OpenGL Notes [a]

Stu Pomerantz
smp@psc.edu
http://www.psc.edu/~smp
September 15, 2004

[a] All material is adapted from: OpenGL ARB, et. al, "The OpenGL Programming Guide", Third Ed., Reading: Addison-Wesley, 1999

The Camera Metaphore

- It is convenient to imagine that OpenGL has a (virtual) camera which is taking a picture of the (virtual) world.

If so, then ...

- Must delimit the part of the world that the camera will see.
  - Can't fit the entire world inside the computer.
  - Can't fit the entire world on the screen.
  - The mathematics forces us to (for near clipping plane).
  - Realistic views don't see everything.
- Must specify *how* the camera sees the world
  - Cameras can have different lenses.

The Camera Metaphore

- Must specify the position and orientation of the camera.
- Must also specify position and orientation of objects in the world that the camera will be "photographing"
- The OpenGL *projection transformation* is concerned with:
  - Delimiting the part of the world that the camera will see.
  - How the camera sees the world.
- The OpenGL *projection transformation* is **not** concerned with:
  - The position and orientation of the camera.
  - The position and orientation of objects in the world.
  - This is the job of the *modelview transformation*

OpenGL Projection Transformations

- Recall: The *projection transformation* defines how a point, or vertex, is transformed from world space to the 2D plane of the screen, or screen space.
  - World space could be an $8\frac{1}{2}$ by 11 sheet of paper in a 2D drawing program or the solar system in a space flight simulator.
  - World space is the two- or three-dimension space which you are modelling in the computer.
- In OpenGL, world coordinate numbers (vertices) are *unitless*. That is, `glVertex2i(2,3)` purposely does not say whether the `2` is in mm, cm, inch, m, km, miles, or light years. The interpretation is up to the application.

OpenGL Projection Transformations

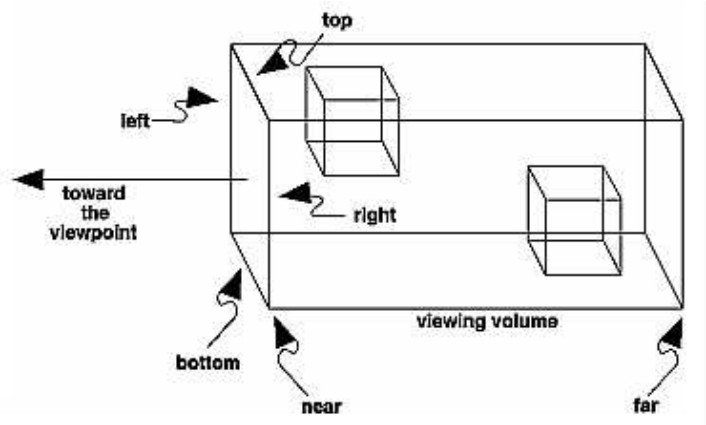
- OpenGL represents the projection transform as a 4x4 matrix.

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{pmatrix}$$

- Many kinds of projection transformations that can be represented in OpenGL.
- There is special support for two commonly used tranformations.
  - Orthographic
  - Perspective

OpenGL Projection Transformations

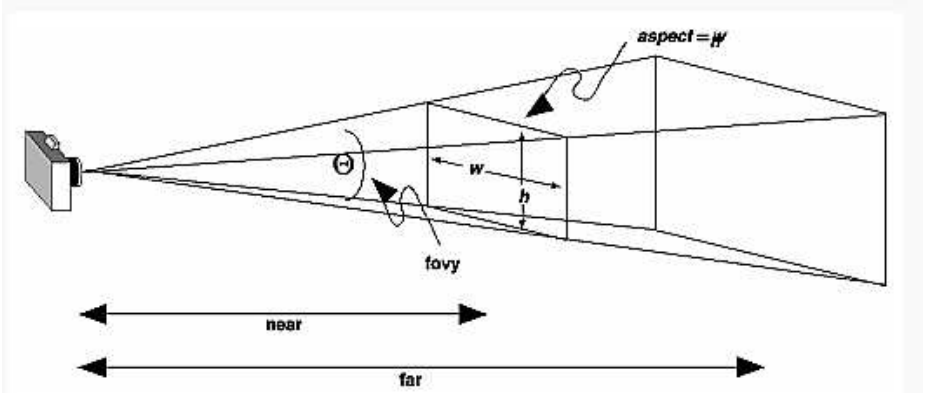
- Projection transformations can be thought of as defining the *shape* of a *viewing volume*.
- An orthographic viewing volume:




```
void glOrtho(GLdouble left,   GLdouble right,
             GLdouble bottom, GLdouble top,
             GLdouble zNear,  GLdouble zFar ) ;
```

