Morpheus_JOGL

Examples for OpenGL/GLSL programming in Java/JOGL.

Version 20171101

Table of Contents

PREAMBLE

Morpheus_JOGL is a Java application designed to provide simple, working examples of OpenGL programming in the Java environment using the JOGL binding (http://jogamp.org/). It is based directly on Morpheus_eProbe and provides the same features as that program in addition to the new graphics examples. The methods developed in this program are expected to be used in the development of new graphics capabilities for the morphometrics program, Morpheus et al. Do let us know if you find the program useful in any way by sending a message in the requested format to the contact address below.

The program is provided free of charge and with no warranty whatsoever. I also include the source code under the Apache 2.0 open source license (1) and JavaDoc documentation.

(1) Open source licenses are bewildering conglomerations of legalese. I have no idea what these really mean, but my intention is to allow you the right to use the source code for your own private purposes and/or freely distributed software so long as proper credit is given. You can read the Apache license here: https://www.apache.org/licenses/LICENSE-2.0

CONTACT INFO

Please address comments, bug reports, ecnomia, etc. to:

morphlab@sc.fsu.edu

And please use the subject line:

Morpheus_JOGL: *

Where '*' is replaced by text indicating the nature of your communication.

DISCLAIMER

And again, this program is provided "as is" and is not guaranteed to do anything whatsoever. Use at your own risk.

ACKNOWLEDGEMENTS

Morpheus_JOGL benefited greatly from work and discussions by others. Detelina Stoyanova wrote a JOGL-based library for me to incorporate into the main Morpheus_et_al program. This project, in fact, was designed to give me the necessary understanding to incorporate that library into the Morpheus program. Whereas, Dr. Stoyanova started from nothing, I have made extensive use of her working code examples. This was particularly important for text rendering and lighting. We both also benefitted greatly from the discussions on the JOGL forum, http://forum.jogamp.org/, on the main JOGAMP site, http://jogamp.org/. Of course, any and all mistakes in the code are of my own doing.

INTRODUCTION

Morpheus_JOGL is an application intended to provide simple, step-by-step examples of OpenGL programming in the Java programming environment. Basic setup and simple drawing is done in the first dialog, Morpheus_JOGL_dlg01_triangle. Subsequent dialogs implement but one or two new features or methods while inheriting the capabilities of those that came before. Each dialog, then, contains only the minimal code to implement the new features so their function can be seen and studied in isolation. Access to OpenGL from within Java is provided by JOGL. That software and more information about it can be found at https://jogamp.org/, and appropriate JOGL libraries must be downloaded and provided to the program in order for it to compile and run. The requisite libraries are discussed below.

This approach was not entirely successful due to the nature of inheritance in Java. Descendent classes actually share the variables in ancestor classes. That is, if a background variable is declared in one class, all descendent classes share that same variable – change background color for one of these descendants and it changes for all. To address this, the code uses `get(...)/set(...)` methods to retrieve or specify the value of a variable. Each dialog has its own unique variable, and overrides the `get(...)/set(...)` functions to work with the local variable. The downside of this is that an increasing proportion of the dialog code is devoted to making local copies of these variables. For instance, in the final dialog, Morpheus_JOGL_dlg15_image, there are over 1600 lines of code, comments, and JavaDoc statements, of which just over 900 are devoted to declaring local variables and overriding their `get(...)/set(...)` functions. This leaves about 700 lines devoted to rendering images as texture either on a rectangle or onto the faces of a cube and providing some degree of new user interaction with that rendering (changing from rectangle to cube rendering).

Given the above, the program should still be useful to some people. Figure 1 shows the running program. The "JOGL\OpenGL Demo|Hot keys" selection has already been made, and hot keys are displayed in the program text area along with the dialogs with which the features are first activated. The dropdown menu shows the dialogs named for their main features. Figure 2 shows the program with all dialogs open and running. Dialogs are shown left-to-right, top-to-bottom after the primary program window.

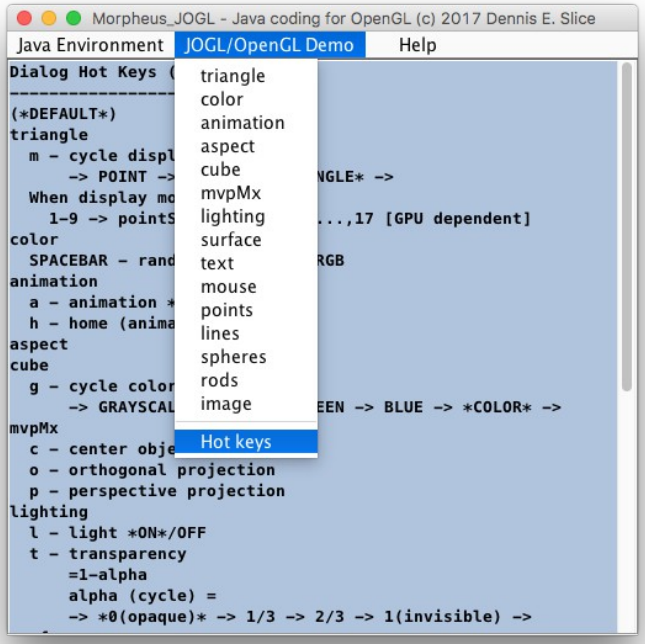


Figure 1: The main Morpheus_JOGL window showing the output of the "JOGL/OpenGL Demo|Hot keys" menu choice.

The first dialog, Morpheus_JOGL_dlg01_triangle, sets up the OpenGL environment and draws an

equilateral white triangle on a black background. The second dialog adds a color background and adds color to the vertices of the triangle, which are interpolated by OpenGL into the interior of the triangle. The third dialog adds animation that gives the appearance of the triangle spinning. The fourth corrects the drawing of the triangle for the aspect (short,wide v. tall,narrow) of the window. The fifth dialog renders a cube, which must be specified as a series of triangles, and the sixth adds a choice between perspective (default) and orthogonal projection. The seventh dialog enhances the geometry to show a smooth sphere (again represented as numerous triangles) and adds lighting effects to the rendering. We simply add reading in a more complicated surface from a file and rendering it in the eighth dialog. Text is added to the scene in dialog nine, and mouse control in dialog ten.

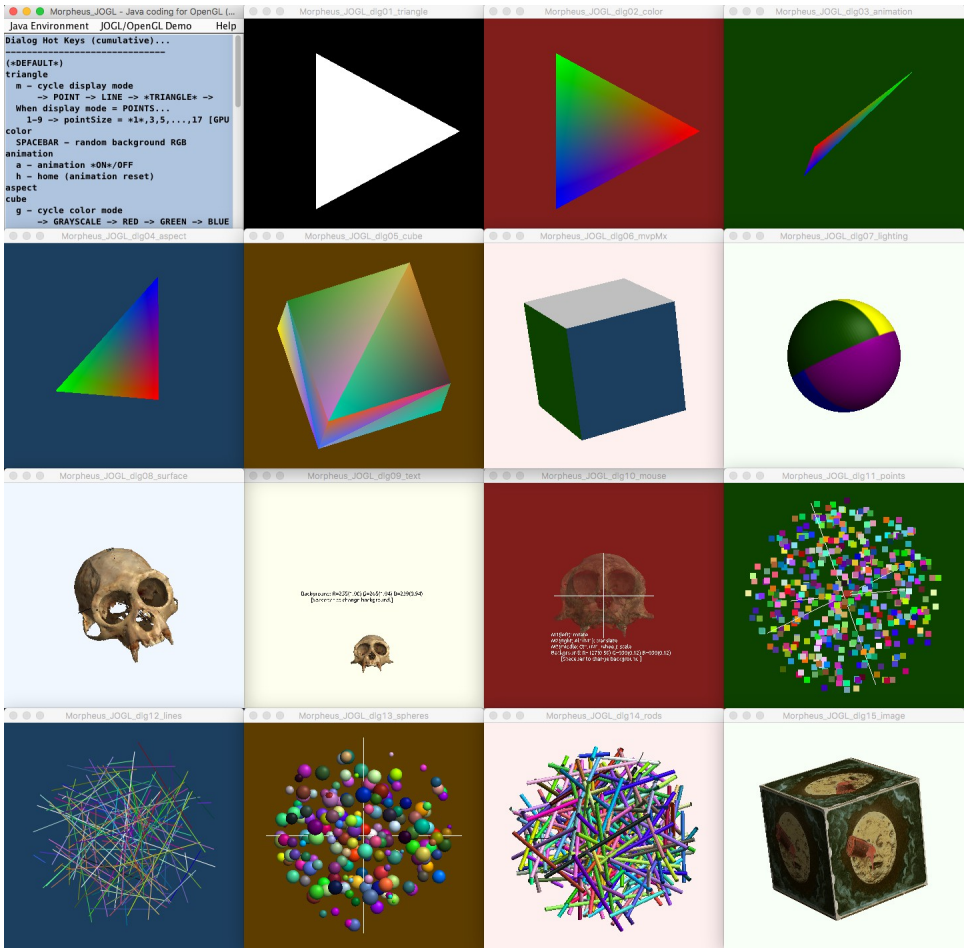


Figure 2: The main Morpheus_JOGL program window and all demo dialogs. Dialog order (1-15) and increasing capabilities are illustrated from left-to-right, top-to-bottom.

