

## glman, a Better View into GLSL

*glman* is a program to allow a better view into the OpenGL Shading Language (GLSL). It does not take the place of an OpenGL program for real shader applications, but it is a nice OpenGL program substitute for running experiments as you learn about shaders.

In the Oregon State University Computer Graphics Education Lab, *glman* is started with:

**Start → All Programs → Shaders → glman.exe**

The starting user interface window looks like this.

At the same time, a console window pops up. There are, at times, useful messages that are displayed there. Most of the time, you can ignore it. Iconify it if you want.

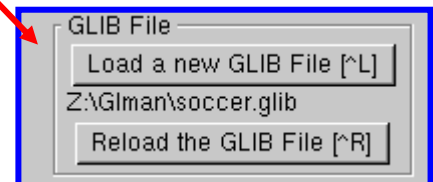
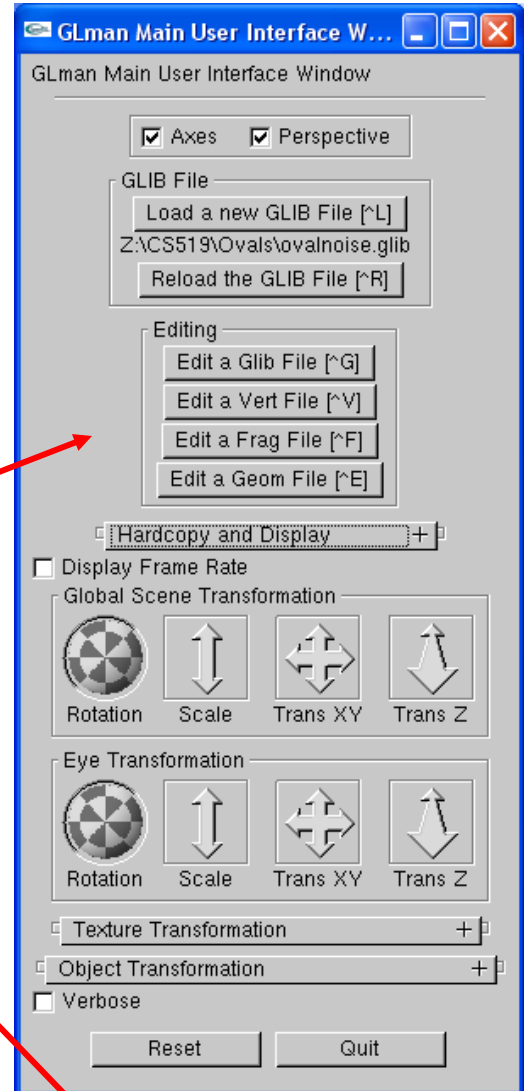
### Loading a GLIB File

*GLIB* stands for *GL Interface Bytestream*. It is an ASCII-encoded input file inspired by the Photorealistic RenderMan RIB file. The GLIB portion of the *glman* menu looks like this.

**Load a new GLIB File** This brings up a dialog box that allows you to select a GLIB file. *glman* parses your GLIB file and creates a scene in a new graphics window.

**Reload the GLIB File** Reloads the currently-loaded GLIB file.

The following commands are allowed in a GLIB file. The command itself is *case-insensitive*, but any text arguments are *case-sensitive*. Numbers in square brackets [ ] show the default values if the parameters are not set. If no default is given, then this command does not do anything without parameters.



***If you read nothing else in this document, read this. Some shaders expect to find the default OpenGL viewing situation, that is, the eye at the origin looking in -Z. So, glman provides that situation when your scene is first displayed. Unfortunately, this often leaves you seeing little or nothing of your scene because you're inside it. Thus, usually your first move upon opening up a new .glib scene should be to use the Eye Transformation "Trans Z" widget to push the scene back into the viewing volume where it is more visible.***

