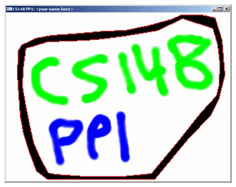
Dan Morris
CS148

Handout #04
Summer 2005

Project 1 – MiniPaint

Due: Tuesday, 7/5/05, 11:59 pm



Project Goals

In this project, we will explore some of the algorithms for scan-conversion covered in class, and introduce OpenGL and GLUT program structure. You will also build a visually dazzling paint program.

Supplies

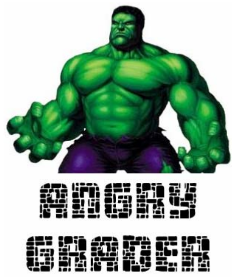
You will use OpenGL and C++ for this project. All projects must compile under Linux when you hand in (specifically, they must compile on the *myth*, *firebird* or *raptor* machines in Sweet Hall). You are welcome to develop on a PC or Mac or digital toaster, but if you do develop on an alternate platform, we strongly advise that you test on the Sweet Hall machines before you hand in your assignments. Those will be the machines we will grade on, and non-compiling programs make graders very sad. GLUT and OpenGL are already in place on the cluster machines.

If you're working on your Windows machine at home, you will probably need to download GLUT from http://www.xmission.com/~nate/glut.html . The stencil zipfile will include a project file for Visual C++ 6.0, which will also work with Visual C++ .net. We put those there to help you, but remember that your program must compile under Linux when you hand in.

A sample solution for Linux can be found at /usr/class/cs148/demos/pp1demo.linux . We will also place a Windows version of the demo in that directory and on the website.

To get you started on the first assignment, we will provide a template on the web site from which you should build your solution. This template includes OpenGL and GLUT setup code and the pixel-drawing routine. Follow the directions in this file to let you know which code you can modify and which code you shouldn't. We do highly encourage you to look at this code and play with it to figure out how OpenGL and GLUT work, but make sure you don't change any of the support code for your submission (broken support code also makes graders very sad).

This is an individual project. You are encouraged to discuss high-level issues with other students, but your code must be your code. If you talk with other student about high-level issues, please document this at the top of your .cpp file. Code piracy makes graders angry. You don't want to make your grader angry…



The Assignment: A Scan-Converting Editor

The goal of this assignment is to finish development of a two-dimensional editor which does its own scan conversion of simple primitives. The user can interact with the editor using the keyboard to specify the primitive and the mouse to actually draw the primitive. The editor allows the user to draw points, lines, circles, polylines (a set of connected line segments), and filled rectangles. The editor also offers a flood-fill operation and a slick-looking airbrush for the true artistes in CS148. The user can change the current paint color from the keyboard.

The following primitive mode keys are supported; pressing one of these will change the value of the `nMode` variable:

‘P’	**Point mode**	Draw a point every time the user clicks the mouse.
‘L’	**Line mode**	Draw a line between the first mouse click and the second mouse click.
‘C’	**Circle mode**	Draw a circle by specifying the center of the circle with the first mouse click and a point on the outside of the circle with the second mouse click.
‘Y’	**Polyline mode**	Draw a series of lines connecting every mouse click to the mouse click before it. The first mouse click does not draw any lines.
‘R’	**Rectangle mode**	Draw a filled rectangle computer with opposite corners specified by two mouse clicks.
‘F’	**Flood fill mode**	Using a mouse click as a seed point, flood fill a region using a 4-connected technique.
‘A’	**Airbrush mode**	Fill a circular region around the mouse click that smoothly blends with the background.

Other control keys:

‘X’	**Clear window**	Draw a point every time the user clicks the mouse.
‘B’	**Toggle pixel size**	To aid your debugging we have created a 'large pixels' mode to help you to see your scan converted primitives without hurting your eyes. Note that throughout this assignment, the ‘pixels’ that you write to don’t correspond to physical pixels on your monitor. To make it easier to debug, we’ve given you a “virtual canvas” to draw on that has larger pixels. All of this is handled transparently, so you don’t need to worry about the physical pixels on your monitor.

‘D’	**Toggle delay**	As another debugging aid, delay mode will cause a short delay after every pixel is drawn to allow you to see what order the pixels are drawn when you scan convert a primitive. Keep in mind that there is no 'right' order for pixels to be drawn--for this project we will only grade you on which pixels are set, not the order that that are set in.
‘0’-‘9’	**Change color**	There are a few default colors defined in pp1.cpp; the first five are white, black, red, green, and blue respectively. Pressing one of these keys will change the value of the `currentDrawColor` variable.
‘Q’	**Quit program**	

Color

Colors in MiniPaint are represented using the `rgb` class, which specifies the red, green, and blue components of a color, all of which vary between 0.0 and 1.0. (0,0,0) is black and (1,1,1) is white. The routines you’ll use to access your pixels use this class. As a quick example, if I have a color called `myColor`, and I wanted to access the red component, I could do:

```
float red_component = myColor.r;
```

Input Files

You can also pass a file to the program by specifying one on the command line, e.g:

```
pp1.exe some_pretty_picture.minipaint
```

This file will be read by a parser (that’s already written for you) and keys and mouse movements will be handed to your program just as if they had come from the user. This can be very helpful for debugging, since it’s painful to try to reproduce a specific series of mouseclicks over and over when you have a bug.

You can look at the sample files that come with the stencil code to see the format of these files. We will give (a whopping) one point of extra credit to the student who turns in the coolest .minipaint file, and we’ll show it in class and give you some delicious candy for your effort. That’s one point out of a hundred, so don’t spend too much time on this.

What You Need To Implement

To begin, copy the skeleton code to your directory. The skeleton code can be found in /usr/class/cs148/asgn/pp1/, or on the syllabus page on the web. On Linux you can do:

```
cp -r /usr/class/cs148/asgn/pp1 <your directory>
```

For Windows, you can copy the files to your own directory or download the .zip file of skeleton code from the course web page.

For circles and lines, we have already done some of the work. Specifically, we set up GLUT and OpenGL, translate mouse values to 'pixel' locations, and do all OpenGL interaction. All you need to do is complete the following functions to draw lines and circles respectively:

```
void drawLine(int startx, int starty, int endx, int endy);
void drawCircle(int x, int y, int radius);
```

In order to do this task, we give you a pixel drawing function (you'll always want to specify 'value' as BLACK):

```
void setPixel(int x, int y, rgb color);
```

For polylines, rectangles, floodfilling, and airbrushing, you will need to add code to the `mouseClick()` function. In order to make them function correctly, you will need to use the variables `nGotStart`, `nStartX`, and `nStartY` (refer to the rest of the function to figure out how to use them). To scan convert the primitives you are allowed (and encouraged) to reuse your line function.

Also, be sure your program can still read files. We have implemented the parser which reads them in and calls your functions for you. However, you should still ensure that it still reads files properly before you hand it in. This is a good way for you to test your program and it is also how we will grade it.

Floodfilling

While the majority of this project is rather simple in order to get you going, floodfilling is not as simple as you might first think. In order to get full credit, we require you to implement a 4-connected flood fill algorithm that will change anything that's the same color as the clicked pixel to whatever the current drawing color is. Furthermore, we ask you to do this *without using a recursive function*.

Each time you make a recursive function call, you add a significant amount of data on to the stack. With flood fill, as it is described in the lecture notes, you may be recursing for every pixel (for the simple algorithm) or each new horizontal run (for the more complicated algorithm). Since the program stack is a finite size and most operating systems do not protect against stack overflow, it is dangerous to use recursion for flood fill. You may implement your own linked list or stack, but you are encouraged to use the `std::list` structure, part of the C++ standard template library, which does most of the work for you.

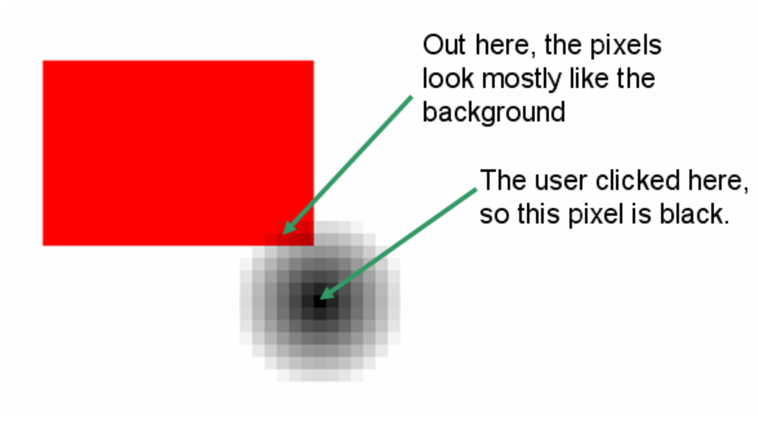
In order to do a flood fill, you'll need to be able to read pixel values in the frame buffer (to see whether they're the same as the seed point or not). The following function can do that for you:

```
rgb getPixel(int x, int y);
```

Airbrushing

The nice paintbrushes in Photoshop generally use a sort of 'airbrushing' technique, where the color of your brush is blended with the color of the background. The pixel right where the user clicked is painted with the paintbrush color, and every pixel within a specified radius is blended between the background and the paintbrush color. The farther a pixel is from the mouse-click point, the more it looks like the background. You should implement this for your MiniPaint assignment. This was not discussed in class, so you'll have to work this out for yourself.

You can hard-code the "radius" of the brush, and you don't need to worry about optimizations at all here. Here's a quick example of what airbrushing should look like; the user's brush is black right now, and the red rectangle was there before he airbrushed:


Out here, the pixels look mostly like the background
The user clicked here, so this pixel is black.

Extra Credit

You are encouraged to add useful new features to your paint program, e.g. new primitives, new interface features, rotated rectangles, etc. Drop the staff a note before you try anything time-consuming, to confirm that it will merit extra credit.

Deliverables

All submissions are due by 11:59 p.m. on the date specified in the syllabus.

You must include a README. The README should document what your program does, any bugs the TA should be aware of, any extra credit you have implemented and any additional information you think would aid the TA in grading your project.

You should submit your program electronically using the submit script. First, while on a Linux machine (**remember, your project must run on Linux!),** you should go to the directory where your code resides. Be sure to remove any executables or .o files as we will compile it ourselves when we grade it. Then you should run:

```
/usr/class/cs148/bin/submit
```

while in the directory. This script will ask you a few questions to make sure all is well in the universe, then it will copy all of your code files, your makefile, and your README and put them in the submissions directory to be graded. The submit script should list all the files submitted. We will be checking time stamps and using this information to track late hours.

Please be aware that your projects in this course will be graded primarily on their output. We will grade your projects by running them through various tests. Because of the number of projects the TA must grade, it will not be possible for them to analyze your programs to determine why they failed on a particular test case. Your best bet is to thoroughly test your program, and compare your output to the sample solutions.