Transformations

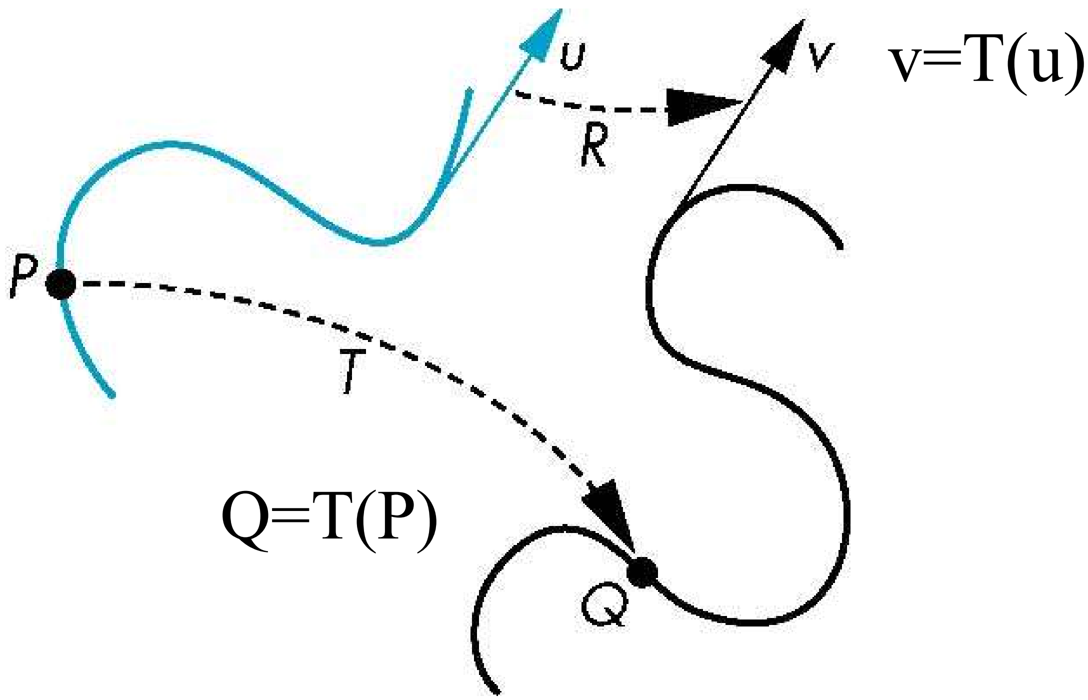
Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

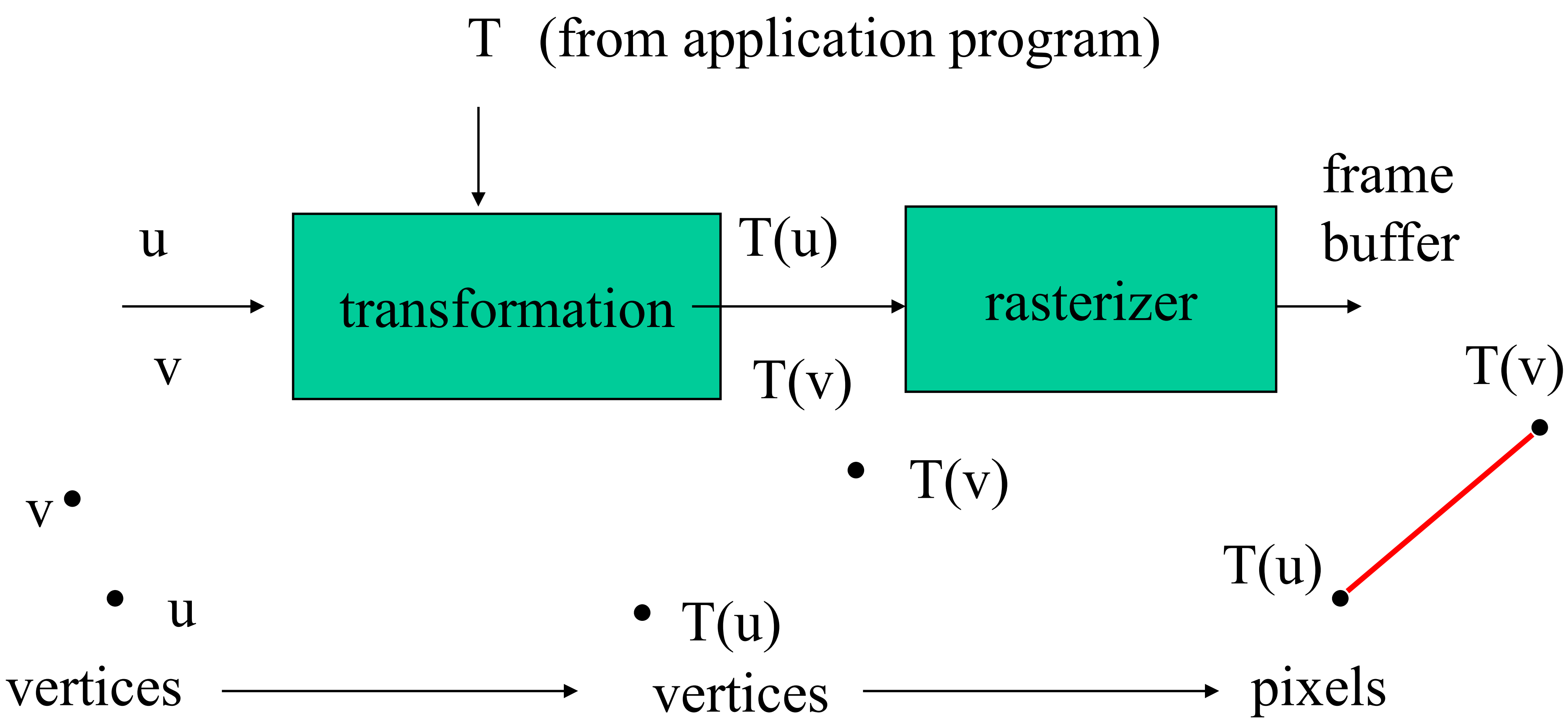
- Introduce standard transformations
  - Rotations
  - Translation
  - Scaling
  - Shear
- Derive homogeneous coordinate transformation matrices
- Learn to build arbitrary transformation matrices from simple transformations

General Transformations

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



Pipeline Implementation



Homogeneous Notation

- 3D points and vectors are represented as 4D points in homogeneous coordinates
  - 3D Vector: [x y z 0]
  - 3D Point: [x y z 1]
- Matrices used in 3D graphics are typically 4x4:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Identity Matrix

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

Matrix Multiplication

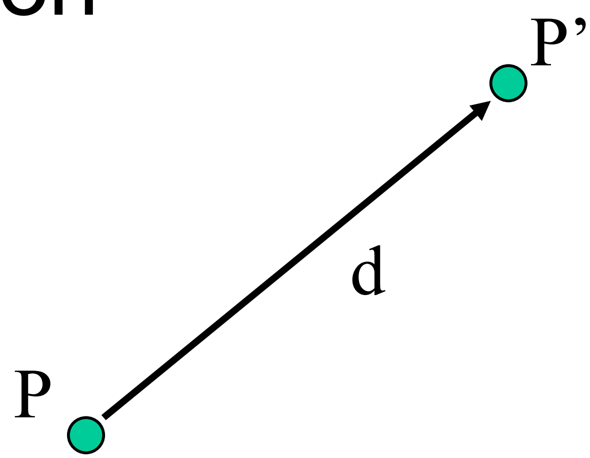
Multiplying a point (or vector) by a matrix:

$$\begin{pmatrix} AX + BY + CZ + D \\ EX + FY + GZ + H \\ IX + JY + KZ + L \\ MX + NY + OZ + P \end{pmatrix} = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

usually done “right to left”

Translation

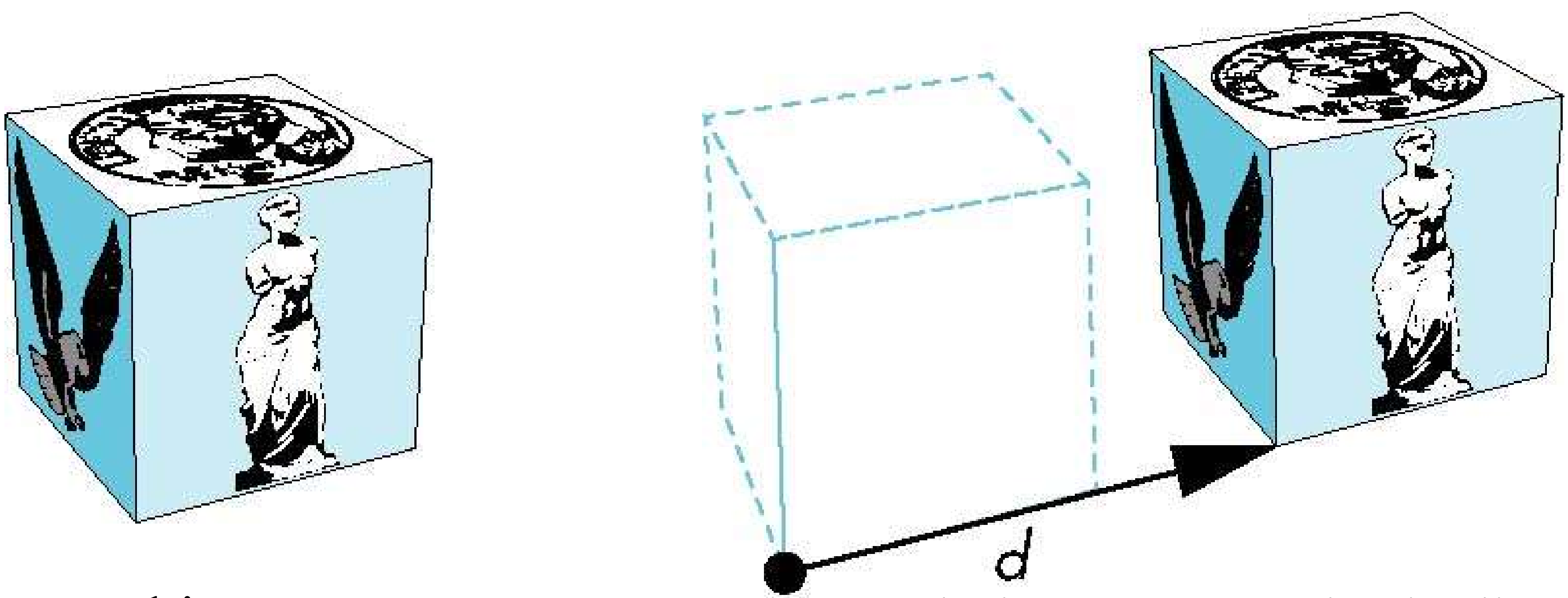
- Move (translate, displace) a point to a new location



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

Object Translation

Every point in object is displaced by same vector



object

translation: every point is displaced by same vector

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

$x' = x + dx$

$y' = y + dy$

$z' = z + dz$

note that this expression is in four dimensions and expresses that 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

Translation Matrix

$$\begin{pmatrix} X + T_X \\ Y + T_Y \\ Z + T_Z \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_X \\ 0 & 1 & 0 & T_Y \\ 0 & 0 & 1 & T_Z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

in GLM:

