1. Tool Tips – WebGL?
2. Geometry, Geometric objects, & Transformations
   Vectors, Matrices
3. Tomorrow’s Lab… Back to shaders……
4. Problem sets 5 and 6 ….. For tomorrow (and/or next Wednesday)
5. Recap Last Week’s Lab & Assignment
   - Ray Tracing Code Review – Isaac or Dani?
   - Comments about stretching the 3D Sierpinski?
6. The rest of the quarter….

Acknowledgements: Ed Angel, Jenny Orr, Ron Metoyer, Mike Bailey

1. 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

2. Geometry

   • Introduce the elements of geometry
     – Scalars
     – Vectors
     – Points
   • Develop mathematical operations among them in a coordinate-free manner
   • Define basic primitives
     – Line segments
     – Polygons

3. Scalars

   • Need three basic elements in geometry
     – Scalars, Vectors, Points
   • Scalars can be defined as members of sets which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutativity, inverses)
   • Examples include the real and complex number systems under the ordinary rules with which we are familiar
   • Scalars alone have no geometric properties

4. 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

5. 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

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

Points

- Location in space
- Operations allowed between points and vectors
  - Point-point subtraction yields a vector
  - Equivalent to point-vector addition

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

Affine Spaces

- Point + a vector space
- Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar operations
- For any point define
  - \( 1 \cdot P = P \)
  - \( 0 \cdot P = 0 \) (zero vector)

Vectors Lack Position

- These vectors are identical
  - Same length and magnitude
- Vectors spaces insufficient for geometry
  - Need points

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:
    \[
    x(\alpha) = \alpha x_0 + (1-\alpha)x_1 \\
    y(\alpha) = \alpha y_0 + (1-\alpha)y_1 
    \]

Affine Sums

Consider the “sum” \( P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n \)

Can show by induction that this sum makes sense iff \( \alpha_1+\alpha_2+\ldots+\alpha_n=1 \)

in which case we have the affine sum of the points \( P_1, P_2, \ldots, P_n \)

If, in addition, \( \alpha_i \geq 0 \), we have the convex hull of \( P_1, P_2, \ldots, P_n \)

Rays and Line Segments

If \( \alpha \geq 0 \), then \( P(\alpha) \) is the ray leaving \( P_0 \) in the direction \( \mathbf{d} \)

If we use two points to define \( \mathbf{v} \), then

\[
P(\alpha) = Q + \alpha \mathbf{v} \\
= Q + \alpha (R-Q) \\
= \alpha R + (1-\alpha)Q 
\]

For \( 0 \leq \alpha \leq 1 \) we get all the points on the line segment joining \( R \) and \( Q \)

Convex Hull

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

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

Curves and Surfaces

- Curves are one parameter entities of the form \( P(\alpha) \) where the function is nonlinear
- Surfaces are formed from two-parameter functions \( P(\alpha, \beta) \)
  - Linear functions give planes and polygons
Planes

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

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

\[
P(\alpha,\beta) = R + \alpha(Q-R) + \beta(P-Q)
\]

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 \( n = u \times v \) and the equivalent form \( (P(\alpha)-P) \cdot n = 0 \).

Linear Independence

- A set of vectors \( v_1, v_2, \ldots, v_n \) is linearly independent if
  \[\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n = 0 \text{ iff } \alpha_1 = \alpha_2 = \ldots = 0\]
- 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, at 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.

*See text, pp 129-133
Evergreen

Representation

• 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

• Which is correct?

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

Example

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

Representation in a Frame

• Frame determined by \((P_0, v_1, v_2, v_3)\)
• Within this frame, every vector can be written as \(v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n\)
• Every point can be written as \(P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n\)
Confusing Points and Vectors

Consider the point and the vector
\[ P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n \]
\[ v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n \]
They appear to have the similar representations
\[ p = [\beta_1, \beta_2, \ldots, \beta_n] \quad v = [\alpha_1, \alpha_2, \ldots, \alpha_n] \]
which confuses the point with the vector
A vector has no position
Vector can be placed anywhere

Homogeneous Coordinates

The homogeneous coordinates form for a three dimensional point \([x \ y \ z]\) is given as
\[ p = [x' \ y' \ z' \ w] \quad v = [x \ y \ z \ w] \]
We return to a three dimensional point (for \(w=0\) by
\[ x = x' / w \]
\[ y = y' / w \]
\[ z = z' / w \]
If \(w=0\), the representation is that of a vector
What is \(w\)?
Note: homogeneous coordinates replace points in three dimensions by lines through the origin in four dimensions
For \(w=1\), the representation of a point is \([x \ y \ z]\)

Homogeneous 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 4x4 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

Change of Coordinate Systems

Consider two representations of the same vector with respect to two different bases. The representations are
\[ a = [\alpha_1, \alpha_2, \alpha_3] \]
\[ b = [\beta_1, \beta_2, \beta_3] \]
where
\[ v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1, \alpha_2, \alpha_3] \quad [v_1 \ v_2 \ v_3] \]
\[ = \beta_1 u_1 + \beta_2 u_2 + \beta_3 u_3 = [\beta_1, \beta_2, \beta_3] \quad [u_1 \ u_2 \ u_3] \]

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
\[ u_1 = \gamma_1 v_1 + \gamma_2 v_2 + \gamma_3 v_3 \]
\[ u_2 = \gamma_2 v_1 + \gamma_2 v_2 + \gamma_3 v_3 \]
\[ u_3 = \gamma_3 v_1 + \gamma_2 v_2 + \gamma_3 v_3 \]

A Single Representation

If we define \(0 \cdot P = 0\) and \(1 \cdot P = P\) then we can write
\[ v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1, \alpha_2, \alpha_3, 0] \quad [v_1 \ v_2 \ v_3 \ P_0] \]
\[ P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [\beta_1, \beta_2, \beta_3, 1] \quad [v_1 \ v_2 \ v_3 \ P_0] \]
Thus we obtain the four-dimensional homogeneous coordinate representation
\[ v = [\alpha_1, \alpha_2, \alpha_3, 0] \quad [T] \]
\[ p = [\beta_1, \beta_2, \beta_3, 1] \quad [T] \]
The matrix $M$ is defined by 

$$
\begin{bmatrix}
Y_{11} & Y_{12} & Y_{13} \\
Y_{21} & Y_{22} & Y_{23} \\
Y_{31} & Y_{32} & Y_{33}
\end{bmatrix}
$$

and the bases can be related by 

$$a = M^T b$$

see text for numerical examples.

Any point or vector can be represented in either frame. 

Consider two frames: $(P_0, v_1, v_2, v_3)$ and $(Q_0, u_1, u_2, u_3)$.

- Any point or vector can be represented in either frame, e.g., we can represent $Q_0, u_1, u_2, u_3$ in terms of $P_0, v_1, v_2, v_3$.

Changes in frame are then defined by 4 x 4 matrices.

In OpenGL, the base frame that we start with is the model-view matrix. When we work with representations, we work with n-tuples or arrays of scalars.

Extending what we did with change of bases:

$$
\begin{align*}
&u_1 = \gamma_1 v_1 + \gamma_2 v_2 + \gamma_3 v_3 \\
u_2 = \gamma_4 v_1 + \gamma_5 v_2 + \gamma_6 v_3 \\
u_3 = \gamma_7 v_1 + \gamma_8 v_2 + \gamma_9 v_3 \\
Q_0 = \gamma_{10} v_1 + \gamma_{11} v_2 + \gamma_{12} v_3 + \gamma_{13} v_0
\end{align*}
$$

Defining a 4 x 4 matrix:

$$
M = \begin{bmatrix}
Y_{11} & Y_{12} & Y_{13} & 0 \\
Y_{21} & Y_{22} & Y_{23} & 0 \\
Y_{31} & Y_{32} & Y_{33} & 0 \\
Y_{41} & Y_{42} & Y_{43} & 1
\end{bmatrix}
$$

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.

Every affine transformation preserves lines.

Every linear transformation is equivalent to a change in frames.

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.

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.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Time-Check....

Affine Transformations

- Line preserving
- Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear
- In graphics: we need only transform endpoints of line segments. Then, let the implementation draw line segment between the transformed endpoints.

Transformations

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

Pipeline Implementation

$T$ (from application program)

$$\begin{align*}
T & \quad \text{transformation} \\
T(u) & \quad \text{rasterizer} \\
\text{frame buffer} & \\
T(v) & \quad \text{pixels}
\end{align*}$$
We will be working with both coordinate-free representations of transformations and representations within a particular frame.

**Notation**

- **P, Q, R:** points in an affine space
- **u, v, w:** vectors in an affine space
- **α, β, γ:** 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

\[
P' = P + d
\]

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

**Translation Using Representations**

Using the homogeneous coordinate representation in some frame:

\[
p = [x \ y \ z \ 1]^T
\]

\[
p' = [x' \ y' \ z' \ 1]^T
\]

\[
d = [dx \ dy \ dz \ 0]^T
\]

Hence \( p' = p + d \) or:

\[
x' = x + dx
\]

\[
y' = y + dy
\]

\[
z' = z + dz
\]

**Translation Matrix**

We can also express translation using a 4 x 4 matrix \( T \) in homogeneous coordinates:

\[
p' = Tp
\]

\[
T = [d_x \ d_y \ d_z \ 0]
\]

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

**Rotation (2D)**

Consider rotation about the origin by \( θ \) degrees:

- radius stays the same, angle increases by \( θ \)

\[
x = r \cos(ϕ + θ)
\]

\[
y = r \sin(ϕ + θ)
\]

\[
x' = x \cos θ - y \sin θ
\]

\[
y' = x \sin θ + y \cos θ
\]

\[
x = r \cos ϕ
\]

\[
y = r \sin ϕ
\]
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
    \[ p' = R_2(\theta) p \]

Rotation Matrix

\[ R = R_2(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

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.

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

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

Scaling

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

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

Reflect

Corresponds to negative scale factors.

\[ s_x = -1, s_y = 1 \]
\[ s_x = -1, s_y = -1 \]
\[ s_x = 1, s_y = -1 \]

Inverses

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations.
  - Translation: \( T^{-1}(d_x, d_y, d_z) = T(-d_x, -d_y, -d_z) \)
  - 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

- 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 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.

Is this because order matters?

Rotation About a Fixed Point other than the Origin

Move fixed point to origin
Rotate
Move fixed point back
\( M = T(p_f) \, R(\theta) \, T(-p_f) \)

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:
  \[ p'^T = p^T C^T B^T A^T \]

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

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:

\[ R(\theta) = R_x(\theta_y) \, R_y(\theta_z) \, R_z(\theta_x) \]

\( \theta_x, \theta_y, \theta_z \) are called the Euler angles.

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

Shear

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

Consider simple shear along x axis

\[ \begin{align*}
x' &= x + y \cot \theta \\
y' &= y \\
z' &= z
\end{align*} \]

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

Pre 3.1 OpenGL Matrices

- Matrices were part of the state
- Multiple types
  - Model-View (GL_MODELVIEW)
  - Projection (GL_PROJECTION)
  - Texture (GL_TEXTURE)
  - Color (GL_COLOR)
- Single set of functions for manipulation
- Select which to manipulate by
  - glMatrixMode(GL_MODELVIEW);
  - glMatrixMode(GL_PROJECTION);

OpenGL Transformations

- Learn how to carry out OpenGL transformations
  - Rotation
  - Translation
  - Scaling
- Introduce mat.h & vec.h 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

```
vertices -> CTM -> vertices
```

CTM operations

- The CTM can be altered either by loading a new CTM or by postmultiplication
  - Load an identity matrix: C ← I
  - Load an arbitrary matrix: C ← M
  - Load a translation matrix: C ← T
  - Load a rotation matrix: C ← R
  - Load a scaling matrix: C ← S
  - Postmultiply by an arbitrary matrix: C ← CM
  - Postmultiply by a translation matrix: C ← CT
  - Postmultiply by a rotation matrix: C ← CR
  - Postmultiply by a scaling matrix: C ← 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 $C = T^{-1} RT$ 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: the last operation specified is the first executed in the program.

Rotation, Translation, Scaling

Create an identity matrix:

```
mat4 m = Identity();
```

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

```
mat4 r = Rotate(theta, vx, vy, vz)
m = m*r;
```

Do same with translation and scaling:

```
mat4 s = Scale(sx, sy, sz)
mat4 t = Translate(dx, dy, dz);
m = m*s*t;
```

Example

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

```
mat 4 m = Identity();
m = Translate(1.0, 2.0, 3.0)*
Rotate(30.0, 0.0, 0.0, 1.0)*
Translate(-1.0, -2.0, -3.0);
```

• Remember that last matrix specified in the program is the first applied.

CTM in OpenGL

• OpenGL had a model-view and a projection matrix in the pipeline which were concatenated together to form the CTM
• We will emulate this process

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 columns
• OpenGL functions that have matrices as parameters allow the application to send the matrix or its transpose
Matrix Stacks

- In many situations we want to save transformation matrices for use later
  - Traversing hierarchical data structures (Chapter 8)
  - Avoiding state changes when executing display lists
- Pre 3.1 OpenGL maintained stacks for each type of matrix
- Easy to create the same functionality with a simple stack class

void main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutCreateWindow("colorcube");
  glutDisplayFunc(display);
  glutReshapeFunc(myReshape);
  glutIdleFunc(spinCube);
  glutMouseFunc(mouse);
  glEnable(GL_DEPTH_TEST);
  glutMainLoop();
}

Reading Back State

- Can also access OpenGL variables (and other parts of the state) by query functions
  
  glGetIntegerv
  glGetFloatv
  glGetBooleanv
  glGetDoublev
  glIsEnabled

Idle and Mouse callbacks

void spinCube()
{
  theta[axis] += 2.0;
  if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
  glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{
  if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    axis = 0;
  if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
    axis = 1;
  if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    axis = 2;
}

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 in a standard way
  - Centered at origin
  - Sides aligned with axes
  
  (Will discuss modeling next week)

Display callback

- can form a matrix in the application and send it to the shader and let shader do the rotation
  
  - can send the angle and axis to the shader and let the shader form the transformation matrix and then do the rotation
  
  More efficient than transforming data in application & resending the data

void display()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glDrawArrays(GL_TRIANGLES, 0, 36);
  glutSwapBuffers();
}
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 a LookAt function
  - Build models of objects
  - The projection matrix is used to define the view volume and to select a camera lens
  - Although these matrices are no longer part of the OpenGL state, it is usually a good strategy to create them in our own applications

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 $R_p, R_t, \ldots, R_h$, find the Euler angles for each and use $R_f = R_{th} R_{th} R_{th}$
    - 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

Interfaces

- One of the major problems in interactive computer graphics is how to use two-dimensional devices such as a mouse to interface with three dimensional objects
- Example: how to form an instance matrix*?
- Some alternatives
  - Virtual trackball
  - 3D input devices such as the spaceball
  - Use areas of the screen
    - Distance from center controls angle, position, scale depending on mouse button depressed
  *p 168-169

Lab/Asst 05: A Simple Program (?)

Generate a square on a solid background

* * p. 186