GEOMETRY & REPRESENTATIONS

Yun Jang
jangy@sejong.edu

Disclaimer

- These slides can only be used as study material for the Computer Graphics at Sejong University
- The slides cannot be distributed or used for another purpose

Basic Elements

- **Geometry** is the study of the relationships among objects in an n-dimensional space
  - In computer graphics, we are interested in objects that exist in **three dimensions**
- Want a minimum set of primitives from which we can build more sophisticated objects
- We will need three basic elements:
  - Scalars
  - Vectors
  - Points

Coordinate-Free Geometry

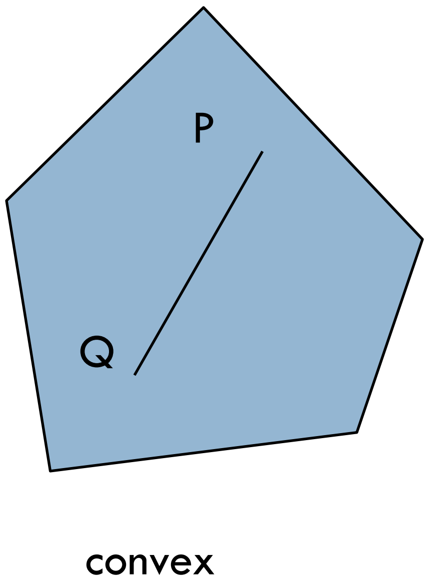
- When we learned simple geometry, most of us started with a **Cartesian** approach
  - Points were at locations in space **p** = (x, y, z)
  - We derived results by algebraic manipulations involving these coordinates
- This approach was nonphysical
  - Physically, points exist regardless of the location of an arbitrary coordinate system
  - Most geometric results are independent of the coordinate system
  - Example Euclidean geometry: two triangles are identical if two corresponding sides and the angle between them are identical

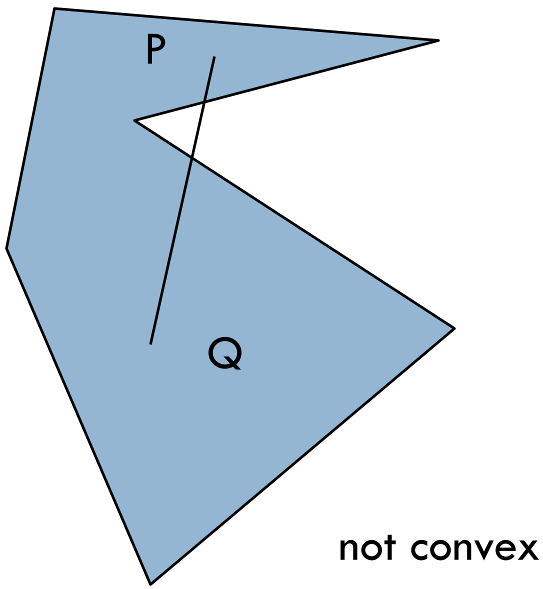
Affine Spaces

- Vector space supporting points with no origin
  - Affine: preserve colinearity and ratio of distances
- Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar operations
  - **Note**: No origin, so cannot add points!
- For any point define
  - $1 \bullet P = P$
  - $0 \bullet P = \mathbf{0}$ (zero vector)

Convexity

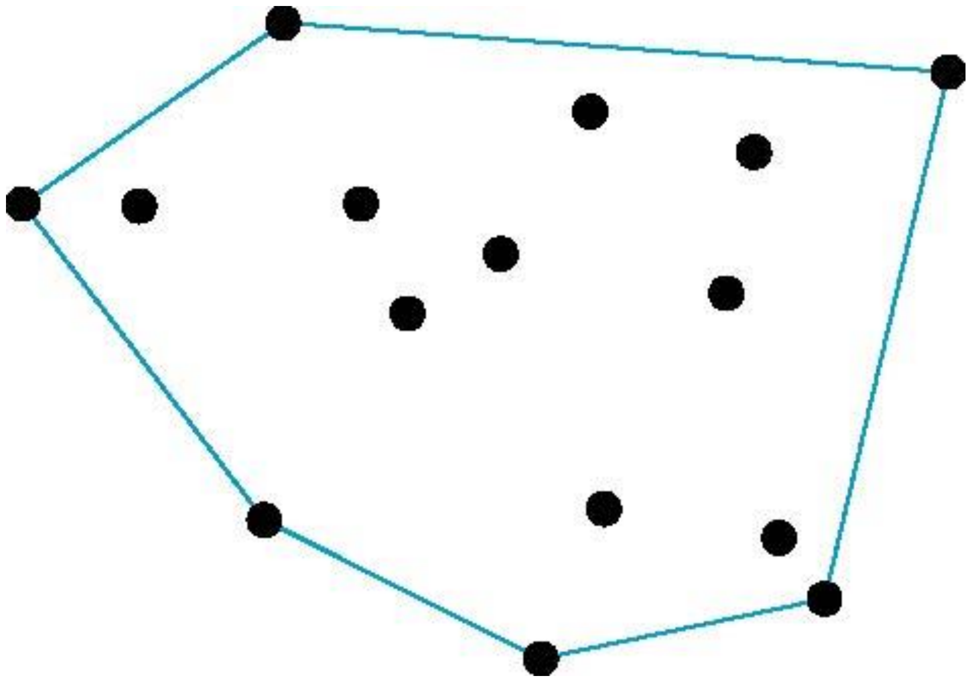
- An object is *convex* **iff** for any two points in the object all points on the line segment between these points are also in the object





Convex Hull

- Smallest convex object containing $P_1, P_2, \ldots\ldots P_n$
- Formed by “shrink wrapping” points



Linear Independence

- A set of vectors v_1, v_2, …, v_n is *linearly independent* if

  $$\alpha_1 v_1+\alpha_2 v_2+.. \alpha_n v_n=0 \text{ iff } \alpha_1=\alpha_2=\ldots=0$$

- If a set of vectors is linearly independent, we cannot represent one in terms of the others
- If a set of vectors is linearly dependent, as least one can be written in terms of the others

