CIS 441/541: Introduction to Computer Graphics
Lecture 12: matrices in OpenGL, project 2B

May 15th, 2013
Hank Childs, University of Oregon
OH

- OH this week: Weds 2-4, Fri 2-4

- Other:
  - Will make signup document this weekend
  - Will grade this weekend
  - Project 2B coming Weds.
    - Get on project 2A!!

- Special colloquium on Thursday
Project #2A (8%), Due Thurs May 16th, 6am

- Goal: OpenGL program that does regular colors and textures
- New VTK-based project2A.cxx
- New CMakeLists.txt (but same as old ones)

Clear what to do?
ModelView and Projection Matrices

- **ModelView idea**: two purposes ... model and view
  - **Model**: extra matrix, just for rotating, scaling, and translating geometry.
    - How could this be useful?
  - **View** Cartesian to Camera transform
- *(We will focus on the model part of the modelview matrix now & come back to others later)*
Common commands for modifying model part of ModelView matrix

- glTranslate
- glRotate
- glScale
NAME

glTranslated, glTranslatef - multiply the current matrix by a translation matrix

C SPECIFICATION

void glTranslated( GLdouble x,
                    GLdouble y,
                    GLdouble z )

void glTranslatef( GLfloat x,
                    GLfloat y,
                    GLfloat z )

PARAMETERS

x, y, z

Specify the x, y, and z coordinates of a translation vector.

DESCRIPTION

glTranslate produces a translation by (x,y,z). The current matrix (see
glMatrixMode) is multiplied by this translation matrix, with the product replacing the current
matrix, as if glMultMatrix were called with the following matrix for its argument:

\[
\begin{align*}
1 & 0 & 0 & x \\
0 & 1 & 0 & y \\
0 & 0 & 1 & z \\
0 & 0 & 0 & 1
\end{align*}
\]
**NAME**

*glRotated, glRotatef* - multiply the current matrix by a rotation matrix

**C SPECIFICATION**

```c
void glRotated( GLdouble angle,
               GLdouble x,
               GLdouble y,
               GLdouble z )

void glRotatef( GLfloat angle,
                GLfloat x,
                GLfloat y,
                GLfloat z )
```

**PARAMETERS**

- `angle` Specifies the angle of rotation, in degrees.
- `x, y, z`
  - Specify the `x, y, and z` coordinates of a vector, respectively.

**DESCRIPTION**

*glRotate* produces a rotation of `angle` degrees around the vector `(x, y, z)`. The current matrix (see *glMatrixMode*) is multiplied by a rotation matrix with the product replacing the current matrix, as if *glMultMatrix* were called with the following matrix as its argument:

\[
\begin{pmatrix}
  x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\
  yx(1-c)+zs & y^2(1-c)+c & yz(1-c)+xs & 0 \\
  xz(1-c)-ys & yz(1-c)+xs & z^2(1-c)+c & 0 \\
  0 & 0 & 0 & 1
\end{pmatrix}
\]

Where \( c = \cos (\text{angle}) \), \( s = \sin (\text{angle}) \), and \( \| (x, y, z) \| = 1 \) (if not, the GL will normalize this vector).
NAME

glScaled, glScalef — multiply the current matrix by a general scaling matrix

C SPECIFICATION

void glScaled( GLdouble x,
               GLdouble y,
               GLdouble z )
void glScalef( GLfloat x,
               GLfloat y,
               GLfloat z )

PARAMETERS

x, y, z

Specify scale factors along the x, y, and z axes, respectively.

DESCRIPTION

glScale produces a nonuniform scaling along the x, y, and z axes. The three parameters indicate the desired scale factor along each of the three axes.

The current matrix (see glMatrixMode) is multiplied by this scale matrix, and the product replaces the current matrix as if glScale were called with the following matrix as its argument:

\[
\begin{pmatrix}
x & 0 & 0 & 0 \\
0 & y & 0 & 0 \\
0 & 0 & z & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
How do transformations combine?

\begin{itemize}
  \item \texttt{glScale}(2, 2, 2)
  \item \texttt{glTranslate}(1, 0, 0)
  \item \texttt{glRotate}(45, 0, 1, 0)
\end{itemize}

→ Rotate by 45 degrees around \((0, 1, 0)\), then translate in \(X\) by 1, then scale by 2 in all dimensions.

→ (the last transformation is applied first)
Which of two of these three are the same?

- Choice A:
  - `glTranslatef(1, 0, 0);`
  - `glScalef(2, 2, 2);`

- Choice B:
  - `glTranslatef(1, 0, 0);`
  - `glScalef(2, 2, 2);`

- Choice C:
  - `glTranslatef(2, 0, 0);`
  - `glScalef(2, 2, 2);`
ModelView usage

dl = GenerateTireGeometry();
glCallList(dl); // place tire at (0, 0, 0)
glTranslatef(10, 0, 0);
glCallList(dl); // place tire at (10, 0, 0)
glTranslatef(0, 0, 10);
glCallList(dl); // place tire at (10, 0, 10)
glTranslatef(-10, 0, 0);
glCallList(dl); // place tire at (0, 0, 10)

each glTranslatef call updates the state of the ModelView matrix.
**NAME**

`glPushMatrix, glPopMatrix` - push and pop the current matrix stack

**C SPECIFICATION**

```c
void glPushMatrix( void )
```

```c
void glPopMatrix( void )
```

**DESCRIPTION**

There is a stack of matrices for each of the matrix modes. In **GL_MODELVIEW** mode, the stack depth is at least 32. In the other two modes, **GL_PROJECTION** and **GL_TEXTURE**, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

`glPushMatrix` pushes the current matrix stack down by one, duplicating the current matrix. That is, after a `glPushMatrix` call, the matrix on top of the stack is identical to the one below it.

`glPopMatrix` pops the current matrix stack, replacing the current matrix with the one below it on the stack.

Initially, each of the stacks contains one matrix, an identity matrix.
glPushMatrix

and

glPopMatrix

dl = GenerateTireGeometry();
glCallList(dl); // place tire at (0, 0, 0)
glPushMatrix();
glTranslatef(10, 0, 0);
glCallList(dl); // place tire at (10, 0, 0)
glPushMatrix();
glTranslatef(0, 0, 10);
glCallList(dl); // place tire at (10, 0, 10) (0, 0, 10)
glPopMatrix();
Matrices in OpenGL

- OpenGL maintains matrices for you and provides functions for setting matrices.

- There are four different modes you can use:
  - Modelview
  - Projection
  - Texture
  - Color (rarely used, often not supported)

- You control the mode using `glMatrixMode`. 
Matrices in OpenGL (cont’d)

- The matrices are the identity matrix by default and you can modify them by:
  - 1) setting the matrix explicitly
  - 2) using OpenGL commands for appending to the matrix

- You can have $\geq 32$ matrices for modelview, $\geq 2$ for others
8.010 How does the camera work in OpenGL?

As far as OpenGL is concerned, there is no camera. More specifically, the camera is always located at the eye space coordinate (0., 0., 0.). To give the appearance of moving the camera, your OpenGL application must move the scene with the inverse of the camera transformation.

8.020 How can I move my eye, or camera, in my scene?

OpenGL doesn't provide an interface to do this using a camera model. However, the GLU library provides the gluLookAt() function, which takes an eye position, a position to look at, and an up vector, all in object space coordinates. This function computes the inverse camera transform according to its parameters and multiplies it onto the current matrix stack.
8.030 Where should my camera go, the ModelView or Projection matrix?

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.

Source: www.opengl.org/archives/resources/faq/technical/viewing.htm
How do you put the Camera Transform in the ModelView matrix?

- No single GL call.

- Options are:
  - (1) you do it yourself (i.e., calculate matrix and load it into OpenGL)
  - (2) you use somebody’s code, i.e., gluLookAt
  - (3) you use a combination of glRotatef, glScalef, and glTranslatef commands.
NAME

`glMatrixMode` - specify which matrix is the current matrix

C SPECIFICATION

```c
void glMatrixMode( GLenum mode )
```

PARAMETERS

- `mode` Specifies which matrix stack is the target for subsequent matrix operations. Three values are accepted: `GL_MODELVIEW`, `GL_PROJECTION`, and `GL_TEXTURE`. The initial value is `GL_MODELVIEW`.

Additionally, if the `GL_ARB_imaging` extension is supported, `GL_COLOR` is also accepted.

DESCRIPTION

`glMatrixMode` sets the current matrix mode. `mode` can assume one of four values:

- **GL_MODELVIEW**
  Applies subsequent matrix operations to the modelview matrix stack.

- **GL_PROJECTION**
  Applies subsequent matrix operations to the projection matrix stack.

- **GL_TEXTURE**
  Applies subsequent matrix operations to the texture matrix stack.

- **GL_COLOR**
  Applies subsequent matrix operations to the color matrix stack.

To find out which matrix stack is currently the target of all matrix operations, call `glGet` with argument `GL_MATRIX_MODE`. The initial value is `GL_MODELVIEW`. 
How do you put the projection transformation in GL_PROJECTION?

- **Two options:**
  - `glFrustum()` (perspective projection)
  - `glOrtho()` (orthographic projection)
NAME

glFrustum — multiply the current matrix by a perspective matrix

C SPECIFICATION

void glFrustum( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar )

PARAMETERS

left, right Specify the coordinates for the left and right vertical clipping planes.

bottom, top Specify the coordinates for the bottom and top horizontal clipping planes.

zNear, zFar Specify the distances to the near and far depth clipping planes. Both distances must be positive.

DESCRIPTION

glFrustum describes a perspective matrix that produces a perspective projection. The current matrix (see glMatrixMode) is multiplied by this matrix and the result replaces the current matrix, as if glMatrixMode were called with the following matrix as its argument:

\[
\begin{pmatrix}
2 & zNear & 0 & A \\
right - left & 0 & 0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
2 & zNear & 0 & B \\
\top - \bottom & 0 & 0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
0 & 0 & C & D \\
0 & 0 & -1 & 0 \\
\end{pmatrix}
\]

A = (right + left) / (right - left)
B = (top + bottom) / (top - bottom)
C = -(zFar + zNear) / (zFar - zNear)
D = -(2 zFar zNear) / (zFar - zNear)

Typically, the matrix mode is GL_PROJECTION, and (left, bottom, -zNear) and (right, top, -zNear) specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, assuming that the eye is located at (0, 0, 0). zFar specifies the location of the far clipping plane. Both zNear and zFar must be positive.

Use glPushMatrix and glPopMatrix to save and restore the current matrix stack.
NAME

**glOrtho** — multiply the current matrix with an orthographic matrix

C SPECIFICATION

```c
void glOrtho( Gldouble left,
               Gldouble right,
               Gldouble bottom,
               Gldouble top,
               Gldouble zNear,
               Gldouble zFar )
```

PARAMETERS

- `left`, `right` Specify the coordinates for the left and right vertical clipping planes.
- `bottom`, `top` Specify the coordinates for the bottom and top horizontal clipping planes.
- `zNear`, `zFar` Specify the distances to the nearer and farther depth clipping planes. These values are negative if the plane is to be behind the viewer.

DESCRIPTION

**glOrtho** describes a transformation that produces a parallel projection. The current matrix (see **glMatrixMode**) is multiplied by this matrix and the result replaces the current matrix, as if **glMultMatrix** were called with the following matrix as its argument:

\[
\begin{bmatrix}
2 & 0 & 0 & tx \\
0 & 2 & 0 & ty \\
0 & 0 & -2 & tz \\
0 & 0 & zFar-zNear & 1 \\
\end{bmatrix}
\]

where

\[
tx = - (right + left) / (right - left)
\]
\[
ty = - (top + bottom) / (top - bottom)
\]
\[
 tz = - (zFar + zNear) / (zFar - zNear)
\]

Typically, the matrix mode is **GL_PROJECTION**, and **(left, bottom, -zNear)** and **(right, top, -zNear)** specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, respectively, assuming that the eye is located at (0, 0, 0). **-zFar** specifies the location of the far clipping plane. Both `zNear` and `zFar` can be either positive or negative.

Use **glPushMatrix** and **glPopMatrix** to save and restore the current matrix stack.
virtual void RenderPiece(vtkRenderer *ren, vtkRenderWindow *renWin)
{
    RemoveVTKOpenGLStateSideEffects();
    SetupLight();

    glMatrixMode(GL_TEXTURE);
    glPushMatrix();
    glScalef(3, 2.5, 1);

    glEnable(GL_TEXTURE_2D);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, p, q, 0, GL_RGB, GL_UNSIGNED_BYTE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, 0);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, 0);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glBegin(GL_QUADS);
    glTexCoord2f(0, 0);
    glVertex3f(0, 0, 0);
    glEnd();
}
Goal: modify ModelView matrix to create dog out of spheres and cylinders

New code skeleton: “project2B.cxx”

No geometry file needed.

You will be able to do this w/ glPush/PopMatrix, glRotatef, glTranslatef, and glScalef.
Contents of project2B.cxx

- Routine for generating spheres
- Routine for generating cylinders
- Routine for generating head, eyes, and pupils

- We will study all of these together for the remainder of the class.
The correct answer is:

- Something that looks like a dog
  - No obvious problems with output geometry.
- Something that uses the sphere and cylinder classes.
  - If you use something else, please clear it with me first.
    - I may fail your project if I think you are using outside resources that make the project too easy.
- Something that uses rotation for the neck and tail.

Aside from that, feel free to be as creative as you want ... color, breed, etc.
For your reference: my dog
We will now look at:

- Code to generate a cylinder
- Code to generate a sphere
- Code that uses spheres to start head.