CS-256—Advanced Programming
Lab 8—31, Mar.

Jim Rogers
jrogers@cs.earlham.edu

Spring 2003

Objectives

1. To learn how to use the CarnegieMellonGraphics package (hereinafter to be referred to as the CMgfx package) a simple C++ 2d graphics package based on OpenGL. This will allow you to add graphical output to your applications. In a later lab you will learn to work with mouse input as well, which will enable you to build applications with fully graphical user interfaces.

2. To modify the tankpuzzle (non-buggy, this time, one hopes) application to provide a graphical display.

3. (Optional challenge) To animate that display.

Background

CMgfx is an object-oriented, C++ based, graphics library. It provides a suite of classes which collectively provide the ability to draw two-dimensional graphics. The largest of these classes is the Window class. This provides the window the graphics are drawn in—the various drawing functions are member functions of this class. We will also use three auxiliary classes Color, which represents color information, Font which represents font information and Style which holds style options for the various drawing functions.

This lab involves several phases. The first phase is the graphical equivalent of “Hello World”—you will write a trivial program that opens a graphics window and use it to try out several of the simple drawing functions. In the second phase you will use these to add a display method to the tank class which will draw a representation of a mixing tank. Next you will modify tankpuzzle to create a window and display the mixing tanks in it. The final phase is an optional challenge that asks you to extend what you have done so that the adding and drawing of the tank contents is animated over time.

There is some (minimal) documentation for the CMgfx library available in the class web at

http://cs.earlham.edu/~jrogers/classes/cs256/cmg2docs/index.html

You will need to consult this during the lab.
Hello World

Here is a pretty nearly minimal CMgfx program (it is provided for you as
line.cpp in the class directory and web):

```cpp
/* line.cpp */
#include <cstdlib>
#include <iostream>
#include "CarnegieMellonGraphics.h"

int main()
{
    Window window(640, 640, "The Magic Line");

    Style linestyle(Color::RED, 4);
    window.drawLine(linestyle, 0, 640, 640, 0);

    char key;
    cout << "'any key' to exit: ";
    cin >> key;

    exit(EXIT_SUCCESS);
}
```

This declares window to be an instance of Window with a width of 640 pixels,
a height of 640 pixels and the title “The Magic Line”.
Drawing in the window
is mostly just a matter of invoking the various drawing methods of the Window
class. Here, we draw a line from the point (0,640) to (640,0). As you will see
when you compile and run this, Y coordinates are screen coordinates, which
means that (0,0) is in the upper-left corner of the window. So this line is the
diagonal from the bottom-left corner to the upper-right corner.

The linestyle parameter determines the style of the line. For the most
part, you will use this to set the color and width of the line. Color::RED is
a defined constant of the Color class. You can also set the color by using
the constructor of the Color class: Style linestyle(Color(255,255,0), 4).
The three values are levels of red, green and blue, respectively, with 255 being
the maximum level. (Which color does this particular combination give you?)
The width parameter is optional, defaulting to 1. The Style parameter of
drawLine is not optional; you must specify a style for the line.

The next part of the program waits for the user to indicate they are finished.
Without this delay, the program would exit immediately, and the window would
be drawn and then disappear immediately. Finally the program terminates by
calling exit(), not by returning. If you end main with a return statement,
the graphics package is likely to segfault.

---

1This window will normally be created without the window manager’s buttons and tabs,
so if you are using VTWM the only place you will see the title is in the icon manager.
Building CMgfx Applications

In addition to including the CarnegieMellonGraphics.h header file for the declarations of the CMgfx classes and functions one has to link against the libraries which contain their implementation. This is made more complicated by the fact that the CMgfx package is built on OpenGL, which is, itself, a suite of graphics libraries. This, in turn, is built on X11, the underlying windows system. As a result, the list of libraries which must be included in the compile command is quite stunning:

```
-lttf -lpng -ljpeg -lglut -1GLU -1GL -1Xi -1Xext -1Xmu -1X11 -lm
-lpthread
```

This is what make is made for.

You are provided with two makefiles: Makefile is a makefile that uses default rules to build any single file cpp program that uses the CMgfx library. To use it, if your program source file is named zowie.cpp you just say `make zowie`. It should find the source file and compile it to produce an executable named zowie. This won’t work for tankpuzzle since it depends on tank.h and tank.o and it must be linked against tank.o. For this, you have been provided with a file Makefile.tank. You can either specify `“-f Makefile.tank”` when you invoke make or you can rename the default Makefile to something else and rename Makefile.tank to Makefile. This has targets for both tankpuzzle and testtank.

Before you move on, you should make sure you can successfully build line.cpp.

Other Drawing Primitives

Explore the documentation for the Window class. Click on Window in the All Classes frame and then scroll down the main frame to the Method Summary. The various drawing primitives are named draw.... You will need, at least, drawRectangleFilled() in addition to drawLine(). The method names in the summary list are hyperlinks to the details you need to know to use the method. Try out several of these.

Setting the Background

The documentation states that, by default, the background of a window will be black, but, in fact, the default background is white. It appears that the only way to set the background is to draw a filled rectangle the size of the window before you draw anything else.

Drawing Text

To write text in the graphics window you have to draw it using

```
drawText( s, f, x, y, text );
```
where $s$ is a Style, $x$ and $y$ are integers giving the position of the beginning of the text, text is the string to be written and $f$ is a Font, which specifies which font to use and what size to make it. There are four built in fonts: HELVETICA, MONO_ROMAN, ROMAN and TIMES, which you can specify (using a constructor of the Font class) with:

Font(HELVETICA, 12)

The first and last of these built in fonts are "bitmapped" fonts and look reasonably good but have fixed sizes. The middle two are "stroke" fonts which means they are drawn with lines and can be rendered in any size. Unfortunately, they aren’t drawn very well.

You can also use any TrueType font you can find on your system. You can locate these using locate .ttf which will likely dump out several screens full of file names. (ttf is the extension of TrueType font files.) Unfortunately, these are not guaranteed to be available on all systems and their locations will vary from system to system, so using them makes your program non-portable. (Although all the ACL machines other than quark will have the same fonts in the same places.)

Two fonts that you can be sure of on the ACL machines are

/usr/local/src/CMUgraphics/examples/arial.ttf
/usr/local/src/CMUgraphics/examples/cour.ttf

which are an arial (proportional) and a courier (fixed pitch) font. You specify these with the alternate constructor:

Font("/usr/local/src/CMUgraphics/examples/cour.ttf", 15)

where, again, the number gives the point size. Since you are likely to use only a few fonts at a few sizes and since these are likely to need to be modified over time, it is usually a good idea to define a Font constant for each one in an place where they can be easily located:

const Font littletext("/usr/local/src/CMUgraphics/examples/arial.ttf", 15);
cost Font bigtext("/usr/local/j2re1.3.1/lib/fonts/LucidaSansOblique.ttf", 30);

**Drawing Numerical Values as Text**

One of the things you will need to be able to do is to display numerical quantities such as the volume or mix in the tank. drawText() is not like <<; it displays only strings. The CMgx library provides a function numberToString() which will convert numerical values to (C++) strings. Note well, this is declared as a member function of the Window class. So it should be invoked as "s = w.numberToString( n )" or some such. There are actually two variants of this function. The first takes an int as its only argument. The second takes a double as its first argument and an (optional) int as its second argument. The second argument selects the number of digits to show following the decimal
point. If it is left out, one gets the default C precision, which is six decimal places.

Before proceeding, extend your “Hello World” program to display text both from a constant string and from a numerical value.

Object-Oriented Graphics

In the O-O approach to graphics one takes the process of drawing the representation of an object to be one of the capabilities (or methods) of the object. So we will implement a display() member function for each class of objects that has a graphical representation. In the case of tankpuzzle there is only one such class, the mixing_tank class. The next step of the lab is to add the display() method to mixing_tank.

In order to display itself, a mixing tank needs to know two things: the identity of the window it is to be displayed in and its position in that window. With both of these it is possible to either keep this information in member variables or to pass it as arguments of the display() function. There are reasonable arguments for both alternatives.

Note that there are significant differences between the relationship of the tank to its position and its relationship to the window. The position of the tank is unique to that tank. The window, on the other hand, will be shared by all objects that will be displayed. The position can be kept as a couple of ints. The window, in contrast, cannot be contained in any one object. Rather, if we are going to store its identity in a member variable we must do so with either a pointer or a reference to window. To simplify things, we have chosen to store the x and y positions as member variables and to pass the window by reference as an argument of display(). We have also added a string member variable name which will be used to label the tank when it is displayed.

The addition of these new member variables requires additional arguments for the initializing constructor. The file tank.h declares the mixing_tank class with these additional member variables, the extended initializing constructor and the display() member function. The file tank.cpp includes the implementation of the initializing constructor but only a stub for display(). You must fill in this stub.

Displaying the Tank in a Fixed Position

The first step is to display the tank in a fixed position. For this, you will ignore the x and y values and simply draw the tank in the top left corner of the screen. Figure 1 gives a suggestion of how you might display the tank. The “position” of the tank is the corner marked “Origin”. Initially you should just take this to be the upper left corner of the window ((0, 0)). Note that the height of the tank should be proportional to the tank’s capacity and the height of the filled section should be proportional to the volume it actually holds. The color of the filled section should be a representation of the mix with liquid ‘A’ being represented,
for instance, by blue (Color(0,0,255)) and ‘B’ being represented, for instance, by red (Color(255,0,0)).

Work out on paper the coordinates of the various vertices. You will also need to work out the formulae for basing these on the tank’s actual values. For the purposes of this lab you may assume that capacity is never greater than 500 units. Figure out the coordinates for this largest tank and then use these to derive a formula for the coordinates for tanks with smaller capacity. Do the same for the volume. For the color of the mix, the blue value should be 255 times the fraction of liquid A, while the red value should be 255 times the fraction of liquid B. (These fractions, of course, will add to one.) Bear in mind, as you do this, that when you add graphics to testtank you will be displaying two tanks. Make sure you size your tank display so that at least two can fit within your graphics window.

You should then write the display() function using these formulae. You must clear the area in which you are drawing the tank first by drawing a rectangle filled with your chosen background color. You should also draw text to display the numerical status of the tank. (Don’t forget to observe the

Figure 1: A possible mixing tank display
You can use the version of testtank that is included in with the lab source files to test your display function. This has been modified to name the tank under test "Tank 1", to position it at (0,0) and to call display() at the beginning of each pass of the command loop. It also has an additional command 'c' which allows you to redefine the tank with a different position and capacity. With this you should be able to test all aspects of your display() function.

Before you continue, get your display function working properly.

**Display Buffering**

By default, CMgfx draws directly to the display. This makes everything you draw visible as soon as it is drawn, but also makes the drawing process at least appear to be slower. The other alternative is "double buffering" in which one draws to a hidden buffer and then swaps it with the buffer that is being displayed when the drawing is done. This looks much better, particularly when the graphics being displayed are animated. To select this mode in CMgfx you must call the disableAutoPageFlip() member function of the window and then call the flipPage() member function when you have finished updating the drawing.

Modify testtank to call disableAutoPageFlip() for the Window w and to call flipPage() immediately after calling the display() function of the tank.\(^2\)

**Displaying the Tank in an Arbitrary Position**

Once you have a display() member function that draws the tank correctly relative to the window origin (0,0) all you need to do to convert it to draw the tank relative to the tank's intended position is to add x to every x value and y to every y value in the function.

**Giving tankpuzzle Graphical Output**

For the most part, you don't have to do much more to add graphical output to tankpuzzle than was done to add it to testtank. You must declare the window and initialize it by drawing the background of your choice and by calling disableAutoPageFlip(). In declaring you tanks, you should select an origin for each that will display them both without overlapping. You then just need to call their display() functions, followed by calling the window's flipPage() function, at the appropriate place.

You should also consider providing a sample of the color of the final mix. You can do this by reserving a portion of the graphics window in which you can draw a rectangle filled with the 24% liquid 'A'. (You will need to work out the red and blue values corresponding to that mix.

\(^2\)In the preliminary version of this lab I had you calling flipPage() from the display() function. But when you are drawing several objects, you only want to call flipPage() once for all of them, so it is better to not put it in the display() functions.
Animating Fill, Pour and Empty

The final phase of the lab is to animate the filling, pouring and emptying of the tanks. You should be able to do this by modifying only the corresponding functions of tankpuzzle.cpp. Don’t worry about setting a specific time interval between drawing successive states of the animation, just draw them one after another letting the creaky drawing speed of the CMgfx package set your frame rate.

This last phase is optional but fun. But don’t spend time on it that you need to work on the current programming assignment.

Assignment

Please submit listings of each of the following:

1. Your final version of line.cpp (or whatever program you used to experiment with the drawing functions).
2. Your final version of tank.cpp.
3. Your final version of testtank.cpp.
4. Your final version of tankpuzzle.cpp.

If you’ve got a particularly flashy version, let me know and we’ll demo it in class.