

CS 365 Homework 3
Drawing 3-D Figures and Adjusting the Synthetic Camera
(Rev. 17 Feb. 2004)

You will write a program that draws several different user-selectable 3-D figures. It also allows the user to adjust the model-view and the projection matrices via the keyboard arrow keys, so you can zoom around to obtain different views of the figure. This program is written in OpenGL/GLUT. You can use any of its facilities except that you must draw the figures using `glBegin()` / `glEnd()`.

What it does

When the program starts, it displays a white 500×500 pixel window with three x, y, z axes and the helix figure displayed. The synthetic camera views this figure according to the default parameters described below (under the HOME button).

User controls are as follows:

- **HOME key** on the keyboard. Resets the model-view matrix so that the eye point (the center of projection in the camera) is $(-1, 0, 5)$ and the at point (where the camera is focused) is $(0, 0, 0)$.

In this program the projection matrix is fixed by the `glFrustum()` parameters: $x_{min} = -1$, $x_{max} = 1$, $y_{min} = -1$, $y_{max} = 1$, $z_{near} = 1.5$, and $z_{far} = 20$.

- **Right mouse button down.** Clicking down with the right mouse button brings up a menu of figures to pick from. The axes do not change, only the figure.
- **Left and right arrow keys.** The left and right keyboard arrow keys adjust the x coordinate of the camera's eye point, decreasing or increasing respectively. Each push of an arrow adjusts the x value by 0.5.
- **Up and down arrow keys.** These adjust the y position of the camera's eye point in the same manner as the left and right arrow keys.
- **Page up and page down keys.** These adjust the z position of the camera's eye point in the same manner as the arrow keys.

- **CTRL+Arrow keys.** Pressing ctrl together with one of the arrow or page up/down keys causes the at point to be adjusted, instead of the eye point.
- **Q key.** Quits the program.
- **H, S, M keys.** Causes the figure to switch between the helix, hemisphere, and möbius strip, respectively.

The Figures

- **Axes.** The x, y, z axes are red, green, and blue lines respectively, showing the interval $[-10, 10]$. Each axis line is a `GL_LINES` drawn with width 1.0. Please put tick marks every 1.0 unit along each axis (you can skip the origin). In my example program the tick marks are simple `GL_POINTS` drawn with width 5.0, but you can use your creativity. For user orientation, please make the tick marks along the negative axis different than the positive axis.
- **Helix.** Please draw a 5-turn helix with a radius of 1.0, starting at $(1, 1, 3)$. It extends back in the $-z$ direction so that it starts at $z = 3$ and ends at $z = -10$. (I don't care about the orientation of the spiral.) Draw this helix using `LINE_STRIP`, using a line width of 3.0. You can draw a line segment every 10° of arc.
- **Hemisphere.** Please draw a hemispherical surface centered at the origin with radius $r = 2$. Draw this with either `TRIANGLE_STRIP` or `QUAD_STRIP`. Use `glPolygonMode()` to set the coloring mode for the outside and inside surfaces. The outside surface of each polygon should be drawn in `GL_FILL` mode, i.e. colored. Each polygon (triangle or quadrilateral) should be colored differently. Compute a color for each polygon that is a function of ϕ and θ , so it varies smoothly along the surface of the hemisphere. (If you make your hemisphere only a single color, it will be hard to see the shape.) The interior surface of the hemisphere should be drawn in `GL_LINE` mode, resulting in the polygons being outlined but not colored.

Draw your hemisphere with the latitude ϕ on the interval $[-70^\circ, +70^\circ]$, and the longitude θ on the interval $[90^\circ, 270^\circ]$. This will result in a hemisphere with the end-caps removed.
- **Möbius Strip.** This is the fun extra credit part of the assignment. Draw it with radius $r = 2$ centered at the origin with a quad strip. Smoothly vary the color of the polygons so as to make the figure easier to see. You can also vary between fill and outline modes to enhance the effect.

Program Organization

- **Main program.** The main program contains calls to the following functions:
 - Call `glutInit()`
 - Call `glutInitDisplayMode()`, `glutInitWindowSize()`, and `glutCreateWindow()` to create a double-buffered 500×500 window in RGB mode. Put your name in the create window call, so that the window has your name on top.
 - Call `glutDisplayFunc()` and `glutReshapeFunc()` to register display and reshape callbacks respectively.
 - Call `glutSpecialFunc()` and `glutKeyboardFunc()` to register keyboard event callbacks. The special key events are the arrows, page up/down, and home keys.
 - Menu initialization is accomplished by calling `glutCreateMenu()` to create a new empty menu and register a menu handler function. Then call `glutAddMenuEntry` several times to add each figure selection to the menu. Finally call `glutAttachMenu()` to cause the menu to activate on the right mouse button.
 - Call `glClearColor()` to set the default window color to white.
 - At the end of the main program call `glutMainLoop()`. After this you have no more work in the main program, but your recall handlers will do the work.
- **Reshape handler.** The reshape handler is responsible for changing the viewport and re-computing the projection matrix. Since the reshape handler is called once before the window is first displayed, there is no need to initialize the projection matrix separately from the main program.