OpenGL Projection Transformations

- A perspective viewing volume:

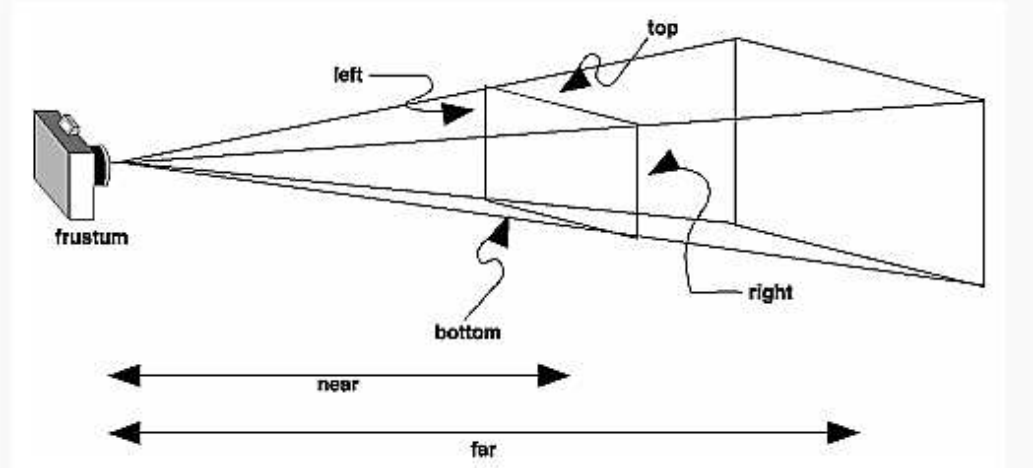



```
void gluPerspective(GLdouble fovy,  GLdouble aspect,
                    GLdouble zNear, GLdouble zFar ) ;
```

- *fovy* is the field of view in degrees.
- *aspect* is the aspect ratio that determines the field of view in the x direction. The ratio of x (width) to y (height).

OpenGL Projection Transformations

- The general viewing volume call:




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

OpenGL Projection Transformations

- When `glOrtho()` `gluPerspective()` or `glFrustum` is executed, two things occur:
  - A matrix which produces a projection transformation for the requested viewing volume is generated.
  - That matrix is *multiplied* into the current OpenGL state.
    - This is an important point which will be detailed in a few slides.

The OpenGL Modelview Transformation

Recall, the modelview transformation is concerned with:

- The position and orientation of the camera.
- The position and orientation of objects in the world.

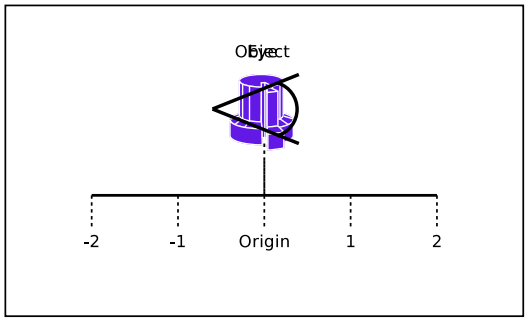
Like the projection transformation, the modelview tranformation is a 4x4 matrix.

But Wait ...

- Why not use two separate transformation matrices, one for the camera (a view transformation) and one for the objects in the world (a model transformation) instead of one composite modelview transformation ?

