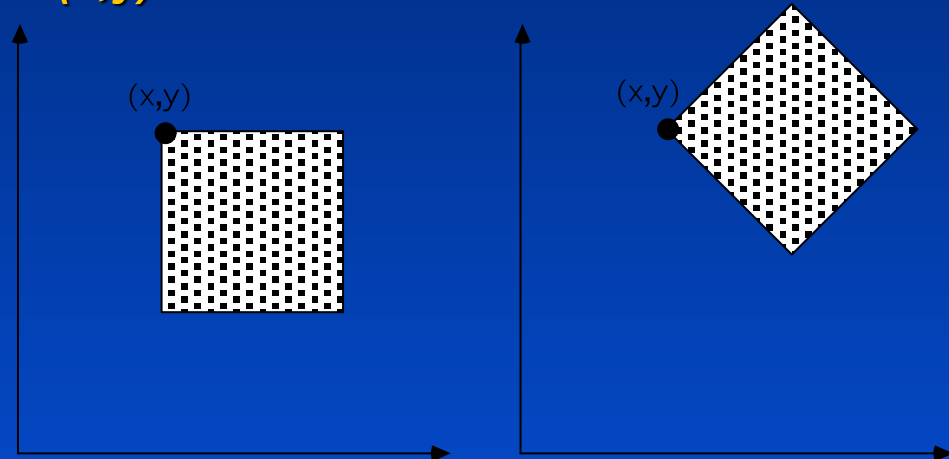
## Rotation of $\theta$ Degrees About Point $(x,y)$

Consist of 3 translations:

1. Translate  $(x,y)$  to origin

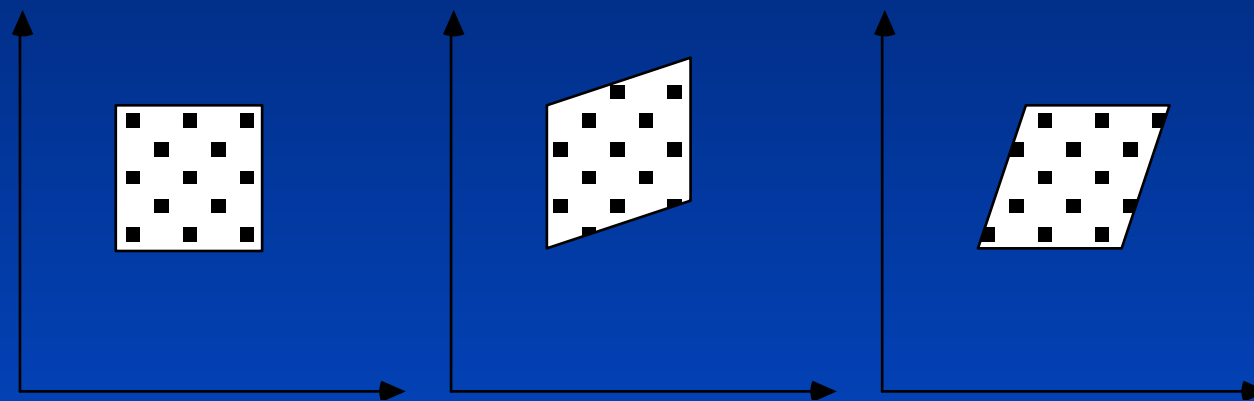
2. Rotate

3. Translate origin to  $(-x,-y)$



$$C = \begin{matrix} \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix} \\ T_{x,y} \end{matrix} \begin{matrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ R_{\theta} \end{matrix} \begin{matrix} \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ 0 & 0 & 1 \end{bmatrix} \\ T_{-x,-y} \end{matrix}$$

# Shears



*Original Data*

*y Shear*

$$\begin{bmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

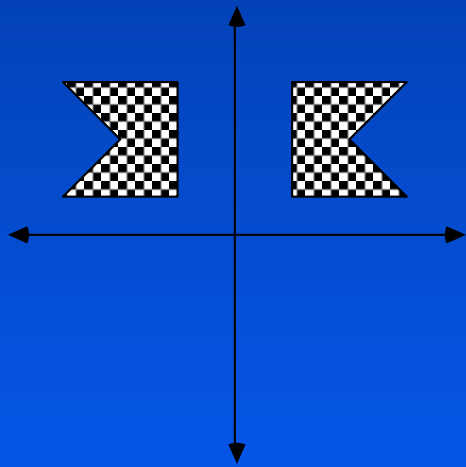
*x Shear*

$$\begin{bmatrix} 1 & b & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

# Reflections

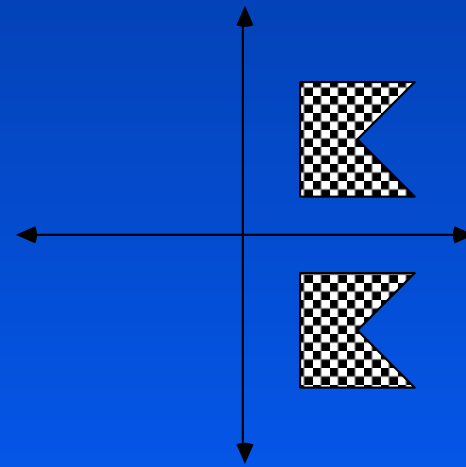
*Reflection about the y-axis*

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



*Reflection about the x-axis*

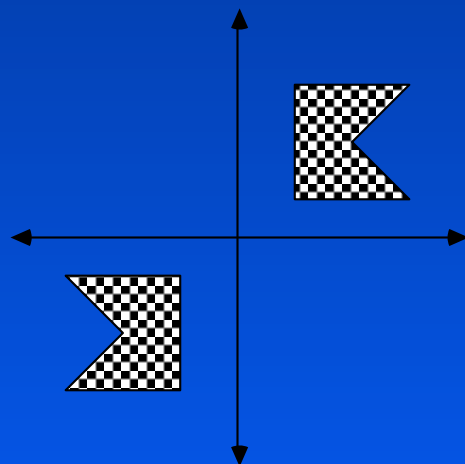
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



# More Reflections

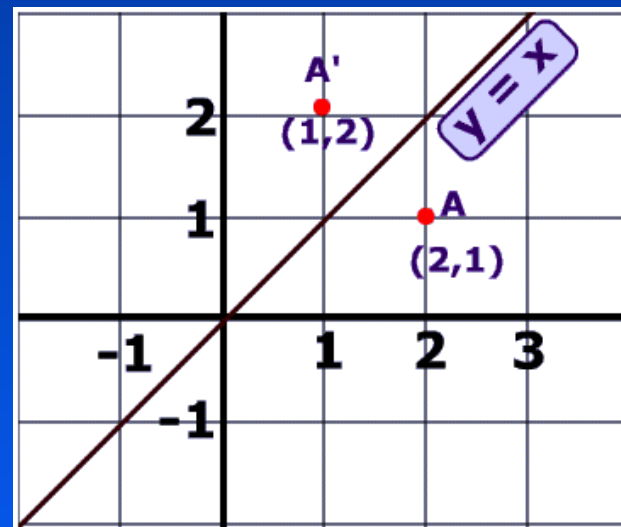
*Reflection about the origin*

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



*Reflection about the line  $y=x$*

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



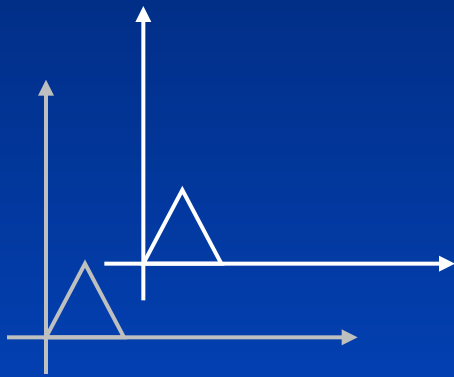
# Transformations as a change in coordinate system

---

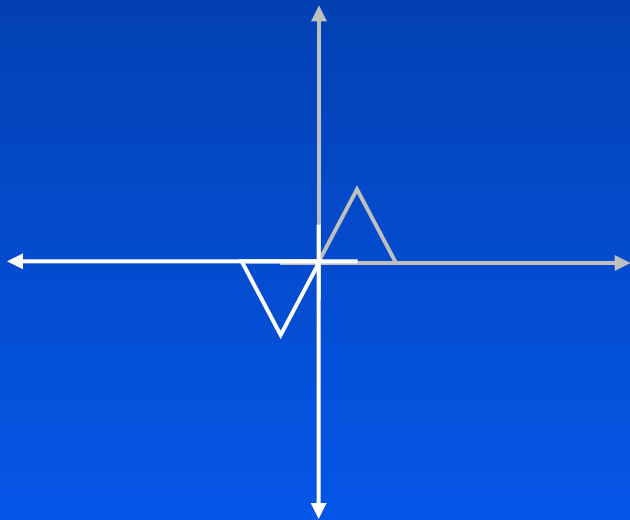
*All transformations we have looked at involve transforming points in a fixed coordinate system (CS).*

