ECE220: Computer Systems & Programming **Fall 2023 ZJUI**
Machine Problem 5 **due: SATURDAY 4 NOVEMBER, 11:59:59 p.m.**

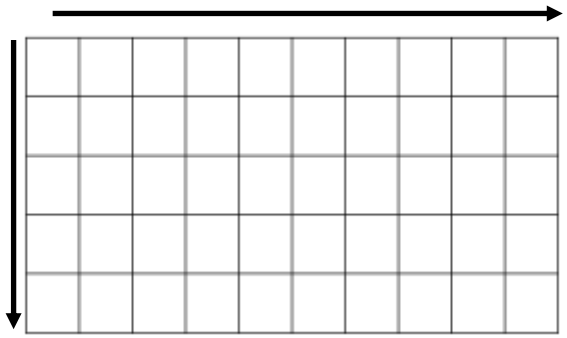
Using Functions to Draw a Picture

Your task this week is to write a selection of functions that act as simple drawing tools to create a PNG image. Specifically, you will write functions that can draw lines, rectangles, triangles, parallelograms, and circles, as well as a color gradient in a rectangular shape. You will then write a function that calls the other functions to draw a picture of anything you like.

The objective for this week is for you to gain some experience with functions in C, particularly with using functions as building blocks for more complex functions. This skill is critical when developing software based on existing libraries and toolkits.

Background

A PNG image is one of several different image formats for computers. In this MP, it can be abstracted to a rectangular grid of pixels like the one shown to the right, where each pixel is represented by an empty square. Each pixel has a location in the grid that can be written as a coordinate pair, *(x,y)*. For the purposes of this MP, the pixel in the top-left corner of the image has a position of *(0,0)*. The pixel directly to the right of that has a position of *(1,0)*, incrementing the x-position by 1. Following this trend, for an image that is WIDTH pixels wide, the pixel in the top-right corner will have a position of *(WIDTH-1,0)*. The y-position increases as the rows go down, so the row on the bottom has a y-position of HEIGHT-1 when the image is HEIGHT pixels tall.



Each pixel has a color value associated with it, completely independent of any other pixels. To facilitate a wide range of colors, the pixel's color is divided into three channels, representing the relative intensities of the colors red, green, and blue in the color of that pixel. Each of these channels can have 256 different levels of intensity. Note that one byte can take on 256 different values, making it a perfect way to represent the level for each channel.

8位二进制 8位 8位 R G B

As might be expected, the greater the number representing the red channel, the more red the pixel will look, whereas a number close to zero means that the pixel is not red at all. This definition is the same with green and blue. 255 red, 0 green, and 0 blue represents pure red, while 0 red, 255 green, and 0 blue is pure green, and 0 red, 0 green, 255 blue is pure blue. 0 for all three channels is a complete absence of color, or black, while 255 for all three channels is white. A great resource for trying out different colors can be found at: https://www.w3schools.com/colors/colors_rgb.asp.

The Task

You will be given the following two C functions to use as tools when you write your code:

```
int32_t draw_dot (int32_t x, int32_t y);

void set_color (int32_t color);
```

The `draw_dot` function the lowest level function in the hierarchy that you must write. It takes two parameters: x and y. It will change the color of the pixel at the point (x,y) to be whatever color you have set and return 1. If the pixel given is out of bounds (the image is `WIDTH` pixels wide and `HEIGHT` pixels high), `draw_dot` returns 0 without drawing anything.

The `set_color` function changes the color that `draw_dot` uses for drawing pixels. Once your code calls `set_color`, every subsequent use of `draw_dot` will use the color set by the most recent call to `set_color`. At the beginning of execution, this color will be white.

The parameter color is an `int32_t`, which means it is 32 bits long, or four bytes. The most significant byte is ignored. The second most significant byte (bits 23-16) represents the red channel of the color. The third most significant byte (bits 15-8) represents the green channel of the color. The least significant byte (bits 7-0) represents the blue channel of the color.

Not all functions that you have to write will use either of these two functions, but some of your functions will require that you make calls to them. Make sure that you are only calling these two functions when writing functions where you are explicitly told to use either `draw_dot` or `set_color`.

Each function that you write returns an `int32_t`. Some of these functions call `draw_dot`. For those that do, the function must return the AND of the return values of all calls to `draw_dot`. This rule means that if any of the pixels drawn are outside the bounds of the picture, your function will return 0, and it will return 1 otherwise. However, you are still required to complete all calls to `draw_dot`, even if the return value is known to be 0.

