B581 — Computer Graphics, Fall 2008 (A. Hanson)

Problem Set 2

Due Date, 11:59pm, Tuesday, October 7, 2008.

This coding problem set has 40 points, about 4% of total grade.

Late submission only by prior arrangement, and maximum score is 50% on late submissions.

Remember: The homework will be graded on a RedHat Linux system. Put all your files into one `tar` file, do not upload files individually. Do NOT send executables; they will be returned ungraded for resubmission. GRADING is based on this written problem statement, not on any demonstrations that may be shown.

We will use `(GLUT_SINGLE | GLUT_RGBA)` mode, with two-dimensional coordinate systems, and the problem part will be specified by typing the "1", "2", "3", or "4" key on the keyboard to change the program state. You may experiment with `(GLUT_DOUBLE)` mode if you like, but the no-erase property of problem [2.1] is difficult to handle, and you probably want to stick to SINGLE for your submitted exercise.

Caution: One often would like to use variable names like `(x0, y0, x1, y1)`; this will get you in trouble if you are using ANY `math.h` routines because `extern double y0, y1 (double);` are Bessel functions defined in `math.h`. Conversely, on many systems (but apparently not the default Linux) if you use any math functions, even square-root, you will get into big trouble if you omit mentioning `math.h`. Include this as good programming practice to avoid mysterious cross-system errors (though note that some inconsistencies exist in some Windows versions, which may require some additional constants like M_PI = π to be hand-defined).

Coordinate System! The `reshape(w,h)` function supplied in the templates has been configured for *right-handed* Cartesian coordinates. One line of transform code in the mouse handlers turns the "upside down" GLUT mouse/screen coordinates right side up. Therefore, you can draw pixels directly in the transformed mouse coordinates in all the following exercises. Look for the variables `main_height`, `main_width`; you may need them!

2.1 (10+5 = 15 points) **“1” key (default startup state). Drawing XOR rubber band lines and logical operations.**

Setup: On entry, draw a *static scene* (a scene that is drawn only the first time or after a Reset by hitting the “r” key). A large colored rectangle is sufficient, though creativity is encouraged. Enable logical operations and start by default in XOR mode.

10 points. Left-Mouse: XOR rubber band task When the left mouse button is pressed and dragged, implement a “Rubber Band Line” with logical operations enabled, using XOR, that draws nondestructively over the static scene, i.e., you may not redraw the scene, only erase and redraw the current line, leaving previous lines on the screen. That is, start the line by pressing the left button; keep the left button down and drag the mouse, erasing the old line using the *same* operation (XOR in the default state) and drawing a new line in the foreground color using XOR each time the mouse moves; *start a new line* when the mouse is released and then depressed at another location. Lines that cross each other may not preserve the color, but that is OK. (*Hint:* you may want to be clever about the value of the source color and the color mask you use; this is your choice, however.)

You may not erase the screen after the very first time your Display routine is called unless the Reset process is invoked with the “r” key. Points will be deducted if you redraw the scene unnecessarily.

5 points. Spacebar = change logical operation. When the spacebar is depressed, *cycle through the logical operations* used to draw and erase the line. Please at least print on the console (printf) the name of the current operation; you may wish to display the name in the graphics window (see Appendix). *Return to the default XOR mode* when the reset, “r” key is pressed. This will give a wide variety of behaviors — can you explain them? Does anything else besides XOR work for non-destructive draw/erase effects?

Notes: (1) It has been observed on some systems that erasing in XOR must occur in the same vertex order as the original drawing; this does not seem to happen on our Linux systems. (2) On some systems, one finds different behaviors when updating the data for drawn objects in the MouseMotion callback vs the Display callback. In principle, robust programming practice requires that variable updating for erased objects as well as all drawing should occur only in the Display routine itself to be system independent.

2.2 (5 points) **"2" key. RGB Additive Color with Masking.**

Set-up. Establish a Black background, and on top of it you want to draw three boxes using emission-appropriate colors, one box red, one green, one blue. For this exercise, you can redraw the whole scene on each redisplay if you like (it will flicker less if you find a way not to redraw).