## GLIB Scene Creation

### Window and Viewing Volume

|                   |   |
|-------------------|---|
| WindowSize wx wy  | Specifies the initial graphics window size. [600. 600.]                                       |
| Ortho xl xr yb yt | Set the current projection to orthographic with the given parameters. [-3. 3. -3. 3.]         |
| Perspective fov   | Set the current projection to perspective with the given field of view angle (degrees). [70.] |

### Transformations

**Like OpenGL itself, these transformations take effect in the reverse order in which they are listed.**

|                       |   |
|-----------------------|---|
| Translate tx ty tz    | Pre-concatenate a translation onto the current matrix                 |
| Rotate angle ax ay az | Pre-concatenate a rotation onto the current matrix (angle in degrees) |
| Scale sx sy sz        | Pre-concatenate a scale onto the current matrix                       |
| PushMatrix            | Push the current matrix on the matrix stack                           |
| TransformBegin        | Same as PushMatrix  |
| PopMatrix             | Pop the current matrix from the matrix stack                          |
| TransformEnd          | Same as PopMatrix   |

### Creating Geometry

|                                    |   |
|------------------------------------|---|
| Box dx dy dz                       | Creates a 3D box. If specified, (dx,dy,dz) are the half-lengths of the sides. [2. 2. 2.]  |
| Cylinder radius height             | Draw a solid cylinder. [1. 1.]  |
| Cone radius height                 | Draw a solid cone. [1. 1.]  |
| DiskXY                             | Creates a unit disk parallel to the XY plane and passing through Z=0.   |
| JitterCloud numx numy numz         | Creates a jittered 3D point cloud. If specified, num* are the number of points to use in each direction.                              |
| LinesAdjacency [v0] [v1] [v2] [v3] | Creates an instance of the new OpenGL GL_LINES_ADJACENCY primitive. This works with geometry shaders and only makes sense if they are |

available on your system. Each vertex consists of an x, y, and z, given in square brackets. So, for instance, [v0] might be: [1. 2. 3.]

|                           |   |
|---------------------------|---|
| Obj filename              | Reads in a .obj geometry filename into lists of GL_TRIANGLES and GL_QUADS. If <i>filename</i> is not given, <i>glman</i> will prompt you for it. The true .obj format can be quite complex. This feature, however, just supports vertices, normals, texture coordinates, and faces. If you don't have a real need, use Obj instead of ObjAdj. The file will read faster and the resulting geometry will display faster. |
| ObjAdj filename           | Reads in a .obj geometry filename into a list of GL_TRIANGLES_ADJACENT. This is useful for use with Geometry Shaders for applications such as smoothing and silhouettes. If <i>filename</i> is not given, <i>glman</i> will prompt you for it. The true .obj format can be quite complex. This feature, however, just supports vertices, normals, texture coordinates, and faces.                                       |
| PointCloud numx numy numz | Creates a regular 3D point cloud. If specified, num* are the number of points to use in each direction.   |
| QuadBox numquads          | Draws a series of 'numquads' quadrilaterals (default=10) parallel to the XY plane. The XYZ coordinates run from (-1.,-1.,-1.) to (1.,1.,1.). The 3D texture coordinates run from (0.,0.,0.) to (1.,1.,1.). This is a good way to test 3D textures.  |
| QuadXY z size nx ny       | Creates a quadrilateral parallel to the XY plane, passing through Z=z. If given, size is the quadrilateral's dimension, going from (-size,-size) to (size,size) in X and Y. If given, nx and ny are the number of sub-quads this quadrilateral is broken into. This is a good way to test 2D textures.  |
| QuadXZ y size nx nz       | Creates a quadrilateral parallel to the XZ plane, passing through Y=y. If given, size is the quadrilateral's dimension, going from (-size,-size) to (size,size) in X and Z. If given, nx and nz are the number of sub-quads this quadrilateral is broken into.  |
| QuadYZ x size ny nz       | Creates a quadrilateral parallel to the YZ plane, passing through X=x. If given, size is the quadrilateral's dimension, going from (-size,-size) to (size,size) in Y and Z. If given, ny and nz are the number of sub-quads this quadrilateral is broken into.  |