The program inheritance trifurcates after Morpheus_JOGL_dlg10_mouse. Dialog eleven and twelve descend directly from this dialog and add the display of points and line segments. This is a distinct branch because the effect of point/line rendering is achieved simply through the manipulation of rendering options – a set of random vertices can be rendered as individual points or one can draw lines between pairs of such points. The second branch of the trifurcation has dialogs thirteen and fourteen

being derived from the mouse dialog. These are fundamentally different in that prior dialogs rendered single geometries – a triangle, a cube, a sphere, a skull surface. Dialog thirteen renders a number of spheres of random size at random locations (within a unit sphere) of random colors. This is done by creating a single set of vertices for a sphere, but defining arrays of colors, locations, and scaling factors to give each a unique appearance. Dialog fourteen does the same thing for rods (closed cylinders), but the computations become more complicated as each rod has to be scaled in the correct direction, rotated to match the orientation of the desired line segment, and moved to the desired location.

Finally, Morpheus_JOGL_dlg15_image, implements the display of images. These are handled as texture. An image is read in to an OpenGL texture data structure. Vertices are then assigned coordinates within the texture, and OpenGL interpolates the texture coordinates for each pixel. Texture color values are return from the texture using the built-in "sampler". Texture coordinates for the vertices are assumed to range from (0, 0) to (1, 1) in both directions. Hence, some care must be taken to ensure proper image aspect, if that is desired.

The program documentation that you are reading at this moment is, by design, rather scant. Instead, the program is more-or-less self-documenting through JavaDoc-ing of all new methods, extensive code comments, and long, explicit variable names. For consistency (or sloth if you wish to read it that way), methods and variables are commented only when they are first introduced or when they are overridden. It was just too hard to try to coordinate identical comments over fifteen dialog source files.

KNOWN ISSUES

The program is not without its flaws – they may, in fact, be numerous,. Here are the known ones.

BUG: Off-origin text is not precisely positioned. This just requires, I think, working through and adjusting for the effects of a number of transformations, but other professional deadlines prevent me from doing this at this time.

INEFFICIENCY: All geometry is passed for rendering as explicit vertex coordinates. It can be more memory-efficient to store the vertex coordinates once and pass faces as indices, instead of actual vertex coordinates, using glDrawElements() in place of glDrawArrays(). Doing so could greatly reduce storage for large geometries with redundant use of vertices in faces, but would require, I think, vertex colors and normals to be constant. To keep the program as simple as possible for downstream inheritance, this was not done. The only geometries likely to see a meaningful efficiency improvement are the external surface and sphere. I might possibly add a dialog using this feature at a later time.

USEFUL REFERENCES

Besides the internet and individuals, much of the OpenGL and JOGL-specific knowledge used to build the program were obtained from:

Shreiner, Sellers, Kessenich, Licea-Kane. 2013. *OpenGL Programming Guide Eighth Edition*. Addison-Wesley.

Wolff. 2011. *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing.

IMAGE CREDITS

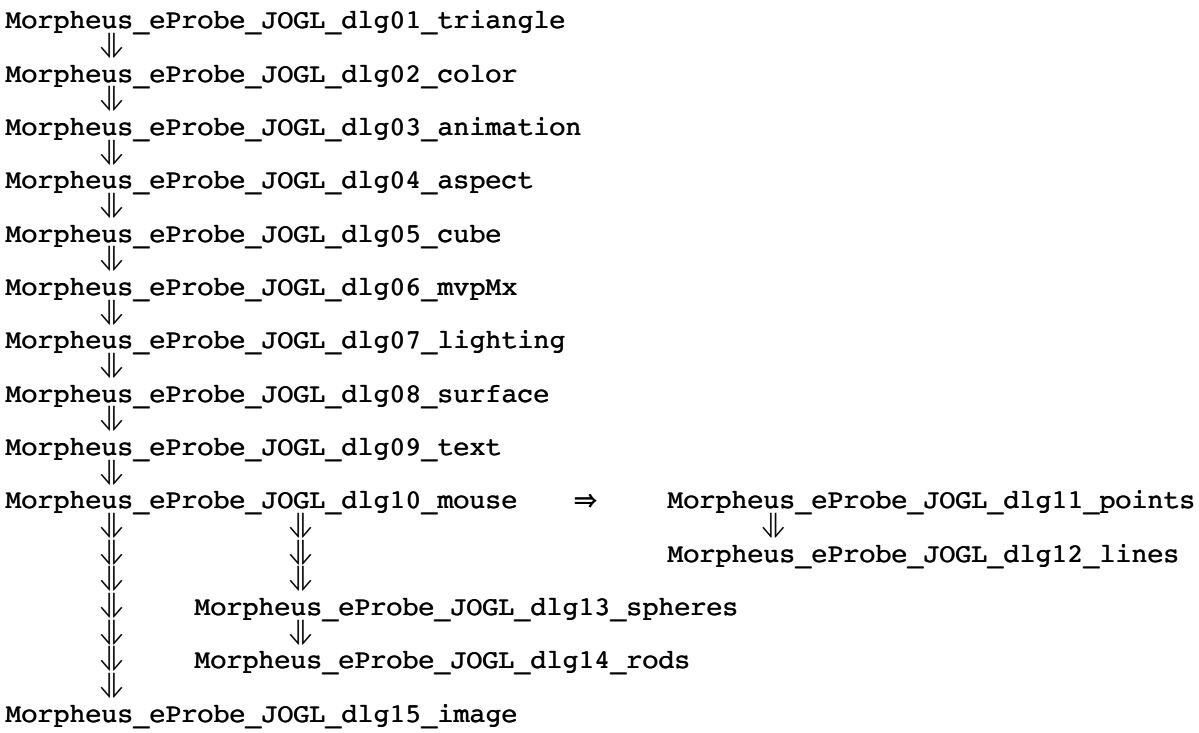
The image of the skull displayed in Morpheus_JOGL_dlg08_surface and subsequent dialogs was generated from data available as part of the publication:

Pomidor, Benjamin J., Jana Makedonska, and Dennis E. Slice. “A Landmark-Free Method for Three-Dimensional Shape Analysis.” PLOS ONE 11, no. 3 (March 8, 2016): e0150368. https://doi.org/10.1371/journal.pone.0150368.

The image used in Morpheus_JOGL_dlg15_image is a frame from a hand-colored print of Georges Méliès's 1902 film *Le voyage dans la lune*. It is available for unrestricted reuse from: https://commons.wikimedia.org/wiki/File:Melies_color_Voyage_dans_la_lune.jpg

INHERITANCE STRUCTURE

Morpheus_JOGL provides fifteen dialogs illustrating incremental aspects of OpenGL programming. Each dialog after the first inherits the capabilities of those above and adds one or two new features. This inheritance is linear from the first, Morpheus_JOGL_dlg01_triangle, through the tenth, Morpheus_JOGL_dlg10_mouse. At that point, the inheritance trifurcates. The first branch begins with Morpheus_JOGL_dlg11_points. This class changes how vertices are generated – random points within a unit sphere, and adjusts the rendering mode to draw these vertices as points. Morpheus_JOGL_dlg12_lines is a direct descendent of this class and further changes the rendering mode to draw simple lines between the pairs of random points. The second branch begins with Morpheus_JOGL_dlg13_spheres to provide a more elegant representation of points in space as spheres. Here, the vertices and faces for a smooth sphere are generated. Then, a number of parameters for different spheres are generated, each with its own location, scaling, and color. Morpheus_JOGL_dlg14_rods extends this class to render rods (closed cylinders) between random points within a unit sphere. This requires more complicated mathematics to compute appropriate scaling, location, and orientation terms to place the common rod geometry to extend from one random point to another. Finally, Morpheus_JOGL_dlg15_image requires new code to read in images as textures and provide coordinates mapping vertex points into the image. This also requires new vertex and fragment and fragment shaders to properly sample the texture for screen rendering. Below is an illustration of the inheritance pattern of all of the dialogs.



PROGRAM LIBRARIES AND CLASSES

This section describes the functionality of each of the Java classes used in the program including those carried over without modification from Morpheus_eProbe. The dialog descriptions outlines the features implemented in each dialog and, when appropriate, aspects of the vertex and fragment shaders used. When no new capabilities are required, a dialog can use previously developed shaders, and this is noted in the description. Furthermore, the descriptions note any hot keys introduced with each dialog and changes to the program defaults for the features connected to these hot keys. The program was developed using Netbeans 8.2, and source includes the .form file that handles the visual editing of Java components such as menus, text areas, etc.

Required libraries...

Morpheus_JOGL uses the JOGL binding to access OpenGL features. As such, the associated libraries are required to compile the program (they are included automatically in the distribution to run the program). In addition, the "eProbe" part of the program requires libraries associated with Java3D to compile, but these are not necessary for the OpenGL demos that are the focus of this program. Java3D-related methods can easily be deleted from the Morpheus_JOGL_00_startup.java class to remove this dependency. The necessary JOGL and Java3D libraries can be obtained from http://jogamp.org/jogl/www/. Those used to build the program are all included in the ./lib directory distributed with Morpheus_JOGL.

