Announcements

- OH today: 230pm-315pm (*not* 3pm-4pm)
- Class canceled Monday Nov. 14th
  - Also Weds Nov 30th
- Midterm Friday Nov 18th
Lectures
- Abhishek: Surfaces and Curves
- Andy: 3D rotation
- Jacob: Animation and Scene Graph
- Nicole: Shader programs
- Chad: GPGPU
- Garrett: modeling
- Kyle: particle effects

541 students: when will we do these?
Projects:

- Manish, Lumin, Vincent: shader programs (x3)
- Ryan: Unity
- Sudhanshu: GPGPU
- Raleigh: WebGL

541 students: when will these be ready?
Most common configuration has CPU and GPU on separate dies

- i.e., plug GPU in CPU

CPU (typically 4-12 cores, ~10GFLOPs)

GPU (typically 100-1000 cores, ~100GFLOPs-~1000GFLOPs)

Peripheral Component Interconnect Express

What are the performance ramifications of this architecture?
Display lists

- Idea:
  - send geometry and settings to GPU once, give it an identifier
  - GPU stores geometry and settings
  - Just pass the identifier for every subsequent render
Display lists

- Generate an identifier:
  \[
  \text{GLUint displayList} = \text{glGenLists}(1);
  \]

- Tell GPU that all subsequent geometry is part of the list:
  \[
  \text{glNewList(displayList,GL_COMPILE)};
  \]

- Specify geometry (i.e., glVertex, etc)

- Tell GPU we are done specifying geometry:
  \[
  \text{glEndList}();
  \]

- Later on, tell GPU to render all the geometry and settings associated with our list:
  \[
  \text{glCallList(displayList)};
  \]
Display lists in action

```c
for (int frame = 0 ; frame < nFrames ; frame++)
{
    SetCamera(frame, nFrames);
    glBegin(GL_TRIANGLES);
    for (int i = 0 ; i < triangles.size() ; i++)
    {
        for (int j = 0 ; j < 3 ; j++)
        {
            glColor3ubv(triangles[i].colors[j]);
            glColor3fv(triangles[i].vertices[j]);
        }
        glEnd();
    }
}
```

```c
GLuint displayList = glGenLists(1);
glNewList(displayList, GL_COMPILE);
glBegin(GL_TRIANGLES);
for (int i = 0 ; i < triangles.size() ; i++)
{
    for (int j = 0 ; j < 3 ; j++)
    {
        glColor3ubv(triangles[i].colors[j]);
        glColor3fv(triangles[i].vertices[j]);
    }
    glEnd();
    glEndList();
}
```

```c
for (int frame = 0 ; frame < nFrames ; frame++)
{
    SetCamera(frame, nFrames);
    glCallList(displayList);
}
```
glNewList

- **GL_COMPILE**
  - Make the display list for later use.

- **GL_COMPILE_AND_EXECUTE**
  - Make the display list and also execute it as you go.
Geometry Specification: glBegin

Name

glBegin — delimit the vertices of a primitive or a group of like primitives

C Specification

```c
void glBegin(GLenum mode);
```

Parameters

`mode`

Specifies the primitive or primitives that will be created from vertices presented between `glBegin` and the subsequent `glEnd`. Ten symbolic constants are accepted: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP`, and `GL_POLYGON`.

C Specification

```c
void glEnd( void);
```

Description

`glBegin` and `glEnd` delimit the vertices that define a primitive or a group of like primitives. `glBegin` accepts a single argument that specifies in which of ten ways the vertices are interpreted. Taking \( n \) as an integer count starting at one, and \( N \) as the total number of vertices specified, the interpretations are as follows:
GL_POINTS

Treats each vertex as a single point. Vertex n defines point n. N points are drawn.

GL_LINES

Treats each pair of vertices as an independent line segment. Vertices 2 \( \times n - 1 \) and 2 \( \times n \) define line n. N 2 lines are drawn.

GL_LINE_STRIP

Draws a connected group of line segments from the first vertex to the last. Vertices n and n + 1 define line n. N - 1 lines are drawn.

GL_LINE_LOOP

Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices n and n + 1 define line n. The last line, however, is defined by vertices N and 1. N lines are drawn.
Geometry Primitives

GL_TRIANGLES

Treats each triplet of vertices as an independent triangle. Vertices \(3 \times n - 2\), \(3 \times n - 1\), and \(3 \times n\) define triangle \(n\). \(N\) 3 triangles are drawn.

GL_TRIANGLE_STRIP

Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. For odd \(n\), vertices \(n\), \(n + 1\), and \(n + 2\) define triangle \(n\). For even \(n\), vertices \(n + 1\), \(n\), and \(n + 2\) define triangle \(n\). \(N - 2\) triangles are drawn.

GL_TRIANGLE_FAN

Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. Vertices \(1\), \(n + 1\), and \(n + 2\) define triangle \(n\). \(N - 2\) triangles are drawn.
Geometry Primitives

GL_QUADS

Treats each group of four vertices as an independent quadrilateral. Vertices $4 \times n - 3$, $4 \times n - 2$, $4 \times n - 1$, and $4 \times n$ define quadrilateral $n$. $N$ $4$ quadrilaterals are drawn.

GL_QUAD_STRIP

Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices $2 \times n - 1$, $2 \times n$, $2 \times n + 2$, and $2 \times n + 1$ define quadrilateral $n$. $N 2 - 1$ quadrilaterals are drawn. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data.

GL_POLYGON

Draws a single, convex polygon. Vertices $1$ through $N$ define this polygon.
What can go inside a `glBegin`?

Only a subset of GL commands can be used between `glBegin` and `glEnd`. The commands are `glVertex`, `glColor`, `glSecondaryColor`, `glIndex`, `glNormal`, `glFogCoord`, `glTexCoord`, `glMultiTexCoord`, `glVertexAttrib`, `glEvalCoord`, `glEvalPoint`, `glArrayElement`, `glMaterial`, and `glEdgeFlag`. Also, it is acceptable to use `glCallList` or `glCallLists` to execute display lists that include only the preceding commands. If any other GL command is executed between `glBegin` and `glEnd`, the error flag is set and the command is ignored.
Lighting

- glEnable(GL_LIGHTING);
  - Tells OpenGL you want to have lighting.

- Eight lights
  - Enable and disable individually
    - glEnable(GL_LIGHT0)
    - glDisable(GL_LIGHT7)
  - Set attributes individually
    - glLightfv(GL_LIGHTi, ARGUMENT, VALUES)


Ten parameters (ones you will use):

- **GL_AMBIENT**
  
  `params` contains four fixed-point or floating-point values that specify the ambient RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped. The initial ambient light intensity is `(0, 0, 0, 1)`.

- **GL_DIFFUSE**
  
  `params` contains four fixed-point or floating-point values that specify the diffuse RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped. The initial value for `GL_LIGHT0` is `(1, 1, 1, 1)`. For other lights, the initial value is `(0, 0, 0, 0)`.

- **GL_SPECULAR**
  
  `params` contains four fixed-point or floating-point values that specify the specular RGBA intensity of the light. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped. The initial value for `GL_LIGHT0` is `(1, 1, 1, 1)`. For other lights, the initial value is `(0, 0, 0, 0)`.

- **GL_POSITION**
  
  `params` contains four fixed-point or floating-point values that specify the position of the light in homogeneous object coordinates. Both fixed-point and floating-point values are mapped directly. Neither fixed-point nor floating-point values are clamped.

  The position is transformed by the modelview matrix when `glLight` is called (just as if it were a point), and it is stored in eye coordinates. If the w component of the position is 0, the light is treated as a directional source. Diffuse and specular lighting calculations take the light's direction, but not its actual position, into account, and attenuation is disabled. Otherwise, diffuse and specular lighting calculations are based on the actual location of the light in eye coordinates, and attenuation is enabled. The initial position is `(0, 0, 1, 0)`; thus, the initial light source is directional, parallel to, and in the direction of the -z axis.
For each light source, we can set an RGBA for the diffuse, specular, and ambient components:

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
GLfloat diffuse0[4] = { 0.7, 0.7, 0.7, 1 };
gllightfv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
… // set ambient, specular, position
glDisable(GL_LIGHT1); // do we need to do this?
…
glDisable(GL_LIGHT7); // do we need to do this?
How do we tell OpenGL about the surface normals?

- **Flat shading:**
  
  ```
  glNormal3f(0, 0.707, -0.707);
  glVertex3f(0, 0, 0);
  glVertex3f(1, 1, 0);
  glVertex3f(1, 0, 0);
  ```

- **Smooth shading:**
  
  ```
  glNormal3f(0, 0.707, -0.707);
  glVertex3f(0, 0, 0);
  glNormal3f(0, 0.707, +0.707);
  glVertex3f(1, 1, 0);
  glNormal3f(1, 0, 0);
  glVertex3f(1, 0, 0);
  ```
glShadeModel — select flat or smooth shading

C Specification

```c
void glShadeModel(GLenum mode);
```

Parameters

`mode`

Specifies a symbolic value representing a shading technique. Accepted values are `GL_FLAT` and `GL_SMOOTH`. The initial value is `GL_SMOOTH`.

