Introduction to Computer Graphics

Assignment Two

September 23, 2002

Due: Tuesday, October 21, 2002, 11:59 PM

Credit (U-grad): Approximately 15 points (Relative, and roughly absolute weighting)
Credit (Grad): Approximately 12 points (Relative, and roughly absolute weighting)

Note to Grad Students and Honor's students: You will be required to do some project work for this course. While I will attempt to structure due dates so that there is time near the end for finishing projects, you will need to start thinking about your projects soon, and take extra care not to get behind on the regular assignments. Likely most projects will be best done in groups.

This assignment may be done in pairs if you prefer.

Hint. Read the assignment *carefully*, and think it through before starting to code.

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.

(The next few paragraphs are similar to ones in 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 parallelepiped. Each face should be a different color (not black). A parallelepiped can be conveniently described by sequence of 4 points, each successive difference defining a new direction. The default parallelepiped to draw is the cube (200,200,200) , (500, 200, 200), (500, 500, 200), (500,500,500). The default parallelepiped may be over-ridden by a line in the input file of the form:

```
pp <x1> <y1> <z1> <x2> <y2> <z2>  <x3> <y3> <z3> <x4> <y4> <z4>
```

Future homework may request additional parallelepipeds, but for this assignment additional “pp” commands **over-ride** the previous one.

HINT: In order that the input/output conditions are met, it is important to consistently color parallelepipeds based on the 4 points, and to keep track of what happens to that sequence of points. In fact, you may find the easiest strategy is to use the 4 points as parallelepiped generators, and never work with planes until you need to.

Parallelepiped corners can be any integer in (INT_MIN, INT_MAX).

Be sure to implement back plane culling. The faces of the parallelepiped which are not visible should not be drawn. **You will need to have a method for deciding which way the normals point.** Having done that, planes are visible if the camera is on the same side as the outside of the plane (back face culling).

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 view port 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**). Note: These actions do **NOT** change the distance the camera is from the origin. It is like the camera is on a string tied to the origin. 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 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.

The x key is to be used to shrink the parallelepiped 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. Scaling should be relative to the center of gravity of the parallelepiped.

Dragging the mouse with the left button down should translate the parallelepiped 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 parallelepiped away from the camera, and the f key is used to translate the parallelepiped towards the camera.

Dragging the mouse with the right button down should rotate the parallelepiped around a line perpendicular to the drag. Rotation direction should be the natural one. The r key is used to rotate the parallelepiped 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 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 program must compile and run on one of the graphics machines (gr01, ... , gr10). Put in the README file the machine which you have verified this on.

Note that the graphics machines can be booted into Windows by people in the lab, so that it is possible that if you are working remotely that you will need to try more than one. We encourage students to use the higher numbered machines for Windows (7 through 10), but this cannot be enforced.

The turnin name is cs433_hw2.

HINT for exams. It may be helpful to understand why you can write out the state of the world with one command, despite all the transforms.