3. Mandelbrot Set

Sample Output

You can compare your graphical output against the image files below using the online image compare tool linked at the top of the page.

Please note that due to minor differences in pixel arithmetic, rounding, etc., it is very likely that your output will not perfectly match ours. It is okay if your image has non-zero numbers of pixel differences from our expected output, so long as the images look essentially the same to the naked eye when you switch between them.

x=50, y=60, size=400, iterations=100, color=orange, range=(-3-2i)...(3+2i)

x=100, y=20, size=400, iterations=50, color=blue, range=(-2-1i)...(1+1i)

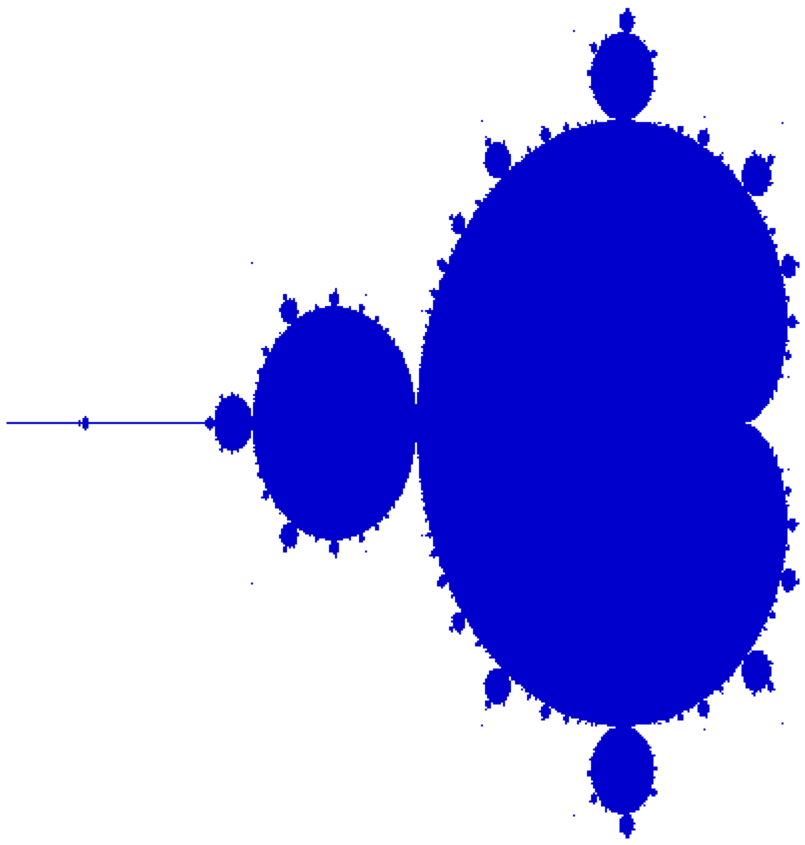
x=100, y=20, size=500, iterations=50, color=blue, range=(-2.5-1i)...(1+1i)

x=100, y=100, size=300, iterations=8, color=green, range=(-2-1i)...(1+1i)

x=150, y=90, size=200, iterations=80, color=red, range=(-2.5-1.5i)...(2+1.5i)

Another fascinating fractal is the "Mandelbrot Set", in tribute to Benoit Mandelbrot, a mathematician who investigated this phenomenon. The Mandelbrot Set is recursively defined as the set of complex numbers, *c*, for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z=0$. In other words, a complex number, *c*, is in the Mandelbrot Set if it does not grow without bounds when the recursive definition is applied. The complex number is plotted as the real component along the x-axis and the imaginary component along the y-axis.

For example, the complex number 0.2 + 0.5*i* is in the Mandelbrot Set, and therefore, the point (0.2, 0.5) would be plotted on an x-y plane. For this problem, you will map coordinates on an image (via a Grid) to the appropriate complex number plane (note that for this problem, we have abstracted away many details to make the coding easier).



Formally, the Mandelbrot set is the set of values of c in the complex plane that is bounded by the following recursive definition:

$$z_{n+1} = z_n^{\,2} + c$$

$$z_0 = 0, \quad n \rightarrow \infty$$

In order to test if a number, c, is in the Mandelbrot Set, we recursively update z until either of the following conditions are met:

A. The absolute value of z becomes greater than 4 (it is diverging), at which point we determine that c is not in the Mandelbrot Set; or,
B. We exhaust a number of iterations, defined by a parameter (usually 200 is sufficient for the basic Mandelbrot Set, but this is a parameter you will be able to adjust), at which point we declare that c is in the Mandelbrot Set.

