CS 103 Lab - 2D Arrays & Image Processing

1 Introduction

In this lab you will complete a program that allows you to draw rectangles and ellipses to a BMP image file.

2 What you will learn

After completing this lab you should be able:

- Utilize 2D arrays and understand their indexing
- Accept multiple values from the keyboard using cin and make decisions
- Utilize multiple source code files
- Utilize a Makefile to compile your code

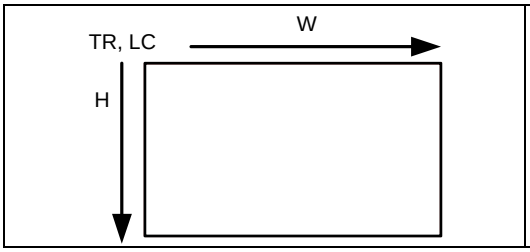
3 Background Information and Notes

In this lab we have provided you code that can save a 256x256 2D array as a grayscale BMP file that you can view on your VM or other machine. You will be asked to write functions that let the user describe a rectangle and/or ellipse and have it "drawn" in black on a white background. We will use a 256-by-256 2D array that will represent an image. Each entry in the array represents 1 pixel. The value of the pixel should be a value of 0 – 255 where 0 is black and 255 is white. Recall that indexing of a 2D array starts with the [0][0] location in the upper left-hand corner (this is different from the Cartesian plane). See the diagram below.

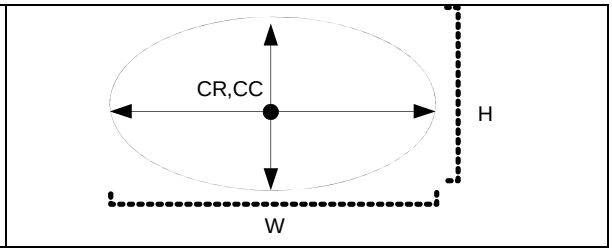
The 2D array for an image uses [row][column] indexing. The first index is the row (top 0, bottom 255), and the second is the column (left 0, right 255).

	Col. 0	Col. 1	Col. 2	…	Col. 255
Row 0:	[0][0]	[0][1]	[0][2]	…	[0][255]
Row 1:	[1][0]	[1][1]	[1][2]	…	[1][255]
Row 2:	[2][0]	[2][1]	[2][2]	…	[2][255]
…	…	…	…	…	…
Row 255:	[255][0]	[255][1]	[255][2]	…	[255][255]

You will write code to draw rectangles and ellipses. A rectangle can be defined by its top-left corner's row and column position, its width, and its height: when we want to draw a rectangle or ellipse, we will ask the user to supply this info. To draw the rectangle, we need to color in some portion of 2 rows and of 2 columns.



TR stands for top row, LC stands for left column. H & W stands for height and width. We will ask the user to enter the TR, LC, H and W in that order. Thus if they enter 10 20 8 5 then we want a rectangle whose upper-left corner is at row 10, column 20 and is 8 rows high and 5 columns wide.



CR & CC stands for Center Row and Column. We ask the user to enter the CR, CC, height, and width values in that order. Thus if they enter 25 30 50 40 we want to draw an ellipse centered at row 25, column 30 that is 50 pixels high and 40 wide (i.e. extends up and down 25 pixels from the center point and 20 pixels left and right)

You should be able to devise an approach to draw the lines representing a rectangle with a few `for` loops. (Note: the rectangle should be hollow, not filled.) However, we defer further discussion of coding to the **Procedure** section, in order to describe the geometry of the ellipse here.

An ellipse can be specified by supplying a center point (row and column) and then its height and width. However, plotting it is more complicated than a rectangle. It is not bad if we use polar coordinates. Let's first consider a circle (which is a special form of an ellipse). Recall that given an angle, Θ and radius, r, we can determine the x, y points on that circle as:

$$x = r\cos\theta$$
$$y = r\sin\theta$$

If we simply go through all the Θ values between 0 and 2π (maybe in increments of .01) we would enumerate all the x, y pixels along the circle and could make them "black" pixels. To do this in C++, you would write a `for` loop that varies theta and marks the appropriate x, y pixels as black.

An ellipse is like a circle that has been squeezed and stretched. Therefore, an ellipse with a horizontal half-axis r_x and vertical half-axis r_y has the parametric equation:

$$x = r_x\cos\theta$$
$$y = r_y\sin\theta$$

In our case, r_x is W/2 and r_y is H/2.

Note:

- We have expressed our equations in terms of Cartesian coordinates x and y where 0, 0 is the center point. You will need to translate (i.e. shift) each x, y value by adding to it the coordinates of the center point.
- When we list a point as (x, y), we are using mathematical Cartesian coordinates. x refers to the horizontal distance and y to the vertical. But in image indexing vertical distance is listed first (i.e. rows) and horizontal is second (i.e. columns). Take care to account for this difference.
- **If a portion of the rectangle/ellipse would fall in an area outside of the image, just draw the portion of the shape that DOES fall in the allowed area. Your program should NOT crash if the coordinates provided fall out of bounds, nor should the shapes "wrap around".**

4 Getting started with *bmplib*

To get started, make a directory for this lab, and download the code. Go to your directory of choice and enter:

```
$ mkdir lab-bmplib
$ cd lab-bmplib
$ wget ftp://bits.usc.edu/cs103/lab-bmplib.tar
$ tar xvf lab-bmplib.tar
```

The files this week include a **bitmap library** (bmplib), a **Makefile** needed for compilation since you will be compiling multiple files together, and skeleton code. You won't need to look at bmplib.cpp, which has implementation details. However if you open **bmplib.h** in a text editor, you'll see it declares several things including

```
const int SIZE = 256;
int writeGSBMP(const char filename[], unsigned char outputImage[][SIZE]);
```

So, if you put `#include "bmplib.h"` in another program, it will know that `SIZE` is 256, and it will be able to call the `writeGSBMP` function on a given filename and 2D image array to save it to disk. For example,

```
writeGSBMP("output.bmp", image);
```

Another function from **bmplib** you may use this week is:

```
void showGSBMP(unsigned char outputImage[][SIZE]);
```

It displays the image to the screen.

'make' and Makefiles: To produce your executable you will need to compile BOTH your program (it will be called `shapes.cpp`) *and* `bmplib.cpp`. There are several ways to do this (as explained in lecture) but we've included a customary set of machine-readable instructions called a **Makefile**. A Makefile is just a text file that has instructions on how to compile your program; feel free to open it and read it if you want, though you will not have to write your own in this course.

To ask this week's makefile to execute its instructions, type:

```
$ make
```

(It may ask you for your password the first time.) This week the Makefile instructions basically say (1) compile `demo.cpp` with `bmplib.cpp` to make the `demo` executable, and (2) compile `shapes.cpp` with `bmplib.cpp` to make the `shapes` executable.

Note: if you need or want to do it yourself, you can. E.g. for (1):

```
$ compile demo.cpp bmplib.cpp -o demo
```

Compiling a single file won't work. (But, see the discussion of "object files" in lecture.)

We have included `demo.cpp`, a sample program to draw a few lines & a circle. View `demo.cpp` by opening it in gedit. When you think you understand the code, compile it by typing `make`. When you run `./demo`, it creates an image file called `cross.bmp` on your VM's hard drive. You can use the "eog" image viewer to open it,

```
$ eog cross.bmp &
```

Optionally, also, you can uncomment the four commented-out parts of demo.cpp, recompile and re-run to see how `showGSBMP()`works.
Note: `showGSBMP()` may not work on machines other than the course VM. You may be able to edit the bottom section of `draw.cpp` to get it working on your machine.

5 Writing your program

Open the skeleton program `shapes.cpp` in gedit.

You will complete it to implement a program that provides a menu to the user to draw a rectangle (command 0), ellipse (command 1), or to quit (command 2). Based on the integer they enter you can prompt them for more info pertinent to

the rectangle or ellipse. Then given this info, draw black pixels to create the indicated rectangle or ellipse. Keep repeating the menu and input process until the user enters the integer 2.

5.1 draw_rectangle()

Write an implementation for the `draw_rectangle()` function with the given input parameters. The 2D image array is a global variable so you don't need to pass it in the function. (Normally global variables are bad style, but it simplifies things this week.) Use appropriate loops to draw a black hollow rectangle.

- **If a portion of the rectangle/ellipse would fall in an area outside of the image, just draw the portion of the shape that DOES fall in the allowed area. Your program should NOT crash if the coordinates provided fall out of bounds, nor should the shapes "wrap around".**

You can do a basic test of your code by adding this to `main`:

```
draw_rectangle(10, 10, 20, 20);
```

Then compile (with `make`), execute, and run `eog output.bmp`. You should see a square near the top left. Delete the added code afterwards.

5.2 draw_ellipse()

Write a function implementation for the draw_ellipse() function with the given input parameters. Use an appropriate loop to draw the indicated ellipse in black.

- **See the bullet point above.**

5.3 main()

Write code that repeatedly loops and displays the instructions to the user, receives the user input and then either draws the shape and repeats or exits if that is what the user chooses. You should print something similar to the following:

```
To draw a rectangle, enter: 0 top left height width
To draw an ellipse, enter: 1 cy cx height width
To save your drawing as "output.bmp" and quit, enter: 2
```

Once the user chooses to quit, execute the `writeGSBMP()` function (already completed in the code skeleton).

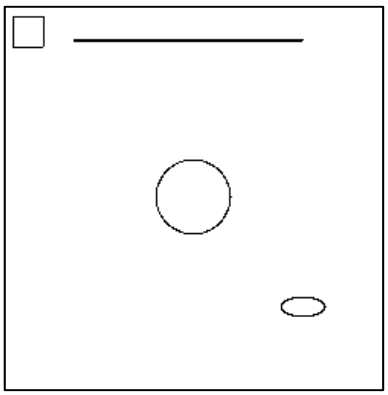
Debugging note: If you want to debug and desire to print out pixel values (i.e. 0's and 255's) you will need to cast the unsigned char pixel values to integers because when cout tries to print an unsigned char it assumes it's ASCII and prints the character code. So you should do something like this:

```
cout << (unsigned int)(image[i][j]) << endl;
```

Example: Here is the expected output from a sample run of the program.

Commands:

```
0 10 10 20 20
0 25 50 1 150
1 128 128 50 50
1 200 200 15 30
2
```



Make sure your program can also create this kind of drawing.

Demonstrate your program and show your TA/CP the two functions you wrote explaining how it works.

5.4 Challenge for the bored

For optional extra practice, use your program to create additional patterns. For instance, you can create the following tessellation using additional for loops: