Introduction to Computer Graphics

Assignment Two

September 19, 2002

Due: Tuesday, October 8, 2002, 11:59 PM

Credit: 15 points (Relative, and roughly absolute weighting)

This assignment may be done in pairs if you prefer.

Only 2D OpenGL *rendering* and input primitives can be used. I would like you to implement the projection, rotations, and back plane culling yourselves. Note that you will need to think about how to do rotations about an arbitrary axis. If this is holding too many people up, we can go into this in class.

(The next few paragraphs are the same as for assignment one.)

Your program should read command lines from standard input. Each line is to be parsed and processed as described below. Depending on these lines, the program may create an interactive graphics window. Once reaching end of file, the program creates an interactive graphics window if it has not already done so, and continues as an interactive program. Note that only one graphics window is created.

When the user quits the program (by typing "q" in the graphics window), the program then writes out to standard output the commands that would create the scene on display at that time.

Thus you should be able to start up the program with the file you just wrote as a new input file, and be exactly where you were.

A typical invocation of the program then would look like:

```
    my_prog < in > out
```

The format of the command lines will always be a single word followed by one or more integers separated by white space. If you like, you can assume that the white space is a single blank. You will need to write code which can count the number of numbers and retrieve them, regardless of how many of them there are. It is recommended that some minimal error checking is done, but the action on error can simply be to print a message and exit. This part of the program is to be regarded as infrastructure, and thus we will simplify things by making the user keep track of what the numbers mean based on their position. Try not to spend too much time on parsing. For this part you are welcome to make use of an external library routine, or open source code (with attribution).

(Now the meat)

You are to implement a perspective view of a box. Each face of the box should be a different color. By default the box has corners (200,200,200) and (500,500,500). The default box may be over-ridden by a line in the input file of the form:

```
box <x1> <y1> <z1> <x2> <y2> <z2>
```

where (x1,y1,z1) is one corner, and (x2, y2, z2) is a diagonally opposite corner.

Future homework may request additional boxes, but for this assignment additional “box” commands over-ride the previous box. Future homework may also allow additional numbers for color, but for now simply chose 6 different colors other than black for the faces.

The world is infinitely big (depending on which cosmological theory you subscribe to!), so the box corners can be any integer in (INT_MIN, INT_MAX).

There may be additional lines in the input file following the specification of an object (for now we only have boxes). The ones that are to be implemented for this assignment are:

```
xrot <theta>
yrot <theta>
zrot <theta>
```

<theta> gives the angle in degrees.

In future it should be an error to rotate if no objects have been specified, since the rotations apply to the object on the previous object line; however, since we have a default box, a reasonable (but not required) action would be to rotate the default box. Regardless, you need to be prepared for an arbitrary collection of rotation commands.

Be sure to implement back plane culling. The faces of the box which are not visible should not be drawn.

The camera is located at the default position of (1000, 1000, 1000). The camera position may be over-ridden with the following command:

```
camera <x> <y> <z>
```

For this assignment, the camera is always pointed to the origin, and VUP is parallel to the Y-axis. Choose a viewport and a projection center to give reasonable results. (You may wish to think about the problem of giving the user more control of the camera.)

Back and front clipping is not required, and OpenGL can be left to take care of the viewport clipping in 2D. (But be sure you understand why we generally do not do it this way for complex 3D worlds!).

Interactive input:

The arrow keys are to be used to move the camera in the directions determined by the screen coordinates (i.e., left arrow moves the camera in the direction of decreasing **u**). The d key is used to decrease the distance of the camera to the origin, and the D key is used to move it further away. Note that as the camera moves, the meaning of the directions in world coordinates changes. You may assume that the user will not move the camera to the Y-axis (why is that not a good idea?), or better yet, block them from doing so.

The x key is to be used to shrink the box in the x direction (world coordinates), and X key is to be used to stretch it. Ditto for y and Y and z and Z.

Dragging the mouse with the left button down should translate the box in the direction of the mouse drag. Note that the drag defines a line in the camera plane. The b key is used to translate the box away from the camera, and the f key is used to translate the box towards the camera.

Dragging the mouse with the right button down should rotate the box around a line perpendicular to the drag. Rotation direction should be the natural one. The r key is used to rotate the box clockwise around the viewing direction. R is used for counter-clockwise.

Try to calibrate key press actions and drags to affect a natural amount of motion. This will require rotations and translations proportional in magnitude to the stroke length.

When the user enters “q” in the graphics window, the program first writes the appropriate commands to standard output, and then exits. If you start up your program with those commands as input, the screen should look exactly as it did on exit.

Extra credit

If you would like to improve on the program, be sure to explain what you did in the README file, and it will be considered for modest extra credit.

Deliverables

You must electronically submit a README containing any relevant information, but at a minimum, your name; an executable (called a2); and a src directory containing source files and a Makefile which can be used to build the executable.

The turnin name is cs433_hw2.