Because the Mandelbrot Set utilizes complex numbers, we have provided you with a **`Complex`** class that contains the following member functions (**`#include "complex.h"`** to use it):

Function	Description
`Complex(double *a*, double *b*)`	Constructor that creates a complex number in the form a + b*i*
`*cpx*.abs()`	returns the absolute value of the number (a double)
`*cpx*.real()`	returns the real part of the complex number
`*cpx*.imag()`	returns the coefficient of the imaginary part of the complex number
`cpx1 + cpx2`	returns a complex number that is the addition of two complex numbers
`cpx1 * cpx2`	returns a complex number that is the product of two complex numbers
`ostr << c`	prints a complex number to an output stream

Implementation Details

Your drawMandelbrotSet function will be passed in a number of parameters that you will use to determine which pixels in the window will be drawn in the Mandelbrot Set. The details of the parameters are described below. Your solution may use loops to traverse the grid, but you must use recursion to determine if a particular point exists in the Mandelbrot Set.

```
void drawMandelbrotSet(GWindow& window, double leftX, double topY, double size,
                       const Complex& min, const Complex& max,
                       int iterations, int color);
```

The **`window`** parameter is the same as in the other parts. In the starter code, we have already created an image that will display the fractal, and created a grid based on that image. The grid is composed of individual pixel values (ints) that you can change based on whether or not a pixel is "in the Mandelbrot Set" or not. How to color the pixels is described below.

The **`leftX`** and **`topY`** parameters are the (x, y) coordinates of the top-left corner of the overall area in which you should draw your Mandelbrot Set. The **`size`** parameter describes the width and height in pixel of the area in which you should draw your Mandelbrot Set. (These can be set by clicking and dragging on the GUI or by entering them in the text boxes.)

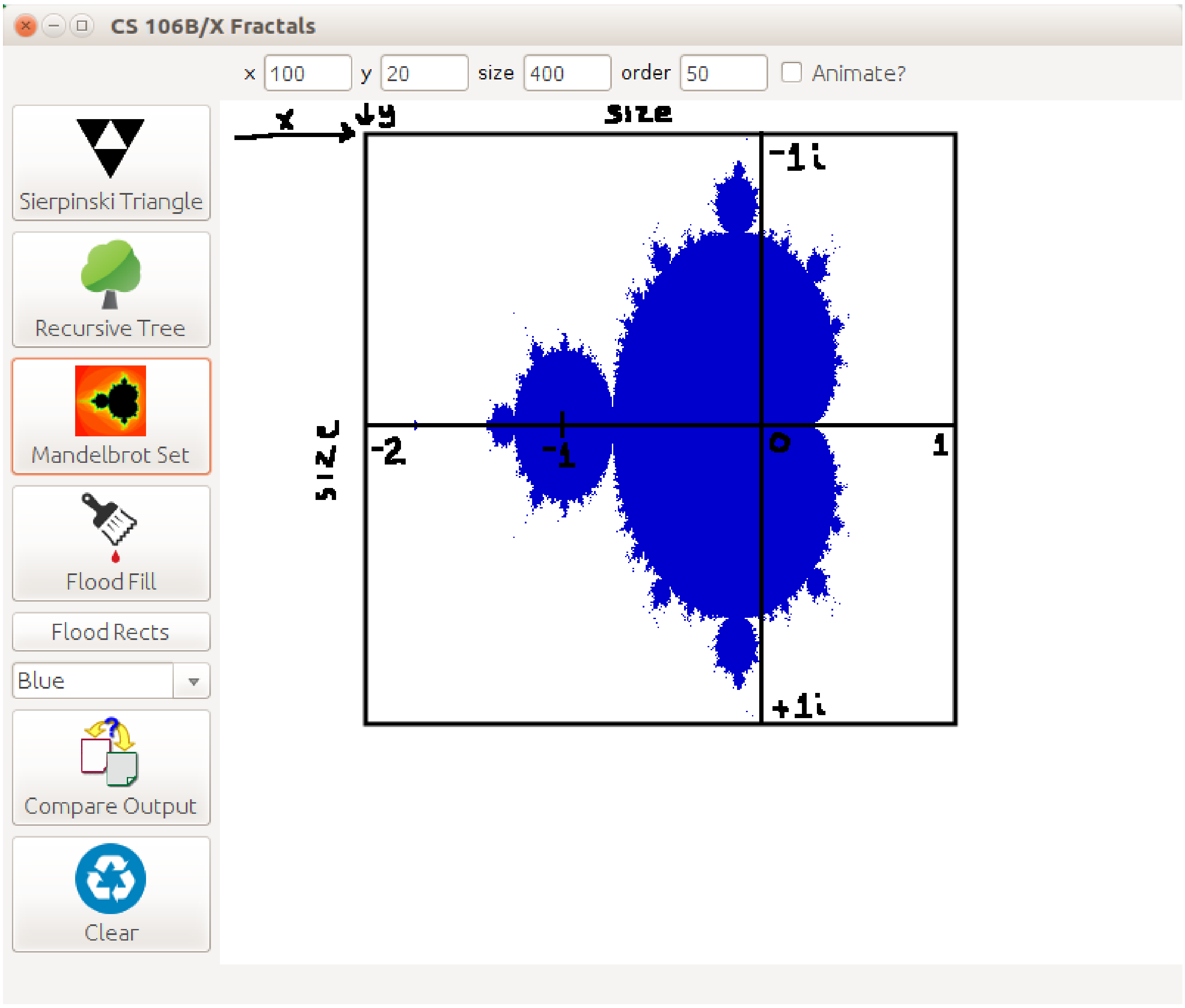
The **`min`** and **`max`** parameters describe the range of complex numbers you should examine. Your code will need to compute a mapping between the x/y pixel range on the screen and the range of complex numbers provided. See the diagram below for more details about performing this mapping. (These are set when you click the Mandelbrot button; the GUI will prompt you for their values.)