|                                   |   |
|-----------------------------------|---|
| Soccerball radius                 | Creates a geometric soccer ball from 12 pentagons and 20 hexagons. As part of this, two uniform variables are defined: <ol style="list-style-type: none"> <li>1. <b>FaceIndex</b>: which face are we on right now. 0-11 are the pentagons, 12-31 are the hexagons.</li> <li>2. <b>Tangent</b>: vec3 pointing in a consistent tangent direction, same as the Sphere uses.</li> </ol> <p>In addition, the S and T texture coordinates are filled with good values for mapping an image to each face. The P value is filled with a normalized radius from the center. The seam is located at P=1. [1.]</p> |
| Sphere radius                     | Draw a solid sphere. This primitive sets the vertex coordinates, the vertex normals, and the vertex texture coordinates. In order to align bump-mapping, it also sets a vec3 called <b>Tangent</b> at each vertex. The <b>Tangent</b> vectors are all tangent to the sphere surface and always point in a consistent direction, towards the North Pole. [1.]  |
| Teapot                            | Create a solid teapot.  |
| Torus innerradius outerradius     | Create a solid torus. [.2 1.]   |
| WireBox dx dy dz                  | Creates a 3D wireframe box. If specified, (dx,dy,dz) are the half-lengths of the sides. [2. 2. 2.]  |
| WireCylinder radius height        | Create a wireframe cylinder. [1. 1.]  |
| WireCone radius height            | Create a wireframe cone. [1. 1.]  |
| WireObj filename                  | Reads in a .obj geometry filename into lists of lines. If <i>filename</i> is not given, <i>glman</i> will prompt you for it. The true .obj format can be quite complex. This feature, however, just supports vertices, normals, texture coordinates, and faces.   |
| WireSphere radius                 | Create a wireframe sphere. [1.]   |
| WireTorus innerradius outerradius | Create a wireframe torus. [.2 1.]   |
| WireTeapot                        | Create a wireframe teapot   |
| Xarrow numsllices                 | Creates an arrow along the X-axis, from X=0. to X=1. If specified, numsllices are the number of individual slices to use along the arrow. [100]   |

## Specifying Textures

|  |  |
|--|--|
| Texture2D texture_unit filename  | This reads a 2D texture from a file. If the filename ends in a .bmp suffix, a BMP image file is assumed, with red, green, and blue read from the file (no alpha). Any other filename pattern implies a raw format, which consists of two binary 4-byte integers giving the X and Y dimensions of the image, and then 4 components per texel specifying the red, green, blue, and alpha of that texel. The components can be all unsigned bytes or all 32-bit floating point. |
| Texture3D texture_unit filename  | This reads a 3D texture from a file in a raw format, which consists of three binary 4-byte integers giving the X, Y, and Z dimensions of the volume, and then 4 components per texel specifying the red, green, blue, and alpha of that texel. The components can be all unsigned bytes or all 32-bit floating point.  |
| CubeMap texture_unit posxfile negxfile posyfile negyfile poszfile negzfile | Generate a cubemap texture on texture unit texture_unit.   |

## Specifying Shaders

|  |  |
|--|--|
| vertex file.vert                         | Specify a vertex shader filename   |
| geometry file.geom                       | Specify a geometry shader filename   |
| fragment file.frag                       | Specify a fragment shader filename   |
| program programname uniformvariables ... | Compile and Link the vertex, fragment, and possibly geometry (see below), shaders into a program and specify the uniform variables for that program (see below) The program command must come last in this group of three or four. It links together the current vertex shader, the current fragment shader, and possibly the current geometry shader. Note that this means that you can re-use a shader in another shader program by simply not redefining another shader of that type.<br><br>If you want to un-specify a shader (that is, no longer use it), just give the vertex, fragment, or geometry command with no arguments. |

**If you are on a system that is enabled for Geometry Shaders, then you must also use the commands:**

|                        |  |
|------------------------|--|
| geometryinputtype type | Specify what type of topology this geometry shader expects to find as input. This can be: GL_POINTS, |
|------------------------|--|

GL\_LINES, GL\_LINES\_ADJACENCY,  
GL\_TRIANGLES, or GL\_TRIANGLES\_ADJACENCY

geometryoutputtype type

Specify what type of topology this geometry shader will be emitting. This can be: GL\_POINTS, GL\_LINE\_STRIP, or GL\_TRIANGLES\_STRIP.

**Like the vertex, geometry, and fragment commands, these must come *before* the program command.**

## Miscellaneous

Background r g b

Sets the current background color

Color r g b

Set the current rendering color to (r,g,b)  
 $0. \leq r,g,b \leq 1.$

Flat

Sets the current shading mode to GL\_FLAT.

LineWidth size

Sets the current line width.

LookAt ex ey ez lx ly lz ux uy uz

Calls the OpenGL `gluLookAt( )` routine. (ex,ey,ez) are the eye position. (lx,ly,lz) are the look-at position. (ux,uy,uz) are the up-vector.

Noise2D res

Specify the resolution of glman's built-in 2D noise texture (see below)

Noise3D res

Specify the resolution of glman's built-in 3D noise texture (see below).

PointSize size

Sets the current point size.

Smooth

Sets the current shading mode to GL\_SMOOTH.

Timer numsecs

Sets the timer period from the default of 10 seconds per cycle to numsecs per cycle

MessageBox An informative text message

Puts up a Message Box with the text message in it. This is useful if you want to tell the user something about what this .glib file is about and what to do with this scene display.

In GLIB files:

- Multiple whitespace characters in a row are treated as a single whitespace character.
- A # causes the rest of the line to be treated as a comment and ignored

- A / causes the rest of the line to be treated as a comment and ignored (so that // will act as expected)
- A backslash (\) at the end of a line causes the carriage return to be ignored. The current line is continued onto the next line.

## Using Textures

As indicated above, there are two ways of inputting a 2D texture: as a 24-bit uncompressed BMP file or as a raw texture file. If you want this to work on any graphics system, be sure the image dimensions are powers of two. The NVIDIA cards in the OSU Computer Graphics Education Lab quietly don't require this to be true, but for now, many systems do.

The 2D texture raw format is very simple. The first 8 bytes are two 4-byte integers, declaring the S and T image dimensions. The following bytes are the RGBA values for each texel. These RGBA values can be unsigned bytes or floats. *glman* will look at the size of the file and do the right thing.

When writing a raw 2D texture file, the order of writing should be:

```
fwrite( &nums, 4, 1, fp );
fwrite( &numt, 4, 1, fp );

for( int t = 0; t < numt; t++ )
{
    for( int s = 0; s < nums; s++ )
    {
        fwrite( &red, 4, 1, fp );
        fwrite( &green, 4, 1, fp );
        fwrite( &blue, 4, 1, fp );
        fwrite( &alpha, 4, 1, fp );
    }
}
```

The 3D texture raw format is just as simple. The first 12 bytes are three 4-byte integers, declaring the S, T, and P volume dimensions. The following bytes are the RGBA values for each texel. These RGBA values can be unsigned bytes or floats. *glman* will look at the size of the file and do the right thing.

When writing a raw 3D texture file, the order of writing should be:

```

fwrite( &nums, 4, 1, fp );
fwrite( &numt, 4, 1, fp );
fwrite( &nump, 4, 1, fp );

for( int p = 0; p < nump; p++ )
{
    for( int t = 0; t < numt; t++ )
    {
        for( int s = 0; s < nums; s++ )
        {
            fwrite( &red, 4, 1, fp );
            fwrite( &green, 4, 1, fp );
            fwrite( &blue, 4, 1, fp );
            fwrite( &alpha, 4, 1, fp );
        }
    }
}

```

*glman* expects raw texture binary byte-ordering to be consistent with an Intel x86 architecture. If you write raw texture files from a UNIX box (e.g., Sun or SGI), you must reverse the byte ordering yourself.