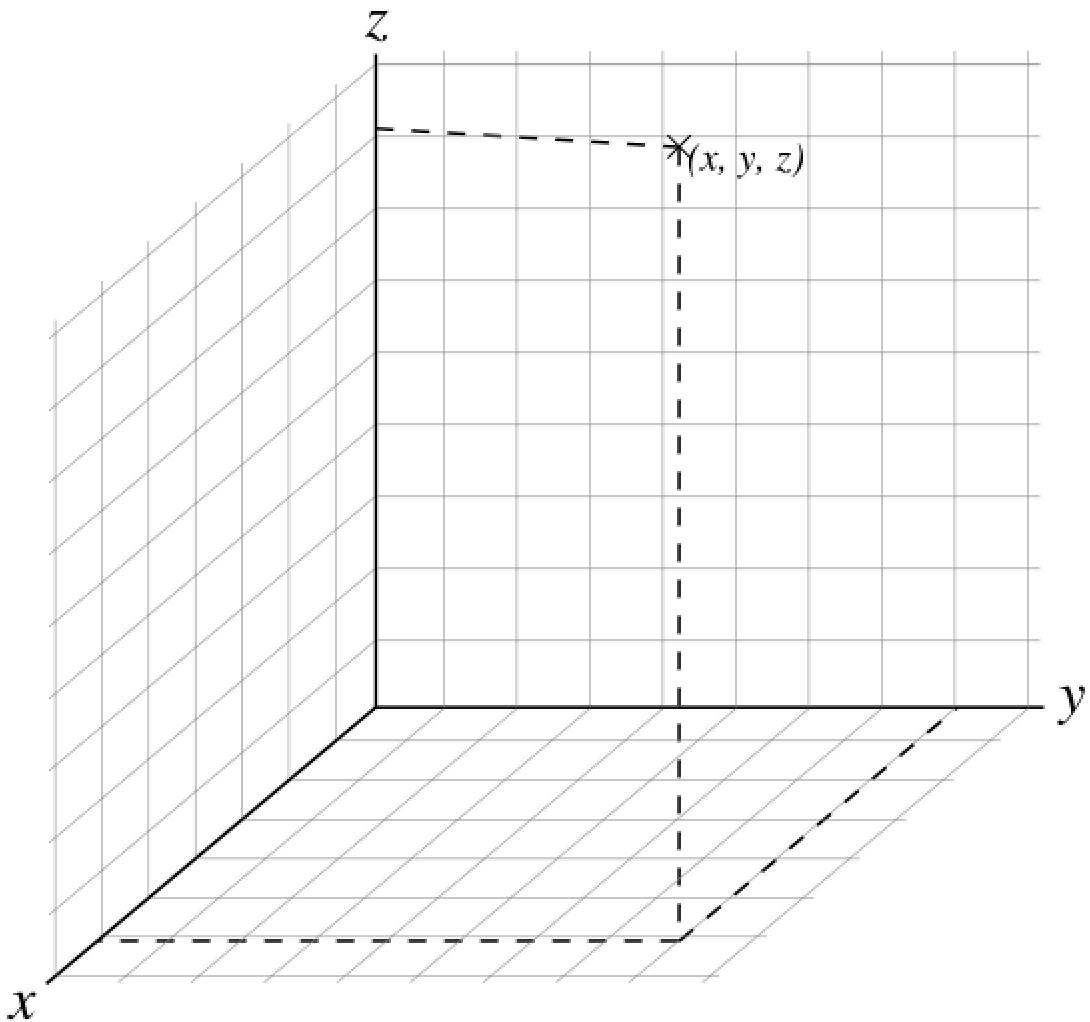
Dimension

- In a vector space, the maximum number of linearly independent vectors is fixed and is called the *dimension* of the space
- In an *n*-dimensional space, any set of n linearly independent vectors form a *basis* for the space
- Given a basis $v_1, v_2, \ldots, v_n$, any vector v can be written as

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

where the $\{\alpha_i\}$ are unique

Example: 3D Cartesian Space

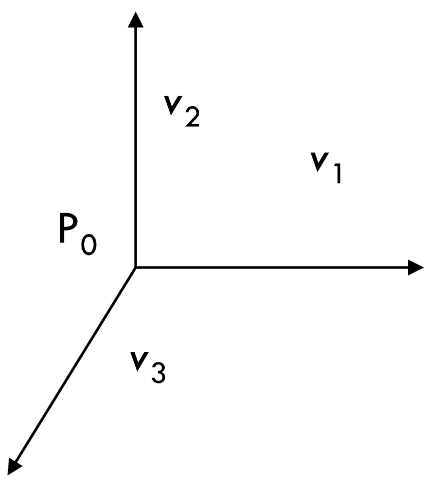


Representation

- Until now we have been able to work with geometric entities without using any frame of reference, such as a coordinate system
- Need a frame of reference to relate points and objects to our physical world
  - For example, where is a point? Can’t answer without a reference system
  - World coordinates
  - Camera coordinates

Frames

- A coordinate system is insufficient to represent points
- If we work in an affine space we can add a single point, the *origin*, to the basis vectors to form a *frame*



Homogeneous Coordinates

The homogeneous coordinates form for a three-dimensional point [x y z] is given as

$\mathbf{p} = [x'\ y'\ z'\ w]^T = [wx\ wy\ wz\ w]^T$

We return to a three dimensional point (for $w \neq 0$) by

$x \leftarrow x'/w$

$y \leftarrow y'/w$

$z \leftarrow z'/w$

If $w = 0$, the representation is that of a vector

Note that homogeneous coordinates replaces points in three dimensions by lines through the origin in four dimensions

For w = 1, the representation of a point is [x y z 1]

Homogenous Coordinates and Computer Graphics

- Homogeneous coordinates are key to all computer graphics systems
  - All standard transformations (rotation, translation, scaling) can be implemented with matrix multiplications using 4 x 4 matrices
  - Hardware pipeline works with 4 dimensional representations
  - For orthographic viewing, we can maintain $w=0$ for vectors and $w=1$ for points
  - For perspective we need a *perspective division*

Example: 2D (3x3) matrices

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x+a \\ y+b \\ 1 \end{pmatrix},$$

Linear transformations:
rotation, reflection, etc

$$\begin{pmatrix} \boxed{\begin{matrix} a & b \\ c & d \end{matrix}} & \begin{matrix} 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & 0 \end{matrix} & 1 \end{pmatrix}.$$

Questions?

- Ask now or e-mail later
- Acknowledgements
  - Previous instructors at Purdue
    - David Ebert, ECE
    - Niklas Elmqvist, ECE
  - Previous instructors at Arizona state university
    - Ross Maciejewski
  - Textbook (Ed Angel)
  - Google Image Search
    - Copyright respective owners

TRANSFORMATIONS AND OPENGL

Yun Jang
jangy@sejong.edu

Disclaimer

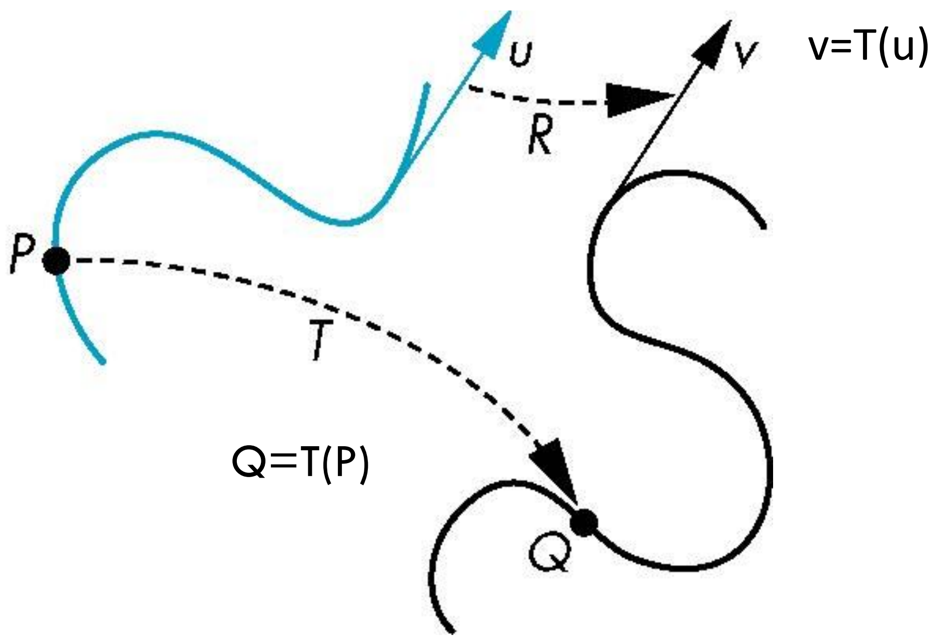
- These slides can only be used as study material for the Computer Graphics at Sejong University
- The slides cannot be distributed or used for another purpose

Objectives

- Introduce standard transformations
  - Rotation
  - Translation
  - Scaling
  - Shear
- Derive homogeneous coordinate transformation matrices
- Learn to build arbitrary transformation matrices from simple transformations

General Transformations

A transformation maps points to other points and/or vectors to other vectors

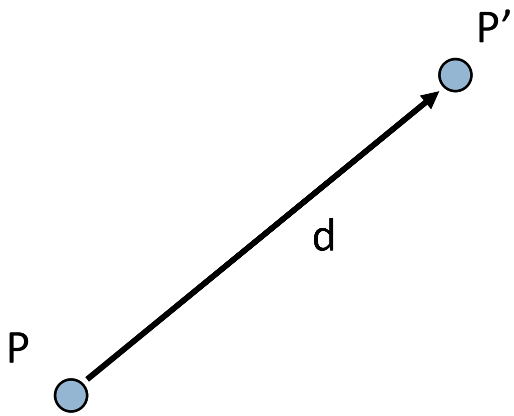


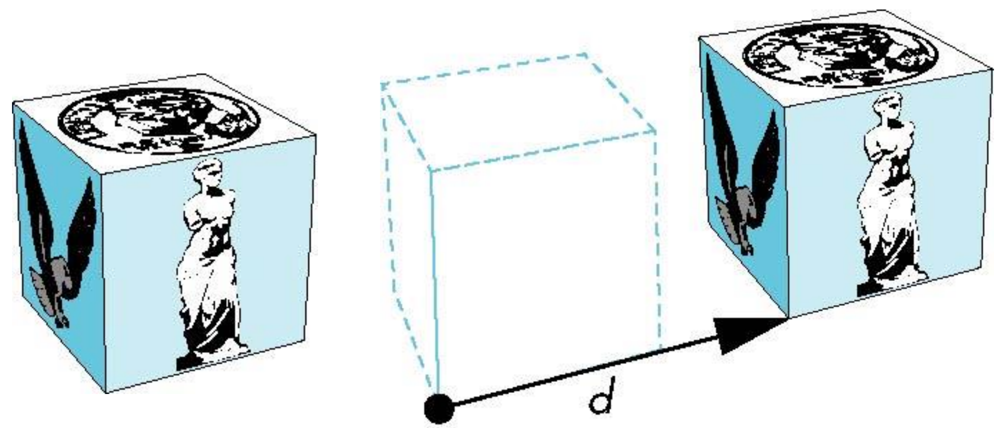
Affine Transformations

- Line preserving
- Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear
- Need only transform endpoints of line segments and let implementation draw line segment between transformed endpoints

Translation

- Move (translate, displace) a point to a new location





- Displacement determined by a vector d
  - Three degrees of freedom
  - P'=P+d

Translation Using Representations

Using the homogeneous coordinate representation in some frame

$\mathbf{p}=[\ x\ y\ z\ 1]^T$

$\mathbf{p}'=[x'\ y'\ z'\ 1]^T$

$\mathbf{d}=[dx\ dy\ dz\ 0]^T$

Hence $\mathbf{p}' = \mathbf{p} + \mathbf{d}$ or

$x'=x+d_x$

$y'=y+d_y$

$z'=z+d_z$

note that this expression is in four dimensions and expresses point = vector + point

Translation Matrix

We can also express translation using a

4 x 4 matrix **T** in homogeneous coordinates

p'=**Tp** where

$$\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

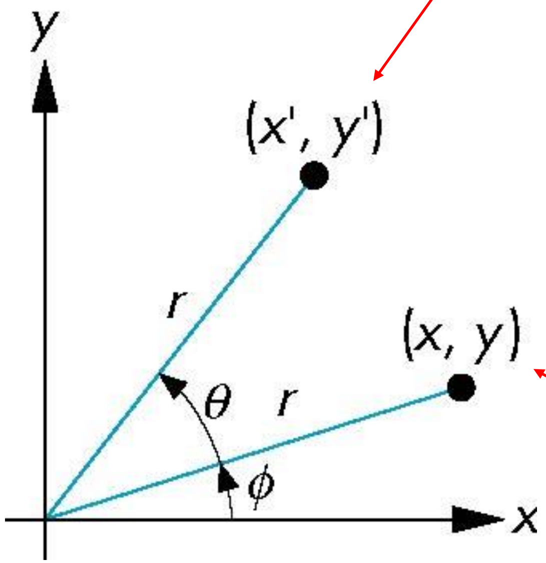
This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together

Rotation (2D)

Consider rotation about the origin by θ degrees

- radius stays the same, angle increases by θ

$x' = r \cos(\Phi + \theta)$
$y' = r \sin(\Phi + \theta)$



$$x' = x \cos\theta - y \sin\theta$$
$$y' = x \sin\theta + y \cos\theta$$

$x = r \cos\Phi$
$y = r \sin\Phi$

Rotation about the z axis

- Rotation about z axis in three dimensions leaves all points with the same z
  - Equivalent to rotation in two dimensions in planes of constant z

$$x' = x \cos\theta - y \sin\theta$$
$$y' = x \sin\theta + y \cos\theta$$
$$z' = z$$

  - or in homogeneous coordinates

$$\mathbf{p}' = \mathbf{R}_{\mathbf{z}}(\theta)\mathbf{p}$$

Rotation Matrix

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

Rotation About x and y axes

- Same argument as for rotation about *z* axis
  - For rotation about x axis, x is unchanged
  - For rotation about y axis, y is unchanged

$$\mathbf{R} = \mathbf{R}_x(q) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

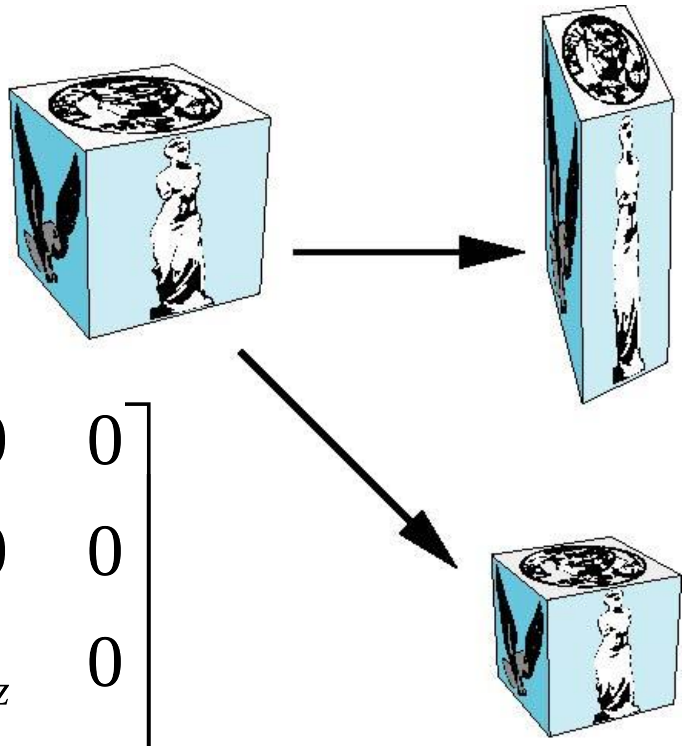
$$\mathbf{R} = \mathbf{R}_y(q) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

Expand or contract along each axis (fixed point of origin)

$x' = s_x x$
$y' = s_y y$
$z' = s_z z$

p'=Sp



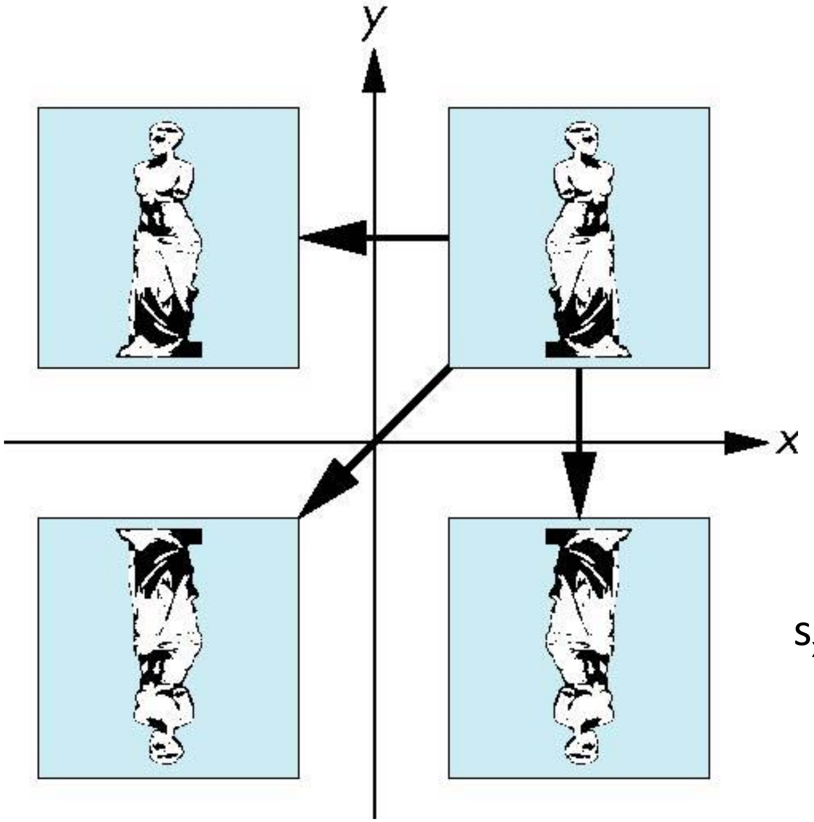
$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Reflection

corresponds to negative scale factors

$s_x = -1$ $s_y = 1$

original



$s_x = -1$ $s_y = -1$

$s_x = 1$ $s_y = -1$

Inverses

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations
  - Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$
  - Rotation: $\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$
    - Holds for any rotation matrix
    - Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$ $\mathbf{R}^{-1}(\theta) = \mathbf{R}^{T}(\theta)$
  - Scaling: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

Order of Transformations

- Matrix on the right is applied first
- Mathematically, the following are equivalent

$$\mathbf{p}' = \mathbf{ABCp} = \mathbf{A}(\mathbf{B}(\mathbf{Cp}))$$

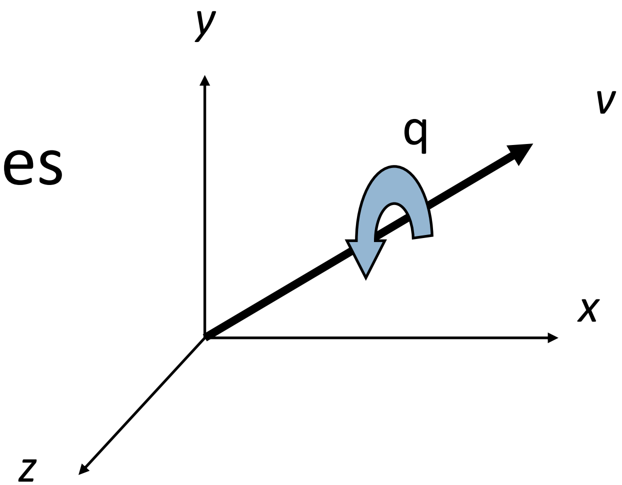
- **Note**: many references use column matrices to represent points
  - In terms of column matrices:

$$\mathbf{p}'^{T} = \mathbf{p}^{T}\mathbf{C}^{T}\mathbf{B}^{T}\mathbf{A}^{T}$$

General Rotation About the Origin

A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the *x*, *y*, and *z* axes

$$\mathbf{R}(q) = \mathbf{R}_z(q_z)\ \mathbf{R}_y(q_y)\ \mathbf{R}_x(q_x)$$



$q_x\ q_y\ q_z$ are called the Euler angles

Note that rotations do not commute
We can use rotations in another order but
with different angles

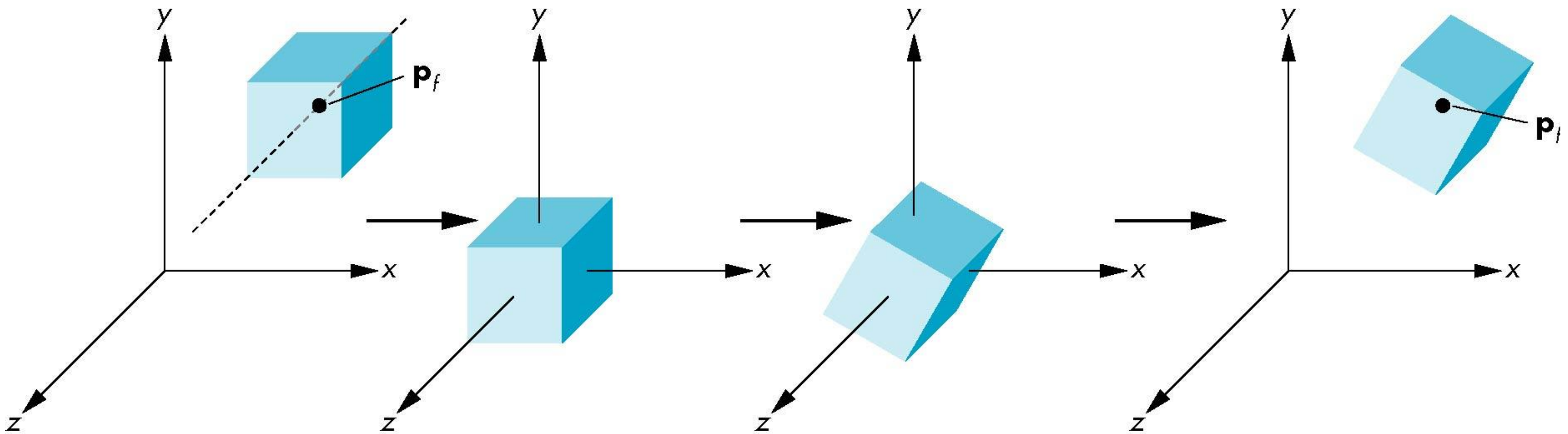
Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

Move fixed point back

$\mathbf{M} = \mathbf{T}(p_f)\,\mathbf{R}(\theta)\,\mathbf{T}(-p_f)$



TRANSFORMATIONS IN OPENGL

Yun Jang
jangy@sejong.edu

Objectives

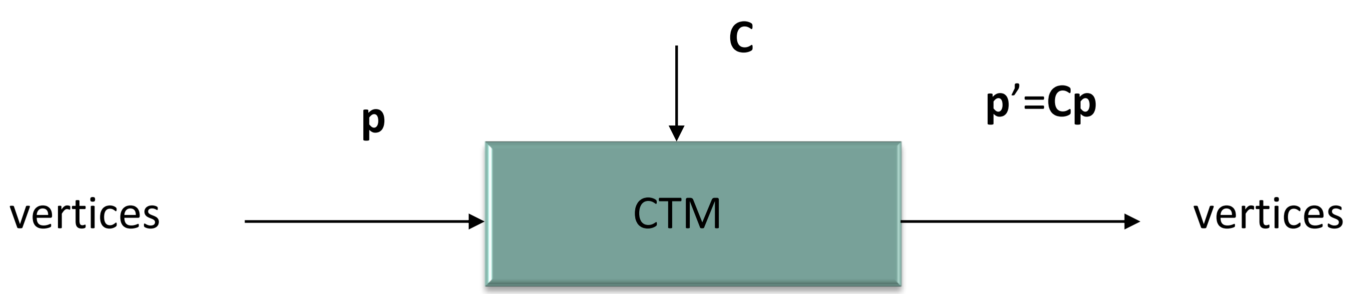
- Learn how to carry out transformations in OpenGL
  - Rotation
  - Translation
  - Scaling
- Introduce OpenGL matrix modes
  - Model-view
  - Projection

OpenGL Matrices

- **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 postmutiplication

Load an identity matrix: $\mathbf{C} \leftarrow \mathbf{I}$
Load an arbitrary matrix: $\mathbf{C} \leftarrow \mathbf{M}$

Load a translation matrix: $\mathbf{C} \leftarrow \mathbf{T}$
Load a rotation matrix: $\mathbf{C} \leftarrow \mathbf{R}$
Load a scaling matrix: $\mathbf{C} \leftarrow \mathbf{S}$

Postmultiply by an arbitrary matrix: $\mathbf{C} \leftarrow \mathbf{CM}$
Postmultiply by a translation matrix: $\mathbf{C} \leftarrow \mathbf{CT}$
Postmultiply by a rotation matrix: $\mathbf{C} \leftarrow \mathbf{C\,R}$
Postmultiply by a scaling matrix: $\mathbf{C} \leftarrow \mathbf{C\,S}$

Rotation About a Fixed Point

Start with identity matrix: $\mathbf{C} \leftarrow \mathbf{I}$

Move fixed point to origin: $\mathbf{C} \leftarrow \mathbf{CT}$

Rotate: $\mathbf{C} \leftarrow \mathbf{CR}$

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

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

This result is a consequence of doing postmultiplications.

Let's try again.

Reversing the Order

We want $\mathbf{C} = \mathbf{T}^{-1}\,\mathbf{R}\,\mathbf{T}$
so we must do the operations in the following order

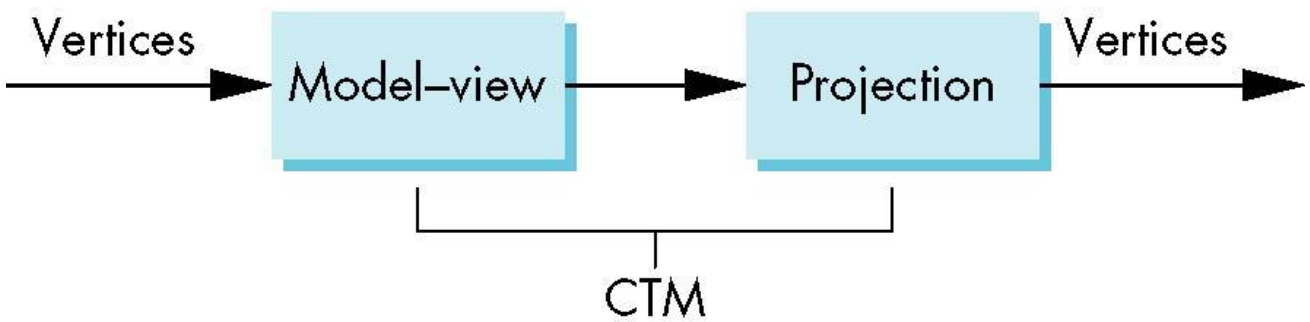
$\mathbf{C} \leftarrow \mathbf{I}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}^{-1}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$

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(const Glfloat *m)
glMultMatrixf(const Glfloat *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 <u>columns</u>
- In **glMultMatrixf**, **m** multiplies the existing matrix on the right

Row vs. Column-major Order

- How to lay out a 2D array in memory?

$$\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$$

- Two options
  - Rows: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p
  - Columns: a, e, i, m, b, f, j, n, c, g, k, o, d, h, l, p
- OpenGL expects column-major
  - C stores in row-major
    - float data[4][4] = { { a, b, c, d}, {e, f, g, h}, ...};
  - Need to transpose before sending to OpenGL

Matrix Stacks

- In many situations we want to save transformation matrices for use later
  - Traversing hierarchical data structures (Chapter 10)
  - Avoiding state changes when executing display lists
- OpenGL maintains stacks for each type of matrix
  - Access present type (as set by **`glMatrixMode)`** by

    `glPushMatrix()`

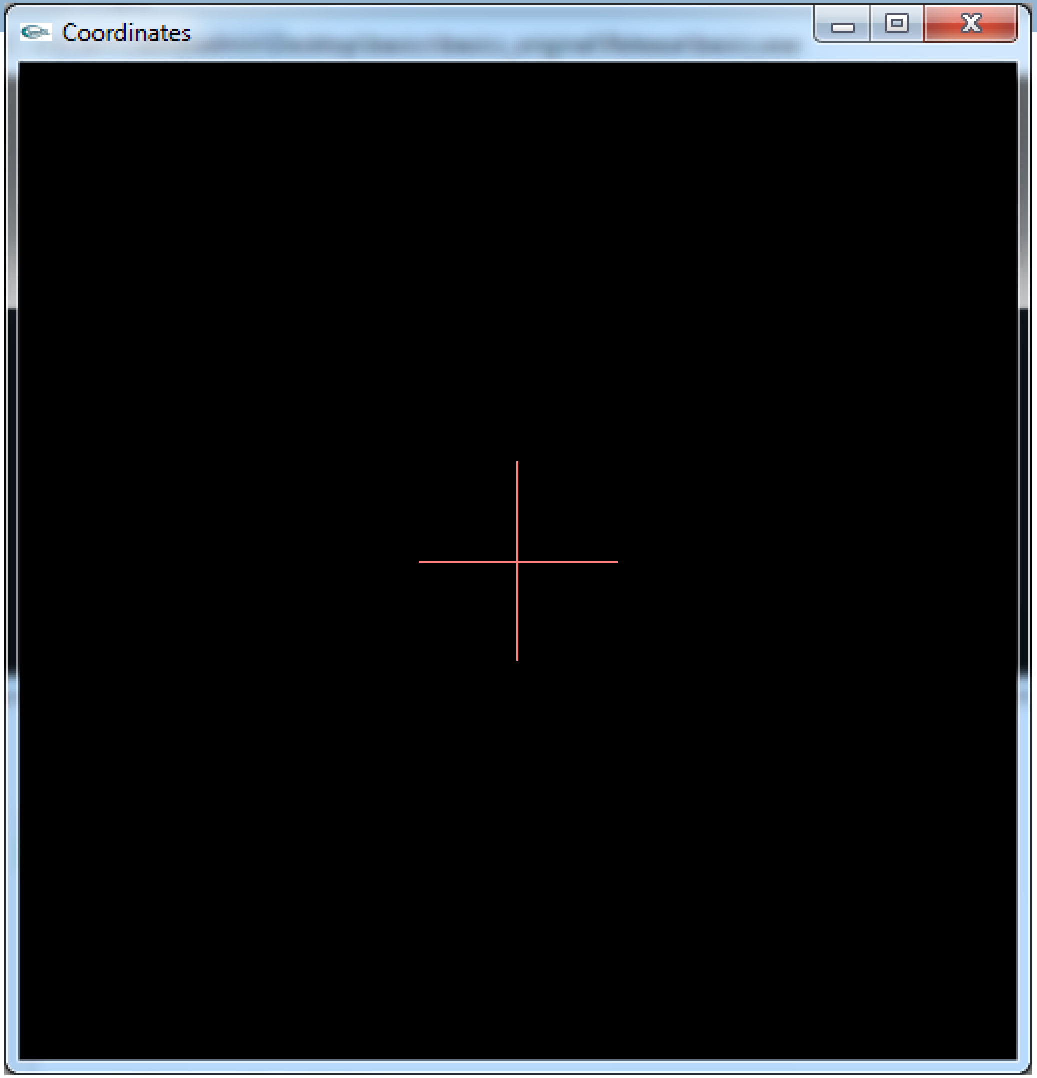
    `glPopMatrix()`

Our Local Coordinate System


```
int main(int argc, char** argv)
{
    ..............
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(200, 200);
    glutCreateWindow("Coordinates");
    ...............
/* create a callback routine for (re-)display */
    glutDisplayFunc(display);
}
```

Our Local Coordinate System

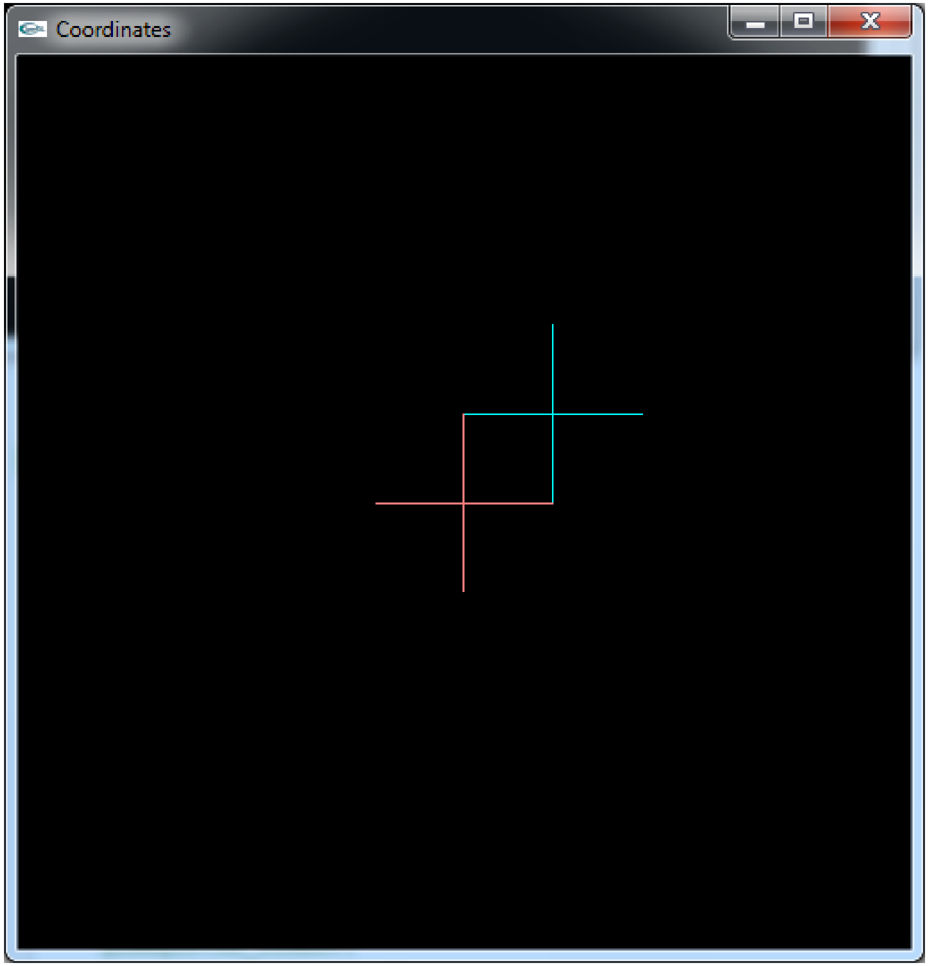

```
void display()
{
	/* clear the screen*/
	glClear(GL_COLOR_BUFFER_BIT);

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

	glColor3f(1, .5, .5);
	drawAxes();
}

Void drawAxes(){
	glBegin(GL_LINES);
		glVertex2f(0, -.2);
		glVertex2f(0, .2);
	glEnd();

	glBegin(GL_LINES);
		glVertex2f(-.2, 0);
		glVertex2f(.2, 0);
	glEnd();
}
```



Our Local Coordinate System

- Now I want to move my local coordinate system to (.2, .2)
- What do I do?