The OpenGL Modelview Transformation

Try this thought experiment:



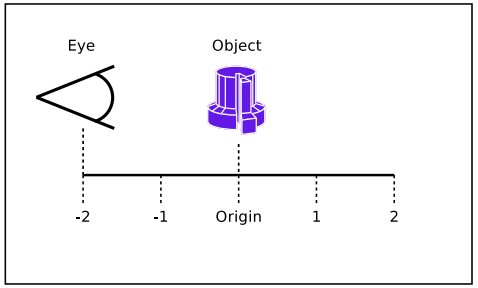
Suppose the camera, or eye, and the object the camera is to view are in the same location, the origin.

In order to view the object either the camera or the object must be moved.

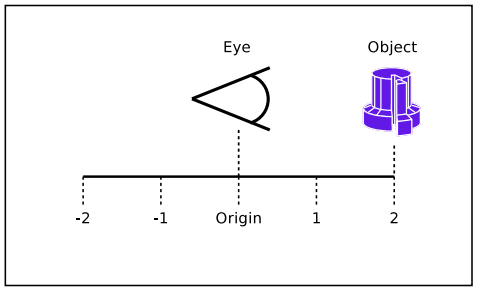
The OpenGL Modelview Transformation

Assuming a constant color background, is there a difference in what the camera sees:

Moving the camera away from the origin:



Moving the object away from the origin:



The OpenGL Modelview Transformation

No Difference!

- The programmer can think of the modelview matrix as transforming the view *or* the models that are being viewed.
- Hence the name *modelview*.
- OpenGL's default state is modelview mode. That is – OpenGL matrix multiplications affect the modelview transformation.

The OpenGL Modelview Transformation

Pointing the camera is pretty simple:

```
void gluLookAt(GLdouble eyeX,  // position of the eye
               GLdouble eyeY,
               GLdouble eyeZ,

               GLdouble centerX, // where the eye
               GLdouble centerY, // is looking
               GLdouble centerZ,

               GLdouble upX, // which way
               GLdouble upY, // is up
               GLdouble upZ ) ;
```

The OpenGL Modelview Transformation

For example:

```
void gluLookAt( 0,0,-5, 0,0,0, 0,1,0 ) ;
```

Puts the camera at -5 on the z axis, looking at the origin, and chooses up to be conincident with the positive y axis. This is a standard view.

Another example:

```
void gluLookAt( 10,0,0, 0,0,0, 0,0,1 ) ;
```

Puts the camera at 10 on the x axis, looking at the origin, and chooses up to be conincident with the positive z axis.

Specifying the View

In order to specify a view to OpenGL:

- Set the viewport.
- Set the projection transformation
- Set the modelview tranformation

In order to do these tasks, there must also be a way to:

- Set the transformation state for the projection mode and the modelview mode back to a default 'no transformation' state. (since doing a transformation call multiplies the current matrix by the new matrix rather than replacing it).
- Change between the projection and modelview states.

Specifying the View

Usually, when the window is resized (e.g. glut `glutReshapeFunc`), these 3 tasks are performed:

- Set the viewport.
- Set some default projection transformation
- Set some default modelview tranformation

Specifying the View

To tell OpenGL to change its state so that OpenGL matrix multiplies affect the projection transformation call:

```
glMatrixMode(GL_PROJECTION) ;
```

To tell OpenGL to change its state so that OpenGL matrix multiplies affect the modelview transformation (what is usually desired) call:

```
glMatrixMode(GL_MODELVIEW) ;
```

Specifying the View

No matter which matrix mode is in use, there must be a way of getting back to a default, no transformation, state.

To return to the default state call:

```
glLoadIdentity() ;
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Which sets the transformation to the identity matrix.

Specifying the View

- Note that `glLoadIdentity() ;` *does not* multiply the current matrix by the identity matrix.
- Rather it sets the current matrix (in either projection matrix mode or modelview matrix mode) *to* the identity, which is the default 'no transformation' state.
- If the identity matrix I is multiplied by transformation a matrix M , then $I \times M = M$
- So, setting the transformation to the identity and then multipling in a new transformation is equivilant to simply setting that new transformation.

Specifying the View

The default projection is:

```
glMatrixMode(GL_PROJECTION) ;
glLoadIdentity() ;
```

which is *equivilant* to

```
glMatrixMode(GL_PROJECTION) ;
glOrtho(-1,1, -1,1, 1,1) ;
```

A common example:

```
glMatrixMode(GL_PROJECTION) ;
glLoadIdentity() ;  // set projection matrix to identity
gluPerspective(45,1,5,100) ;  // multiply in a
                              // perspective transform
```

Specifying the View

The default modelview is:

```
glMatrixMode(GL_MODELVIEW) ;
glLoadIdentity() ;
```

which is *equivilant* to

```
glMatrixMode(GL_MODELVIEW) ;
gluLookAt(0,0,0, 0,0,-1, 0,1,0) ;
```

A common example:

```
glMatrixMode(GL_MODELVIEW) ;
glLoadIdentity() ;  // set modelview matrix to identity
gluLookAt(0,0,10, 0,0,-1, 0,1,0) ;  // multiply in a
                                    // camera transform
```

Specifying the View

All together now in a reshape callback example:

```
void reshape(int x, int y) {
  glViewPort(x,y) ;

  glMatrixMode(GL_PROJECTION) ;
  glLoadIdentity() ;
  gluPerspective(45,1,5,100) ;

  glMatrixMode(GL_MODELVIEW) ;
  glLoadIdentity() ;
  gluLookAt(0,0,10, 0,0,-1, 0,1,0) ;
}
```

Specifying the View

Beware! Although you know *how* to do it, you don't yet know the mathematics behind *why* it works.

Modelview Transformations

- OpenGL transformations are all specified in three dimensional space.
- Examples here will be in two dimensions (for now).
- Only the *translation* and *rotation* transformations will be discussed (for now).
- Transformation calls in OpenGL multiply a matrix which represents the transformation into the current OpenGL state.
- This approach could easily get out of hand if there were no way to save and restore the modelview matrix state. Fortunately, there is.

Modelview Transformations

- Saving the Modelview state:

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

- Restoring the Modelview state:

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

Commonly used to make temporary changes to the modelview transformation so a particular object can be further transformed without affecting the overall modelview state.

Modelview Transformations

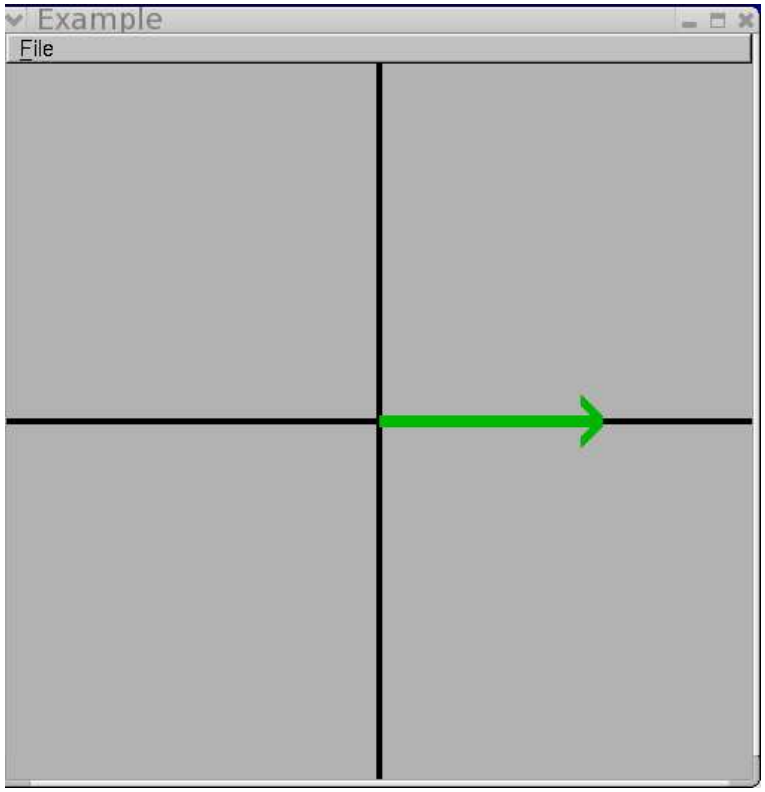
- Recall the arrow example:

```
// arrow vertices
float pt[6][2] ={ {0,0},{1,0},
                  {1,0},{0.9,0.1},
                  {1,0},{0.9,-0.1} } ;


// drawing a green arrow
glColor3f(0,.6,0) ;
glBegin(GL_LINES) ;
for(i = 0 ; i < 6 ; i++ )
  glVertex3f(pt[i][0],pt[i][1],1) ;
glEnd() ;
```

Modelview Transformations

Draws this figure:

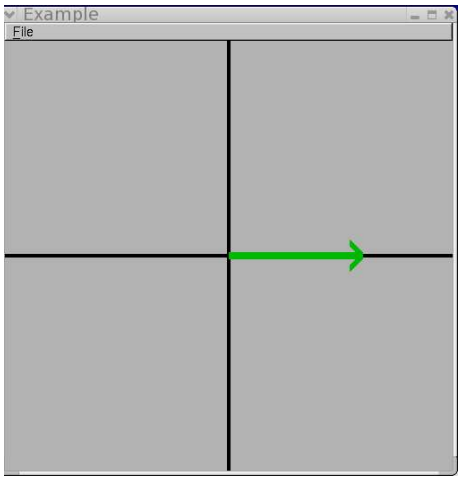


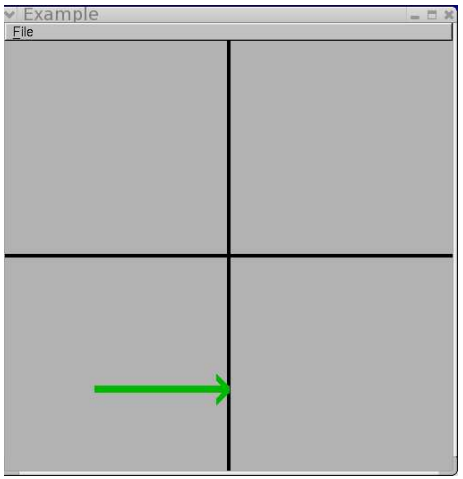
Modelview Transformations

- Translation:

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

Example (using gluLookAt(0,0,7, 0,0,0, 0,1,0)):





No translation vs. `glTranslatef(-1,-1,0)`

Modelview Transformations

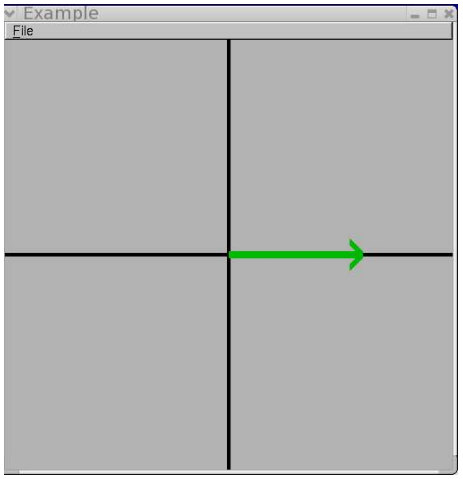
- Translation is always in three dimensional space. It is easy to hold one dimension constant and do a two dimensional translation as the previous example did.
- `glTranslatef()` calculates a translation matrix which has the desired affect.
- This matrix is multiplied into the current matrix transformation mode (usually the modelview mode).