The second argument on the Texture2D and Texture3D lines is the OpenGL texture unit to assign this texture to. You then need to tell your shaders what that texture number is. For example, the GLIB line might be:

```
Program Texture TexUnit 7
```

And your fragment shader might look like:

```

uniform sampler2D TexUnit;

void
main( void )
{
    vec4 rgb = texture2D( TexUnit, gl_TexCoord[0] ).rgb;
    gl_FragColor = vec4( rgb, 1. );
}

```

**BTW, don't use texture units 2 and 3 yourself. *glman* uses these already to tell your shaders about its built-in 2D and 3D noise textures.**

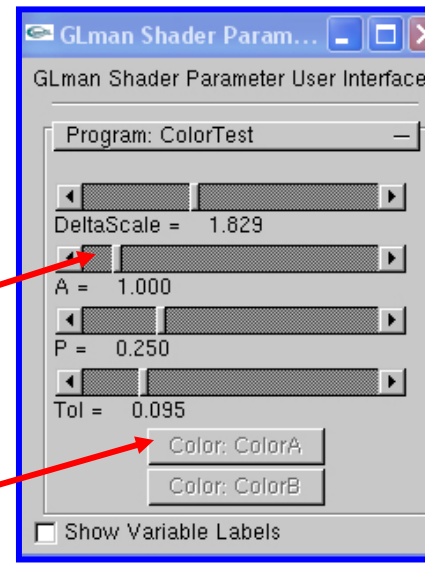
Interestingly enough, a limitation of GLSL is that you cannot hardcode the texture unit number in the call to `texture2D( )`. For example, you *cannot* say:

```
vec4 rgb = texture2D( 7, gl_TexCoord[0] ).rgb;
```

## Specifying Uniform Variables

Uniform variables are specified on the Program command line in a tag-value pair format:

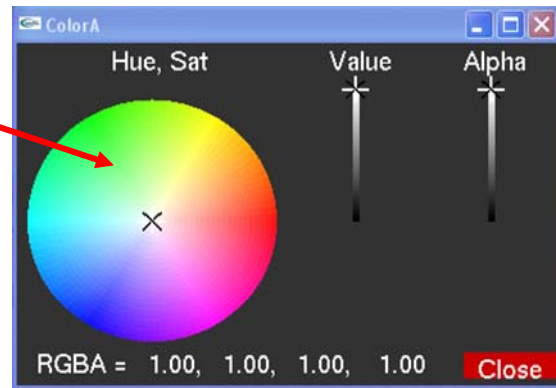
- Scalar variables are just listed as numbers.
- Array variables are enclosed in square brackets. [ ]
- Range variables are enclosed in angle brackets. < >  
These are scalar variables, but will generate a slider in the Uniform Variable user interface so that you can then change this value on the fly. The 3 values in the brackets are : <min current max>. This can be used for variables that are type *int* or *float*. *glman* will look at your symbol table and do the right thing.



- Color variables are enclosed in curly brackets. { }  
They are:

{red green blue alpha}

and will generate a button in the UI that, when clicked, brings up a color selector window. The color selector allows you to change this color variable on-the-fly.



- The < > angle brackets can also be used for Boolean variables. Enclose a single number in the angle brackets – 0 for false and non-zero for true. (The words <true> and <false> can also be used in the angle brackets for readability. This will generate a checkbox in the user interface. This must be used only for variables that are type *bool*. *glman* will look at your symbol table to confirm this.
- Multiple vertex-geometry-fragment-program combinations are allowed in the same GLIB file. If there is more than one combination, then they will appear as separate rollout panels in the user interface. The first program rollout will be open, and all the others will be closed. Open the ones you need when you need them.

For example, the user interface window shown above was created as a result of these lines in the GLIB file:

```
Vertex   colortest.vert
Fragment colortest.frag
Program  ColorTest  DeltaScale <0. 0. 5.>  A <0 1. 10>  P <0. .25 1.>  \
          Tol <0. 0. .5>  ColorA {1. 1. 1.}  ColorB {1. .5 0}
```

## Noise

*glman* automatically creates a 3D noise texture and places it into Texture Unit 3. Your vertex or fragment shader can get at it through the pre-created uniform variable called `Noise3`. Reference it in your shader as:

```
uniform sampler3D Noise3;
    . . .
vec3 stp = ...
vec4 nv = texture3D( Noise3, stp );
```

The noise texture is a `vec4`. The [0] component is the low frequency noise. The [1] component is twice the frequency and half the amplitude of the [0] component, and so on for the [2] and [3] components. Each component is centered around a value of .5, so that if you want a plus-or-minus effect, subtract .5 from each component. To get a nice 4-octave noise value between 0. and 1. (useful for noisy mixing), add up all four components, subtract 1., and divide the whole thing by 2.

| Component | Term                 | Term Range                           |
|-----------|----------------------|--------------------------------------|
| 0         | <code>nv[0]</code>   | $0.5 \pm ,5000$                      |
| 1         | <code>nv[1]</code>   | $0.5 \pm ,2500$                      |
| 2         | <code>nv[2]</code>   | $0.5 \pm ,1250$                      |
| 3         | <code>nv[3]</code>   | $0.5 \pm ,0675$                      |
|           | <b>sum</b>           | <b><math>2.0 \pm \sim 1.0</math></b> |
|           | <b>sum - 1</b>       | <b><math>1.0 \pm \sim 1.0</math></b> |
|           | <b>(sum - 1) / 2</b> | <b><math>0.5 \pm \sim 0.5</math></b> |

```
float sum = nv[0] + nv[1] + nv[2] + nv[3]; // range is 1. -> 3.
sum = ( sum - 1. ) / 2.; // range is now 0. -> 1.
```

If you would like to have a 4-octave noise function that ranges from -1. to 1., then do this instead:

```
float sum = nv[0] + nv[1] + nv[2] + nv[3]; // range is 1. -> 3.
sum = sum - 2.; // range is now -1. -> 1.
```

Accessing the 2D noise texture works the same as the 3D one, except you use:

```
uniform sampler2D Noise2;
    . . .
vec2 st = ...
vec4 nv = texture2D( Noise2, st );
```

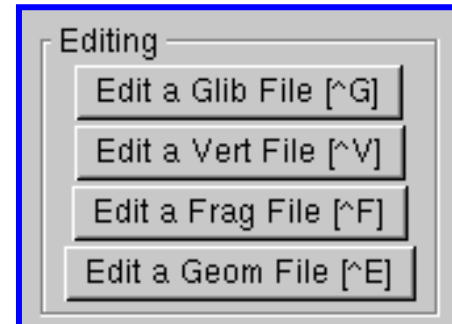
By default, the noise textures are 64x64 and 64x64x64. You can change this by putting a command in your GLIB file of the form:

```
Noise2D 128
or
Noise3D 128
```

or whatever resolution you want (up to around 400). The first time *glman* creates a 2D or 3D noise texture for you, it will take a few seconds. But, it then writes it to a file. The next time *glman* needs this noise texture, it will read it directly from the file, which is a lot faster.

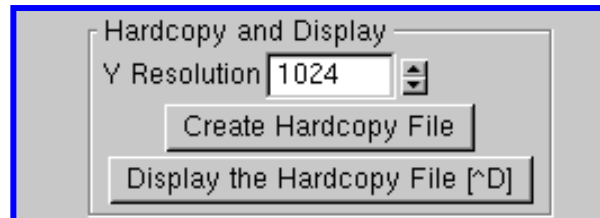
## Editing Files

The .glib, .vert, .frag, and .geom files can be edited any way you want. If you want to easily open a WordPad editing window on a file, click on one of these buttons and then select the file. You can have as many of these WordPad windows open at one time as you want. If your system does not support Geometry Shaders, then the “Edit a Geom File” button will not appear.



## Generating and Displaying an Image Hardcopy of the Screen

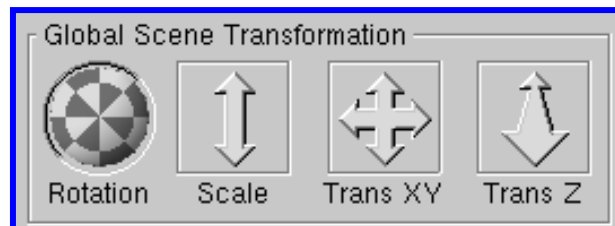
**Create Hardcopy File** As you will be doing cool things with *glman*, you will often want to copy the graphics window to an image file. The Create Hardcopy button will do that, and will let you specify the name of the .bmp file it will be written into. Because *glman* does this by rendering to an internal off-screen framebuffer, the resolution can be (almost) arbitrary. This is useful if you need more pixels in your hardcopy image than you have on your physical screen, as you might need for a high-quality print image or for a large poster.



**Display the Hardcopy File** To confirm what you got, and to perhaps send the image to a printer, click here. This will be grayed out until you actually create a Hardcopy.

## Global Scene Transformation

These widgets allow you to transform the entire scene in the graphics window. The scene can also be rotated by holding down the left mouse button with the cursor in the graphics window. The scene can also be scaled by holding down the middle mouse button in the graphics window.

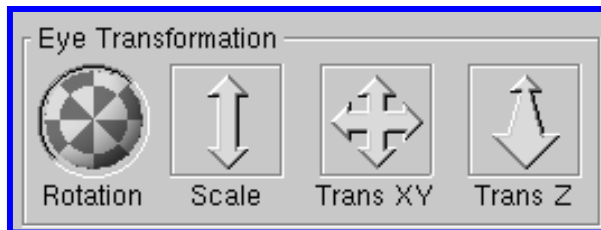


It is important to realize that, unlike what is normally done in an OpenGL program, these transformations *do not end up in the ModelView matrix*. In *glman*, they end up in the *Projection matrix* so that they have no impact on anything your shaders do in Eye Coordinates. That is, these scene transformations can be used to see the back of a scene, without changing the Eye Coordinate behavior of the shaders.

## Eye Transformation

These widgets also allow you to transform the entire scene in the graphics window.

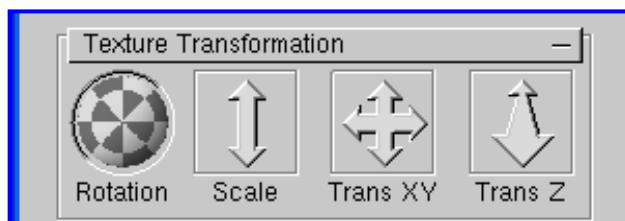
It is important to realize that these transformations do end up in the ModelView matrix, just as if the OpenGL `gluLookAt()` routine had been called. That is, these scene transformations will change the Eye Coordinate behavior of the shaders.



**Usually your first move upon opening up a new .glib scene is to use the "Trans Z" widget to push the scene back into the viewing volume where it is more visible.**

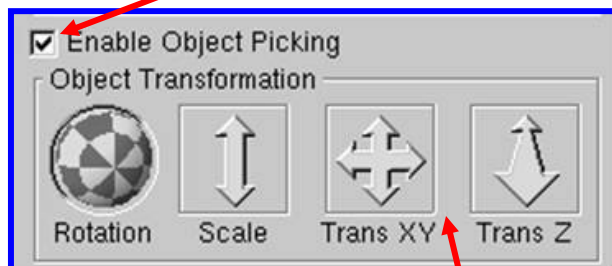
## Texture Transformation

These widgets also allow you to put a transform into the texture matrix. Note that, once you are no longer using the fixed-function pipeline, the texture matrix does not automatically take effect. You must do this yourself in your shader.