Description

GL primitives can have either flat or smooth shading. Smooth shading, the default, causes the computed colors of vertices to be interpolated as the primitive is rasterized, typically assigning different colors to each resulting pixel fragment. Flat shading selects the computed color of just one vertex and assigns it to all the pixel fragments generated by rasterizing a single primitive. In either case, the computed color of a vertex is the result of lighting if lighting is enabled, or it is the current color at the time the vertex was specified if lighting is disabled.

Flat and smooth shading are indistinguishable for points. Starting when `glBegin` is issued and counting vertices and primitives from 1, the GL gives each flat-shaded line segment its computed color of vertex \( i + 1 \), its second vertex. Counting similarly from 1, the GL gives each flat-shaded polygon the computed color of the vertex listed in the following table. This is the last vertex to specify the polygon in all cases except single polygons, where the first vertex specifies the flat-shaded color.
Transforms in GL
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)
SLIDE REPEAT: Our goal

Add additional transforms here….

World space:
- Triangles in native Cartesian coordinates
- Camera located anywhere

Camera space:
- Camera located at origin, looking down -Z
- Triangle coordinates relative to camera frame

Image space:
- All viewable objects within $-1 \leq x, y, z \leq +1$

Screen space:
- All viewable objects within $-1 \leq x, y \leq +1$

Device space:
- All viewable objects within $0 \leq x \leq \text{width}, 0 \leq y \leq \text{height}$
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

```c
void glTranslated( GLfloat x,
                  GLfloat y,
                  GLfloat z )
```

```c
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:

```
1 0 0 x
0 1 0 y
0 0 1 z
0 0 0 1
```

Note: this matrix transposed from what we did earlier
**glRotate**

**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:

```
| 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  |
```

Where `c = cos (angle)`, `s = sin (angle)`, and `||x, y, z|| = 1` (if not, the GL will normalize this vector).
glScale

**NAME**

`glScaled`, `glScalef` - multiply the current matrix by a general scaling matrix

**C SPECIFICATION**

```c
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:

```
  x 0 0 0  
 0 y 0 0  
0 0 z 0  
0 0 0 1  
```

How do transformations combine?

`glScale(2, 2, 2)`

`glTranslate(1, 0, 0)`

`glRotate(45, 0, 1, 0)`

→ 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:**
  - `glScalef(2, 2, 2);`
  - `glTranslate(1, 0, 0);`

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

- **Choice C:**
  - `glTranslate(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.
dl = GenerateTireGeometry();
glCallList(dl); // place tire at (0, 0, 0)
glPushMatrix();
glTranslatef(10, 0, 0);
glCallList(dl); // place tire at (10, 0, 0)
glPopMatrix();
glPushMatrix();
glTranslatef(0, 0, 10);
glCallList(dl); // place tire at (10, 0, 10) (0, 0, 10)
glPopMatrix();

Why is this useful?
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.
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.

Source: www.opengl.org/archives/resources/faq/technical/viewing.htm
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.
glMatrixMode

NAME

glMatrixMode - specify which matrix is the current matrix

C SPECIFICATION

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)
glFrustum

NAME

- multiply the current matrix by a perspective matrix

C SPECIFICATION

```c
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 `glMultMatrix` were called with the following matrix as its argument:

\[
\begin{bmatrix}
2 & z\text{Near} & 0 & A \\
right - left & 0 & 0 & 0 \\
2 & z\text{Near} & -1 & 0 \\
\text{top} - \text{bottom} & 0 & 0 & B \\
0 & 0 & C & D \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

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 \((\text{left}, \text{bottom}, -z\text{Near})\) and \((\text{right}, \text{top}, -z\text{Near})\) 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 \((0, 0, 0)\). \(-z\text{Far}\) specifies the location of the far clipping plane. Both \(z\text{Near}\) and \(z\text{Far}\) 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, pwidth, pheight, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glBegin(GL_QUADS);
    glTexCoord2f(0, 0);
    glVertex2f(0, 0);
    glTexCoord2f(1, 1);
    glVertex2f(pwidth, pheight);
    glTexCoord2f(1, 0);
    glVertex2f(pwidth, 0);
    glTexCoord2f(0, 1);
    glVertex2f(0, pheight);
    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.
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 with 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
What is the correct answer?

- 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