# Introduction to Computer Graphics

# Assignment One

August 27, 2002 (last revision, August 29)

Due: Tuesday, September 17, 2002, Midnight

Credit: 12 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 should be done individually.

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

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 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. 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.

(Additional clarification of the grid will be given in class).

Your program is to implement two drawing primitives in this block-pixel world. Monochrome lines ((R,G,B)==k(100,100,100)) and filled polygons. Lines should be anti-aliased according to the cone approximation of Gaussian sampling discussed in class (and in the text section 3.14.3). As also discussed in class, you may approximate the integral of the cone weighting function applied to a line by the area of a triangle. (If you use some other way to compute the integral, make a note of it in your README file). The radius and the height of the cone are both one “block” pixel. Lines should be drawn so that their brightness is normalized to (200, 200, 200) times the output of the weighting filter. So if the line goes right through the sample point, the brightness is (200, 200, 200), otherwise it is less bright.

Note: A non-anti-aliased, or differently anti-aliased line is better than no line at all. If you are unable to achieve the cone filter result, then drawing the line some other way will earn partial credit (note the method used in the README file).

Polygons should be filled by interior pixels only, as defined by the conventions discussed in class, and consistent with section 3.4 of the text. 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> <y3> (<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 firsts 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.

Hints

Modest inefficiencies are OK, especially in the anti-aliasing code, but you should not compute the brightness of pixels which are obviously too far from the line to be anything other than black.

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.

Deliverables

You must electronically submit a README containing any relevant information, but at a minimum, your name, and the platform (if not Linux), an executable (called a1); a src directory containing source files and a Makefile which can be used to build the executable.

The turnin name is cs433_hw1.