CIS 441/541: Introduction to Computer Graphics
Lecture 12: transforms in GL, Project 2B
Announcements

- 1D redux: cancelled
  - Will be 1E & 1F
- Reminder: class canceled on Weds Nov 12th
  - OH canceled too
  - OH canceled Weds Nov 19th as well
Outline

- Review
- Display lists
- Transforms in GL
- Project 2B
Outline

- Review
- Display lists
- Transforms in GL
- Project 2B
Textures with GL_REPEAT

class vtk44PolyDataMapper : public vtkOpenGLPolyDataMapper
{
    public:
        static vtk44PolyDataMapper *New();

    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        GLubyte Texture3[9] = {
            0, 0, 255, // blue
            255, 255, 255, // white
            255, 0, 0, // red
        };
        glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, 3, 0, GL_RGB,
                     GL_UNSIGNED_BYTE, Texture3);
        glEnable(GL_COLOR_MATERIAL);
        glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

        glEnable(GL_TEXTURE_1D);
        float ambient[3] = { 1, 1, 1 };
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        glTexCoord1f(-2);
        glVertex3f(0,0,0);
        glTexCoord1f(0);
        glVertex3f(0,1,0);
        glTexCoord1f(4.);
        glVertex3f(1,1,0);
        glEnd();
    }
};
1D, 2D, 3D textures

- 2D textures most common
- 1D textures: color maps (e.g., what we just did)
- 3D textures: “volume rendering”
  - Use combination of opacity and color (i.e., RGBA)
2D Textures

- Pre-rendered images painted onto geometry
- `glTexImage1D` → `glTexImage2D`
- `GL_TEXTURE_WRAP_S` → `GL_TEXTURE_WRAP_S` + `GL_TEXTURE_WRAP_T`
- `glTexCoord1f` → `glTexCoord2f`
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
    public:
        static vtk441PolyDataMapper *New();
    
        virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
            vtkJPEGReader *rdr = vtkJPEGReader::New();
            rdr->SetFileName("HankChilds_345.jpg");
            rdr->Update();
            vtkImageData *img = rdr->GetOutput();
            int dims[3];
            img->GetDimensions(dims);
            unsigned char *buffer = (unsigned char *) img->GetScalarPointer(0,0,0);
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, dims[0], dims[1], 0, GL_RGB,
                           GL_UNSIGNED_BYTE, buffer);
            glEnable(GL_COLOR_MATERIAL);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
            glEnable(GL_TEXTURE_2D);
            float ambient[3] = { 1, 1, 1 };
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
            glBegin(GL_TRIANGLES);
            glVertex3f(0,0,0);
            glVertex3f(0,0,0);
            glVertex3f(1,0);
            glVertex3f(0,1,0);
            glVertex3f(1,0,0);
            glVertex3f(1,1,0);
            glEnd();
    }
};
Minifying: texture bigger than triangle.

- How to map multiple texture elements onto a pixel?
  - GL_NEAREST: pick closest texture
  - GL_LINEAR: average neighboring textures

Magnifying (GL_TEXTURE_MAG_FILTER): triangle bigger than texture

- How to map single texture element onto multiple pixels?
  - GL_NEAREST: no interpolation
  - GL_LINEAR: interpolate with neighboring textures
GL_TEXTURE_MAG_FILTER with NEAREST and LINEAR

class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
  static vtk441PolyDataMapper *New();

  virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
  {
    GLubyte Texture3[9] = {
      0, 0, 255,   // blue
      255, 255, 255, // white
      255, 0, 0,    // red
    };
    glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, 3, 0, GL_RGB, GL_UNSIGNED_BYTE, Texture3);
    glEnable(GL_COLOR_MATERIAL);
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    //-----
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    // OR
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    //-----
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

    glEnable(GL_TEXTURE_1D);
    float ambient[3] = {1, 1, 1};
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
    glBegin(GL_TRIANGLES);
    glTexCoord1f(0.);
    glVertex3f(0,0,0);
    glVertex3f(0,0,0);
    glVertex3f(0,1,0);
    glVertex3f(1,0,0);
    glVertex3f(1,1,0);
    glEnd();
  }
};
Geometry Specification: glBegin

Name

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

C Specification

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

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 \( 2n - 1 \) and \( 2n \) 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.
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.
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.
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)`
glLightfv parameters

- **Ten parameters (ones you will use):**

1. **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).