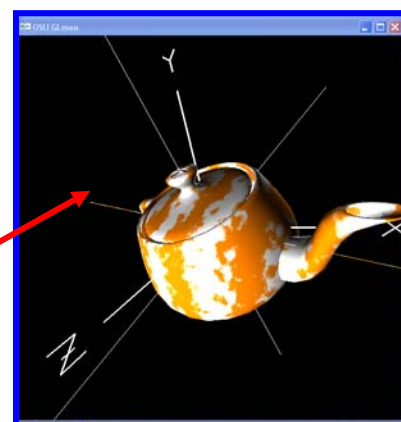
## Object Picking and Transformation

Individual objects in the scene can be picked and independently transformed. This is a good way to test shaders that operate in Eye Coordinates rather than Model Coordinates. To do this, you must first enable object picking by turning on this checkbox.



that it is selected. The 3D cursor will be drawn using the same shaders as the selected object is, so its appearance might look a little weird.

Then, clicking on a 3D object in the scene with the left mouse button will cause that object to be selected. A large 3D cursor becomes centered on the object to show



At that time, the Object Transformation widgets will become active. These widgets allow you to transform the selected object. The object transformations go into the ModelView matrix, where they will impact any shader that performs operations in Eye Coordinates.

To de-select an object, either click in an open area of the graphics window, or uncheck the Enable Object Picking checkbox.

## Monitoring the Frame Rate

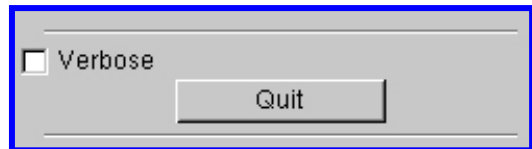
It is sometimes useful to get an idea of how much certain shader operations affect the overall speed of the graphics pipeline. For example, certain math functions are implemented in hardware, some in software. If-tests often cause a slowdown. Low count for-loops are often better unrolled. To see what your current frame rate is, click this checkbox on. This causes *glman* to time your display as you interact. After you turn this option on, you will see two things: (1) a frames-per-second (FPS) number will be superimposed on top of the graphics window, and (2) your display speed will drop sharply. This speed drop is because *glman* loops through multiple instances of your display to acquire more precision on the timing. Your speed will go back to normal once you turn this option off.



The timing does not include the initial setting and clearing of the frame buffers. Nor does it include the swapping of the double buffers. It just includes the drawing of your scene.

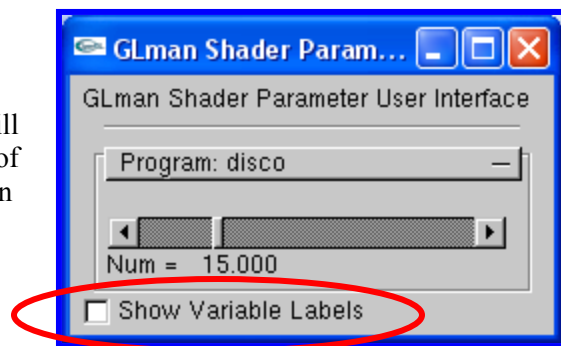
## Miscellaneous

**Verbose** Normally, the messages in the console window are things that you might really need to know. But, if you would like to see more of what is really going on behind the scenes, click this checkbox on. It can be, at times, voluminous. Don't say I didn't warn you.



**Quit** Need I say more?

**Show Variable Labels** This checkbox shows up in the Uniform Variable user interface window. Clicking it will superimpose the values of the uniform variables on top of your graphics scene. This is very handy for doing screen captures of the graphics scene and being able to later recall what uniform variable values made this scene.



## Keyboard Shortcuts

|           |                                     |
|-----------|-------------------------------------|
| Control-D | Display the most recent screen dump |
| Control-E | Edit a geometry file                |
| Control-F | Edit a fragment file                |
| Control-G | Edit a glib file                    |
| Control-L | Load a new glib file                |
| Control-R | Reload the current glib file        |
| Control-S | Screen dump to a file               |
| Control-V | Edit a vertex file                  |

## Questions? Comments?

Direct *gman* questions, comments, and suggestions to:

Prof. Mike Bailey  
Computer Science  
Oregon State University  
2117 Kelley Engineering Center  
Corvallis, OR 97331-5501  
541-737-2542  
F: 541-737-1300  
mjb@cs.oregonstate.edu

