Introduction to Computer Graphics

Assignment One

August 26, 2004

Due: Tuesday, September 14, 2004, Midnight

Credit (U-grad): Approximately 9 points (Relative, and roughly absolute weighting)
Credit (Grad): Approximately 7 points (Relative, and roughly absolute weighting)

This assignment has three objectives. First, to use the early part of the term, where little
graphics content has been covered, to build some infrastructure which will be common to
several other assignments. Second, to use the early part of the term to become very
familiar with OpenGL and GLUT. Third, learn about some of the issues in constructing
graphics primitives.

Any OpenGL and GLUT routines may be used for this assignment.

This assignment must be done individually (that you can share parsing code if you
acknowledge where you got it from). There will be some scope for group work in future
assignments.

The program structure described below will be common to most other assignments. It is
to make testing and debugging easier for you, and marking easier for your grader.

(The next few paragraphs describe the behavior in the general case---for a particular
assignment, some words such as “may” could be more properly replaced by “will”).

Your program should begin by reading input lines from a file which is set to standard
input (using the unix shell redirection symbol “<”). Each line will contain a command to
the program and should be parsed and processed as described below. The program may
create an interactive graphics window as a consequence of processing one of them. 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. Only one graphics window is
created.

If you have no input file, then standard input will be normally be attached to the terminal.
This can result in the program simply waiting for the user to type something. Possibly
the easiest way to handle this possibility is to test stdin to see if it is a terminal using the
issatty() system call, and then have the program behave as though EOF was entered in this case.

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
```

Once the program is running as an interactive program, the input is through mouse clicks and simple one character directives which are typed will the graphics window is in the foreground. The command line input format is only for the initial read and final write.

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). In fact, for this part, you are welcome to share code with others (just make a note in your code, i.e., “John, Mary, and I collaborated on the parser, and only the parser”).

(Now the meat---specific to this assignment)

You are to implement a 21 by 21 grid consisting of squares 30 pixels wide. The pixel width (e.g., 30) should be easy to change. It may look better being larger or smaller. You can choose a different size if you like. Each square is surrounded by a border which is one pixel wide. The border is counted as part of the 30 pixels. This means that there is a grid pattern with grid lines which are 2 pixels wide, because the borders of neighboring squares team up. Each square is centered on an integral (x,y) pair. The origin is in the middle of the grid. If you move left/right/up/down from the origin, then you will encounter a border pixel after 14 pixels, the non-border part of the next square after 16 pixels, and you will be at another integral (x,y) location at 30 pixels. These numbers will be different if you choose a different square size.

The grid establishes a coordinate system in the block pixel world. The location of the origin at the middle does not affect the interactive part, but it will affect the meaning of the numbers that are read in at the start, and written out at the end. We will orient the axis as done in math where x is horizontal and increases from left to right, and y is vertical and increases from bottom to top. Thus the lower left hand block is (-10,10) and the upper right hand block is (10,10).
Note that this is a common point of confusion! It is quite common for graphics packages and image display programs to use a different coordinate system. In fact, in OpenGL Y increases top to bottom. It is also quite common that the first coordinate is a row going down, and the second is a column going to the right, so X and Y are reversed, as well as the first coordinate increasing downwards, not upwards.

Dealing with this problem in particular, and mapping between coordinate systems in general is a standard activity that graphics practitioners need to get used to.

Important point: If you want a perfect looking grid, you will have to be aware of the off-by-one problem explained on the web page. We have not exactly characterized the problem, but basically, you will likely find that a window of dimension 100 is actually 101 pixels wide, due to counting starting at 0.

Your program is to implement two drawing primitives in this block-pixel world. Monochrome (bright grey) lines ((R,G,B)==(200,200,200)) and filled polygons.

Polygons should be filled by interior pixels only, as defined by the conventions discussed in class. Your program should handle all polygon types.

You can assume that the command line input will not be outside the grid.

There will be both command line and interactive input.

The command lines to be supported are:

```
line [ x1 ] [ y1 ] [ x2 ] [ y2 ]
poly [ x1 ] [ y1 ] [ x2 ] [ y2 ] [ x3 ] [ x4 ] ( [x] [y])*
```

In the case of “poly” the polygon is completed by drawing from the last point to the first.

Each polygon should be filled with a different color, which can be neither monochrome nor red which are used for other purposes. You can recycle colors after 5 or so.

For interactive input, implement a menu activated by the right button which puts the program into either line mode or polygon mode. The initial state is line mode. In line mode, lines are added by simply selecting 2 points in succession with clicks of the left mouse button. When the first point is selected, the grid square is set to red. On the second click, the monochrome line is drawn. The initial red grid square should no longer be red.

In polygon mode, polygons are added edge by edge. For user feedback we will draw the lines. Thus you will want to make use of your line drawing routine. The first click creates a red square at the appropriate box, just as in the case for lines. Each new click causes a line to be drawn from the previous point to the current point. Furthermore, the new most recently clicked point is marked as red (this part is slightly different than how we do lines). When the user clicks the very first point selected for a second time, the operation
is complete, all lines are erased, and the polygon is filled with the next polygon color. Give a little thought as to what you would like to happen if the user clicks the initial point on the second or third click.

If the user enters “c” in the graphics window, all objects are forgotten about, and the grid is cleared of objects.

When the user enters “q” in the graphics window, the program first writes the appropriate “line” and “poly” commands to standard output, and then exits. Note that the order of the commands makes a difference. 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. An example would be infinite undo while drawing polygons, and for objects added. Another example would be more clever feedback while drawing objects. A third example would be some implementation of anti-aliasing.

Hints

In the above description, the word “erase” need not be taken literally. You may find it easier to simply remove the items from a some data structure which is redrawn when appropriate.

There is a fair bit of programming here---start soon! As always, care given to the design will help. Also, note that the style of input and output will be in common with some of the other assignments.

Deliverables

You must electronically submit a README containing any relevant information, but at a minimum, your names; an executable (called a1); 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 (gr07 through gr10), but this cannot be enforced.

The turnin name is cs433_hw1.