Task (Any-Mouse-Drag): When the left mouse button is pressed and dragged, move the left hand box (Red around the screen. Middle mouse, the middle box (Green). Right mouse, the right box (Blue).

Requirement. When the boxes overlap, you should see the additive color model combinations. There is an easy way to do this with masks; you probably do not need logical operations unless you choose not to erase the screen with each mouse action.

2.3 (5 points) **"3" key. Subtractive Color with LogicOps.**

Set-up. Establish a White background, and on top of it you want to draw three boxes using printer-appropriate colors, one box cyan, one magenta, one yellow. For this exercise, you can redraw the whole scene on each redisplay if you like (it will flicker less if you find a way not to redraw).

Task (Any-Mouse-Drag): When the left mouse button is pressed and dragged, move the left hand box (Cyan) around the screen. Middle mouse, the middle box (magenta). Right mouse, the right box (yellow).

Requirement. When the boxes overlap, you should see the subtractive color model combinations. There is an easy way to do this with RGB masks *combined* with logical operations.

2.4 (15 points) **"4" key. Line Scan Conversion with Error visualization.**

Set-up. Adapt your rubber-band line code from the first exercise to get a starting and ending point on the screen. **Note:** *Unlike part [2.1] you may clear and redraw the entire screen at each call to the Display callback in this application.*

Implement Line Scan Conversion. Pass the two points obtained from the mouse-down point and the mouse-dragged point to your *personal* implementation of the line scan conversion algorithm (see class notes or any text for algorithm details). You are *required* only to implement positive slope less than one, but you are *encouraged* to do a full 360-degree implementation. Plot the points using GL_POINTS; obviously, you are not permitted to use the hardware line-drawing mode GL_LINES in this exercise.

Implement a plot of the per-pixel error measure. Using a method of your choice (across the middle of your viewport, at the bottom, in a separate viewport, etc. – as long as it is clearly visible), plot the values of the error-measure being used at each pixel to decide whether the y-value stays the same or is incremented at the next x-value (swapped for large slope if you are implementing 360 degrees). Update this plot as well as the line segment pixel drawing each time the mouse moves while the button is depressed. Leave your drawing on the screen when the button goes up.

2.5 *Personal homework, nothing to submit:*

In addition to implementing the line scan-conversion algorithm, read carefully the description of the Midpoint or Bresenham scan conversion algorithm for lines in the lecture notes or in any of the available textbooks. Be prepared on an exam to write down the basic principles of the algorithm in your own words.

Appendix: writing labels

To write text labels, in case you want to label logical operations on the screen, for example, use BitmapCharacter. Using GL_XOR as the temporary logical operation for this might be advisable. An example of some code that can be used for labeling is given below. One interesting thing about characters in C is that "char" defines individual 8-bit characters, e.g., ' ' is the space character 0x40 (use "man ascii" to see the rest), while "char *" defines a pointer to a string, or array of char's like `"graphics"` which is actually nine characters long, with a null character (0x00) at the end, and that is how the code below detects the end of the string.

```
int i;
char ch;
char * chptr;
static char* ops[2] = {"HI", "THERE"};
glRasterPos2i(xpos,ypos);
glutBitmapCharacter(GLUT_BITMAP_9_BY_15, 'x');
glutBitmapCharacter(GLUT_BITMAP_9_BY_15,' ');
i = 0; chptr = ops[something]; ch = chptr[i];
while ((ch != 0x00)&&(i<12))
  { printf("Debug: char =[%c] [%d]",ch,ch);
    glutBitmapCharacter(GLUT_BITMAP_9_BY_15,ch);
    i = i+1;
    ch= chptr[i];
  }
```

Instructions: Submit separate working Linux files with a Makefile.linux. Include a plain text "README.txt" if there is anything special for the grader to note. Supply ONLY complete source code *(no executables)* in a single tar file as follows:

Use "tar cvf ps$< N >< login >$.tar *" to create a tar file.

$< N >$ – the number of the assignment

$< login >$ – your user name

For example: A student whose username is xxyy works on ps2. The submitted file should be called `ps2xxyy.tar`.