Functions that call other functions you have written must find the AND of the return values of each of the functions called, and return this value. Once again, you are required to complete all function calls, even if the return value is known to be 0.

The functions you are required to write are as follows:

There are two similar functions for drawing lines called `near_horizontal` and `near_vertical`. The function `near_horizontal` should only be called for lines that have a slope between -1 and 1 inclusive. Such slopes look shallow to a human eye. On the other hand, `near_vertical` should be called on lines that have a slope larger than 1 or smaller than -1. Given two pixels at coordinates (x_1,y_1) and (x_2,y_2), the slope is calculated by $\frac{y2-y1}{x2-x1}$. Note that you do not need to perform the division to check which type should be drawn. When $(x_1,y_1) = (x_2,y_2)$, call `near_vertical`.

```
int32_t near_horizontal (int32_t x_start, int32_t y_start,
                         int32_t x_end, int32_t y_end);
```

For this function, you will use point-slope form to select a y-coordinate that corresponds to each x-coordinate between `x_start` and `x_end`, including those two points. Begin by deciding which point is (x_1,y_1) and which is (x_2,y_2). Depending on how you choose to order the two points, one part of your

code will be simpler, while another will be more complex. Think carefully about the loop structure and the algebra. You will then use `draw_dot` to fill in the pixel at each location. Point-slope is as follows:

$$y - y_1 = \frac{(y_2 - y_1)(x - x_1)}{(x_2 - x_1)}$$

However, we want the lines to be smooth, so we need to round off the integer division. We first multiply both numerator and denominator by two, then add another half of the denominator to the numerator. Since most processors round towards zero, however, we need to "add" the extra half in the right direction. In particular, we need a factor of $\text{sgn}(y_2 - y_1)$, the sign of $y_2 - y_1$, which is 1 when $y_2 > y_1$, 0 when $y_2 = y_1$, and -1 when $y_2 < y_1$. You do not need to implement the 0 case, however. We obtain:

$$y - y_1 = \frac{2(y_2 - y_1)(x - x_1) + (x_2 - x_1)\text{sgn}(y_2 - y_1)}{2(x_2 - x_1)}$$

This equation can be rearranged to be:

$$y = \left[\frac{(2(y_2 - y_1)(x - x_1) + (x_2 - x_1)\text{sgn}(y_2 - y_1))}{(2(x_2 - x_1))}\right] + y_1$$

It is imperative that the function evaluate these expressions using the parentheses as specified. This expression will generate correct intermediate values that will yield a final output identical to the one generated in the gold code.

```
int32_t near_vertical(int32_t x_start, int32_t y_start,
                      int32_t x_end, int32_t y_end);
```

Unlike the previous function, for this one, you will use point-form to select an x-coordinate that corresponds to each y-coordinate between `y_start` and `y_end`, including those two points. As before, you must first choose which point is (x_1,y_1) and which is (x_2,y_2). Also unlike the previous function, the two points may be identical, in which case your function should draw a single dot (it is up to you to guarantee that `near_horizontal` does not receive identical points when called from your code). You will then use `draw_dot` to fill in the pixel at each location. You can use the equation in this form:

$$x = \left[\frac{(2(x_2 - x_1)(y - y_1) + (y_2 - y_1)\text{sgn}(x_2 - x_1))}{(2(y_2 - y_1))}\right] + x_1$$

Again, it is imperative that you evaluate the expression using the exact parentheses specified in this equation.

```
int32_t draw_line (int32_t x_start, int32_t y_start,
                   int32_t x_end, int32_t y_end);
```

This function must call one of `near_horizontal` or `near_vertical` depending on the slope of the line between (`x_start, y_start`) and (`x_end, y_end`). Remember that given two pixels at coordinates (x_1,y_1) and (x_2,y_2), the slope is calculated by $\frac{y2-y1}{x2-x1}$. Also, remember that `near_horizontal` should only be called for lines that have a slope between -1 and 1 inclusive, and `near_vertical` should be called on lines that have a slope larger than 1 or smaller than -1. For lines for which the slope is not defined (that is, for which the two points are the same), your code must call `near_vertical`.

```
int32_t draw_rect (int32_t x, int32_t y, int32_t w, int32_t h);
```