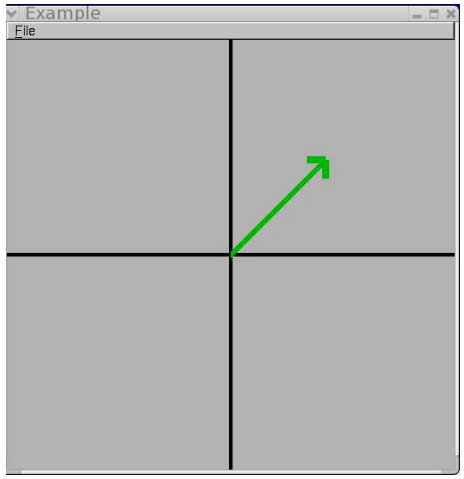
Modelview Transformations

- Rotation:

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

Example (using gluLookAt(0,0,7, 0,0,0, 0,1,0)):





No rotation vs. `glRotatef(45,0,0,1)` ; Note that the camera is facing the x-y plane, so rotation in the x-y plane is about the z axis.

Modelview Transformations

- Rotation:

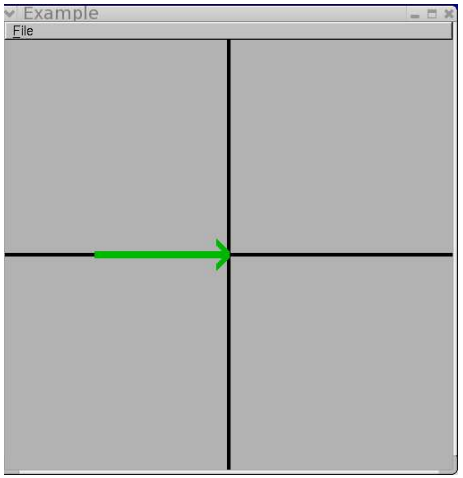
  `glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)`

- *angle* specifies the angle of rotation, in degrees.
- *x, y, z* specify the x, y, and z coordinates of the vector, about which you are rotation.

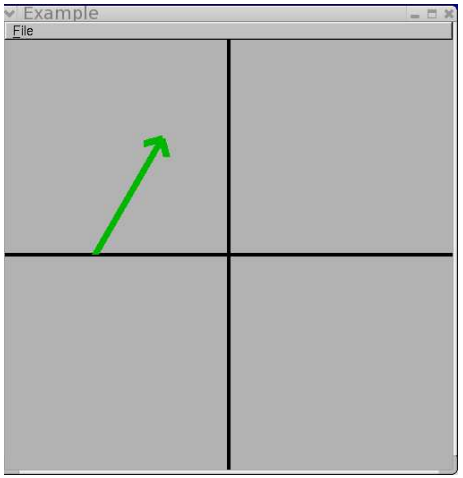
Modelview Transformations

- Translation followed by Rotation:

Example (using gluLookAt(0,0,7, 0,0,0, 0,1,0)):



glTranslatef(-1,0,0) ;



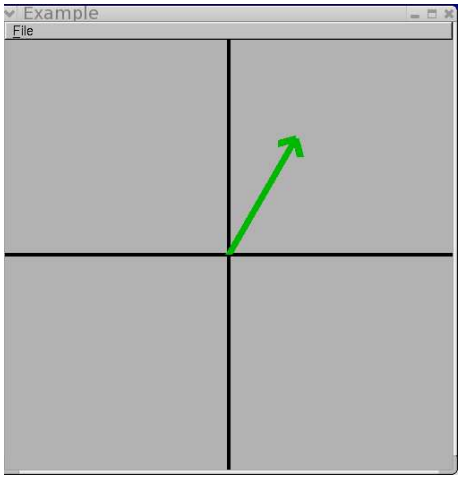
glTranslatef(-1,0,0) ;
glRotatef(60,0,0,1) ;

Translation vs Translation followed by Rotation

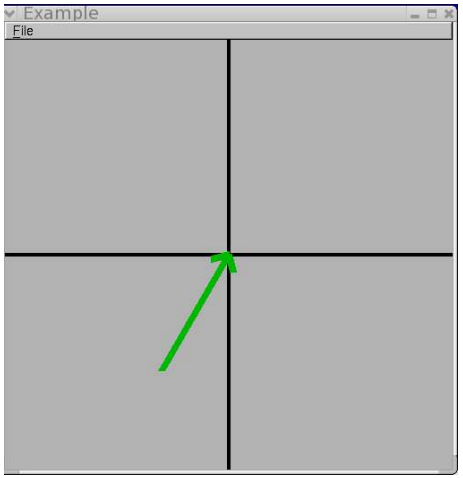
Modelview Transformations

- Rotation followed by Translation:

Example (using gluLookAt(0,0,7, 0,0,0, 0,1,0)):



glRotatef(60,0,0,1) ;



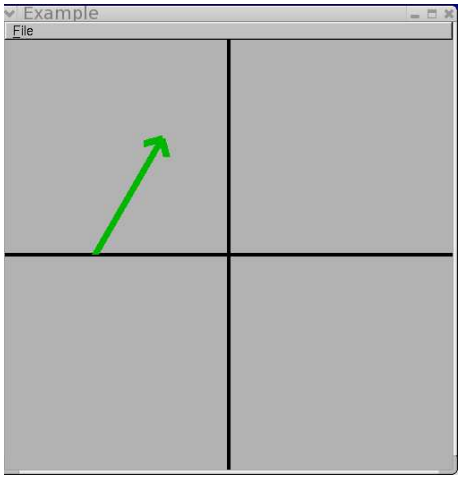
glRotatef(60,0,0,1) ;
glTranslatef(-1,0,0) ;

Rotation vs Rotation followed by Translation

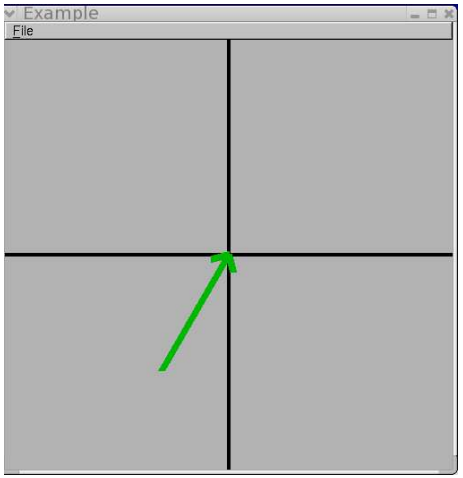
Modelview Transformations

- Translation and Rotation are *order dependant*!

Example (using gluLookAt(0,0,7, 0,0,0, 0,1,0)):



glTranslatef(-1,0,0) ;
glRotatef(60,0,0,1) ;



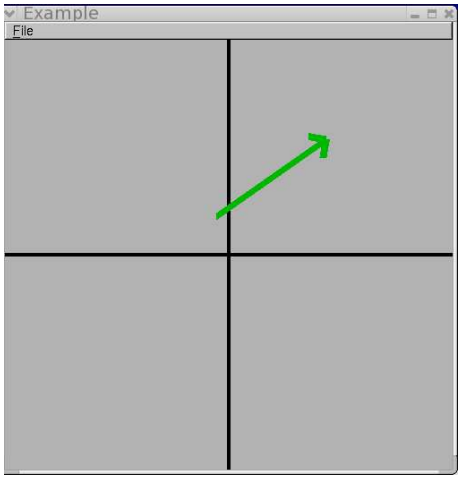
glRotatef(60,0,0,1) ;
glTranslatef(-1,0,0) ;

Translation then Rotation vs Rotation then Translation

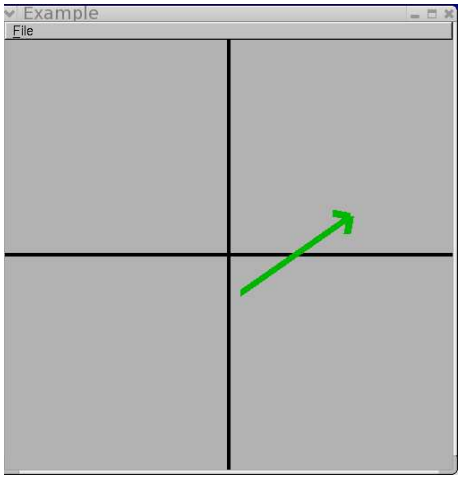
Modelview Transformations

- Rotation about an arbitrary point (center of the arrow)

Example (using gluLookAt(0,0,7, 0,0,0, 0,1,0)):



glTranslatef(-0.5,0,0) ;
glRotatef(35,0,0,1) ;
glTranslatef(0.5,0,0) ;



glTranslatef(0.5,0,0) ;
glRotatef(35,0,0,1) ;
glTranslatef(-0.5,0,0) ;

Modelview Transformations

Beware! Although you know *how* to do it, you don't yet know the mathematics behind *why* it works.