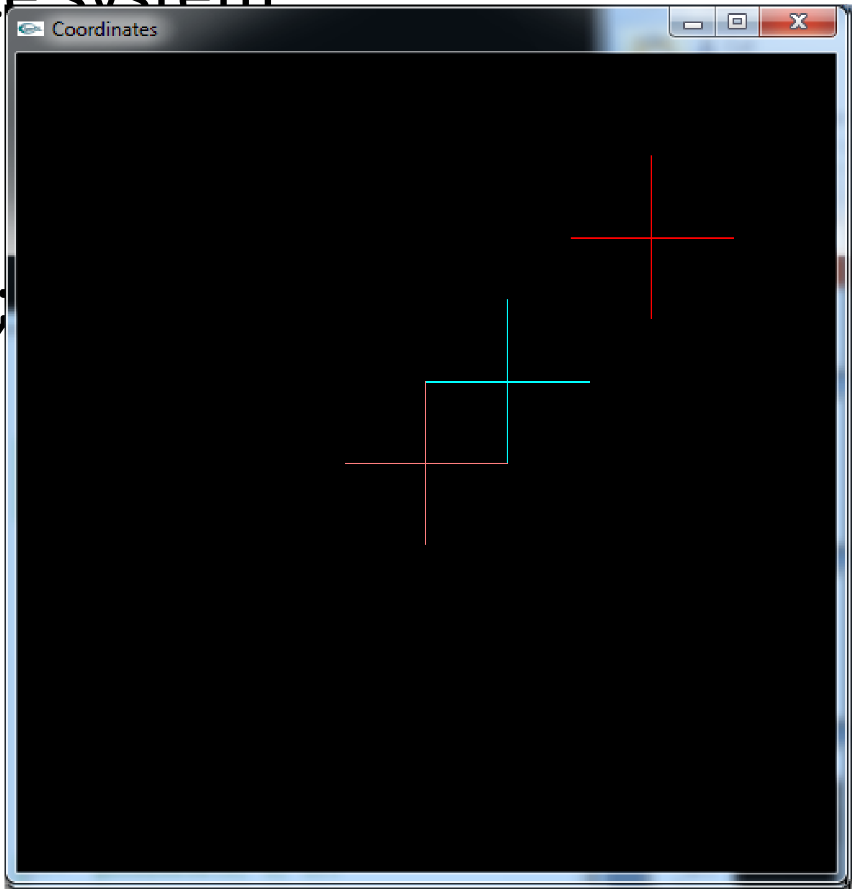
```
glTranslatef(.2, .2, 0);
glColor3f(0, 1, 1);
drawAxes();
```



Our Local Coordinate System

- Now I want to rotate an object about the center of my new local coordinate system
- What do I do?

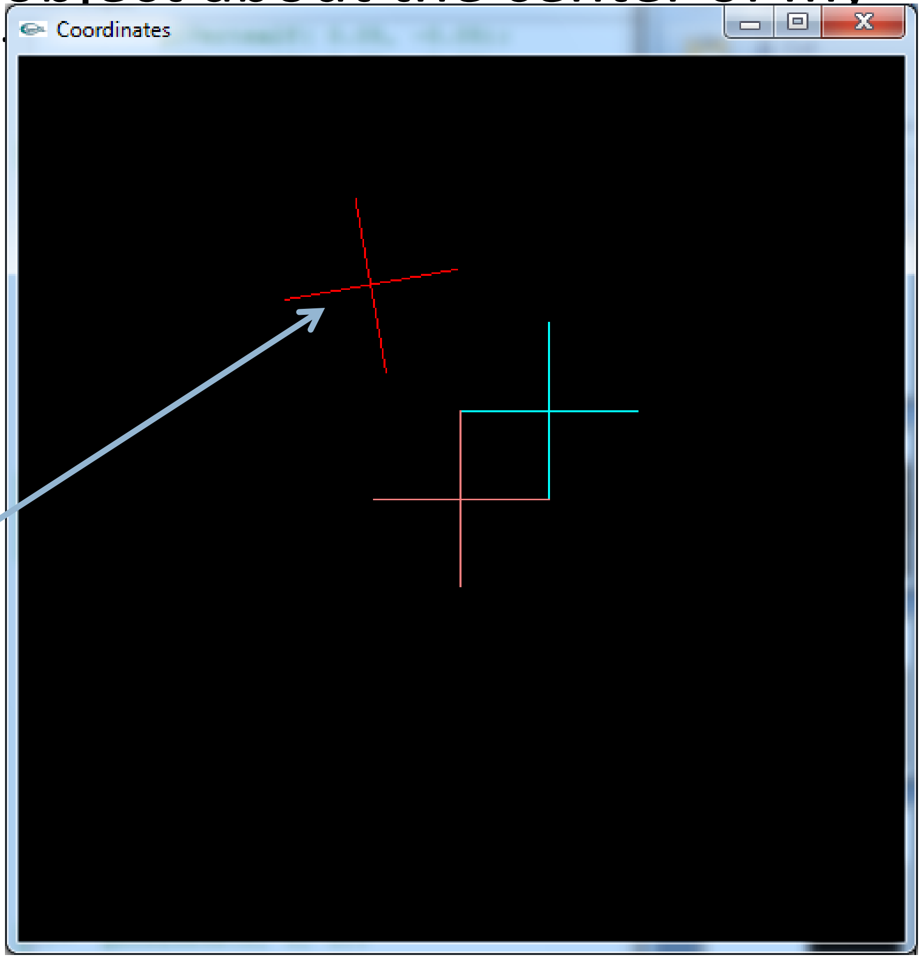
```
glTranslatef(.35, .35, 0);
glRotatef(angle, 0, 0, 1)
drawAxes()
```



Our Local Coordinate System

- Now I want to rotate an object about the center of my new local coordinate system
- What do I do?

```
glRotatef(angle, 0, 0, 1);
glTranslatef(.35, .35, 0);
glRotatef(-angle, 0, 0, 1);
drawAxes()
```



Your new local coordinate system is rotated if you don't add this in!