Project - Creating Beautiful Images

This project is all about conjuring up images. In this course, so far, we have primarily worked with text input. While textual data is easy to work with, let's be honest; it's not particularly exciting to write programs which only interact with you via words.

Writing Image data

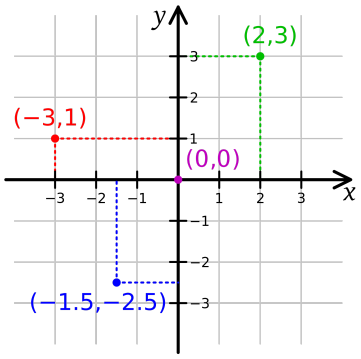
To write image data to a file, we will use the '.pgm' (black and white image) or '.ppm' (color image) file format.
You can read a short description about both here: PGM, PPM
Essentially, you will write to a file like always but in a particular specified format and with an extension in the file name(.ppm or .pgm). Your computer would then look at the extension of the file and, while opening it, interpret it as an image file. This is similar to how your computer would know that anything with say '.pdf' in the file name is a PDF document.

The format goes like this:
<Code> # P2 for pgm and P3 for ppm
Width Height # No of columns and rows in your image separated by a space
255 # This is the value for each pixel in your image, 255 for our purposes
<Pixel values> # here we start listing pixel values in your image, left to right, top to bottom

You will get more practice about printing out images in recitation

Prerequisites - User Coordinates, Generating Images

Before we do any kind of drawing, we need to be able to specify where we want things in our image to be drawn. In high school, you might have learnt about coordinate systems which provide us with a nice way to specify a location in a 2D plane (we are only concerned with 2D for this project).



How do we generate something like this in our code?
Look at the following code snippet.

```
5   int main () {
6       int width = 10;
7       int height = 10;
8
9       for (int i = 0; i < height; i++) {
10          for(int j = 0; j < width; j++) {
11              cout << "(" << i << ", " << j << ")" << " ";
12          }
13          cout << endl;
14      }
15  }
```

You have two for-loops - the outer one going from 0 to height and the inner one going from 0 to width. Inside the loop, you are printing out the loop variables ‘i’ and ‘j’ where you hold ‘i’ constant and vary ‘j’ and then update ‘i’ and repeat the process. The output looks like the following:

```
(0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (0, 6) (0, 7) (0, 8) (0, 9)
(1, 0) (1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (1, 8) (1, 9)
(2, 0) (2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7) (2, 8) (2, 9)
(3, 0) (3, 1) (3, 2) (3, 3) (3, 4) (3, 5) (3, 6) (3, 7) (3, 8) (3, 9)
(4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (4, 5) (4, 6) (4, 7) (4, 8) (4, 9)
(5, 0) (5, 1) (5, 2) (5, 3) (5, 4) (5, 5) (5, 6) (5, 7) (5, 8) (5, 9)
(6, 0) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (6, 6) (6, 7) (6, 8) (6, 9)
(7, 0) (7, 1) (7, 2) (7, 3) (7, 4) (7, 5) (7, 6) (7, 7) (7, 8) (7, 9)
(8, 0) (8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (8, 6) (8, 7) (8, 8) (8, 9)
(9, 0) (9, 1) (9, 2) (9, 3) (9, 4) (9, 5) (9, 6) (9, 7) (9, 8) (9, 9)
```

Okay, so we have something resembling X-Y coordinates. Now, imagine, hypothetically, you had an image with ‘height’ number of rows and ‘width’ number of columns, you can specify a location in that image using the loop variables in a double for-loop like the one above. (alternatively, think of an image like a 2D array and the coordinates as indices of elements in that 2D array)

So far so good.

But there is a problem. Notice (0,0), the origin, exists at the top left of our coordinate frame. We are used to having the origin at the center of our frame of reference (Look at the image of the XY coordinate system above). How do we fix that?

Look at the following code:

```cpp
#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;


int main () {
    int width = 10;
    int height = 10;

    for (int i = 0; i < height; i++) {
        for(int j = 0; j < width; j++) {

            int x = j - (width/2);
            int y = i - (height/2);

            cout << "(" << setw(2) << x << ", " << setw(2) << y << ")" << " ";
        }
        cout << endl;
    }
}
```

We have created two additional variables in the loop (‘x’ and ‘y’) which depend on ‘j’ and ‘i’. The variable ‘x’ equals (j - (width/2)). Therefore, when j = 0 (first column), ‘x’ will be equal to -5 (we have x = 0 - (10/2)).

Similarly, when j = 9 (last column), ‘x’ will be equal to 4. A similar calculation can be done for ‘y’ depending on i. We have ‘x’ and ‘y’ ranging from -5 to 4. You can look at what this code prints out below:

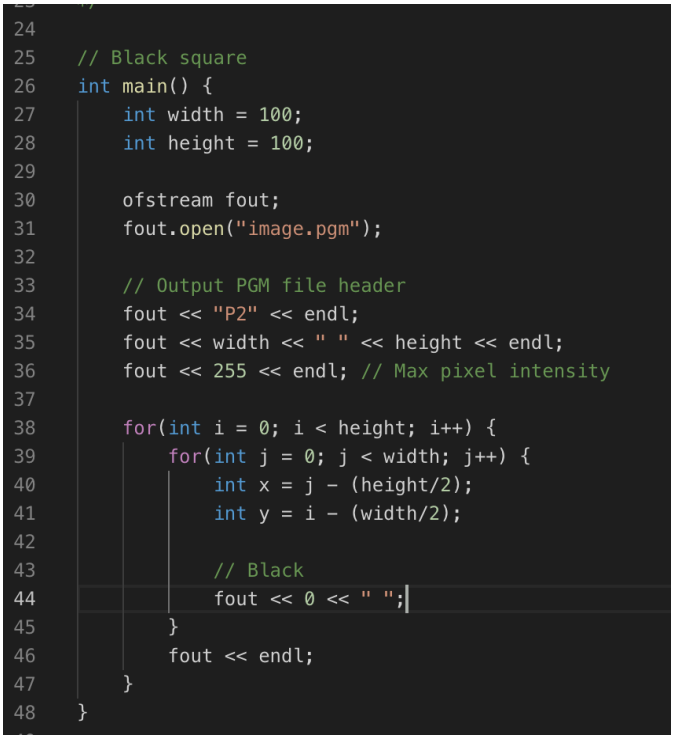
```
(−5, −5) (−4, −5) (−3, −5) (−2, −5) (−1, −5) ( 0, −5) ( 1, −5) ( 2, −5) ( 3, −5) ( 4, −5)
(−5, −4) (−4, −4) (−3, −4) (−2, −4) (−1, −4) ( 0, −4) ( 1, −4) ( 2, −4) ( 3, −4) ( 4, −4)
(−5, −3) (−4, −3) (−3, −3) (−2, −3) (−1, −3) ( 0, −3) ( 1, −3) ( 2, −3) ( 3, −3) ( 4, −3)
(−5, −2) (−4, −2) (−3, −2) (−2, −2) (−1, −2) ( 0, −2) ( 1, −2) ( 2, −2) ( 3, −2) ( 4, −2)
(−5, −1) (−4, −1) (−3, −1) (−2, −1) (−1, −1) ( 0, −1) ( 1, −1) ( 2, −1) ( 3, −1) ( 4, −1)
(−5,  0) (−4,  0) (−3,  0) (−2,  0) (−1,  0) ( 0,  0) ( 1,  0) ( 2,  0) ( 3,  0) ( 4,  0)
(−5,  1) (−4,  1) (−3,  1) (−2,  1) (−1,  1) ( 0,  1) ( 1,  1) ( 2,  1) ( 3,  1) ( 4,  1)
(−5,  2) (−4,  2) (−3,  2) (−2,  2) (−1,  2) ( 0,  2) ( 1,  2) ( 2,  2) ( 3,  2) ( 4,  2)
(−5,  3) (−4,  3) (−3,  3) (−2,  3) (−1,  3) ( 0,  3) ( 1,  3) ( 2,  3) ( 3,  3) ( 4,  3)
(−5,  4) (−4,  4) (−3,  4) (−2,  4) (−1,  4) ( 0,  4) ( 1,  4) ( 2,  4) ( 3,  4) ( 4,  4)
```

Notice where (0, 0) is now. It is at the center (well not quite, but that is because height and width are even numbers).

In summary, we have ‘transformed’ our coordinate system from having an origin at the top left to having an origin at the center of our image (we haven’t drawn an image yet, but we will). Henceforth, we will call the coordinate system with the origin at the center, ‘user coordinates’ (and the other one as ‘device coordinates’).

You might be wondering - why is doing something like this important? Aren’t coordinates just coordinates no matter where the origin is? That is true and many computer graphics systems don’t actually use coordinate systems with origin at the center, but, as we shall see, this choice of coordinate system, simplifies our code considerably and it also matches our intuition (most people would describe image location with respect to the center rather than the top left). You are free to use any coordinate system you like in this project, but the course staff will be using the system with the origin at the center throughout the course to explain things.

Phew! And I thought this project was about printing images! So let's actually get to it shall we?



```
24
25  // Black square
26  int main() {
27      int width = 100;
28      int height = 100;
29
30      ofstream fout;
31      fout.open("image.pgm");
32
33      // Output PGM file header
34      fout << "P2" << endl;
35      fout << width << " " << height << endl;
36      fout << 255 << endl; // Max pixel intensity
37
38      for(int i = 0; i < height; i++) {
39          for(int j = 0; j < width; j++) {
40              int x = j - (height/2);
41              int y = i - (width/2);
42
43              // Black
44              fout << 0 << " ";
45          }
46          fout << endl;
47      }
48  }
```

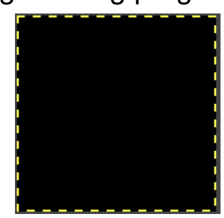
We are basically trying to print out a 100x100 image (notice the 'width' and 'height' variables). We open a file we want to write to. Let's call it 'image.pgm'. We use the extension '.pgm' because we want to print out a black and white image right now, to keep things simple. Then, we output to the file: the first three lines which need to be included in every '.pgm' file (code, dimensions and maximum pixel intensity). After that, we can start assigning values to every single pixel in our image (there will be 100x100 pixels in our image).

We just 'fout' the value 0 at every pixel location to get an all black image (0 represents black and 255 represents white, everything in between represents shades of gray). Notice there is a space after every pixel value. This is to delineate where one pixel intensity ends and where another begins.
Note: We are printing out a newline after every outer loop iteration. This is not necessary. You CAN have every pixel value on one line - left to right, top to bottom. We just have it in there for easier visualization/readability.

This is what ‘image.pgm’ looks like (this is a 20x20 example for easier visualization because 100x100 values would not fit in a single screen):

```
P2
20 20
255
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

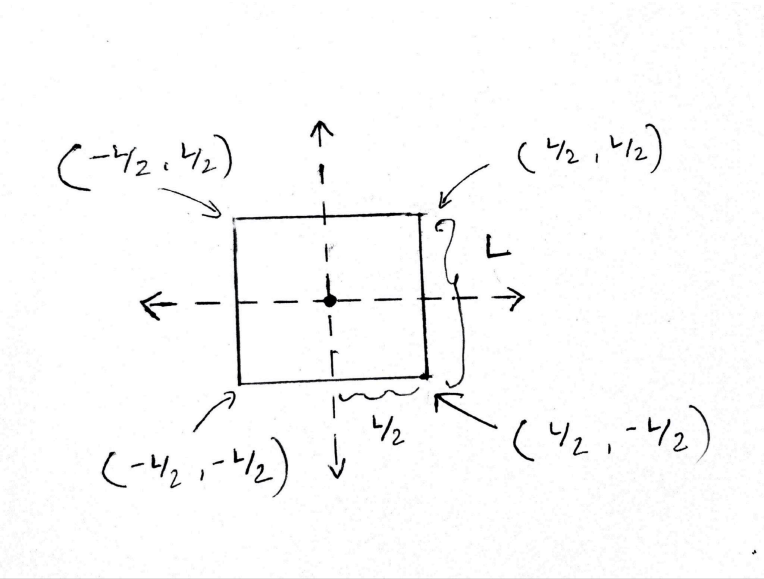
When we open the image file in an image editing program like GIMP. We see:



Yay! Our first image!
Printing out black images isn’t exactly super exciting so how can we do something more? Notice how in the above code we didn’t exactly use user coordinates. Why? Because we didn’t need them! We didn’t have to specify what to print and where since every single pixel had the same value. Let’s change that.

Suppose our goal is to print out a square. More specifically, a white square in a black background. We already have the black background part down (see above!). Let’s assume we want the square to be at the center of our image. Therefore, we now need to color pixels differently depending on where they are located in the image (think ‘if-condition’ on user coordinates).

Let’s review some geometry before we move forward:

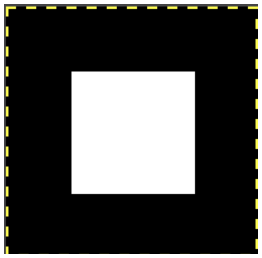


As we can see above, if we have a square centered at (0, 0), we can write the coordinates of the vertices of the square (with side length L) as (L/2, L/2), (L/2, -L/2), (-L/2, -L/2) and (-L/2, L/2).

How do we implement something like this in code?
Basically, we want to say that anything that lies within the square should be colored differently from points outside the square.
The points inside the square are those with: -L/2 < x < L/2 and -L/2 < y < L/2.

```
52  // White square
53  int main() {
54      int width = 100;
55      int height = 100;
56
57      int side_length = 50;
58
59      ofstream fout;
60      fout.open("image.pgm");
61
62      // Output PGM file header
63      fout << "P2" << endl;
64      fout << width << " " << height << endl;
65      fout << 255 << endl; // Max pixel intensity
66
67      for(int i = 0; i < height; i++) {
68          for(int j = 0; j < width; j++) {
69              int x = j - (width/2);
70              int y = i - (height/2);
71
72              if ((x > (-side_length)/2) && (x < (side_length)/2) && (y > (-side_length)/2) && (y < (side_length)/2)) {
73                  // Inside Square : White
74                  fout << 255 << " ";
75              }
76              else {
77                  // Outside Square : Black
78                  fout << 0 << " ";
79              }
80          }
81          fout << endl;
82      }
83  }
```

We’ve assumed a side of length 50. The output image we see:



Obviously, you can center the square anywhere in the image. If your square is centered at (h,w), then, you need to shift your coordinates by that amount.

Problem 1 - Printing Flags (50 points)

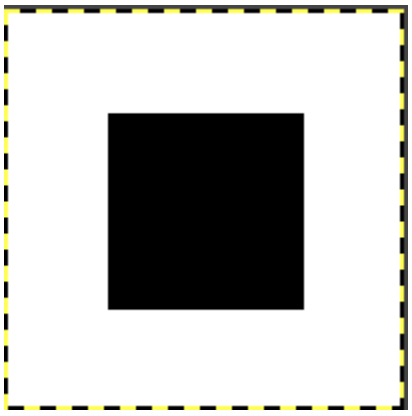
We will start with a problem which allows us to create images of flags. To build up complexity gently, we will break this problem down into subproblems.

Problem 1a (10 points)

Produce a 100x100 image with a black square at the center with a white background.
Specifications:

- Size of image: 100X100
- Size of square: 50x50
- Color of square: Black
- Color of background: White
- File format: PGM

Your output should look like this:



Problem 1b (10 points)

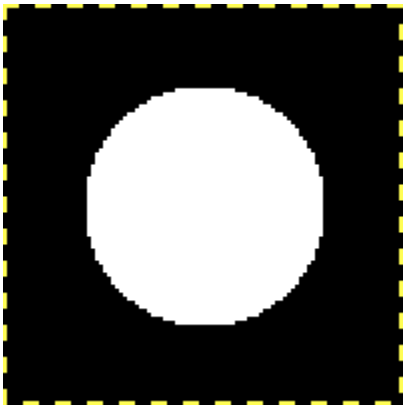
We can draw other shapes using this problem formulation as well.

Recall, the equation of a circle centered at the origin of radius 'r' is : $x^2 + y^2 = r^2$
Therefore, if we color all points inside differently from those outside, we can get the image of a circle!

Produce a 100x100 image with a white circle at the center with a black background.
Specifications:

- Size of image: 100X100
- Radius of circle: 30
- Color of circle: White
- Color of background: Black
- File format: PGM

Your output should look something like this:



Color Images:

There are only two differences when trying to print color images instead of black and white ones:

1. The file extension should be '.ppm' instead of '.pgm'
2. Instead of 1 pixel intensity value per pixel, we need to print out 3 values per pixel between 0 and 255 - Red, Blue, Green (referred to as RGB values).

(Almost) All colors can be described by different combinations of RGB values.

The most obvious combinations are:

Red = (255, 0, 0)
Green = (0, 255, 0)
Blue = (0, 0, 255)

Further, some other common color combinations would be:
White = (255, 255, 255)
Black = (0, 0, 0)
Yellow = (255, 255, 0)
Purple = (128, 0, 128)
Crimson = (220, 20, 60)

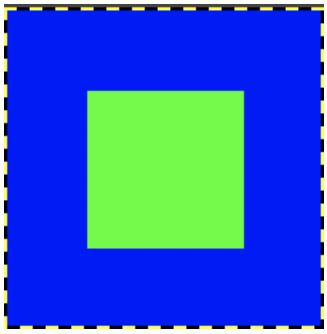
Let us try to produce a green square on a blue background. The code looks like the following:

```
122  // Green square on Blue background
123  int main() {
124      int width = 100;
125      int height = 100;
126
127      int side_length = 50;
128
129      ofstream fout;
130      fout.open("image.ppm");
131
132      // Output PPM file header
133      fout << "P3" << endl;
134      fout << width << " " << height << endl;
135      fout << 255 << endl; // Max pixel intensity
136
137      for(int i = 0; i < height; i++) {
138          for(int j = 0; j < width; j++) {
139              int x = j - (width/2);
140              int y = i - (height/2);
141
142              if ((x > (-side_length)/2) && (x < (side_length)/2) && (y > (-side_length)/2) && (y < (side_length)/2)) {
143                  // Inside Square : Green
144                  // RGB Values
145                  fout << 0 << " " << 255 << " " << 0 << " ";
146              }
147              else {
148                  // Outside Square : Blue
149                  // RGB Values
150                  fout << 0 << " " << 0 << " " << 255 << " ";
151              }
152          }
153          fout << endl;
154      }
155  }
```

The only changes we've made are:

1. Change the file extension to '.ppm' on line 130.
2. Change the image header code from 'P2' to 'P3' on line 133.
3. Change the 'fout' statements on lines 145 and 150 to output 3 values corresponding to the color we want instead of just one pixel value.

The output looks like:



Problem 1c (15 points)

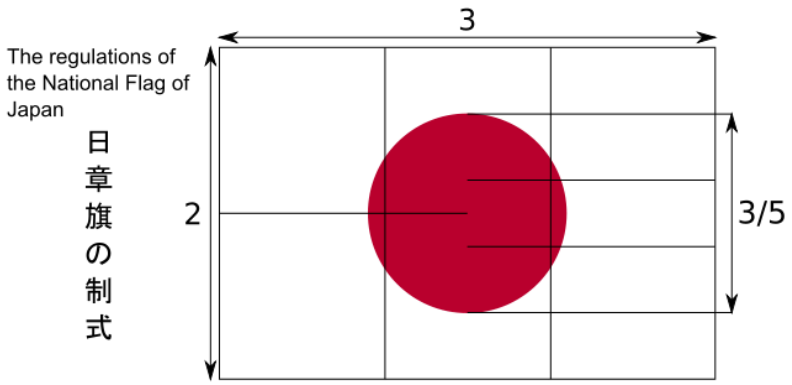
Produce a 100x100 image with a yellow circle at the center with a red background.
Specifications:

- Size of image: 100X100
- Radius of circle: 30
- Color of circle: Yellow
- Color of background: Red
- File format: PPM

Your output should look something like this:

Flags:

Now that we are able to generate basic shapes, we can recreate flags of several countries pretty easily.
Here is what the Japanese flag looks like:



寸法の割合及び日章の位置縦横の三分の二。日章直径縦の五分の三。
地白色；日章紅色。

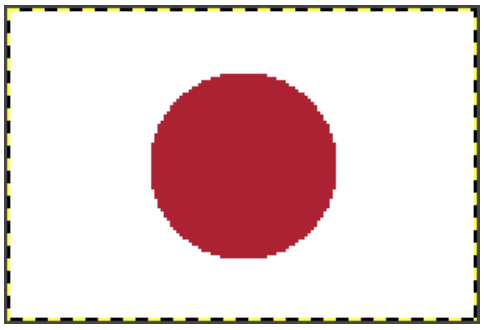
The area of the Japanese flag has a ratio of 2 units high and 3 units wide. The size of the sundisc is 3/5ths of the height of the flag. The color of the field is white and the sundisc is colored red.

源：国旗及び国歌に関する法律（１９９９）
Source: Law Concerning the National Flag and Anthem (1999)

To be as accurate as possible (and as respectful as possible), we need to obey the ratios of dimensions in these flags. As we can see above, the japanese flag (most flags really) is rectangular with a width:height ratio of 3:2 and the diameter of the circle in the center is ⅗ of the height.

Flag colors are also extremely specific.
The japanese flag uses:
Japan Red = (188, 0, 45)
White = (255, 255, 255)

Now, if we just change the dimensions of the code from problem 1c (and the colors!), we will get something like the image below. The dimensions used were 150x100 (3:2 ratio) and a radius of 30 (diameter = 60, which is ⅗ of height).



There is some pixelation in the image but we can fix that easily by just upping the size.

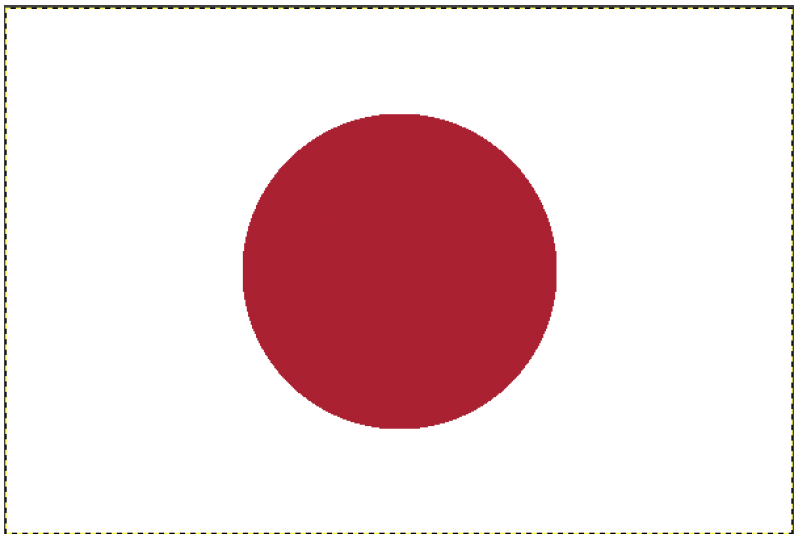
Problem 1d (15 points)

Produce a 600x400 image of the Japanese Flag.

Specifications:

- Size of image: 600X400
- Follow the color codes and dimensions listed above to get a final image compliant with Japanese flag specs.
- File format: PPM

Your output should look like:



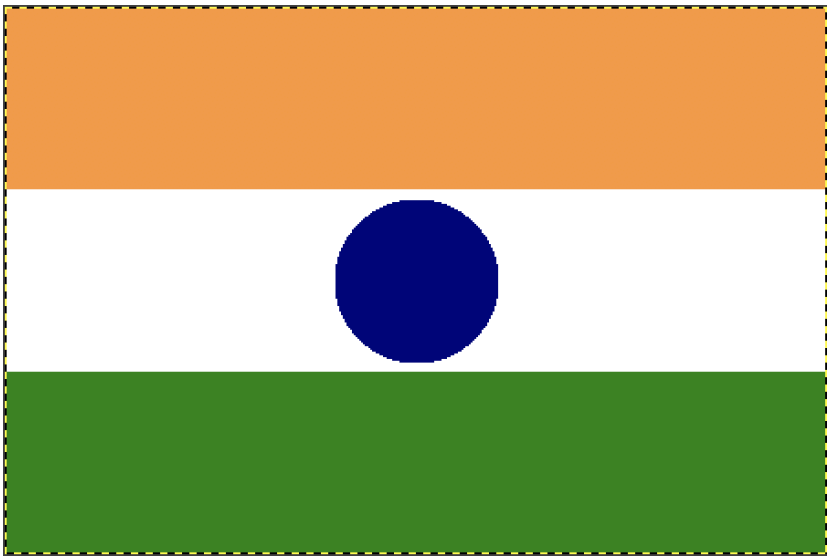
Extra Credit: Problem 1e (10 points)

Produce a 450x300 image of the indian flag (without the spokes).
Specifications:

- Size of image: 450x300
- Follow the color codes and dimensions listed below to get a final image compliant with Indian flag specs.
- File format: PPM

The width:height ratio is 3:2 (so 450:300 is ok). The diameter of the circle for a 450x300 is 90 (radius = 45).
There are 3 stripes in the flag which are of equal height (so 100 in our case)
Colors:
Navy Blue (Circle) = (0, 0, 128)
Saffron (First Stripe) = (255, 153, 51)
White (Second Stripe) = (255, 255, 255)
India Green (Third Stripe) = (19, 136, 8)

Your final image should look like:



Extra Extra Credit: Problem 1f (10 points)

Produce an image of a flag of your choice (real or fictional).
To get credit, your flag must have more than just squares and rectangles and must be at least equivalent in difficulty to the Indian flag. It should have more than one type of geometrical object (circles, squares, triangles etc).
The final decision to award credit lies with the course staff.
Before creating your image, you can get your idea approved from the instructor so that you don't waste time.

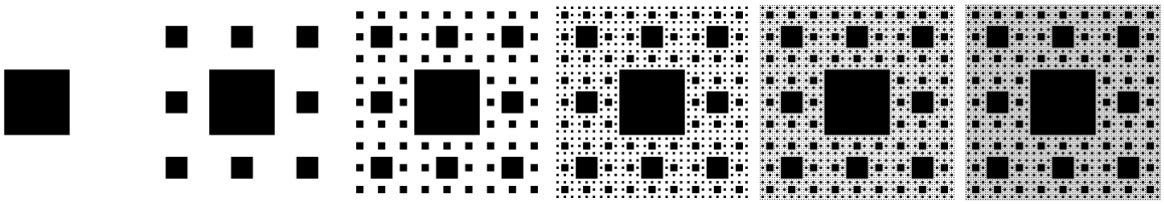
Problem 2 (75 points)

In this problem, we will be creating an image of a ‘fractal’. Fractals are self repeating patterns obtained by applying a mathematical formula over and over again. You can read more here.

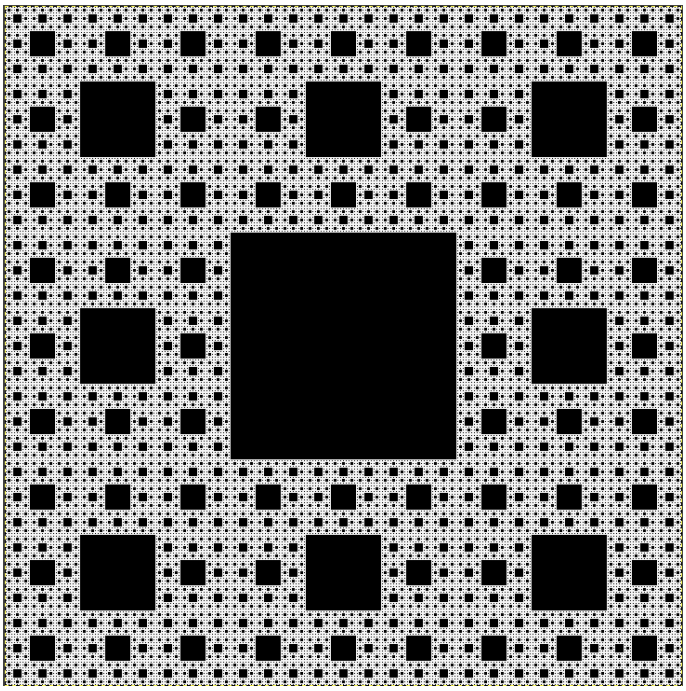
The specific fractal we are creating in this problem resembles a 2D carpet design.

The construction of the carpet begins with a square. The square is cut into 9 congruent subsquares in a 3-by-3 grid, and the central subsquare is removed. The same procedure is then applied recursively to the remaining 8 subsquares, *ad infinitum*.

In the image below, we start with a white square and ‘remove’ the part in between (i.e color it with black) to get a carpet of order 1. Then, we apply the same process to each of the 8 non-empty sub-squares to get the pattern for order 2 and so on.



Your job is to write a C++ program that creates the carpet of order 7 and stores the result in a ‘.pgm’ file (black and white).
Your program needs to take input from the user. The user will enter the ‘order’ of the carpet to be produced. You can pass these inputs either as command line arguments or via cin.
The output should look like this:



We would recommend solving this problem via a recursive formulation and using a 2D array to keep track of pixel values. Of course, it is possible to solve this without an array or recursion. The recursive strategy involves applying the function to 8 sub-squares and painting the center sub-square empty (black in our case).

Problem 2a (75 points)

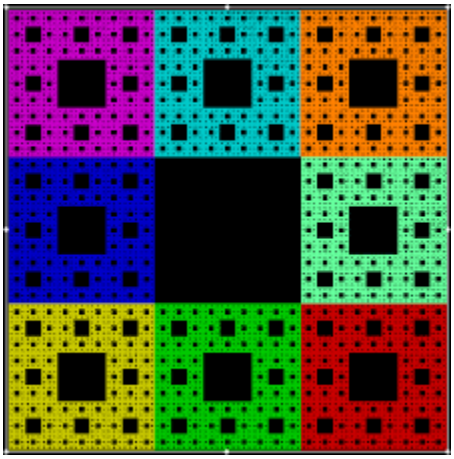
Produce an image of the above described carpet of order 7 where empty space is colored with black while all other space is white.

Specification:

- Image Size: Can depend on the order of the carpet. (You are allowed to submit a .pgm file of any size, might be easier to use a power of 3)
- Color of Background: Black
- Color of Squares: White
- File type: PGM

Extra Credit: Problem 2b (10 points)

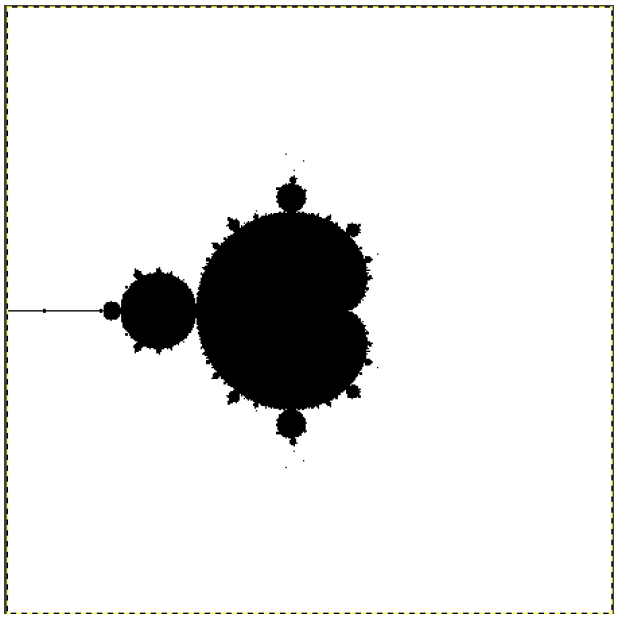
Modify your code to output a carpet such that sub-squares at the last level are printed with different colors. See image below:



You may choose any colors of your choice as long as they are distinct and distinguishable. Submission format is the same as above.

Problem 3 (75 points)

Write a program to create the image of the fractal pictured below.



Complex Numbers

The above fractal uses complex numbers (see tutorial here if you have never seen these before).

To use complex numbers in C++, you need to include the <complex> header.
To declare a variable of type complex number of doubles called ‘v’, we write:
complex<double> v(0.0, 0.0); // This variable is initialized to v = 0.0 + i*0.0

Some more examples:
complex<double> x(1.0, 3.0) // This variable is initialized to x = 1.0 + i*3.0

To get the real and imaginary parts of a complex number variable (called say c), we do:
double real_part = real(c);
double imaginary_part = imag(c);

To get the absolute value of a complex number c:
double abs_value = sqrt(real(c) * real(c) + imag(c) * imag(c))

More details here. We don’t need a lot of knowledge about complex numbers, just how to initialize them and get their real and imaginary values (and by extension, find their absolute value).

How to calculate the inside of the set shown above?

To get the required set, you start by picking a complex number c. Then, we repeatedly apply the equation: $z_{n+1} = z^2_n + c$ with $z_0 = 0$ as the starting point. This produces a sequence of "iterates" (i.e. z_i's).

For some values of c, the iterates keep getting bigger and bigger until they reach infinity. These iterates are unbounded: for any size you pick, there'll be some later iterate that is bigger still.

For other values of c, the iterates remain bounded. Mathematically, the set we're looking for consists of the c values that remain bounded. In the image above, the starting points with iterates which are bounded are colored differently from those whose iterates are unbounded.

It's possible to show that if any iterate gets bigger than 2, then it'll go to infinity eventually (we will not prove this statement). So when we are applying $z_{n+1} = z^2_n + c$ repeatedly, the points which are inside the set, have iterates who have absolute value <= 2.

Therefore, one way to check that a point is not in the set is to keep iterating and see if it "escapes" (absolute value becomes greater than 2 at any point of time).

But for any particular c value, we don't know for sure if it is in the set or not unless we run $z_{n+1} = z_n + c$ infinite number of times.

So that we don't have to wait beyond the end of time to see our picture of the set, we choose to give up after a certain maximum iteration count. If the point still hasn't 'escaped', it might or might not be in the set.

But if the point escapes, we can be sure the point is not in the set. We can draw these two cases with two different colors, giving us the picture above.

Your solution to this problem will use the same concepts as problem 1 (user coordinates, double for-loop etc). The code structure will remain remarkably similar.

However, two things change from our formulation in Problem 1:

User coordinate scaling:

Note that the escape radius is always of value 2 (due to a mathematical theorem). Therefore, if we tried to create an image of size 300x300, with a figure of radius 2, it is barely going to be visible. This gets worse as the size of dimensions increases.
How to fix this? We can scale user coordinates so that they lie in the range (-1.0, 1.0) like so:

```
for (int i = 0; i < h; i++) {
    for(int j = 0; j< w; j++){
        double y = (i - h/2.0) / (h/2.0);
        double x = (j - w/2.0) / (w/2.0);
```

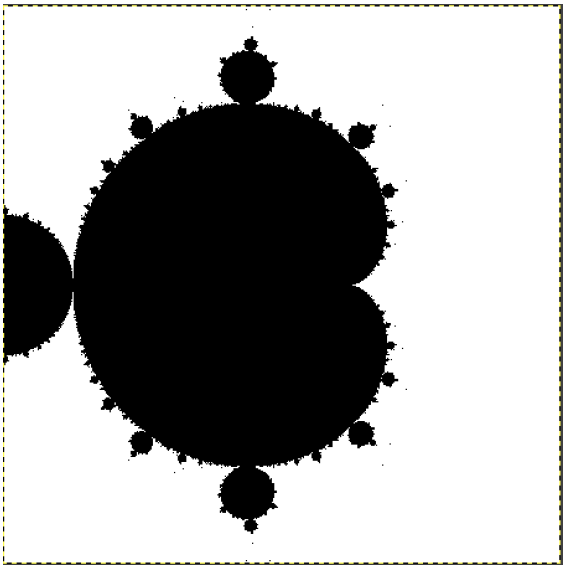
Convince yourself ‘x’ and ‘y’ are now in the range -1.0 to 1.0. (Use pen and paper if needed!). Our coordinates are of type double now. This is natural. Even in high school geometry, X-Y coordinates had fractional parts.

Initial Value c:

The next thing we need to figure out is: how to fix the initial value c?
The most natural thing to do from the description so far is do:

```
complex<double> c = complex<double>(x, y);
```

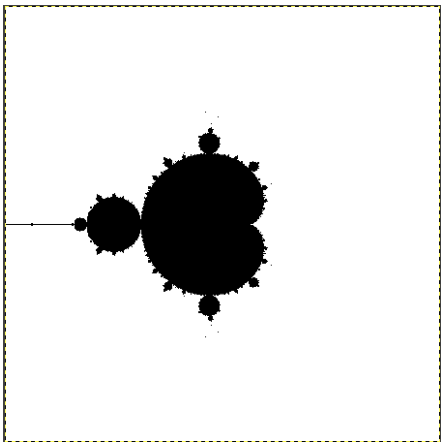
Basically, we are setting the initial value c to be (x, y) in our usual double for-loop. This produces the image below. Note you might be able to produce the correct image using the above setting depending on what you do elsewhere.



So we seem to be producing the right shape, it's just that the shape is too large to fit into our image (Width x Height). Hence, just like user coordinates, we scale the value like below.

```
complex<double> c = zoom * complex<double>(x, y);
```

If we set zoom = 2 (just an example), we get the image we're looking for

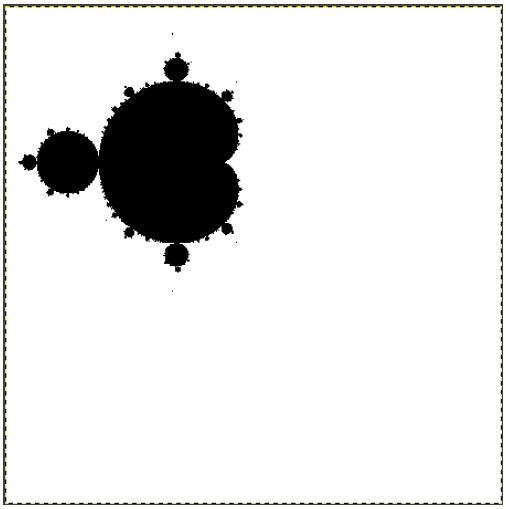


Now, we are essentially done with the problem but what if we want to move the set around? We can set the value of c like the following:

```
complex<double> c = center + zoom * complex<double>(x, y);
```

Here 'center' is also a complex number whose initial value would be set to (0.0, 0.0).

If we change the 'center' to be (0.5, 0.75), we get:



Notice how the figure moved towards the top left.

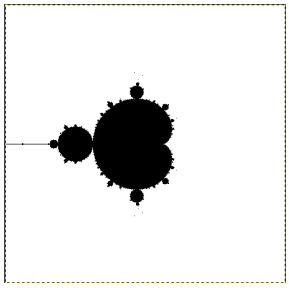
Finally, note, this problem involves applying an equation to itself repeatedly (think recursion). We can implement this problem recursively as well, but the above description assumes an

iterative solution. You can choose either. If you decide to use recursion, you might want to use a 2D array to make printing to a file easier.

Problem 3a (50 points)

Produce an image of the above described set of size 500x500
Specification:
Size of image: 500X500
Center: (0, 0)
Zoom (if needed): 2.0
Color of points outside set: White
Color of points inside set: Black
File format: PGM

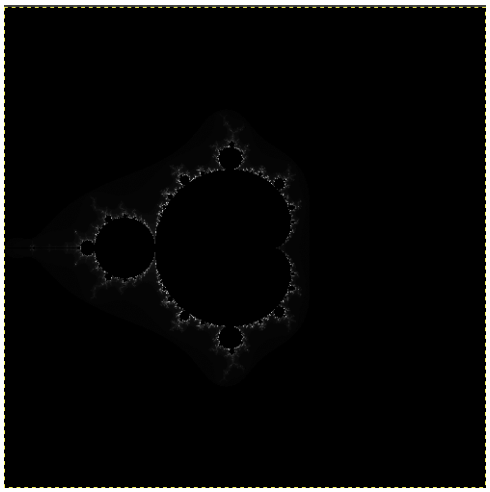
Output should look like this:



Continuous Coloring

We know that points painted white in the above picture are ‘escaping’ (i.e. their iterates go beyond 2 when applying the master equation). We can keep track of how many iterations it takes for each point to escape and then use that to color that point. Essentially, we convert a loop iteration value (where we break) to a value between 0 and 255 (color).

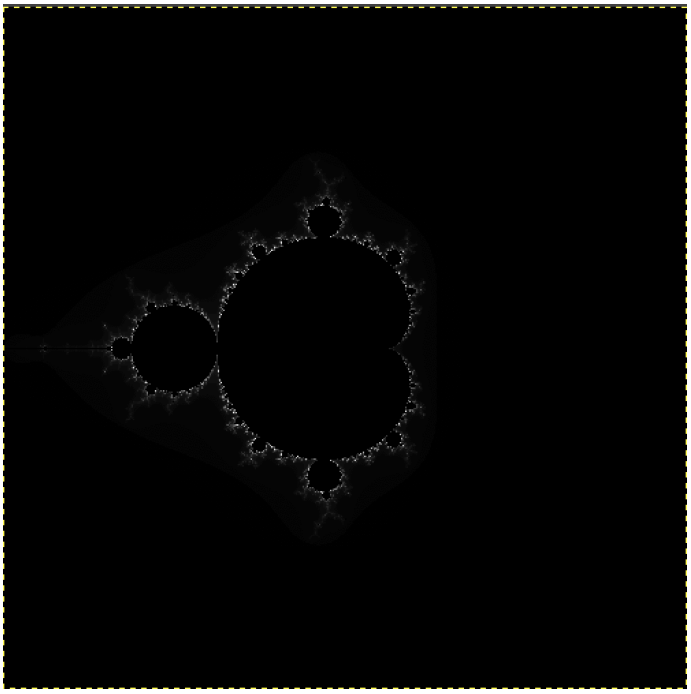
We just need to scale the iteration number to be between 0 and 255 since that is the range of pixel intensities.



Problem 3b (15 points)

Produce an image of the fractal of size 500x500 with continuous coloring (black and white)
Specification (Same as 3a):
Size of image: 500X500
Center: (0, 0)
Zoom (if needed): 2.0
Color of outside set: White
Color of points inside set: Black
File format: PGM

Output should look like the following:



Notice how black turns to shades of gray around the edge of the set.

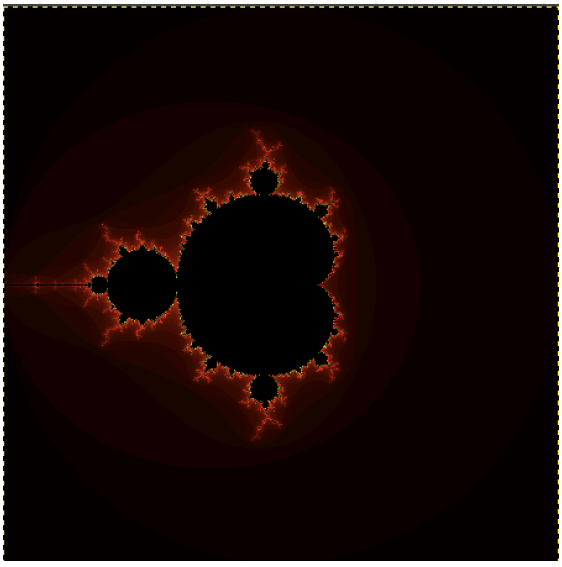
Colored Fractal

Now, we create a colored version of the set. From 3b, we just need to print out 3 values per pixel instead of 1 value. You can choose to generate these 3 values in any way you wish. One particular scheme could be:

```
R = bw_pixel * 8;   // where bw_pixel is a value you used in 3b (between 0 and 255)
G = bw_pixel;
B = bw_pixel / 8;
```

Another scheme could be to keep two of the channels constant and just vary just one channel out of RGB. This is your chance to experiment!

An example output using the first scheme is:



Problem 3c (10 points)

Produce a **colored** image of the fractal of size 500x500 with continuous coloring

Specification:

- Size of image: 500X500
- Center: (0, 0)
- Zoom (if needed): 2.0
- Color of outside set: Color scheme of your choice
- Color of points inside set: Black
- File format: PPM

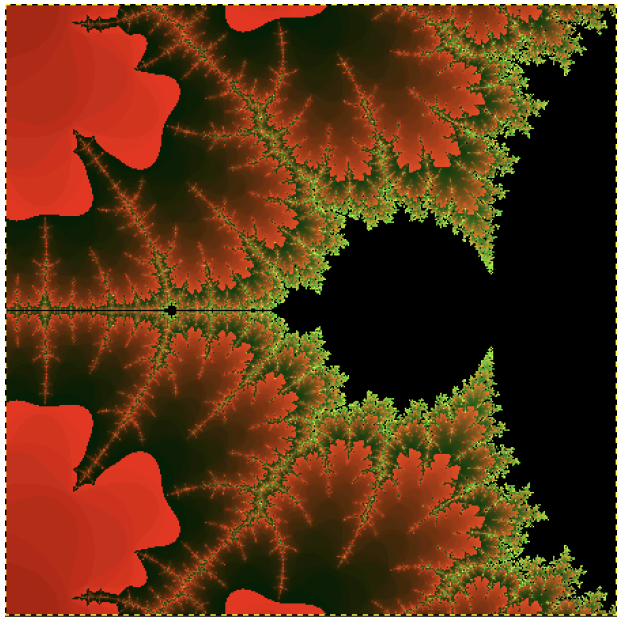
See picture above for an example of the output.

Extra Credit - Problem 3d (20 points)

This is your opportunity to create an image you can hang in your bedroom. You already know a few ways in which you can change your image:

1. Color scheme
2. Center
3. Zoom

Here is an example image with center (-1.4, 0.0) and zoom = 0.01



You can run a loop over a range of ‘center’ and ‘zoom’ values, produce a bunch of images and then choose the most aesthetically pleasing one.

You might be able to see some pixelation in the above picture. More generally, as you zoom further and further, you will run into this issue. This is caused by the limited precision of the type double (there’s only so many values a double variable can represent!). One workaround is to use the type ‘long double’ instead of ‘double’ for added precision.

One more idea: We have only colored points outside the set, can we color points inside as well?

Everyone will be allowed to enter their best picture into a contest where others will get to vote on everyone else’s submissions. The post with the most votes will get extra credit. If the course staff thinks there are more than one deserving submissions, we will award them extra credit too.

Note since the requirements for this component are vague, you may receive partial extra credit depending on how good your submission is. For exceptional submissions, we will consider even more credit.

Submission Instructions:

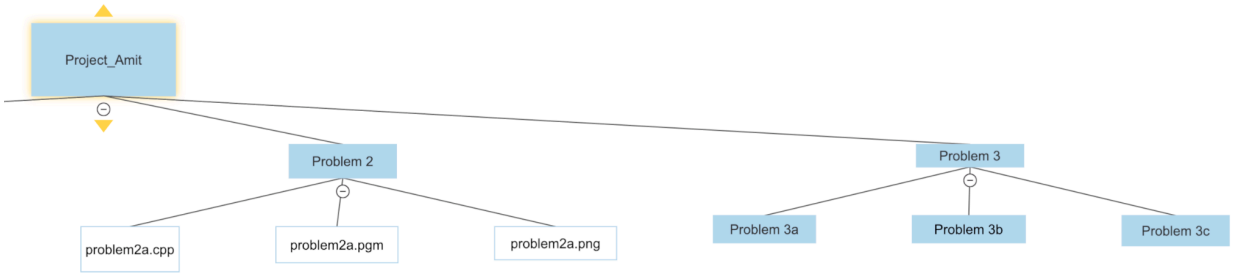
You need to submit your project in a zip file on Canvas. **The project is due at midnight on Tuesday, July 23**. These problems are hard and will require a decent amount of time so please get started ASAP. You should attempt problem 1 first and then you can attempt problems 2 and 3 in any order.

The project carries a total of 200 points (50 + 75 + 75) with lots of potential extra credit (50 points!).

You need to include all your code and image files (.ppm and .pgm). You should also include '.png' versions of all images you generate. You can use an online converter such as this (google for others) to convert your files.

Files associated with each problem (and subproblem) should be in separate folders. So, your zip file should have three main folders inside it (with additional sub-folders further inside). You should have code associated with each image generated in a separate '.cpp' file along with a '.pgm'/'.ppm' and '.png' file.

Project structure :



The above image shows just a small part of what your project structure should look like.
Note if you have completed extra credit problems, you will have more folders.
Not having the right folder structure would automatically be -10% off (i.e. 20 points off)

Your submissions will be graded on code quality as well as the final image produced. Code should be written with consistent indentation, descriptive variable names and comments explaining the code briefly. Follow good coding practices highlighted in class.
Code quality will be 10% of your final grade.

There is no partial credit for any question. You either produce an image upto specification or you get no credit for that problem.

There is no need for a write up. But if you want to explain what you tried for the extra credit problem feel free to include a short note. Finally, **there will be interview grading for the project conducted after the submission deadline. This is an individual project so any**

plagiarism will not be tolerated. You will fail this course if we find any instance of cheating.