The **`iterations`** parameter is the maximum number of iterations your algorithm should try on each pixel to decide whether that pixel's corresponding complex number diverges. If the number diverges, you should leave its corresponding pixel untouched. If the number does not diverge after that many iterations, it is in the Mandelbrot Set. (The max number of iterations is set by modifying the "Order" parameter in the GUI. The default of 1 iteration is far too few; you'll need to set a higher number, such as 50 or 200, to see a sharp image.)

The **`color`** parameter is parameter is an integer representing an RGB color that you should use to color any pixels that do not diverge in your Mandelbrot Set. Just set the pixel's value in the grid to that color. The grid's pixels are colored white by default.

The following diagram summarizes the mapping you must perform from onscreen x/y coordinates to the complex number space. The figure below is drawn with a top-left pixel value of (x=100, y=20) and a size of 400. The complex numbers min and max are specified to be (-2-i) and (1+i) respectively. Recall that in this exercise the complex numbers are mapped to the x/y coordinate space with their real components on the x-axis and the imaginary i components on the y-axis. This means that for this specific example, your code would need to map the space [-2 .. 1] onto the x-coordinates [100 .. 500], and map the space [-1i .. 1i] onto the y-coordinates [20 .. 420].

So, for example, the pixel (x=100, y=20) represents complex number (-2 -i). Its right-side neighbor, (x=101, y=20), represents roughly (-1.9925 -i), and the next pixel to the right of that, (x=102, y=20), represents roughly (-1.985 -i). Its downward neighbor, (x=100, y=21), represents roughly (-2 -0.995i), and the next pixel below that, (x=100, y=22), represents roughly (-2 -0.99i). The origin of the complex number space, (0+0i), would be at a pixel position of roughly (x=367, y=220). We continue in this fashion until the bottom-right corner of the x/y region, (500, 420), which represents complex number (1+i).



Mapping coordinates: (100, 20)x400 => (-2-i to 1+i)

Note that your **`drawMandelbrotSet`** function itself is not directly recursive. You can use loops to iterate over the the range of pixels of interest in the window. But the code that examines each pixel to determine whether or not it diverges must be recursive for full credit and should not use loops. You should probably put such code into a helper function that is called by **`drawMandelbrotSet`**.

Debugging: It can be hard to debug this problem because if your calculations are not quite right, you may see no graphical output at all. If you are not seeing any helpful output, try inserting a **`cout`** statement for each pixel to verify whether you have properly figured out its corresponding complex number. Also try printing out the number of iterations until each complex number diverges, to help you see whether you are computing that value properly.

4. Plasma Fractals

A Plasma fractal, also known as random midpoint displacement, is a recursive algorithm used to generate superbly cool-looking psychadelic gradients. Take a look: