Geometry and transformations
Overview

– Basic elements
– Vector spaces and affine spaces
– Coordinate systems and frames
– Representation
– Transformations
– Transformations in OpenGL
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
Scalars

• Scalars can be defined as members of sets which:
  – can be combined by two operations
    • addition and multiplication
  – obey some fundamental axioms
    • associativity, commutivity, inverses

• Examples include the real and complex number systems under the ordinary rules with which we are familiar

• Scalars alone have no geometric properties
Vectors

• Physical definition: a vector is a quantity with two attributes
  – Direction
  – Magnitude
• Examples include
  – Force
  – Velocity
  – Directed line segments
    • Most important example for graphics
    • Can map to other types
Vector Operations

• Every vector has an inverse
  – Same magnitude but points in opposite direction
• Every vector can be multiplied by a scalar
• There is a zero vector
  – Zero magnitude, undefined orientation
• The sum of any two vectors is a vector
  – Use head-to-tail axiom
Linear Vector Spaces

• Mathematical system for manipulating vectors
• Operations
  – Scalar-vector multiplication \( u = \alpha v \)
  – Vector-vector addition: \( w = u + v \)
• Expressions such as
  \( v = u + 2w - 3r \)
Make sense in a vector space
Vectors Lack Position

• These vectors are identical
  – Same length and magnitude

• Vectors spaces insufficient for geometry
  – Need points
Points

• Location in space

• Operations allowed between points and vectors
  – Point-point subtraction yields a vector
  – Equivalent to point-vector addition

P = v + Q

Q

P

v = P - Q

v

P = v + Q
Affine Spaces

• Operations
  – Vector-vector addition
  – Scalar-vector multiplication
  – Point-vector addition
  – Scalar-scalar operations

• For any point define
  – $1 \cdot P = P$
  – $0 \cdot P = \mathbf{0}$ (zero vector)
Lines

• Consider all points of the form
  \[ P(\alpha) = P_0 + \alpha \mathbf{d} \]
  – Set of all points that pass through \( P_0 \) in the direction of the vector \( \mathbf{d} \)
Parametric Form

- This form is known as the parametric form of the line
  - More robust and general than other forms
  - Extends to curves and surfaces

- Two-dimensional forms
  - Explicit: $y = mx + h$
  - Implicit: $ax + by + c = 0$
  - Parametric:
    
    $x(\alpha) = \alpha x_0 + (1-\alpha)x_1$
    $y(\alpha) = \alpha y_0 + (1-\alpha)y_1$
Planes

- A plane can be defined by a point and two vectors or by three points.

\[ \mathbf{P}(\alpha, \beta) = \mathbf{R} + \alpha \mathbf{u} + \beta \mathbf{v} \]

\[ \mathbf{P}(\alpha, \beta) = \mathbf{R} + \alpha (\mathbf{Q} - \mathbf{R}) + \beta (\mathbf{P} - \mathbf{Q}) \]
Normals

• Every plane has a vector \( n \) normal (perpendicular, orthogonal) to it

• From point-two vector form \( P(\alpha,\beta) = R + \alpha u + \beta v \), we know we can use the cross product to find

\[
 n = u \times v
\]
Linear Independence

• A set of vectors \( \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n \) is \textit{linearly independent} if

\[ \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \ldots + \alpha_n \mathbf{v}_n = 0 \text{ iff } \alpha_1 = \alpha_2 = \ldots = 0 \]

• 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, at least one can be written in terms of the others.
Dimension

• 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.
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
Coordinate Systems

- Consider a basis \( \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n \)
- A vector is written \( \mathbf{v} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \ldots + \alpha_n \mathbf{v}_n \)
- The list of scalars \( \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \) is the representation of \( \mathbf{v} \) with respect to the given basis
- We can write the representation as a row or column array of scalars
  \[
  \mathbf{a} = [\alpha_1 \quad \alpha_2 \quad \ldots \quad \alpha_n]^T =
  \begin{bmatrix}
  \alpha_1 \\
  \alpha_2 \\
  \vdots \\
  \alpha_n
  \end{bmatrix}
  \]
Example

• \( v = 2v_1 + 3v_2 - 4v_3 \)
• \( a = [2 \ 3 \ -4]^T \)
• Note that this representation is with respect to a particular basis
• OpenGL: start by representing vectors using the object basis
  – later the system needs a representation in terms of the camera or eye basis
Coordinate Systems

- Which is correct?

- Both are because vectors have no fixed location
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*
Representation in a Frame

• Frame determined by \((P_0, v_1, v_2, v_3)\)

• Within this frame, every vector can be written as
  \[ \mathbf{v} = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \]

• Every point can be written as
  \[ \mathbf{P} = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n \]
Confusing Points and Vectors

• Consider the point and the vector

\[ \mathbf{p} = \mathbf{P}_0 + \beta_1 \mathbf{v}_1 + \beta_2 \mathbf{v}_2 + \ldots + \beta_n \mathbf{v}_n \]

\[ \mathbf{v} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \ldots + \alpha_n \mathbf{v}_n \]

• They appear to have the similar representations

\[ \mathbf{p} = [\beta_1 \beta_2 \beta_3] \quad \mathbf{v} = [\alpha_1 \alpha_2 \alpha_3] \]

which confuses the point with the vector

A vector has no position

Vector can be placed anywhere

point: fixed
A Single Representation

If we define $0 \cdot P = 0$ and $1 \cdot P = P$ then we can write

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1 \alpha_2 \alpha_3 0] [v_1 v_2 v_3 P_0]^T$$

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [\beta_1 \beta_2 \beta_3 1] [v_1 v_2 v_3 P_0]^T$$

Thus we obtain the four-dimensional homogeneous coordinate representation

$$v = [\alpha_1 \alpha_2 \alpha_3 0]^T$$

$$p = [\beta_1 \beta_2 \beta_3 1]^T$$
Homogeneous Coordinates

The homogeneous coordinates form for a three dimensional point \([x \ y \ z]\) is given as
\[
p = [x' \ y' \ z' \ w]^T = [wx \ wy \ wz \ w]^T
\]
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 \([x \ y \ z \ 1]\)
Homogeneous Coordinates

• 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
Change of Coordinate Systems

- Consider two representations of a the same vector with respect to two different bases. The representations are

\[
\begin{align*}
\mathbf{a} &= [\alpha_1 \alpha_2 \alpha_3] \\
\mathbf{b} &= [\beta_1 \beta_2 \beta_3]
\end{align*}
\]

where

\[
\begin{align*}
\mathbf{v} &= \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \alpha_3 \mathbf{v}_3 = [\alpha_1 \alpha_2 \alpha_3] [\mathbf{v}_1 \mathbf{v}_2 \mathbf{v}_3]^T \\
\mathbf{u} &= \beta_1 \mathbf{u}_1 + \beta_2 \mathbf{u}_2 + \beta_3 \mathbf{u}_3 = [\beta_1 \beta_2 \beta_3] [\mathbf{u}_1 \mathbf{u}_2 \mathbf{u}_3]^T
\end{align*}
\]
Representing second basis in terms of first

Each of the basis vectors, $u_1, u_2, u_3$, are vectors that can be represented in terms of the first basis

$$u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$$
$$u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$$
$$u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$$
Matrix Form

The coefficients define a 3 x 3 matrix

\[ M = \begin{bmatrix}
\gamma_{11} & \gamma_{12} & \gamma_{13} \\
\gamma_{21} & \gamma_{22} & \gamma_{23} \\
\gamma_{31} & \gamma_{32} & \gamma_{33}
\end{bmatrix} \]

and the bases can be related by

\[ a = M^T b \]

see text for numerical examples
Change of Frames

• We can apply a similar process in homogeneous coordinates to the representations of both points and vectors

Consider two frames:

(P₀, v₁, v₂, v₃)
(Q₀, u₁, u₂, u₃)

• Any point or vector can be represented in either frame
• We can represent Q₀, u₁, u₂, u₃ in terms of P₀, v₁, v₂, v₃
Representing One Frame in Terms of the Other

Extending what we did with change of bases

\[
\begin{align*}
u_1 &= \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3 \\
u_2 &= \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3 \\
u_3 &= \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3 \\
Q_0 &= \gamma_{41}v_1 + \gamma_{42}v_2 + \gamma_{43}v_3 + \gamma_{44}P_0
\end{align*}
\]

defining a 4 x 4 matrix

\[
M = \begin{bmatrix}
\gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\
\gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\
\gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\
\gamma_{41} & \gamma_{42} & \gamma_{43} & 1
\end{bmatrix}
\]
Working with Representations

Within the two frames any point or vector has a representation of the same form

\[ a = [\alpha_1 \alpha_2 \alpha_3 \alpha_4] \] in the first frame
\[ b = [\beta_1 \beta_2 \beta_3 \beta_4] \] in the second frame

where \( \alpha_4 = \beta_4 = 1 \) for points and \( \alpha_4 = \beta_4 = 0 \) for vectors and

\[ a = M^T b \]

The matrix \( M \) is 4 x 4 and specifies an affine transformation in homogeneous coordinates.
Affine Transformations

• Every linear transformation is equivalent to a change in frames
• Every affine transformation preserves lines
• However, an affine transformation has only 12 degrees of freedom because 4 of the elements in the matrix are fixed and are a subset of all possible 4 x 4 linear transformations
The World and Camera Frames

• When we work with representations, we work with n-tuples or arrays of scalars
• Changes in frame are then defined by 4 x 4 matrices
• In OpenGL, the base frame that we start with is the world frame
• Eventually we represent entities in the camera frame by changing the world representation using the model-view matrix
• Initially these frames are the same ($M=I$)
Moving the Camera

If objects are on both sides of $z=0$, we must move camera frame.

$$
M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & -d \\
0 & 0 & 0 & 1
\end{bmatrix}
$$
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
• Importance in graphics is that
  – Need only transform endpoints of line segments
  – Let implementation draw line segment between the transformed endpoints
Notation

We will be working with both coordinate-free representations of transformations and representations within a particular frame

$P, Q, R$: points in an affine space
$u, v, w$: vectors in an affine space
$\alpha, \beta, \gamma$: scalars
$p, q, r$: representations of points
  - array of 4 scalars in homogeneous coordinates
$u, v, w$: representations of points
  - array of 4 scalars in homogeneous coordinates
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

  \[ \mathbf{p} = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^T \]
  \[ \mathbf{p}' = \begin{bmatrix} x' & y' & z' & 1 \end{bmatrix}^T \]
  \[ \mathbf{d} = \begin{bmatrix} dx & dy & dz & 0 \end{bmatrix}^T \]

- 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

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

- This form is better for implementation because
  - all affine transformations can be expressed this way
  - multiple transformations can be concatenated together
Rotation (2D)

Consider rotation about the origin by $\theta$ degrees
– radius stays the same, angle increases by $\theta$

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

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

$x = r \cos \phi$
$y = r \sin \phi$
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
  \[
p' = R_z(\theta)p
  \]
Rotation Matrix

\[ R = R_z(\theta) = \begin{bmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]
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

\[
R = R_x(\theta) = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
R = R_y(\theta) = \begin{bmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Scaling

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

\[ x' = s_x x \]
\[ y' = s_y x \]
\[ z' = s_z x \]
\[ p' = S p \]

\[ S = S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Reflection

corresponds to negative scale factors

$s_x = -1 \ s_y = 1$

$s_x = -1 \ s_y = -1$

$s_x = 1 \ s_y = -1$
Inverses

- Could compute inverse matrices by general formulas
- But simple geometric observations are easier
  - Translation: \( T^{-1}(d_x, d_y, d_z) = T(-d_x, -d_y, -d_z) \)
  - Rotation: \( R^{-1}(\theta) = R(-\theta) \)
    - Holds for any rotation matrix
    - Note that since \( \cos(-\theta) = \cos(\theta) \) and \( \sin(-\theta) = -\sin(\theta) \)
      \( R^{-1}(\theta) = R^T(\theta) \)
  - Scaling: \( S^{-1}(s_x, s_y, s_z) = S(1/s_x, 1/s_y, 1/s_z) \)
Concatenation

- Can form arbitrary affine transformation matrices by multiplying
  - rotation, translation, and scaling matrices
- Same transformation is applied to many vertices?
  - cost of forming a matrix $M=ABCD$ is not significant compared to the cost of computing $Mp$ for many vertices $p$
- The difficult part is how to form a desired transformation from the specifications in the application
Order of Transformations

- Note that matrix on the right is the first applied.
- Mathematically, the following are equivalent:
  \[ p' = ABCp = A(B(Cp)) \]
- Note many references use column matrices to represent points. In terms of column matrices:
  \[ p'^T = p^T C^T B^T A^T \]
General Rotation About the Origin

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

$$R(\theta) = R_z(\theta_z) \ R_y(\theta_y) \ R_x(\theta_x)$$

$\theta_x \ \theta_y \ \theta_z$ 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

\[ M = T(p_f) \cdot R(\theta) \cdot T(-p_f) \]
Instancing

• In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size

• We apply an instance transformation to its vertices to
  Scale
  Orient
  Locate

\[ M = TRS \]
Shear

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

Consider simple shear along $x$ axis

\[
x' = x + y \cot \theta \\
y' = y \\
z' = z
\]

\[
H(\theta) = \begin{bmatrix}
1 & \cot \theta & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
OpenGL Matrices

• In OpenGL matrices are part of the state
• Multiple types
  – Model-View (GL.GL_MODELVIEW)
  – Projection (GL.GL_PROJECTION)
  – Texture (GL.GL_TEXTURE) (ignore for now)
  – Color(GL.GL_COLOR) (ignore for now)
• Single set of functions for manipulation
• Select which to manipulated by
  – gl.glMatrixMode(GL.GL_MODELVIEW);
  – gl.glMatrixMode(GL.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
  - applied to all vertices that pass down the pipeline
- The CTM is defined in the user program and loaded into a transformation unit

\[ p' = C p \]
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:

```
gl.glLoadIdentity()
```

Multiply on right:

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

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

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

```
gl.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)

```c
    glMatrixMode(GL.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

\[
\text{glLoadMatrixf}\,(m) \\
\text{glMultMatrixf}\,(m)
\]

• 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 \text{glMultMatrixf}, \( m \) multiplies the existing matrix on the right
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()`
Reading Back Matrices

• Can also access matrices (and other parts of the state) by *query* functions

  ```
  glGetIntegerv
  glGetFloatv
  glGetBooleanv
  glGetDoublev
  glEnable
  ```

• For matrices, we use as

  ```
  double m[] = new double[16];
  gl.glGetFloatv(GL.GL_MODELVIEW, m);
  ```
Using Transformations

• Example: use idle function to rotate a cube and mouse function to change direction of rotation

• Start with a program that draws a cube (**Cube.java**) in a standard way
  – Centered at origin
  – Sides aligned with axes
  – Will discuss modeling in next lecture
Using the Model-view Matrix

• In OpenGL the model-view matrix is used to
  – Position the camera
    • Can be done by rotations and translations but is often easier to use *gluLookAt*
  – Build models of objects
• The projection matrix is used to define the view volume and to select a camera lens
Model-view and Projection Matrices

• Although both are manipulated by the same functions
  – be careful because incremental changes are always made by postmultiplication
  – For example, rotating model-view and projection matrices by the same matrix are not equivalent operations.
  – Postmultiplication of the model-view matrix is equivalent to premultiplication of the projection matrix
Model-view and Projection Matrices

• From the OpenGL Technical FAQ (http://www.opengl.org/resources/faq/technical/viewing.htm)

  – The GL_PROJECTION matrix should contain only the projection transformation calls it needs to transform eye space coordinates into clip coordinates.
  – The GL_MODELVIEW matrix, as its name implies, should contain modeling and viewing transformations, which transform object space coordinates into eye space coordinates. Remember to place the camera transformations on the GL_MODELVIEW matrix and never on the GL_PROJECTION matrix.
  – Think of the projection matrix as describing the attributes of your camera, such as field of view, focal length, fish eye lens, etc. Think of the ModelView matrix as where you stand with the camera and the direction you point it.
Smooth Rotation

• From a practical standpoint, we are often want to use transformations to move and reorient an object smoothly
  – Problem: find a sequence of model-view matrices $M_0, M_1, \ldots, M_n$ so that when they are applied successively to one or more objects we see a smooth transition

• For orientating an object, we can use the fact that every rotation corresponds to part of a great circle on a sphere
  – Find the axis of rotation and angle
  – Virtual trackball (see text)
Incremental Rotation

• Consider the two approaches
  – For a sequence of rotation matrices $R_0, R_1, \ldots, R_n$, find the Euler angles for each and use $R_i = R_{iz} R_{iy} R_{ix}$
    • Not very efficient
  – Use the final positions to determine the axis and angle of rotation, then increment only the angle
• Quaternions can be more efficient than either
• Next time:
  – Computer Viewing and Projection, Derivation of the Simple Normalization Matrix