Basically the reshape handler should look like this:

```
void reshape(int w, int h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    /* Now set the projection
       matrix using glLoadIdentity()
       followed by glFrustum() or
       gluPerspective() */
    glMatrixMode(GL_MODELVIEW);
    return;
}
```

- **Display handler.** The display handler recomputes the model-view matrix using `glLoadIdentity()` followed by `gluLookAt()`.

Keep in mind that many user actions change either the eye or at coordinates, so you cannot assume this is constant. It also is responsible for clearing the canvas using `glClear()`, then drawing the selected figure (helix, hemisphere, or möbius strip) and the axes. It ends with `glutSwapBuffers()`.

- **Figure Drawing Routines.** These routines should push/pop the current drawing state, as illustrated here:

```
void draw_helix () { /* for example ... */
    ... declarations ...
    glPushAttrib(GL_CURRENT_BIT | GL_LINE_BIT);
    glLineWidth(3.0); /* Line width */
    glColor3f(0.0, 0.0, 0.0); /* drawing color=black */
    ... do it ...
    glPopAttrib();
    return;
}
```

The general form for drawing polygons, points, or lines is to call `glBegin()` with the drawing mode, then draw `glVertex` one or several (or even many many) times, then call `glEnd()`. The drawing modes we have discussed in class are:

- `GL_POINTS` draws one point for each `glVertex`. `glPointSize()` sets the point size in pixels.
- `GL_LINES` draws one line segment for each pair of `glVertex` points. Use `glLineWidth()` to set the width in pixels.
- `GL_LINE_STRIP` draws one line segment for the first two vertices, then one more line segment for each new vertex. If you plot vertices $p_1, p_2, p_3 \dots$, the first line segment is p_1, p_2 , the next is p_2, p_3 , and the i -th vertex creates a line segment p_{i-1}, p_i .
- `GL_TRIANGLE_STRIP` draws one triangle for the first three vertices, then one more triangle for each new vertex. If you plot vertices p_1, p_2, p_3, \dots , the first triangle is p_1, p_2, p_3 , the next is at p_2, p_3, p_4 , and the i -th vertex creates a triangle at p_{i-2}, p_{i-1}, p_i .
- `GL_QUAD_STRIP` draws one quadrilateral for the first four vertices, then one more quadrilateral for each two new vertices drawn. If you

plot vertices $p_1, p_2, p_3, p_4, p_5, p_6 \dots$, the first is p_1, p_2, p_3, p_4 , the next is at p_3, p_4, p_5, p_6 , and the i -th vertex (where i is even) creates a quad at $p_{i-3}, p_{i-2}, p_{i-1}, p_i$.

- **Keyboard and menu handlers.** The current eye and at positions, as well as which figure is being drawn, are kept in global variables. These user interaction routines do not need to draw anything or change any matrices: they merely change the global variables so that the picture will be changed the next time it is displayed. The pattern is to change the variables, then call `glutPostRedisplay()` before exiting so that the picture is redrawn in due course. The special key handler takes care of the arrow, page up/down, and home keys. The special key handler can find out whether the control-key is being pressed by executing:

```
glutGetModifiers() & GLUT_ACTIVE_CTRL
```

The regular key handler takes care of the Q, H, S, and M keys. The menu handler changes which figure is being drawn.

Note About Matrices in OpenGL

You are using two OpenGL transformation matrices in this program:

- The projection matrix, which defines the view volume (a frustum). This matrix can be adjusted by calling `glFrustum()` or `gluPerspective()`.
- The model-view matrix, which positions the picture relative to the camera (the eye and at points). This matrix can be adjusted by calling `gluLookAt()`.

Modifying a matrix in OpenGL requires that you know the *current matrix*. Whenever you issue a matrix modification command it is this current matrix that is modified. You change which matrix is the current matrix by calling `glMatrixMode()`. If you accidentally call `glFrustum()` while the model-view matrix is the current one, you will be adjusting the wrong matrix even though that is clearly not your intent.

The default current matrix is the model-view matrix. One way to organize your OpenGL program is to keep this the default, and temporarily change the current matrix in the resize handler while you are changing the projection matrix. That is what I have illustrated in the above sample resize-handler code.

Another unobvious feature of OpenGL is that `glFrustum` and `gluLookAt` do not *set* the current matrix, they *adjust* the current matrix by multiplying with the desired transformation. For example, if you call `gluLookAt` several times in a row the camera will move farther and farther from the object. So to set a particular camera angle (model-view) or view volume (projection) matrix, you first call `glLoadIdentity()` then immediately adjust the matrix. Again, the sample code for the reshape handler illustrates this.

Submission

Turn in your program by posting the source code to Coursevu/Blackboard, as before. The program must work on the Suns.

You are encouraged to make your program work on a Linux machine also, as this seems to smoke out more bugs. Details on how to use the Linux machines in front of my office will be provided in class.