*Can also think of them as a transformation of the CS itself*

# Transforming the CS - examples



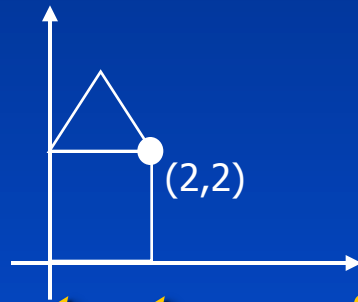
Translate(4,4)



Rotate(180°)

# Why transform the CS?

*Objects often defined in a “natural” or “convenient” CS*



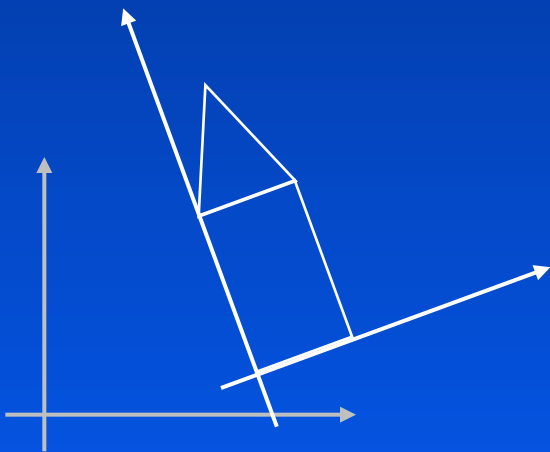
*To draw objects transformed by  $T$ , we could:*

- Transform each vertex by  $T$ , then draw
- Or, draw vertices in a transformed CS



# Drawing in transformed CS

*Tell system once how to draw the object,  
then draw in a transformed CS to transform  
the object*



House drawn in a CS  
that's been translated,  
rotated, and scaled

$$M = S_{x,y} R_d T_{x,y}$$

# Mapping between systems

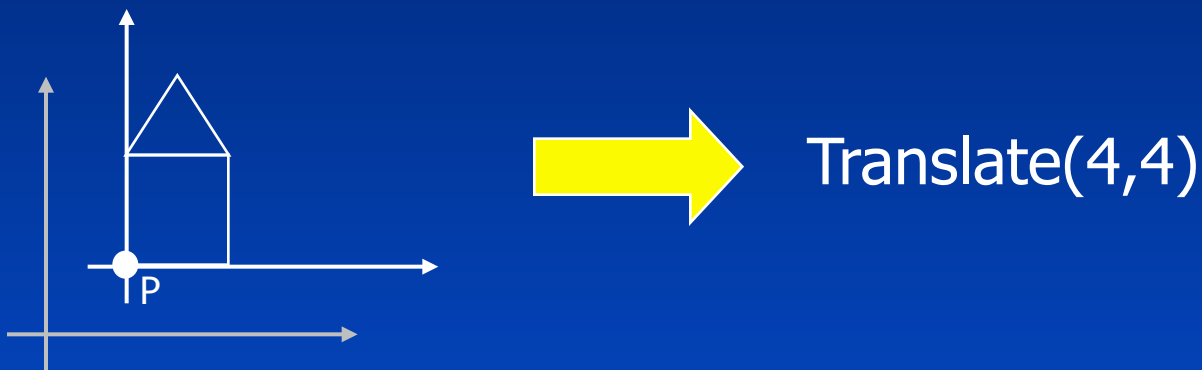
---

## ***Given:***

- The vertices of an object in  $CS_2$
- A transformation matrix  $M$  that transforms  $CS_1$  to  $CS_2$

***What are the coordinates of the object's vertices in  $CS_1$ ?***

# Mapping example



Point P is at (0,0) in the transformed CS ( $CS_2$ ). Where is it in  $CS_1$ ?

Answer: (4,4)

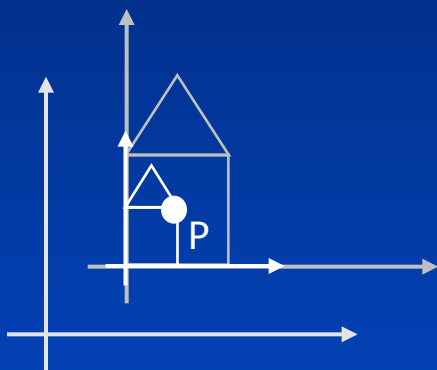
\*Note:  $(4,4) = T_{4,4} P$

## Mapping rule

---

*In general, if  $CS_1$  is transformed by a matrix  $M$  to form  $CS_2$ , a point  $P$  in  $CS_2$  is represented by  $MP$  in  $CS_1$*

# Another example



Translate(4,4), then  
Scale(0.5, 0.5)

Where is P in  $CS_3$ ?

(2,2)

Where is P in  $CS_2$ ?

$S_{0.5,0.5} (2,2) = (1,1)$

Where is P in  $CS_1$ ?

$T_{4,4} (1,1) = (5,5)$

\*Note: to go directly from  $CS_3$  to  $CS_1$  we can  
calculate  $T_{4,4} S_{0.5,0.5} (2,2) = (5,5)$

## General mapping rule

*If  $CS_1$  is transformed consecutively by  $M_1$ ,  $M_2$ , ...,  $M_n$  to form  $CS_{n+1}$ , then a point  $P$  in  $CS_{n+1}$  is represented by*

$$M_1 M_2 \dots M_n P \text{ in } CS_1.$$

*To form the composite transformation between CSs, you **postmultiply** each successive transformation matrix.*

# OpenGL Transformations

*Learn how to carry out transformations in OpenGL*

- Rotation
- Translation
- Scaling

*Introduce OpenGL matrix modes*

- Model-view
- Projection

# OpenGL Matrices

*In OpenGL matrices are part of the state*

*Multiple types*

- Model-View (`GL_MODELVIEW`)
- Projection (`GL_PROJECTION`)
- Texture (`GL_TEXTURE`) (ignore for now)
- Color (`GL_COLOR`) (ignore for now)

*Single set of functions for manipulation*

*Select which to manipulated by*

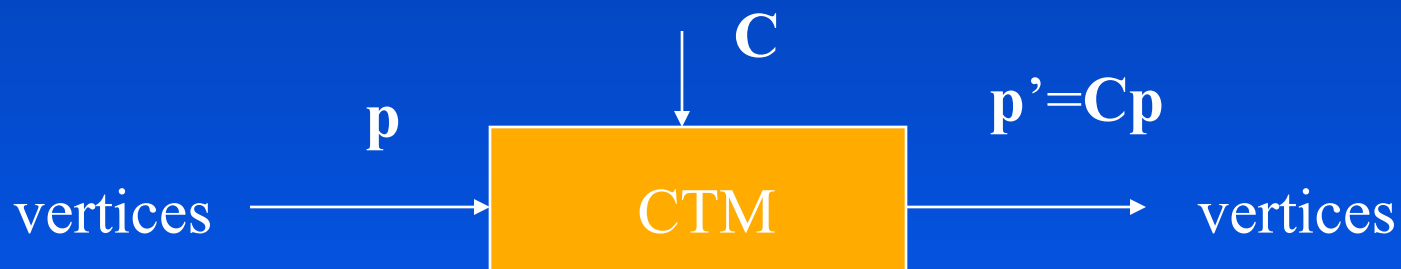
- `glMatrixMode (GL_MODELVIEW) ;`
- `glMatrixMode (GL_PROJECTION) ;`



# Current Transformation Matrix (CTM)

*Conceptually there is a 4 x 4 homogeneous coordinate matrix, the current transformation matrix (CTM) that is part of the state and is applied to all vertices that pass down the pipeline*

*The CTM is defined in the user program and loaded into a transformation unit*



# CTM operations

***The CTM can be altered either by loading a new CTM or by postmultiplication***

Load an identity matrix:  $C \leftarrow I$

Load an arbitrary matrix:  $C \leftarrow M$

Load a translation matrix:  $C \leftarrow T$

Load a rotation matrix:  $C \leftarrow R$

Load a scaling matrix:  $C \leftarrow S$

Postmultiply by an arbitrary matrix:  $C \leftarrow CM$

Postmultiply by a translation matrix:  $C \leftarrow CT$

Postmultiply by a rotation matrix:  $C \leftarrow CR$

Postmultiply by a scaling matrix:  $C \leftarrow CS$

# Rotation about a Fixed Point

*Start with identity matrix:  $C \leftarrow I$*

*Move fixed point to origin:  $C \leftarrow CT$*

*Rotate:  $C \leftarrow CR$*

*Move fixed point back:  $C \leftarrow CT^{-1}$*

*Result:  $C = TR T^{-1}$  which is backwards.*

*This result is a consequence of doing **postmultiplications**.*

*Let's try again.*

## Reversing the Order

**We want  $C = T^{-1} R T$**

**so we must do the operations in the following order**

$$C \leftarrow I$$

$$C \leftarrow CT^{-1}$$

$$C \leftarrow CR$$

$$C \leftarrow CT$$

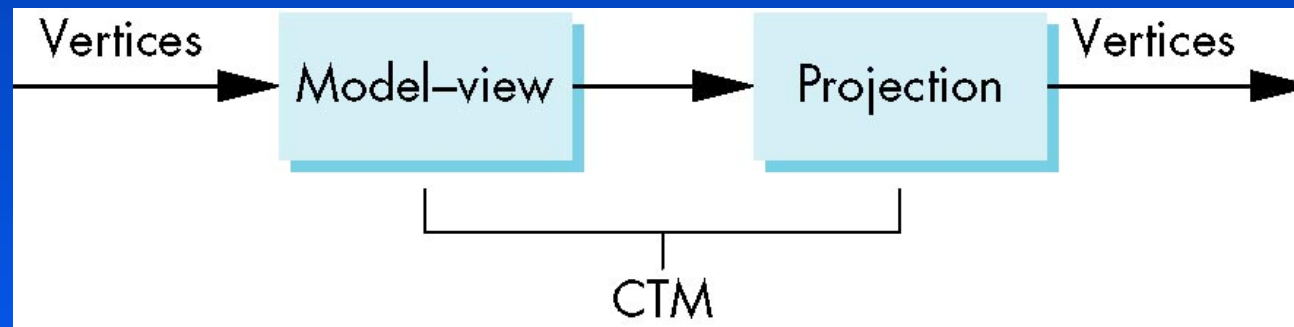
**Each operation corresponds to one function call in the program.**

**Note that the last operation specified is the first executed in the program**

# CTM in OpenGL

*OpenGL has a model-view and a projection matrix in the pipeline which are concatenated together to form the CTM*

*Can manipulate each by first setting the correct matrix mode*



# Rotation, Translation, Scaling

Load an identity matrix:

```
glLoadIdentity()
```

Multiply on right:

```
glRotatef(theta, vx, vy, vz)
```

*theta* in degrees, *(vx, vy, vz)* define axis of rotation

```
glTranslatef(dx, dy, dz)
```

```
glScalef( sx, sy, sz)
```

Each has a float (f) and double (d) format (*glScaled*)

# Example

***Rotation about z axis by 30 degrees with a fixed point of (1.0, 2.0, 3.0)***

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glTranslatef(1.0, 2.0, 3.0);  
glRotatef(30.0, 0.0, 0.0, 1.0);  
glTranslatef(-1.0, -2.0, -3.0);
```

***Remember that last matrix specified in the program is the first applied***

# Arbitrary Matrices

***Can load and multiply by matrices defined in the application program***

```
glLoadMatrixf (m)  
glMultMatrixf (m)
```

***The matrix  $m$  is a one dimension array of 16 elements which are the components of the desired 4 x 4 matrix stored by columns***

***In `glMultMatrixf`,  $m$  multiplies the existing matrix on the right***



# Transformations in OpenGL

*OpenGL makes it easy to do transformations to the CS, not the object*

*Sequence of operations:*

- Set up a routine to draw the object in its “base” CS
- Call transformation routines to transform the CS
- Object drawn in transformed CS

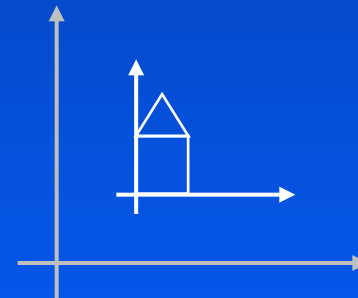
# OpenGL transformation example

```
drawHouse () {  
    glBegin (GL_LINE_LOOP);  
    glVertex2i (0,0);  
    glVertex2i (0,2);  
    ...  
    glEnd ();  
}
```

Draws basic house

```
drawTransformedHouse () {  
    glMatrixMode (GL_MODELVIEW);  
    glTranslatef (4.0, 4.0, 0.0);  
    glScalef (0.5, 0.5, 1.0);  
    drawHouse ();  
}
```

Draws transformed house



# Matrix Stacks

*In many situations we want to save transformation matrices for use later*

- Traversing hierarchical data structures
- Avoiding state changes when executing display lists

*OpenGL maintains stacks for each type of matrix*

- Access present type (as set by `glMatrixMode`) by

```
glPushMatrix()
```

```
glPopMatrix()
```

# OpenGL matrix stack example

```
glLoadMatrixf(m0);  
glPushMatrix();  
glMultMatrixf(m1);  
glPushMatrix();  
glMultMatrixf(m4);  
render chair2;  
glPopMatrix();  
glPushMatrix();  
glMultMatrixf(m3);  
render chair1;  
glPopMatrix();  
render table;  
glPopMatrix();  
glPushMatrix();  
glMultMatrixf(m2);  
render rug;  
glPopMatrix();  
render room;
```