- *glm::translate(x,y,z)*
- *mat4 * vec4*

Scaling

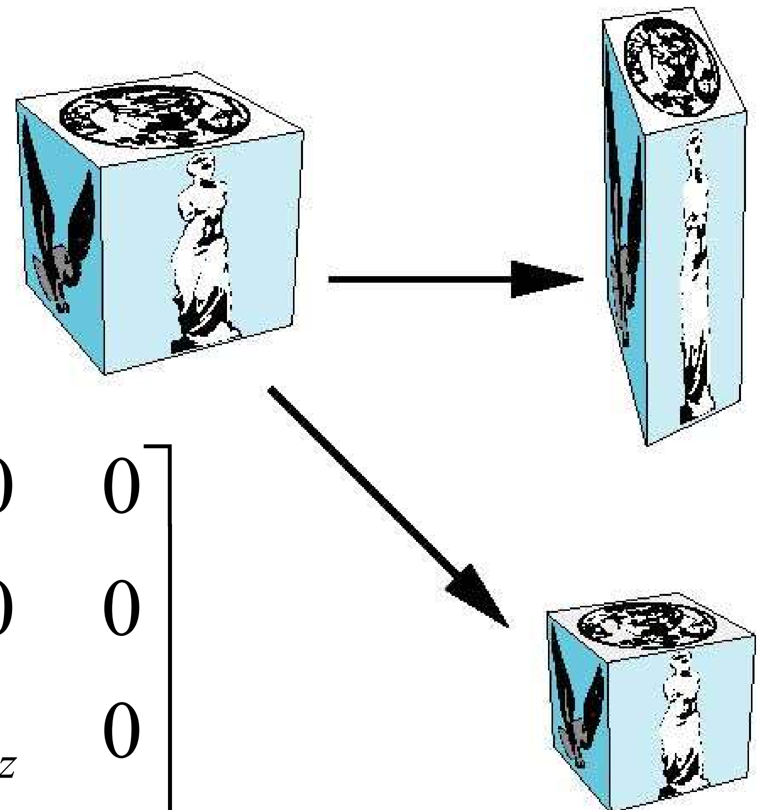
Expand or contract along each axis (fixed point of origin)

$x' = s_x x$

$y' = s_y y$

$z' = s_z z$

$\mathbf{p}' = \mathbf{S}\mathbf{p}$



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

Scaling

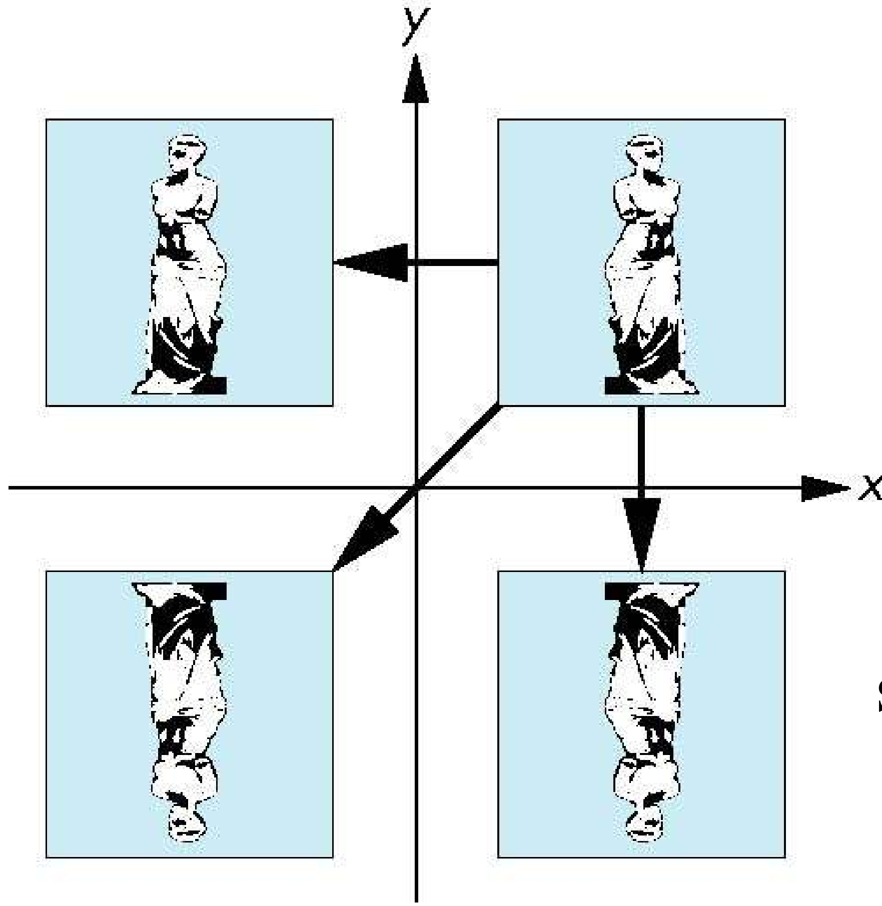
$$\begin{pmatrix} X * S_X \\ Y * S_Y \\ Z * S_Z \\ 1 \end{pmatrix} = \begin{bmatrix} S_X & 0 & 0 & 0 \\ 0 & S_Y & 0 & 0 \\ 0 & 0 & S_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

in GLM:

- *glm::scale(x,y,z)*
- *mat4 * vec4*

Reflection

corresponds to negative scale factors



$s_x = -1 \; s_y = 1$

original

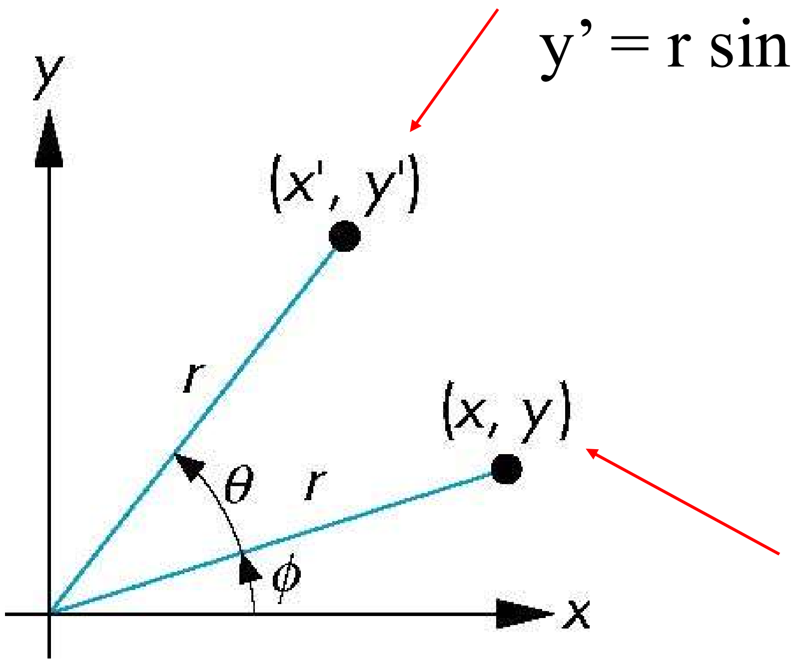
$s_x = -1 \; s_y = -1$

$s_x = 1 \; s_y = -1$

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

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

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

Rotation Matrices

Rotation around X by θ degrees

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Rotation around Y by θ degrees

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Rotation around Z by θ degrees

$$\begin{pmatrix} X' \\ Y' \\ Z' \\ 1 \end{pmatrix} = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

- *glm::rotate(mat4, θ, x, y, z)*
- *mat4 * vec4*

Euler Angles

In the mid-1700s, the mathematician Leonhard Euler showed that a rotation around any desired axis could be specified instead as a combination of rotations around the X, Y, and Z axes.

These three rotation angles, around the respective axes, have come to be known as Euler angles.

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: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

Concatenation

- We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices
- Because the same transformation is applied to many vertices, the cost of forming a composite matrix $\mathbf{M}=\mathbf{ABCD}$ is not significant compared to the cost of computing $\mathbf{Mp}$ for many vertices $\mathbf{p}$
- The difficult part is how to form a desired transformation from the specifications in the application

Mulipying a Matrix by a Matrix

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} * \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

$$= \begin{bmatrix} Aa+Be+Ci+Dm & Ab+Bf+Cj+Dn & Ac+Bg+Ck+Do & Ad+Bh+Cl+Dp \\ Ea+Fe+Gi+Hm & Eb+Ff+Gj+Hn & Ec+Fg+Gk+Ho & Ed+Fh+Gl+Hp \\ Ia+Je+Ki+Lm & Ib+Jf+Kj+Ln & Ic+Jg+Kk+Lo & Id+Jh+Kl+Lp \\ Ma+Ne+Oi+Pm & Mb+Nf+Oj+Pn & Mc+Ng+Ok+Po & Md+Nh+Ol+Pp \end{bmatrix}$$

Matrix Multiplication is Associative

New Point = $Matrix_1$ * ($Matrix_2$ * ($Matrix_3$ * Point))

New Point = ($Matrix_1$ * $Matrix_2$ * $Matrix_3$) * Point

and thus, equivalently:

$Matrix_C$ = $Matrix_1$ * $Matrix_2$ * $Matrix_3$

New Point = $Matrix_C$ * Point

In this example, $Matrix_C$ is often called the <u>concatenation</u> of $Matrix_1$, $Matrix_2$, and $Matrix_3$

Order of Transformations

- Note that matrix on the right is the first applied
- Mathematically, the following are equivalent

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

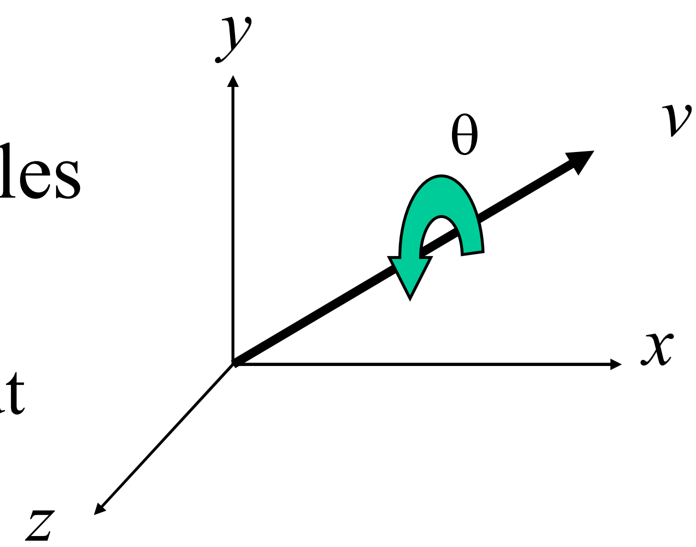
- Note many references use column matrices to present points. In terms of column matrices

$$\mathbf{p}^{T\prime} = \mathbf{p}^T\mathbf{C}^T\mathbf{B}^T\mathbf{A}^T$$

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

$$\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z)\, \mathbf{R}_y(\theta_y)\, \mathbf{R}_x(\theta_x)$$



$\theta_x\ \theta_y\ \theta_z$ are called the Euler angles

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

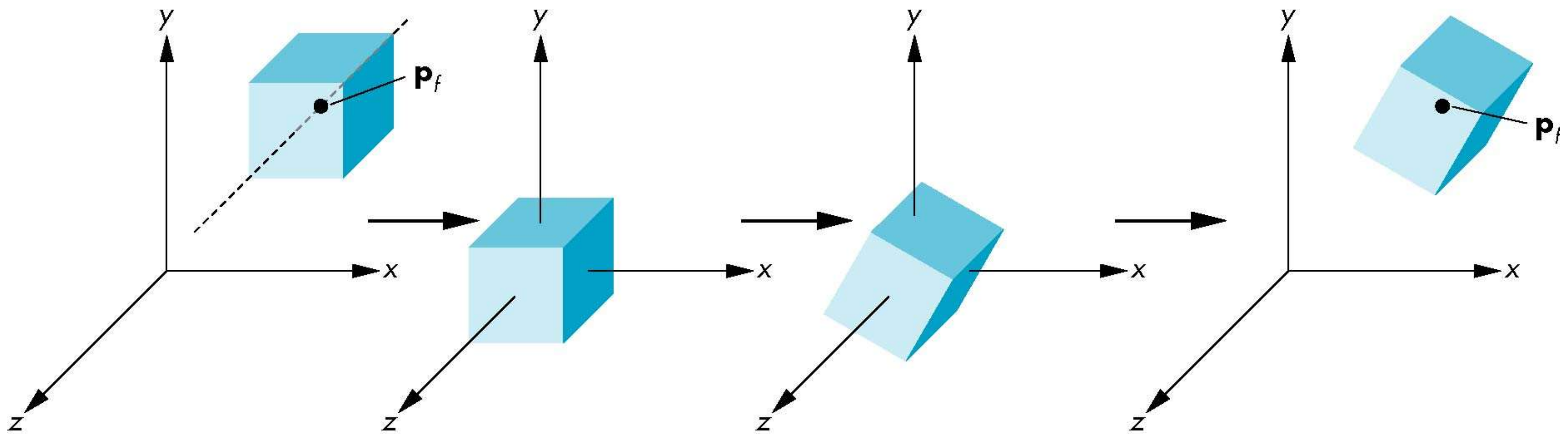
Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

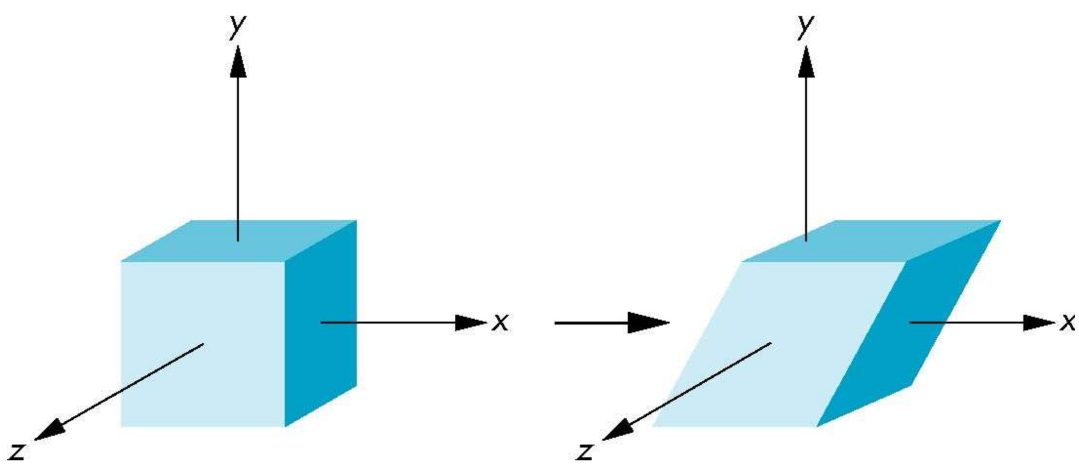
Move fixed point back

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



Shear

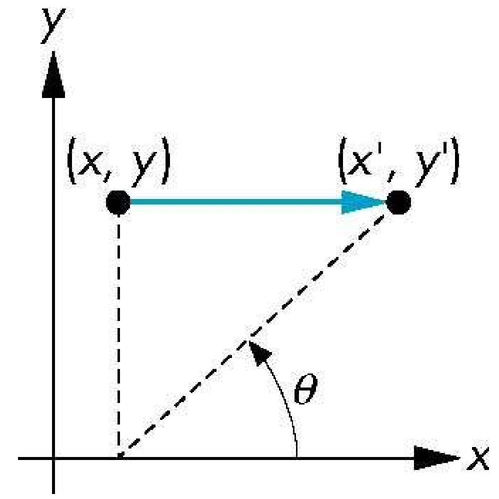
- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions



Shear Matrix

Consider simple shear along x axis

$$x' = x + y \cot \theta$$
$$y' = y$$
$$z' = z$$



$$\mathbf{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D Transformations

- A vertex is transformed by 4×4 matrices
- All matrices are stored column-major in OpenGL
  - this is opposite of what "C" programmers expect
- Matrices are always post-multiplied
  - product of matrix and vector is $\mathbf{M}\vec{v}$

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \qquad v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}$$

Affine Transformations

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Characteristic of many important transformations
  - Translation
  - Rotation
  - Scaling
  - Shear
- Line preserving

OpenGL Transformations

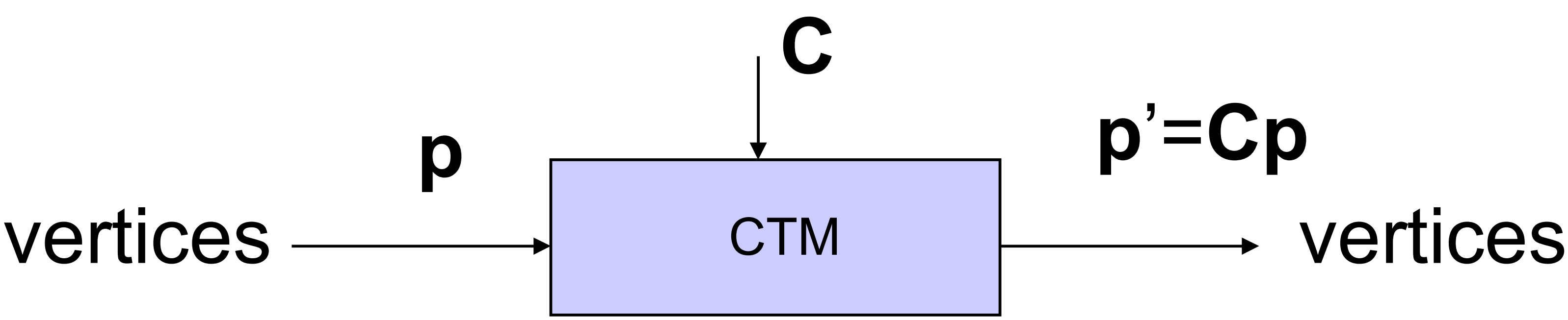
Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Learn how to carry out transformations in OpenGL
    - Rotation
    - Translation
    - Scaling
- Introduce QMatrix4x4 and QVector3D transformations
    - Model-view
    - 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.

The last operation specified is the first executed in the program!

Rotation, Translation, Scaling

Create an identity matrix:

```
QMatrix4x4 m;
m.setToIdentity();
```

Multiply on right by rotation matrix of **theta** in degrees where (**vx**, **vy**, **vz**) define axis of rotation

```
m.rotate(theta, QVector3D(vx, vy, vz));
```

Do same with translation and scaling:

```
m.scale(sx, sy, sz);
m.translate(dx, dy, dz);
```

Example

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

```
QMatrix4x4 m;
m.setToIdentity();
m.translate( 1.0, 2.0, 3.0);
m.rotate   (30.0, QVector3D(0.0, 0.0, 1.0));
m.translate(-1.0,-2.0,-3.0);
```

- Remember that the last matrix specified is the first applied

Arbitrary Matrices

- Can load and multiply by matrices defined in the application program
- Matrices are stored as one dimensional array of 16 elements which are the components of the desired 4 x 4 matrix stored by <u>columns</u>
- OpenGL functions that have matrices as parameters allow the application to send the matrix or its transpose

Vertex Shader for Rotation of Cube (1)

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
```

Vertex Shader for Rotation of Cube (2)

```
// Remember: these matrices are column-major

mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                0.0,  c.x,  s.x, 0.0,
                0.0, -s.x,  c.x, 0.0,
                0.0,  0.0,  0.0, 1.0 );

mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );
```

Vertex Shader for Rotation of Cube (3)

```
    mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                    s.z,  c.z, 0.0, 0.0,
                    0.0,  0.0, 1.0, 0.0,
                    0.0,  0.0, 0.0, 1.0 );
    color = vColor;
    gl_Position = rz * ry * rx * vPosition;
}
```

Sending Angles from Application

```
GLuint thetaLoc; // theta uniform location
vec3  theta;     // axis angles

void display( void )
{
   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
   glUniform3fv( thetaLoc, 1, theta );
   glDrawArrays( GL_TRIANGLES, 0, NumVertices );
}
```