JOGL – the OpenGL Java bindings. Files with the text "linux", "macosx", and "windows" provides support for Linux, OS X, and MS Windows operating systems, respectively. Others are non-OS-specific libraries. As written, text capabilities require the font file to be in the "atomic" subdirectory.

```
./atomic/jogl-fonts-p0.jar
gluegen-rt-natives-linux-amd64.jar
gluegen-rt-natives-linux-i586.jar
gluegen-rt-natives-macosx-universal.jar
gluegen-rt-natives-windows-amd64.jar
gluegen-rt-natives-windows-i586.jar
gluegen-rt.jar
jogl-all-natives-linux-amd64.jar
jogl-all-natives-linux-i586.jar
jogl-all-natives-macosx-universal.jar
jogl-all-natives-windows-amd64.jar
jogl-all-natives-windows-i586.jar
jogl-all.jar
```

JAVA3D – the Java3D libraries. Required for full "eProbe" functionality, but not required for the OpenGL demos.

```
j3dcore.jar
j3dutils.jar
vecmath.jar
```

Morpheus_JOGL_00_startup.java – the initial class that contains the `main(...)` function to initialize and display the main program window. It also contains a commented out function to open all dialogs and tile them on screen. If the `fullscreen` parameter passed to that function is `true`, the main window and the fifteen dialogs are tiled to fill the entire screen to the extent possible. If this

parameter is `false`, then the height of the screen is filled, but the tiled windows are square. This was included to easily develop images for the documentation (Figure 2 above) and to check that all dialogs were functioning properly after programming changes.

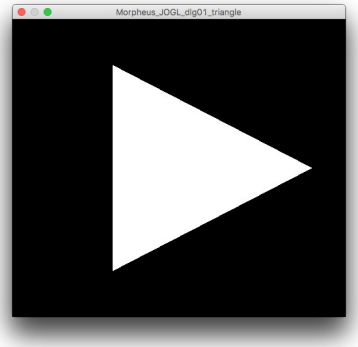
Morpheus_JOGL_01_Java.java – a class directly carried over from Morpheus_eProbe that collects non-graphical system information and has a function that can set graphical information strings from classes that do support graphics.

Morpheus_JOGL_02_JOGL.java – a class from Morpheus_eProbe to test for the existence of the Java OpenGL binding, JOGL. It requires the main eProbe Java class be passed in its constructor so it can set the proper parameter strings.

Morpheus_JOGL_03_Java3D.java – a class from Morpheus_eProbe to test for the existence of the Java 3D API. It requires the main eProbe Java class be passed in its constructor so it can set the proper parameter strings.

Morpheus_JOGL_04_Jframe.java – the main Morpheus_JOGL program window. It provides a text output area for reporting system information and Hot key occurrence and usage. There is an associated .form file that is used by Netbeans for visual Java control editing.

Morpheus_JOGL_dlg01_triangle.java – draws a white, equilateral triangle on a black background. this first dialog provides the most basic capabilities for OpenGL/JOGL programming. It defines some generally useful constants and the first set of class-specific variables and their `get(...)`/`set(...)` methods that will be overridden in descendent dialogs so that each level of the lineage has its own unique variables – the body of the code uses the `get(...)`/`set(...)` functions for access to them.



As in all dialogs, there is a `main(...)` function defined that allows the standalone running of the dialog independently of the main Morpheus_JOGL – this is very handy for debugging.

This class defines an `initOverrides(...)` method that is called after dialog creation – Java strongly discourages the use of overridden functions in class constructors. Among other things, this function creates the GLCanvas object, which is what provides access via JOGL to the OpenGL system. This is added as the drawing area for the dialog.

The `setupDialog(...)` method is called after `initOverrides(...)` to position and display the dialog.

The `generateGeometry(...)` method generates the vertex coordinates and whatever else might be needed for a displayed graphical object in later, overridden functions. Here, the coordinates are of an

equilateral triangle circumscribed by an origin-centered circle of radius 2.

The class then overrides the four methods required of a GLEventListener – `init(...)`, `reshape(...)`, `display(...)`, and `dispose(...)`.

The `loadBuffers(...)` method, called from `init(...)`, sets up and stores data in the OpenGL environment.

The `display(...)` method handles the OpenGL drawing, but the heavy lifting is actually passed off to a `drawScene(...)` method that can be more flexibly overridden in subsequent dialogs.

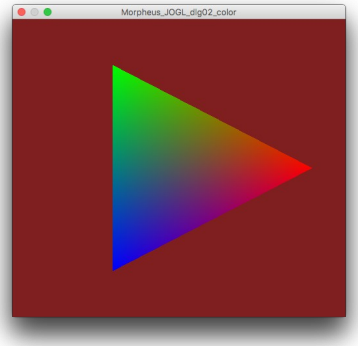
A number of utility methods are also defined including: `loadShaderFile(...)` that reads in a shader program as a text string, `initVertexShader(...)` that compiles a string read in from a vertex shader file, `initFragmentShader(...)` that compiles a string read in from a fragment shader file, `initShaderProgram(...)` that combines the compiled vertex and fragment shaders into a single OpenGL program, and `printShaderLog(...)` and `printProgramLog(...)` that are general debugging methods for shader and OpenGL program testing.

Several methods are overridden that are required for a KeyListener: `keyTyped(...)`, `keyPressed(...)`, and `keyReleased(...)`. Morpheus_JOGL only uses the `keyPressed(...)` method. In this case, it is used to change the display mode from solid triangle to vertex points to outlined triangle. Keys are also captured to set the size of the points when the vertices are rendered as single points.

Each OpenGL GLSL program must provide two shaders – a vertex shader that processes data coming into OpenGL from the program before passing it along, and a fragment shader, that accepts data from the OpenGL system and determines final pixel color. For this first dialog, the vertex shader only receives vertex coordinate data and passes it on in proper (homogeneous coordinate) form, e.g., (x,y,z,1.0). The fragment shader does nothing but set the color for any pixel requested to an opaque white: (red=1.0, green=1.0, blue=1.0, alpha=1.0). The "alpha" controls pixel transparency with 0.0 being invisible and 1.0 being opaque. All values are floats.

Hot Keys: 'm' – mode. Cycles the display mode from filled faces to vertex points to face edges, and back to filled faces, etc.

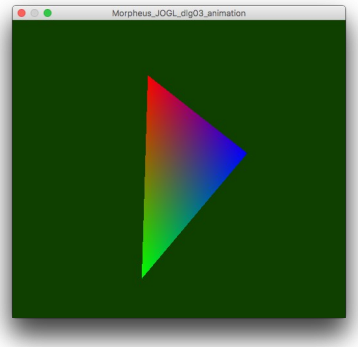
'1'-'9' – point size. When in point display mode, this sets the size of the points rendered.



Morpheus_JOGL_dlg02_color.java – adds color to the vertices of the triangle created in the first dialog and adds a color background. New variables are added to identify and hold the background and vertex colors. A method is provided to generate random background colors. Other methods are overridden as appropriate (see the "METHODS" section of the Appendix). The vertex shader is modified to receive and pass on the vertex color as an (r,g,b) 3-vector. The fragment shader simply passes out the color it has been given augmented with a constant alpha=1.0 value.

Hot Keys: 'space bar' – changes the background to a randomly generated color.

Morpheus_JOGL_dlg03_animation.java – animates the color triangle display with continuous rotation. This requires an animator and a rotation matrix which are provided as new local variables with `get(...)/set(...)` methods and a frame counter to change the direction of rotation every so often. The animation effect is provided by a JOGL FPSAnimator that attempts to call the `display(...)` method at a desired frame-per-second rate, which is set to 24fps in this program. At each call to the `display(...)` method, the animation rotation matrix is updated by multiplication by incremental rotation around the x, y, and z axes. After so many frames, these parameters are changed to add visual interest. Other methods are added or overridden to support this.



The animation matrix is actually passed to the vertex shader as part of a general mvp transformation matrix. This provides for transformation of the geometric model to "world" or "data" space, the transformation from that space to the "view" space, which in OpenGL is -1 to +1 mapping of the screen x, y display and 0 to +1 in the z direction. Rotation, translation, and scaling can all be applied through the construction of these intermediate matrices and composited into a single mvp matrix prior to rendering. The vertex shader is modified to receive this matrix and apply the transform to the vertex coordinates before passing them back to OpenGL. For the time being, the mvp matrix is just a copy of the animation rotation matrix. Later on, a whole series of matrices will be defined to separate the various components of different transformations and user-driven mouse interaction.

The mvp matrix is defined as a "uniform" variable within the vertex shader. Unlike, say, colors that may change with each vertex element, "uniform" variables are set once for processing of all vertices currently being processed.

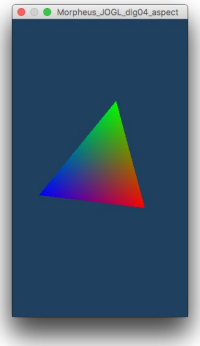
Changes are made to the shaders to support color being passed out of the vertex shader and into the fragment shader as a 4-vector (r,g,b,a).

A utility method is also defined that prints out a given matrix and associated string in a format that can be copied and pasted directly into R for testing transformation math.

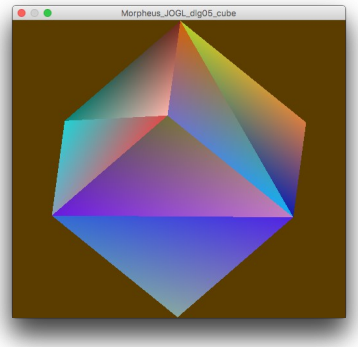
Hot Keys: 'a' – animation. Turns animation ON (`true`) or OFF (`false`). The default ON.