2. **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).

3. **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).

4. **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.
glLightfv in action

For each light source, we can set an RGBA for the diffuse, specular, and ambient components:

```c
glEnable(GL_LIGHTING);

// Enable the specific light source
glEnable(GL_LIGHT0);

// Define the RGBA values for the diffuse component
GLfloat diffuse0[4] = { 0.7, 0.7, 0.7, 1 };  // Example values

// Set the diffuse component for the light source
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse0);

...  // set ambient, specular, position

// Disable the light source to see the effect
glDisable(GL_LIGHT1);  // do we need to do this?

...

// Disable the last light source
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);
  glVertex3f(1, 0, 0);
  glVertex3f(0, 0.707, +0.707);
  glVertex3f(1, 1, 0);
  glVertex3f(1, 0, 0);
  glVertex3f(1, 0, 0);
What happens with multiple lights?

```c
glEnable(GL_LIGHT0);
glEnable(GL_LIGHT1);
```

- The effects of these lights are additive.
  - Individual shading factors are added and combined
  - Effect is to make objects brighter and brighter
    - Same as handling of high specular factors for 1E
glMaterial: coarser controls for color

- You specify how much light is reflected for the material type.

- Command:
  
  `glMaterialfv(FACE_TYPE, PARAMETER, VALUE(S))`

- `FACE_TYPE =`
  
  - `GL_FRONT_AND_BACK`
  
  - `GL_FRONT`
  
  - `GL_BACK`

(We will talk about this on Friday **NEXT WEEK**)


**glMaterialfv Parameters**

GL_AMBIENT

`params` contains four fixed-point or floating-point values that specify the ambient RGBA reflectance of the material. The values are not clamped. The initial ambient reflectance is (0.2, 0.2, 0.2, 1.0).

GL_DIFFUSE

`params` contains four fixed-point or floating-point values that specify the diffuse RGBA reflectance of the material. The values are not clamped. The initial diffuse reflectance is (0.8, 0.8, 0.8, 1.0).

GL_SPECULAR

`params` contains four fixed-point or floating-point values that specify the specular RGBA reflectance of the material. The values are not clamped. The initial specular reflectance is (0, 0, 0, 1).
glMaterialfv Parameters

GL_EMISSION

`params` contains four fixed-point or floating-point values that specify the RGBA emitted light intensity of the material. The values are not clamped. The initial emission intensity is \((0, 0, 0, 1)\).

GL_SHININESS

`params` is a single fixed-point or floating-point value that specifies the RGBA specular exponent of the material. Only values in the range \([0, 128]\) are accepted. The initial specular exponent is 0.

GL_AMBIENT_AND_DIFFUSE

Equivalent to calling `glMaterial` twice with the same parameter values, once with `GL_AMBIENT` and once with `GL_DIFFUSE`. 
OpenGL: very complex model for lighting and colors

*glMaterial* not used

*glColor* is no-op

---

Is OpenGL Lighting Enabled?

*glEnable(GL_LIGHTING)*;
*glDisable(GL_LIGHTING)*;

---

Are Colour-driven Materials Enabled?

*glEnable(GL_COLOR_MATERIAL)*;
*glDisable(GL_COLOR_MATERIAL)*;

---

No

Final polygon colour is determined by: *glColor*(..);

---

Yes

Final polygon colours for Ambient, Diffuse, Specular and Emission components are determined by: *glMaterial*(..);

---

No

---

Yes

Which colour components are set by *glColor* and which by *glMaterial*?

*glColorMaterial*(..)

---

Final polygon colours for components specified by *glColorMaterial* are set by *glColor*(..); the remaining components are set by *glMaterial*();
The OpenGL light model presumes that the light that reaches your eye from the polygon surface arrives by four different mechanisms:

- AMBIENT
- DIFFUSE
- SPECULAR
- EMISSION - in this case, the light is actually emitted by the polygon - equally in all directions.
Difference between lights and materials

- There are three light colours for each light:
  - Ambient, Diffuse and Specular (set with glLight)
- There are four colors for each surface
  - Same three + Emission(set with glMaterial).
- All OpenGL implementations support at least eight light sources - and the glMaterial can be changed at will for each polygon

http://www.sjbaker.org/steve/omniv/opengl_lighting.html
Interactions between lights and materials

- The final polygon colour is the sum of all four light components, each of which is formed by multiplying the glMaterial colour by the glLight colour (modified by the directionality in the case of Diffuse and Specular).

- Since there is no Emission colour for the glLight, that is added to the final colour without modification.

http://www.sjbaker.org/steve/omniv/opengl_lighting.html
Outline

- Review
- Display lists
- Transforms in GL
- Project 2B
glBindTexture: tell the GPU about the texture once and re-use it!