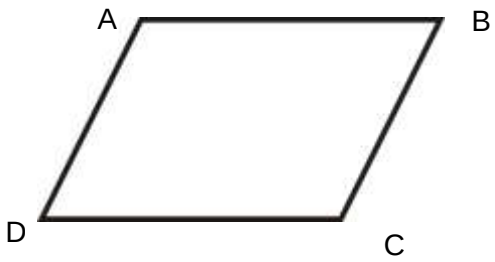
This function draws...you guessed it, a rectangle! It draws a rectangle of a height and width specified in the parameters as h and w, respectively, with x and y being the coordinates of the top left corner of the rectangle. The upper right corner is at (x+w,y), and so forth. If either the height or the width is negative, the function must immediately return 0 without altering the image in any way. The sides of the rectangle must be drawn by calling the `draw_line` functions.

```
int32_t draw_triangle (int32_t x_A, int32_t y_A,
                       int32_t x_B, int32_t y_B,
                       int32_t x_C, int32_t y_C);
```

A triangle is any shape that has three sides. For this function, you will be given three x-values and three y-values with the suffixes A, B, and C corresponding to the three vertices of the triangle. You are expected to use the `draw_line` function to connect each of the three pairs of vertices with a line. This technique will form a triangle.

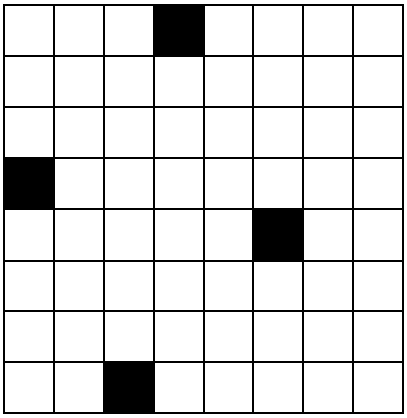
```
int32_t draw_parallelogram (int32_t x_A, int32_t y_A,
                            int32_t x_B, int32_t y_B,
                            int32_t x_C, int32_t y_C);
```

A parallelogram is a four-sided shape such that opposite sides are parallel, as shown below.



For this function, you will be given three points of the parallelogram in the form of three sets of x-values and three sets of y-values with the suffixes A, B, and C to show which point they correspond to. The sides AB and BC are part of the parallelogram, but it is important to note that AC is not. It will be up to your function to determine the location of the final point, D, and use it to draw the two remaining sides, CD, and DA. You must use `draw_line` to accomplish this task. If any of the pixels in the parallelogram lie out of bounds, you must return 0. Otherwise, return 1.

An important piece of information to note while drawing your parallelogram is that for any side, the relationship in space between the two end points is exactly the same as the relationship between end points on the opposite side. For example, in the parallelogram to the right, the point on the left is three squares down and three squares to the left of the point on the top. Likewise, the point on the bottom is three squares down and three squares to the left of the point on the right.



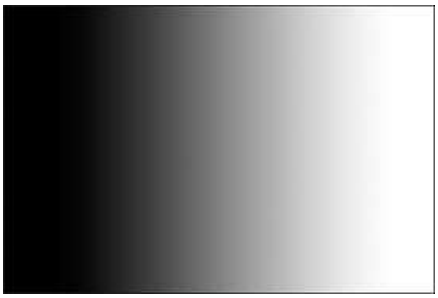
```
int32_t draw_circle (int32_t x, int32_t y,
                     int32_t inner_r, int32_t outer_r);
```

This function draws a circle centered at *(x,y)*. The function must first verify that the inner radius given is greater than or equal to 0. The outer radius must be greater than or equal to the inner radius. If either of these constraints do not hold, the function must return 0 without altering the image in any way. The circle must use `draw_dot` to fill in every pixel in between the inner radius and outer radius (inclusive on both sides). To determine whether a pixel is in the given range, use the Pythagorean theorem/distance formula:

$$(x_2 - x_1)^2 + (y_2 - y_1)^2 = \text{distance}^2$$

You must verify that the pixel to be filled has a distance that is less than or equal to the outer radius and greater than or equal to the inner radius.

```
int32_t rect_gradient (int32_t x, int32_t y, int32_t w, int32_t h,
                       int32_t start_color, int32_t end_color);
```



For this function, like `draw_rect`, you are given the coordinates of the pixel at the top-left corner and the height and width of the rectangle to draw. If the height is less than 0, or the width is less than 1, this function must immediately return 0 without altering the image in any way. This function is unique, because instead of using `draw_line` to draw the sides of the rectangle, you will be using a combination of `draw_dot` and `set_color` to create a rectangular region in the image that forms a gradient from `start_color` to `end_color` from left to right in the rectangle. The image to the left shows a rectangular region with a gradient from black to white. (This image also has a black border for clarity. Your function should not draw a border.)

In order to draw the gradient, you will need find the intensities of the three channels in each of the colors given, then use the following equation to determine for each pixel in the image the amount of red, green, and blue will be in the color.

$$level = \left[\frac{(2(x - x_1)(level_2 - level_1) + (x_2 - x_1)\mathrm{sgn}(level_2 - level_1))}{(2(x_2 - x_1))}\right] + level_1$$

Here, x is the x-location of the pixel for which you are finding the color, x_2 is the x-location of the right edge of the rectangle, and x_1 is the x-location of the left edge of the rectangle. *level* refers to the value 0-255 representing the intensity of one of the three color channels at this pixel. Similarly, $level_1$ and $level_2$ represent the corresponding levels at the left and right edges of the rectangle, respectively. Using this equation to calculate the level for all three channels at a pixel will allow you to form the correct color for that pixel and pass it to `set_color`, and then you can use `draw_dot` to draw a pixel of the correct color to the image. Using this approach for every pixel in the rectangle will result in a smooth transition from `start_color` to `end_color`. As before, the function must evaluate the expression using the parentheses specified in order to generate the proper output.

```
int32_t draw_picture ();
```

`draw_picture` takes no parameters and may call any or all of the other functions to draw a picture of your choice. When you write this function, you have two choices. The first choice is that you may call the other functions you have written to draw a five-or-more-letter word in the image (but not "IllINI"—sorry!). The second choice is to draw a picture of your own design that has greater or equal complexity compared with a five-or-more-letter word. Use your best judgement when adopting this choice; at least one member of the course staff must agree in advance that the picture is of suitable complexity in order for it to be judged adequate.

Pieces

Your program will consist of a total of two files:

`mp5.h`	This header file provides function declarations and a brief description of the functions that you must write for this assignment.
`mp5.c`	The main source file for your code (you must write it yourself). Include the **`mp5.h`** header file and be sure that your function matches the one in the header.

Two other files are also provided to you:

`main.c`	A source file that interprets commands and calls your function.
`Makefile`	A file that allows you to use `make` commands to compile your code.

You need not read these files, although you are welcome to do so.

Specifics

You should read the description of the functions in the header file before you begin coding.

- Your code must be written in C and must be contained in the **`mp5.c`** file in the **`mp/mp5`** subdirectory of your repository. We will NOT grade any other files. **Changes made to any other files WILL BE IGNORED during grading.** If your code does not work properly without such changes, you are likely to receive 0 credit.
- The image given is 624 pixels wide and 320 pixels high. There are two `#define` statements in `mp5.h` that define `WIDTH` to be 624 and `HEIGHT` to be 320. Use these names in your code if needed.
- Each function that you write will return 1 if all of the pixels it draws are within the bounds of the image and 0 if any of the pixels are out of bounds.
    - For functions that call `draw_dot` directly, you will return the AND of all calls to `draw_dot`. This rule means that if any call to `draw_dot` passes a pixel location that is out of bounds, the return value for that call will be 0, making the total return value 0. But if all pixels are in the image, the total return value will be 1. You will not return from the function until all calls to `draw_dot` have been made, so even if one of the calls return 0, you must still make all of the remaining calls. For example, if `draw_circle` draws a circle which has a right half that is out of bounds and your code determines that the final return value is 0 before it has drawn any of the bottom half of the circle, you must still draw the rest of the circle. This approach will result in the appearance that whatever image you have drawn is completely drawn, but clipped off the side of the screen.
    - Functions that do not call `draw_dot` will make calls to other functions that you have written. These functions will return 1 or 0. The return value of the function that calls them will be the AND of each of the functions called. You are still required to call all of the functions and only return after attempting to draw all parts of the shape.
    - For `near_vertical`, your function must handle the case of two identical points by drawing a single dot (using one call to `draw_dot`).
    - For `draw_rect` and `rect_gradient`, if the given height or width is negative, or if the width is 0, your function must return 0 and should not modify the image in any way.
    - For `draw_circle`, `inner_r` must be greater than or equal to 0. Likewise, `outer_r` must be greater than or equal to `inner_r`. If either of these two relations is not observed, your function must return 0 and should not modify the image in any way.