'h' – home. Reinitializes the animation rotation matrix to its home position – the animation rotation matrix is set to the identity matrix. Unless the animation is stopped, however, rotation will continue from this orientation.

Morpheus_JOGL_dlg04_aspect.java – previous matrices mapped the equilateral triangle to fill the window, which OpenGL maps to -1 to +1 in the x, y directions (and 0 to 1 in the z). In this dialog, an aspect adjustment matrix is defined that scales the data in the x or y direction so that the aspect of the data is preserved regardless of window shape, say, tall-narrow vs. short, wide, relatively speaking. This adjustment matrix is set in the overridden `reshape(...)` method and composited into the mvp matrix. As such, this dialog can use the unmodified vertex and fragment shaders from Morpheus_JOGL_dlg03_animation.



Morpheus_JOGL_dlg05_cube.java – this class constructs more complicated geometry, a cube, from the most complex geometry support by OpenGL, the triangle. A cube has six, square faces, and each face can be defined by two triangles. Thus, to represent a cube, one needs to specify twelve triangles. The `generateGeometry(...)` method is overridden to call a `makeCube(...)` method that redefines the vertex and color array to represent a cube. Vertex colors are set randomly for illustration. Note, too, that when triangles are rendered the ordering of their vertices determines the front and back of the triangle. A face showing ABC in a counterclockwise direction would be the front face. That showing ABC in a clockwise direction is considered the back face. Details of cube specification are found in the code comments.

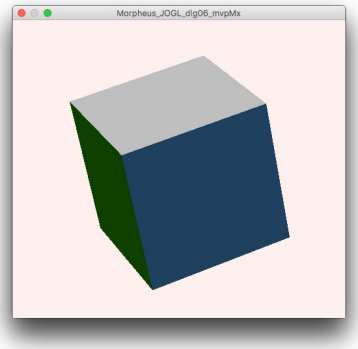


Since the above required relatively little coding, the ability to change the handling of color during rendering was added. The default is to color the vertices according to how they are specified. Alternatives include mapping those colors to grayscale or red- or blue- or green-scale, sequentially, before returning to color. Code comments describe the mapping used.

Again, no changes were required for the shaders and the dialog uses those from Morpheus_JOGL_dlg03_animation.

Hot Keys: 'g' – grayscale. Cycles the color mode from color to grayscale to red to green to blue, then back to color, etc.

Morpheus_JOGL_dlg06_mvpMx.java – this dialog introduces projection, either perspective or orthogonal, onto the view port, a.k.a. the dialog window. Previous dialogs assumed data were already within the OpenGL frustum (view volume) and the animation transformation simply rotated the object is place. Here we introduce all of the matrices for various transformations, and leave all as identity except those for projection of the objects onto the view port. There are two options. Perspective projection approximates real world viewing in which things farther away appear closer together. Parallel lines, e.g. railroad tracks, seem to converge in the distance. Orthogonal projection projects the data straight onto the view screen such that parallel lines going off into the distance remain parallel after rendering onto the viewport and distance between points at varying distance from the viewer maintain their apparent distance.



The full compliment of matrices introduced are the model matrix that transforms the generated geometry into the desired scale, orientation, and position, the view matrix that transforms the resulting data coordinates to fit within the OpenGL viewing volume (-1 to +1; -1 to +1; 0 to +1) for (x, y, z), and the projection matrix that transforms to give the desired type of projection. Each of these can be composited from several other matrices. The model matrix, for instance, is the product of model scaling, rotation, and translation matrices applied in that order – order is critical! The view matrix consists of the product of the view scaling, rotation, and translation matrix followed by the animation rotation matrix and the mouse scaling, rotation, and translation matrices. Finally, the projection matrix

consists of a base viewing matrix to reposition the data in the frustum for projection and a projection matrix, which could be orthogonal or perspective, which effects the desired projection, and a final aspect correction matrix.

The coordinates leave the vertex shader after the mvp transformation has been applied. That is:

```
coords_out      = mvp * data_coords

                = perspective * view * model * data_coords
                (terms reversed because of how mx mult carried out)

                = aspect
                * projection
                * projection_base
                * mouse_translation * mouse_rotation * mouse_scaling
                * animation_rotation
                * view_translation * view_rotation * view_scaling
                * model_translation * model_rotation * model_scaling
                * data_coords
```

The `generateGeometry(...)` method is overridden to provide solid face colors and to tweak the location and orientation a bit for testing and visual interest. There is a function, `tweakGeometry(...)`, to apply a certain amount of translation and scaling and a function for determining the extent of the geometry to be rendered, `setModelToViewMx(...)`. This allows the user the choice of centering that display on the object or on the origin. The min/max of the data accounting for centering preference are saved for later plotting of axes.

A utility function is retained that shows a transformation matrix for debugging.

Changes are made only to the mvp matrix. Hence, this dialog still uses the Morpheus_JOGL_dlg03_animation vertex and fragment shaders.

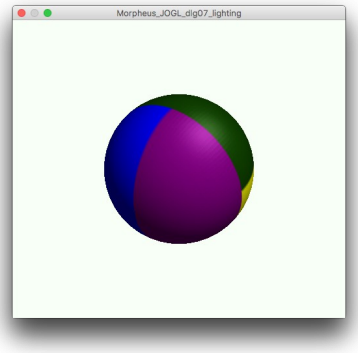
Hot Keys: 'c' – center. This toggles between centering the rendered scene on the origin or the center of the object being rendered. The default may change from dialog to dialog depending upon which display is most appropriate for the feature being developed.

'p' – perspective. Renders scene using perspective projection.

'o' – orthogonal. Renders scene using orthogonal projection.

The home key, 'h', is overridden to account for different matrix initialization.

Morpheus_JOGL_dlg07_lighting.java – adds lighting to the scene. Previous dialogs simply colored pixels interpolated between vertices by the color specified at the vertices or interpolated from them. This dialog colors each pixel according to its illumination from various light sources: ambient, diffuse, and specular. Ambient light is that which is reflected all around and therefore illuminates all pixels to some degree. Diffuse light is directional and more brightly illuminates pixels that are part of surfaces more directed toward the light source. Specular light, finally, is the focused light coming from a specific light source that provides highlights on the surface of the object. A reasonable



discussion of these light sources and how they are implemented in GLSL can be found at http://www.opengl-tutorial.org/beginners-tutorials/tutorial-8-basic-shading/.

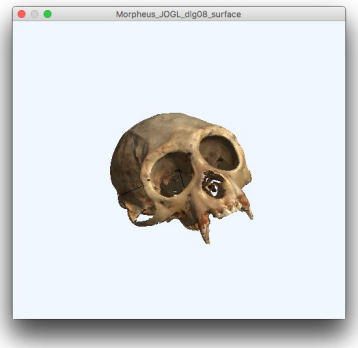
The computation of all of these components is somewhat involved and requires the calculations of the pixel location relative to the viewer and the light source. The necessary code changes were adapted from the examples provided in David Wolff's "OpenGL 4.0 Shading Language Cookbook" (2011).

A key new component necessary for the computation of lighting is the "normal" vector at each pixel. If you consider a planar table top, the normal vector for any spot on the table top is a (unit) vector that points upwards at a right angle from the plane of the top. Light hitting this point at some angle bounces off of the point at the same angle attenuated somewhat by the shininess of the material at that point. Each vertex has a normal calculated for it relative to the triangle of which it is a part. Thus, vertices that are part of several triangles will have normals for each of those triangles just as it has a color assigned for each triangle. The main body of code for this class includes variables and methods for computing the normals for the vertices of each triangle and for storing these values and passing them into the vertex shader.

In addition, one must compute the transformations of the normals within the vertex shader, as well as the position of the viewer (eye position). The general solution for this requires additional mv (model-view) and mvit (model-view inverse transpose) matrices. These are passed into the shader as uniform variables (unchanging during geometry rendering) as well as some variables for using the lighting calculations (or not), `lightson`, and a transparency specification, "alpha" - previous dialogs used only opaque rendering. Since shaders do not support boolean variables, `lightson` is passed as a float that is interpreted as `true` if the value is >0.5 and as `false` for any other value. If `lightson` is `false` pixel color is unaltered with transparency set according to the alpha level.

Hot Keys: 'l' – lighting computations ON (`true`)/OFF (`false`). They are ON by default.

't' – transparency. Cycles through opaque (alpha=1.0) through various levels (alpha=0.6, 0.3) of face transparency until face is invisibly rendered (alpha=0.0), then back to opacity and so forth. The rendering of true transparency is a very complicated issue – the distance and occlusion relationships of all elements must be taken into account. Morpheus_JOGL only approximates transparency by changing the opacity of individual rendering elements.



Morpheus_JOGL_dlg08_surface.java – loads a more complicated geometry from an external file and adds axis plotting. Rendered geometry thus far has been generated mathematically within the program. In this dialog an external file is read containing vertices, vertex color, and face definitions. The provided file, Morpheus_JOGL_sample.surface, is in a very simple, non-standard format. The first line gives the number of vertex definition lines to follow immediately. Each of these lines contains the coordinates of a vertex and its color as RGB ranging from 0-255. This is followed by a line containing the number of face definition lines, each of which contains three vertex indices, i,j,k. This is actually a pared down version of data from a .ply file. However, a method for the general reading of .ply files or .obj, or .stl, or other standard formats is beyond the scope of this project.

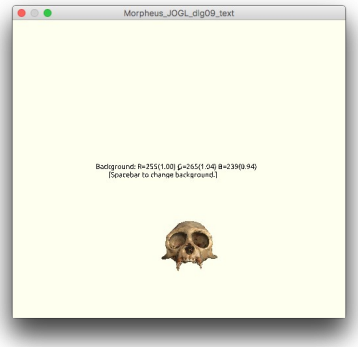
Since the above only requires an override and rewrite of the `generateGeometry(...)` method, this dialog also adds the rendering of axes – either at the origin extending -1 to +1 on each axis or at the center of the object and extending over the min/max of the object in each dimension. Which is used depends upon the centering option chosen for display – uncentered rendering shows the axes at origin, centered rendering shows that axes centered within the geometry.

Code modifications are mainly an override of `generateGeometry(...)` and the provision and manipulation of axis drawing variables and associated methods. The overridden `drawScene(...)` method calls a `drawAxis(...)` method if requested.

The dialog uses the shader files from Morpheus_JOGL_dlg07_lighting vertex and fragment shaders.

Hot Keys: 'x' – axis. Turns ON (`true`)/OFF (`false`) the plotting of axes. Default may change from dialog to dialog.

Morpheus_JOGL_dlg09_text.java – adds text to the scene. Specifically, this dialog displays a string showing the background color in both 0-255 RGB values and the OpenGL compatible 0.0 to 1.0 values along with instructions for randomly changing the background color. I find I often come across pleasing colors through random generation that I could not directly appreciate with color-picking swatches or explicit, manual blending. This provides the code values to reproduce those colors.



Text drawing is not a well-defined component of OpenGL. In fact, it is not even indexed in the OpenGL Programming Guide or the OpenGL 4.0 Shading Language Cookbook. The routines used here were modified from online examples posted in the JOGL forum as noted in the program comments.

There is one known bug in this code. Off-origin text is not positioned properly. This likely requires just sitting down and working through the mathematics of the transformation matrices for text, but other hard deadlines prevent me from doing this at the moment, and I don't want to delay distribution of the program/code just for this.

Also, it is worth noting that the text drawing routines disable blending, which is necessary for the pseudo-transparency achieved through an alpha<1.0 in the pixel color specification. This must be re-enabled at the end of the drawText method.

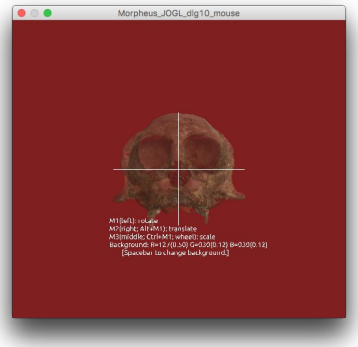
No changes are required to the Morpheus_JOGL_dlg07_lighting vertex and fragment shaders.

Hot Keys: 's' – string. Turns the display of the program-generated string ON (`true`) or OFF (`false`).

Object centering and animation default to OFF for this dialog.

Morpheus_JOGL_dlg10_mouse.java – provides mouse support for user manipulation of the scene. The left mouse button (click and drag) controls rotation, the right button controls translation, and the middle button or wheel controls scaling. These functions can also be accessed via modification of the

left mouse button: left button = rotation, left+alt = translation, and left+ctrl = scaling.



The computations for intuitive rotation are rather complex and emulate the manipulation of a trackball. Thus, the screen coordinates of the mouse position must be projected onto the ball and rotation computed through changes in this position. The relevant code was modified from various sources with discussions of the procedure found in links provided in the program comments.

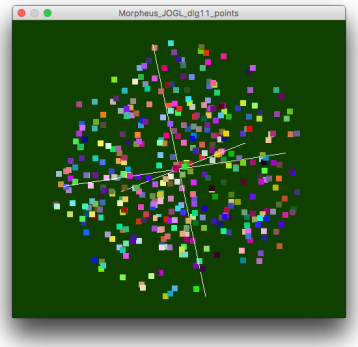
The string-generation method is overridden to add hints on mouse usage to the user.

Again, the Morpheus_JOGL_dlg07_lighting vertex and fragment shaders are used.

This dialog also marks a trifurcation in dialog inheritance. The immediate branch generates plots of simple points and lines mostly through manipulation of the drawing mode. The second branch deals with the more complex issue of rendering multiple geometries, spheres and rods (closed cylinders). The third handles images as textures mapped to either a rectangle (preserving image aspect) or a cube (not preserving aspect).

Hot Keys: 'r' – reset. Resets the mouse transformation matrices to their original states.

Object centering is turned ON for rendering, and the surface from the previous dialog is rendered in its most pseudo-transparent, but visible, mode.

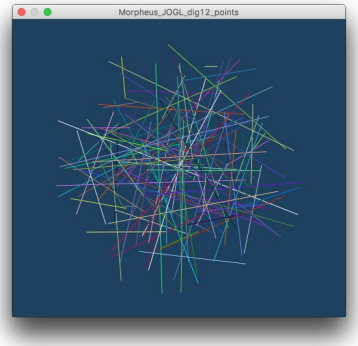


Morpheus_JOGL_dlg11_points.java – illustrates the plotting of simple points in random colors through the manipulation of the rendering mode. A method is provided that generates random points within the unit sphere. The `initOverrides(...)` method is overridden to set the rendering mode to POINTS, and the `setModelToViewMx(...)` is overridden to bypass the centering computations, which are unnecessary since all points are within the -1 to +1 limits of OpenGL plotting (adjusted for aspect). Fake normals are generated for the points, and the only programming complexity is that point coordinates are added sequentially to the vertex array until a specified number (500) is reached, then replaced starting at the beginning of the array. Point generation is linked to the animator's call to the `display(...)` function, so when animation is turned OFF, point generation stops. Also, mouse control of the image forces a repaint and, thus, also calls the point generation routine. The dialog uses the Morpheus_JOGL_dlg07_lighting vertex and fragment shaders.

Hot Keys: As with the existing POINT display mode, the number keys '1'-'9' set the point size from 1 to 17. The default for this dialog is '5'.

Lighting is turned OFF by default because the fake normals don't provide full lighting of the back of the points, but this can be turned on to see the effect.

Morpheus_JOGL_dlg12_lines.java – generates random lines within a unit sphere. The only substantial change to Morpheus_JOGL_dlg11_points is that random points are generated in pairs and the `drawScene(...)` method is overridden to set the drawing mode to GL_LINES that treats the array as pairs of line segment endpoints. The specified number of lines is 120, and segment generation is again linked to the animator. Segment generation is linked to the `display(...)` functions, hence stopping the animator stops segment generation and mouse manipulation forces a repaint, which calls segment generation.



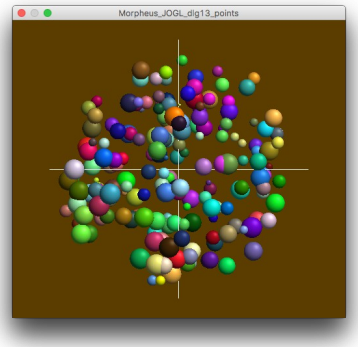
One noteworthy aspect is that setting the width of rasterized lines through `glLineWidth(...)` is not a required component of an OpenGL implementation. To avoid using this, it is suggested lines be drawn as narrow polygons, but this has not been implemented here.

The dialog uses the Morpheus_JOGL_dlg07_lighting vertex and fragment shaders.

Hot Keys: The mode key, 'm', has been disabled to force line segment drawing.

Lighting is turned OFF by default for the same reasons as in the previous dialog.

Morpheus_JOGL_dlg13_spheres.java – the second branch of the trifurcation at Morpheus_dlg10_mouse plots spheres of random size and color at random locations within a unit sphere. Each sphere uses the vertex and normal arrays generated in the method introduced in Morpheus_JOGL_dlg07_lighting dialog, but has its own scaling and location transformations set through individual model matrices. These model matrices are then used in the construction of the mvp matrix prior to rendering each sphere. This better approximates the actual plotting of data points likely to be used in an application.

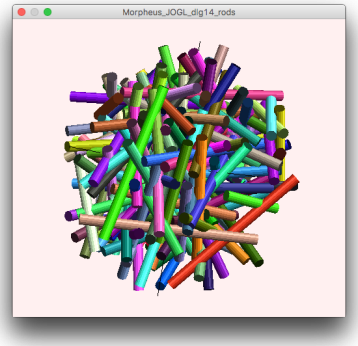


Spheres are added to the scene up to a specified number (200). Afterwards, they are replaced sequentially. Because of the complexity (number of faces to achieve a smooth look) of each sphere, animation rotation is turned OFF. Sphere generation is, however, handled through calls to the `display(...)` method by the animator, so turning the animator OFF stops sphere generation, while mouse interaction, which requires a repaint, calls the sphere generator. The dialog uses the Morpheus_JOGL_dlg07_lighting vertex and fragment shaders.

Hot Keys: No new hotkeys are introduced.