```cpp
class vtk441PolyDataMapper : public vtkOpenGLPolyDataMapper
{
public:
    static vtk441PolyDataMapper *New();
    GLuint texture;
    bool initialized;

    vtk441PolyDataMapper()
    { initialized = false; }
    void SetUpTexture()
    {
        glGenTextures(1, &texture);
        glBindTexture(GL_TEXTURE_2D, texture);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

        vtkJPEGReader *rdr = vtkJPEGReader::New();
        rdr->SetFileName("HankChilds_345.jpg");
        rdr->Update();
        vtkImageData *img = rdr->GetOutput();
        int dims[3];
        img->GetDimensions(dims);
        unsigned char *buffer = (unsigned char *) img->GetScalarPointer(0,0,0);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, dims[0], dims[1], 0, GL_RGB,
                     GL_UNSIGNED_BYTE, buffer);
        initialized = true;
    }

    virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
    {
        if (!initialized)
            SetUpTexture();
        glEnable(GL_COLOR_MATERIAL);
        glBindTexture(GL_TEXTURE_2D, texture);
        glEnable(GL_TEXTURE_2D);
        float ambient[3] = { 1, 1, 1 };  // Default ambient color
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);
        glBegin(GL_TRIANGLES);
        glTexCoord2f(0,0);
        glVertex3f(0,0,0);
        glVertex3f(1,0);
        glVertex3f(0,1,0);
        glVertex3f(0,0,1);
        glVertex3f(1,1,0);
        glVertex3f(1,1,1);
        glVertex3f(1,0,0);
        glVertex3f(0,1,0);
        glEnd();
    }
};
```
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:
  \[
  \text{GLU}n \text{t} \text{ displayList} = \text{glGenLists}(1);
  \]

- 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();
}

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();
gEndList();

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.
Outline

- Review
- Display lists
- Transforms in GL
- Project 2B
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`
glTranslate

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
glmMatrixMode) 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:

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

glRotated, glRotatelf - multiply the current matrix by a rotation matrix

C SPECIFICATION

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

void glRotatelf(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:

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

Where \( c = \cos \) (angle), \( s = \sin \) (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

```c
void glScaled( GLfloat x,
              GLfloat y,
              GLfloat 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?

```gl
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);
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.
glPushMatrix, glPopMatrix

NAME

glPushMatrix, glPopMatrix - push and pop the current matrix stack

C SPECIFICATION

void glPushMatrix( void )

C SPECIFICATION

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`.
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)
glFrustum - multiply the current matrix by a perspective matrix

C SPECIFICATION

void glFrustum( double left,
                 double right,
                 double bottom,
                 double top,
                 double zNear,
                 double 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:

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

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

void glOrtho( Gdouble left,
             Gdouble right,
             Gdouble bottom,
             Gdouble top,
             Gdouble zNear,
             Gdouble 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:

\[
\begin{bmatrix}
  2 / (right - left) & 0 & 0 & tx \\
  0 & 2 / (top - bottom) & 0 & ty \\
  0 & 0 & -2 / (zFar - zNear) & tz \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

where

\[
\begin{align*}
  tx &= - (right + left) / (right - left) \\
  ty &= - (top + bottom) / (top - bottom) \\
  tz &= - (zFar + zNear) / (zFar - zNear)
\end{align*}
\]

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, vtkCamera *cam)
{
    RemoveVTKOpenGLStateSideEffects();
    SetupLight();

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

    glEnable(GL_TEXTURE_2D);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, p_width, p_height, 0, GL_RGB, GL_UNSIGNED_BYTE, p_image);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glBegin(GL_QUADS);
    glTexCoord2f(0,0);
    glVertex3f(10, -10, -10);
Outline

- Review
- Display lists
- Transforms in GL
- Project 2B
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.
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