- The top-left pixel in the image has coordinates (0,0).
    - Pixels increase in x-location as they go to the right.
    - Pixels increase in y-location as they go down.
- The image begins with every pixel being white.
- The color of pixels drawn with `draw_dot` will initially be white. Use `set_color` to change the color of pixels drawn.
- You must be conscientious of these edge cases.
    - A line with a start point and end point that are the same point should just draw that singular point.
    - A rectangle with 0 width or 0 height will just be a line.
    - A rectangle with 0 width and 0 height will just draw a single dot.
    - A triangle where all three given points are collinear (lie on the same line) will just draw a line.
    - A parallelogram where all three given points are collinear (lie on the same line) will just draw a line.
    - A circle with an inner radius and outer radius of 0 will just draw a single dot (the center).
    - You may assume that we will not call `rect_gradient` with `w` (width) equal to 0.
- Each function may only call functions directly specified in the documentation, and **must** call these functions.
- Your routines' return values and outputs must be correct.
- Each function in `mp5.c` is given to you in the following manner (using `near_horizontal` as an example):

  ```
  int32_t
  near_horizontal (int32_t x_start, int32_t y_start, int32_t x_end,
  int32_t y_end)
  ```

  The autograder relies on `near_horizontal` being at the beginning of the line as shown here. You must not alter this part of the code. Only modify the file in the function bodies. Do not add additional subroutines to implement your functions.
- Your code must be well-commented. Follow the commenting style of the code examples provided in class and in the textbook.

Library Needed for the Fall 2020 VM Provided by TAs

If you want to use the VM provided by the TAs, you must first install the PNG library by typing:

```
sudo apt -y install libpng-dev
```

in a terminal window.

Compiling and Executing Your Program

Once you have written the functions into the **`mp5.c`** file, you can compile your code by typing one of:

```
make
make all
```

Either command will compile the program and make an executable called `mp5`. You can also type:

```
make clean
```

This command will delete all files generated by the compiler. These `make` commands are specified by the **`Makefile`**, which you can view if you are curious.

The **`mp5`** program takes one command line argument:

```
./mp5 <r1>
```

The argument specifies one of eight prebuilt tests that can use to test each of your functions individually. These tests are as follows:

```
1-near_horizontal                    5-draw_parallelogram
2-near_vertical                      6-draw_rect
3-draw_line                          7-draw_circle
4-draw_triangle                      8-rect_gradient
```

These commands each produce an output called `out#.png` where # is replaced with the number you entered. There will be files called `gold#.png` available for you to compare to, to see if you have produced the correct output. These files are located in the `gold_out` directory.

If no parameter is given, or the given parameter is not 1-8, the program will call `draw_picture`.

Note that the minimum required for the program to compile is for all of the functions to be defined in `mp5.c` and for them all to return something. If you want to test one of your functions before you have written any or all of the other ones, you can simply set the others to return 1, and they will be valid functions from the compiler's point of view, allowing you to test the function you are writing.

Note that these tests are not comprehensive, and **you are strongly urged to write your own test code**.

To test your code further, you can use the `gold` executable provided to you to run any of the required functions. To do this, type `./gold <r1> <r2> <r3> <r4> <r5> <r6> <r7>`, where `<r1>` represents a number 1-8 to indicate which function you would like to test (using the same key as the test outputs for mp5). The remaining arguments to the executable are the arguments to whichever function you would like to call. For functions that only have four arguments, `<r6>` and `<r7>` are ignored and are not necessary. The output will be put in a file called `gold_out.png`.

Grading Rubric

We put a fair amount of emphasis on style and clarity in this class, as reflected in the rubric below.

Functionality (70%)

- 8% - `near_horizontal` produces the correct output
- 8% - `near_vertical` produces the correct output
- 4% - `draw_line` produces the correct output
- 4% - `draw_triangle` produces the correct output
- 4% - `draw_rect` produces the correct output
- 4% - `draw_parallelogram` produces the correct output
- 10% - `draw_circle` produces the correct output
- 12% - `rect_gradient` produces the correct output
- 12% - `draw_picture` produces an acceptable output (as specified in the function explanation)
- 4% - all functions have proper return values

Style (15%)

- 15% - All functions call the functions they are required to call and do not call any of the other functions

Comments, Clarity, and Write-up (15%)

- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 10% - code is clear and well-commented, and compilation generates no warnings (note: any warning means 0 points here)

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if you code does not compile, you will receive no functionality points. As always, your functions must be able to be called many times and produce the correct results, so we suggest that you avoid using any static storage (or you may lose most/all of your functionality points).

Appendix

This picture shows all of the functions that you will write or interact with in this program. An arrow from one function to another means that the function at the source of the arrow is required to call the function at the destination of the arrow. `draw_picture`, being open-ended, may call any of the functions defined in the MP.