Morpheus_JOGL_dlg14_rods.java – renders rods (closed cylinders) between random points in random colors within a unit sphere. Similar to the Morpheus_JOGL_dlg13_spheres dialog, the rods are

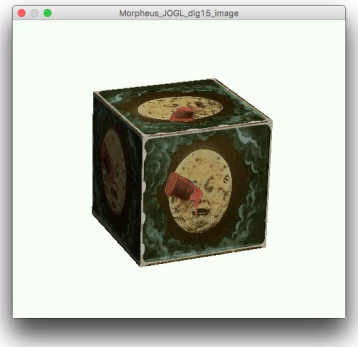
based on a single generated geometry and individually transformed using the model scale, rotation, and translation matrices.



Construction of the base rod is a bit tedious. It has unit length and unit diameter, which are altered by the model matrices. Each rod is scaled to a length matching the distance between the two random points along its axis, but scaled according to a diameter specification in the plane orthogonal to that. It is oriented in the direction parallel to the vector between the random points, and positioned such that one end is located at the first random point. Given the scaling and rotation, the rod thus extends between the two points. The primary computational complexity arises in the computation of a rotation transformation to align the rod with the vector between the two points. See code comments for details.

Animation rotation is restored, and again, the dialog uses the Morpheus_JOGL_dlg07_lighting vertex and fragment shaders.

Hot Keys: 'd' – diameter. Cycles through a number of rod diameters: 0.005 (default), 0.01, 0.02, 0.04, 0.08 relative to the unit sphere.



Morpheus_JOGL_dlg15_image.java – the final branch of the trifurcation at Morpheus_JOGL_dlg10_mouse illustrates the handling of external, two-dimensional images. The images are loaded as textures, and the triangles making up the rendered geometry are assigned coordinates within the texture/image. The texture is mapped internally from 0.0 to 1.0 in both the x and y dimensions. The initial scene maps the texture/image to a rectangle adjusted for the aspect ratio of the original image as one might do for general image display. An alternative rendering maps the entire image to the faces of a cube.

Code modifications from Morpheus_JOGL_dlg10_mouse, involve restarting the animator in the `initOverrides(...)` method and generating the appropriate geometry (rectangle or cube) according to the `asImage` variable. When `true`, the geometry is an aspect-corrected rectangle, when `false` it is a cube. The texture coordinates assigned to the corners of the rectangle are (0.0, 0.0), (0.0, 1.0), (1.0, 1.0), and (1.0, 0.0), and similarly for each face of the cube.

A `loadImage(...)` method is provided that reads in the image and loads it into an OpenGL compatible texture. Two other images illustrating different image aspects (tall vs. wide) can be accessed by commenting out the default image loading line and uncommenting one of the other lines of code. The previous `makeCube(...)` method is overridden to supply required, but unused, vertex coordinates and texture coordinates for each of the constituent vertices. A `makeRectangle(...)` method provides similar functions. Checks are included to see if geometry regeneration is required due to changes in the requested geometry – rectangle vs. cube. Special handling is required to adjust for the requested color mode (color, grayscale, red, green, blue) without reloading the original image.

Shader setup methods are also overridden to provide the additional texture coordinate data. The vertex shader adds new variables to receive the texture coordinates for a vertex and a toggle to indicate if these coordinates are to be used or not. The latter allows the same shaders to be used for drawing the

axes (`useTexture` OFF) as for rendering the image (`useTexture` ON). For general applications, one might have separate shaders for drawing these different components.

The fragment shader uses the default sampler provided by OpenGL to retrieve appropriate texture color if `useTexture` is ON or uses the provided color if it is OFF. Once the base pixel color is set, processing proceeds as before according to the state of the `LightsOn` toggle.

Hot Keys: 'i' – image. Swaps display from mapping image onto an aspect-corrected rectangle to mapping it onto the faces of a cube and vice versa.

APPENDICES

CODE, JAVADOC, AND COMMENT LINES

It can be meaningless to report lines of code for a project since a large number of lines could indicate either program complexity or programmer inefficiency. These are reported here to provide an indication of the actual coding involved to achieve particular features because an increasing amount of dialog lines were taken up by providing local variables and overridden `get(...)/set(...)` methods. Eleven of the fifteen dialog classes have well over 1000 lines, but only four have more than 500 lines devoted to adding new features or capabilities. These line counts, too, include all code, JavaDoc, and extensive inline comments.

Dialogs	Total Lines	Local Variables	Net New Lines
Morpheus_JOGL_dlg01_triangle	1035	363	672
Morpheus_JOGL_dlg02_color	402	280	122
Morpheus_JOGL_dlg03_animation	644	381	263
Morpheus_JOGL_dlg04_aspect	460	379	81
Morpheus_JOGL_dlg05_cube	826	411	415
Morpheus_JOGL_dlg06_mvpMx	1319	551	768
Morpheus_JOGL_dlg07_lighting	1067	664	403
Morpheus_JOGL_dlg08_surface	1234	782	452
Morpheus_JOGL_dlg09_text	1222	867	355
Morpheus_JOGL_dlg10_mouse	1553	918	635
Morpheus_JOGL_dlg11_points	1289	913	376
Morpheus_JOGL_dlg12_lines	1227	914	313
Morpheus_JOGL_dlg13_spheres	1316	1016	300
Morpheus_JOGL_dlg14_rods	1496	1002	494
Morpheus_JOGL_dlg15_image	1628	921	707

Vertex Shaders	Total Lines
Morpheus_JOGL_dlg01_triangle.vert	27
Morpheus_JOGL_dlg02_color.vert	28
Morpheus_JOGL_dlg03_animation.vert	85
Morpheus_JOGL_dlg07_lighting.vert	97
Morpheus_JOGL_dlg15_image.vert	80

Fragment Shaders	Total Lines
Morpheus_JOGL_dlg01_triangle.frag	23
Morpheus_JOGL_dlg02_color.frag	22
Morpheus_JOGL_dlg03_animation.frag	21
Morpheus_JOGL_dlg07_lighting.frag	92
Morpheus_JOGL_dlg15_image.vert.frag	88

CONSTANTS, TYPEDEFS, & VARIABLES

The dialog inheritance trifurcates at Morpheus_JOGL_dlg10_mouse, which implements mouse interaction. In the following table, red-tinted cells indicate the point-line branch (feature implemented through simple manipulation of rendering mode), the green-tinted cells indicate the sphere/rod branch (multiple geometry rendering), and the blue-tinted cells indicate the image/texture branch. The '+' indicates when an element is first introduced into the code. The '*' indicates when the variable is locally replaced via `get(...)`/`set(...)` methods in descendent dialogs. The `get(...)`/`set(...)` methods are not, themselves, shown. If an element is never overridden, it may, in some cases, be considered a constant within the program. I generally try to name constants in all caps, e.g., NDIM, but in some cases this was not done. In those cases, either a longer, mix-case name seemed more readable or the element would likely be a variable in a more general program.

	Dialog														
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
`NDIM`	+														
`noAnimatorJWA`	+														
`trianglePositionArray`	+														
`triangleScaleFactor`	+														
`JOGLColor`	+														
`WHITE`	+														
`LIGHTGRAY`	+														
`MEDIUMGRAY`	+														
`DARKGRAY`	+														
`BLACK`	+														
`DARKRED`	+														
`PURERED`	+														
`LIGHTRED`	+														
`DARKGREEN`	+														
`PUREGREEN`	+														
`LIGHTGREEN`	+														
`DARKBLUE`	+														
`PUREBLUE`	+														
`LIGHTBLUE`	+														
`DARKYELLOW`	+														
`DARKBROWN`	+														
`PUREYELLOW`	+														
`LIGHTYELLOW`	+														
`LIGHTBROWN`	+														
`DARKPURPLE`	+														

PUREPURPLE	+														
LIGHTPURPLE	+														
DisplayModeType	+														
nDisplayModes	+														
defaultDisplayMode	+														
GLCanvas	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
vShaderID	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
fShaderID	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
shaderProgramID	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
vaoIDBuffer	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
vertexIDBuffer	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
vertexPositionID	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
vertexPositionBuffer	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
shaderBaseString	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
titleString	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
bgColor	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
vertexPositionArray	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
displayMode	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
enablePointSize	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
pointSize	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*
vertexColorID		+	*	*	*	*	*	*	*	*	*	*	*	*	*
vertexColorIDBuffer		+	*	*	*	*	*	*	*	*	*	*	*	*	*
vertexColorBuffer		+	*	*	*	*	*	*	*	*	*	*	*	*	*
vertexColorArray		+	*	*	*	*	*	*	*	*	*	*	*	*	*
MXDIM			+												
MAX_FRAME_COUNT			+												
animator			+	*	*	*	*	*	*	*	*	*	*	*	*
frameCount			+	*	*	*	*	*	*	*	*	*	*	*	*
mvpMatrixID			+	*	*	*	*	*	*	*	*	*	*	*	*
animationRotationMx			+	*	*	*	*	*	*	*	*	*	*	*	*
animationThetaXDelta			+	*	*	*	*	*	*	*	*	*	*	*	*
animationThetaYDelta			+	*	*	*	*	*	*	*	*	*	*	*	*
animationThetaZDelta			+	*	*	*	*	*	*	*	*	*	*	*	*
mvpMx			+	*	*	*	*	*	*	*	*	*	*	*	*
animatorJWA			+												
aspectMx				+	*	*	*	*	*	*	*	*	*	*	*
colorMode					+	*	*	*	*	*	*	*	*	*	*
colorModeChanged					+	*	*	*	*	*	*	*	*	*	*

mMx						+	*	*	*	*	*	*	*	*	*
vMx						+	*	*	*	*	*	*	*	*	*
pMx						+	*	*	*	*	*	*	*	*	*
modelRMx						+	*	*	*	*	*	*	*	*	*
modelHMx						+	*	*	*	*	*	*	*	*	*
modelTMx						+	*	*	*	*	*	*	*	*	*
viewRMx						+	*	*	*	*	*	*	*	*	*
viewHMx						+	*	*	*	*	*	*	*	*	*
viewTMx						+	*	*	*	*	*	*	*	*	*
mouseRMx						+	*	*	*	*	*	*	*	*	*
mouseHMx						+	*	*	*	*	*	*	*	*	*
mouseTMx						+	*	*	*	*	*	*	*	*	*
vpMatrix						+	*	*	*	*	*	*	*	*	*
toggleCenterData						+	*	*	*	*	*	*	*	*	*
perspectiveProjection						+	*	*	*	*	*	*	*	*	*
positionMinMaxArray						+	*	*	*	*	*	*	*	*	*
viewLeft						+									
viewRight						+									
viewTop						+									
viewBottom						+									
viewNear						+									
viewFar						+									
viewBaseMx						+									
perspectiveProjMx						+									
orthogonalProjMx						+									
DEFAULT_ALPHA_LEVEL							+								
DEFAULT_N_ALPHA_LEVELS							+								
mvMatrixID							+	*	*	*	*	*	*	*	*
mvitMatrixID							+	*	*	*	*	*	*	*	*
mvMx							+	*	*	*	*	*	*	*	*
mvitMx							+	*	*	*	*	*	*	*	*
alphaID							+	*	*	*	*	*	*	*	*
alphaLevel							+	*	*	*	*	*	*	*	*
lightsOnID							+	*	*	*	*	*	*	*	*
lightsOn							+	*	*	*	*	*	*	*	*
vertexNormalID							+	*	*	*	*	*	*	*	*
vertexNormalIDBuffer							+	*	*	*	*	*	*	*	*
vertexNormalArray							+	*	*	*	*	*	*	*	*

vertexNormalBuffer							+	*	*	*	*	*	*	*	*
showAxes								+	*	*	*	*	*	*	*
axisVAOBuffer								+	*	*	*	*	*	*	*
axisVertexPositionIDBuffer								+	*	*	*	*	*	*	*
axisVertexPositionArray								+	*	*	*	*	*	*	*
axisVertexPositionBuffer								+	*	*	*	*	*	*	*
axisVertexColorIDBuffer								+	*	*	*	*	*	*	*
axisVertexColorArray								+	*	*	*	*	*	*	*
axisVertexColorBuffer								+	*	*	*	*	*	*	*
axisVertexNormalIDBuffer								+	*	*	*	*	*	*	*
axisVertexNormalArray								+	*	*	*	*	*	*	*
axisVertexNormalBuffer								+	*	*	*	*	*	*	*
showText									+	*	*	*	*	*	*
textShaderProg									+	*	*	*	*	*	*
vaoTextBuffer									+	*	*	*	*	*	*
textRegionUtil									+	*	*	*	*	*	*
renderState									+	*	*	*	*	*	*
regionRenderer									+	*	*	*	*	*	*
font									+						
fontSet									+						
fontFamily									+						
fontStyleBits									+						
fontSize									+						
sampleCount									+						
firstMouseDrag										+	*	*	*	*	*
oldMouseX										+	*	*	*	*	*
oldMouseY										+	*	*	*	*	*
newMouseX										+	*	*	*	*	*
newMouseY										+	*	*	*	*	*
maxPointCount											+				
pointCount											+				
currentPoint											+				
maxLineCount												+			
lineCount												+			
currentLine												+			
maxShapeCount													+	*	
currentShape													+	*	
shapeCount													+	*	

modelRMxArray													+	*	
modelHMxArray													+	*	
modelTMxArray													+	*	
modelColorArray													+	*	
nRodRadii														+	
rodRadiusIndex														+	
rodRadius														+	
rodScaleFactor														+	
nRodSections														+	
textureID															+
texture															+
originalTextureData															+
texCoordArray															+
texCoordIDBuffer															+
texCoordBuffer															+
texCoordShaderLoc															+
iSample															+
asImage															+
needsGeometryRegeneration															+
useTextureID															+
useTexture															+

METHODS

The dialog inheritance trifurcates at Morpheus_JOGL_dlg10_mouse, which implements mouse interaction. In the following table, red-tinted cells indicate the point/line branch (feature implemented through simple manipulation of rendering mode), the green-tinted cells indicate the sphere/rod branch (multiple geometry rendering), and the blue-tinted cells indicate the image/texture branch. The '+' indicates when a method is first introduced into the code. The '*' indicates when it is overridden in descendent dialogs. If no '+' is associated with a method, this indicates a system method that must be overridden, e.g. a required JOGL method or listener, within the program. Each dialog has a `main(...)` method, not shown, to allow for standalone execution.

	Dialog														
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
`IncDisplayMode(...)`	+														
`pointSizeFunction(...)`	+														
`toggleEnablePointSize(...)`	+														
`getVertexNumber(...)`	+														
`setBGColor(...)`	+														
`initOverrides(...)`	+		*			*			*	*	*		*		*
`setupDialog(...)`	+														
`generateGeometry(...)`	+				*	*	*	*			*		*	*	*
`init(...)`	*	*	*		*		*		*		*	*	*		*
`reshape(...)`	*			*					*						
`display(...)`	*	*			*						*		*		*
`dispose(...)`	*								*						*
`loadBuffers(...)`	+	*					*	*							*
`drawScene(...)`	+		*					*	*			*	*	*	
`loadShaderFile(...)`	+														
`initVertexShader(...)`	+														
`initFragmentShader(...)`	+														
`initShaderProgram(...)`	+														
`printShaderLog(...)`	+														
`printProgramLog(...)`	+														
`keyTyped(...)`	*														
`keyPressed(...)`	*	*	*		*	*	*	*	*	*				*	*
`keyReleased(...)`	*														
`randomGLBackgroundColor(...)`		+													
`incFrameCount(...)`			+												
`setRandomRotation(...)`			+												
`updateAnimationRotationMx(...)`			+												

setUniforms3D(...)			+				*								*
initMx(...)			+			*									
buildMVPMx(...)			+	*		*	*						*		
printMx(...)			+												
printMxR(...)			+												
makeCube(...)					+										*
adjustForColorMode(...)					+										*
tweakGeometry(...)						+									*
setModelToViewMx(...)						+		*	*		*		*		
reInitMx(...)						+									*
buildAndGetPMx(...)						+									
buildAndGetVMx(...)						+									
buildAndGetMMx(...)						+									
setAxisMinMax(...)						+									
showTransformedData(...)						+									
setColorArray(...)							+								
calculateNormalVectors(...)							+								
makeSphere(...)							+								
generateAxisGeometry(...)								+							
loadAxisBuffers(...)								+							
drawAxes(...)								+							*
buildAxisMVPMx(...)								+					*		
getString(...)									+	*					
drawText(...)									+						
getTextPVMMx(...)									+						
mouseMxReset(...)										+					
mouseClicked(...)										*					
mousePressed(...)										*					
mouseReleased(...)										*					
mouseEntered(...)										*					
mouseExited(...)										*					
mouseDragged(...)										*					
mouseMoved(...)										*					
mouseWheelMoved(...)										*					
mouseRotation(...)										+					
mouseTranslation(...)										+					
mouseScaling(...)										+					
windowToViewCoordinates(...)										+					

quaternionRotationMatrix(...)										+					
arcBall(...)										+					
updateGeometry(...) (first branch)											+	*			
getCoordsInSphere(...) (first branch)											+				
setModelRMxArray(...)													+		
setModelHMxArray(...)													+		
setModelTMxArray(...)													+		
setModelColorArray(...)													+		
updateGeometry(...) (second branch)													+		
getCoordsInSphere(...) (second branch)													+		
loadImage(...)															+
regenerateGeometry(...)															+
makeRectangle(...)															+
adjustImageAspect(...)															+
setupImage(...)															+

DOCUMENTATION – Morpheus_eProbe

IMPORTANT: Morpheus_JOGL is built directly atop Morpheus_eProbe. Hence, the following installation, program requirements, execution, source, JavaDoc, and Mac-specific sections from the Morpheus_eProbe User's Guide apply equally well to Morpheus_JOGL, *simply substitute "JOGL" for "eProbe" in the text and figures where appropriate.* Morpheus_JOGL-specific icons are provided.

Installation

To install the program, download the latest .zip file and unzip it. Note that on Windows there may be a distinction of simply looking inside of an archive versus actually extracting it. The program archives are named as:

morpheus_eProbe_YYYYMMDD.zip

The year of the revision is substituted for YYYY, then month for MM, and the day for DD, e.g., morpheus_eProbe_20161221.zip.

Unzipping this file will produce a directory with the same name as the archive sans ".zip". The program and associated files are found within that directory.

Program Requirements

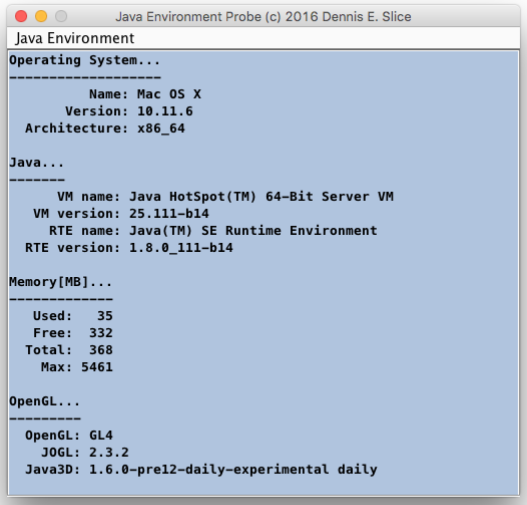
It is only assumed that you have an installed a properly configured Java environment to run this program. Mac users should see the "Note to Mac OS X Users" section below.

Execution

To run the program, navigate into the directory created above and double-click on the file (2),

morpheus_eProbe.jar

The program should execute, and you should see a screen like this containing a concise listing of the available information:



As you can see, the program shows the name and version of the detected operating system and the system architecture upon which it is running. It also shows the names and versions of the Java Virtual Machines (VM) and Runtime Environment (RTE).

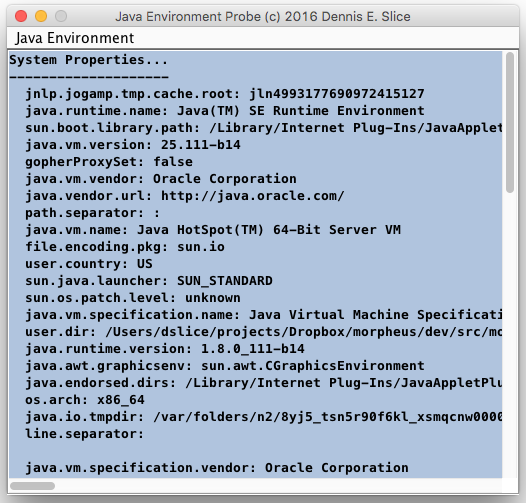
The program also reports on the memory environment of the Java VM.

As distributed, the program will report on the versions of the available OpenGL, JOGL, and Java3D environments. Access to this functionality, which the program does not require, requires the availability of JOGL libraries. Java3D is another library set that, in turn, requires JOGL. And JOGL is required to query the OpenGL capabilities. The necessary libraries are found in the ./lib subdirectory. However, the program, morpheus_eProbe.jar, can run without these libraries (just move the .jar file to another directory). If the JOGL and Java3D libraries are available on the system's Java search path, then their versions will be reported. If not, a simple 'not detected' message will be seen.

The "Java Environment" submenu allows the choice of two output formats...

"Concise" – as seen above, this is the basic information about the Java environment.

"Verbose" – is a dump to the screen of everything the probe captures about the environment. Here is a partial look at a verbose listing:



The “Exit” submenu item shuts down the program, as does the window-close button, whatever that may look like on your system.

The ./icons directory contains some images/icons you can use to pretty-up the program. The image was listed in Google search as “Labeled for reuse with modification”. I cropped and rescaled the original image and converted it to .png and .icns (Mac icon) formats. The original was found here:

http://www.accesspaymentsystems.com/wp-content/uploads/2012/05/MagnifyingGlass_20747271.jpg

I am not sure how to change the program icon on Windows or Linux environments, but a .png file is provided for this purpose. On an OS X system, you can right-click on the .jar file and select the “Get Info” menu choice. When that window opens, simply drag and drop the .icns file into the upper, left area of the info window (over the current icon).

(2) Note that the program is setup to only work properly if executed from this directory. I have not provided the .bat and .sh files, as with Morpheus et al., that would preserve the originating directory needed to find the associated library files. These files are not necessary, however, so if you move just this .jar file, the program will operate and report on the system-wide availability of Java-based OpenGL capabilities.

Source

The program was developed as a NetBeans project. I did not spend too much time investigating how to properly package project source for distribution. Instead, I just zipped the source directory. NetBeans user’s should be able to figure it out. Others can figure out how to make use of the .java files. The .form file is what NetBeans uses to manage the visual editing of the main program window.

Compiling the program, as is, will require the inclusion in the build of JOGL and Java3D files. The necessary files can be found at http://jogamp.org/jogl/www/

The main Morpheus_eProbe does not require these, but the two probes, for JOGL and Java3D, do. To divorce Morpheus_eProbe from the JOGL and Java3D libraries, the JOGL and Java3D-specific probes accept as a constructor parameter a Morpheus_eProbe object. They then do their work, and use the probe’s public functions to set the OpenGL-specific strings. To compile the program without these

libraries, simply comment out the indicated parts in the Morpheus_eProbe_startup.java file and don't include the JOGL and Java3D probe classes.

Internally, the results of the probe are returned as a single newline-delimited, formatted string that can, in turn, be sent to any variety of text display devices – edit window, console display, etc. For instance, the return string might look like, where "\t" and "\n" indicate tab and newline characters, respectively:

```
"Operating System...\n-------------------\n\t          Name: Mac OS X"
```

and when displayed would look like:

```
Operating System...
-------------------
              Name: Mac OS X
```

Future versions may include query methods to return individual string or numeric information.

JavaDoc

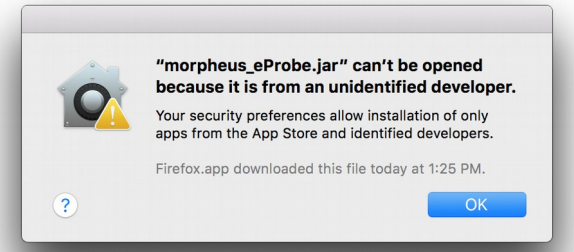
The source code has been extensively commented and documented using the JavaDoc markup system. This is then used to produce HTML code documentation. You can view these files by double-clicking on the index.html file in the javadoc directory created when the archive was unpacked. You can also access these files by choosing to open a file in your browser, navigating to the javadoc subdirectory of the Morpheus_eProbe program directory, and opening index.html.

Note to Mac OS X Users

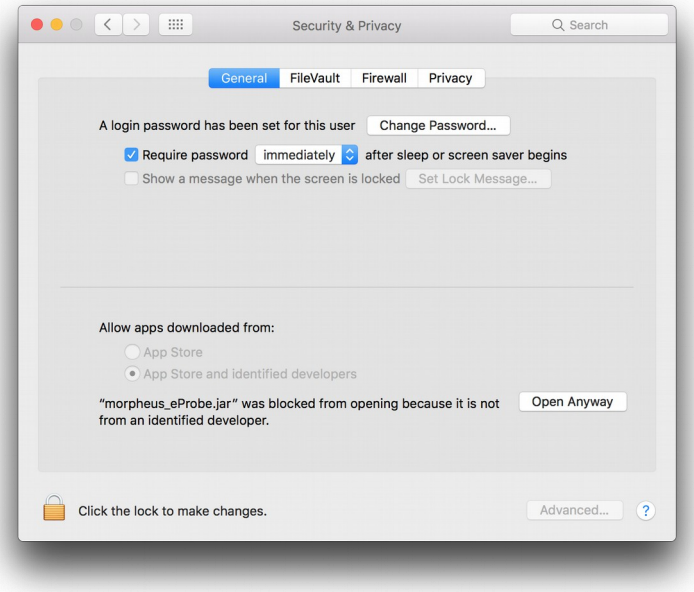
I develop on a Mac, so there may be issues of which I am not aware running the program on Linux and Windows platforms. If you find any issues on these platforms, please let me know at the address given in the "Contact Info" section below.

For the Mac, there may be a couple of issues you need to consider.

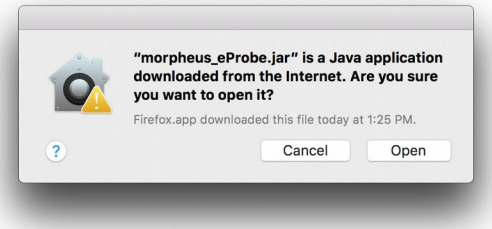
1. You should have your OS X system set up to not run just any old software you download from the web, such as morpheus_eProbe. In this case, when you try to run morpheus_eProbe, you will see a message like the following:



If you get this, open the "System Preferences" under the Apple icon in the upper left corner of the screen. Then, click on the "Security & Privacy" icon on the top row. You should see on the panel that opens a message about Morpheus_eProbe being blocked and a button that says "Open Anyway":



Clicking on this button will bring up another window confirming your request:



Click "Open", and the program should run. You will then be able to run the program from then on until you download a new version or re-unzip the old one.

2. The second possible issue you might have is that Apple used to distribute outdated versions of Java3D libraries with its operating system. I cannot find these files any longer on any of my systems

running OS X 10.11.6 or 10.12. I cannot ascertain if they are no longer distributed or if I have somehow permanently deleted them. You should be able to tell if they (or some other inappropriate versions) are there by dragging the morpheus_eProbe.jar out of its own directory (where the latest library files are stored). If you then run the program and get any message other than “not detected” for the OpenGL information, you might have to move or rename some files – and you should probably do so anyway.

These files are found in the /System/Library/Java/Extensions directory. They include any file with j3d in its name and the file, vecmath.jar. Just create another directory, say hold_j3d, and move the files into it. You may need administrator privileges to make these changes. There is an app, mac_osx_prepare, distributed with Morpheus that does this, and another, mac_osx_unprepare, that undoes it. I have not included this app here because I think it is becoming less of an issue for general users with newer operating systems.