**ECE2036** **Fall Semester, 2012**

# Mandelbrot Set

*Assigned: Nov 16, 2012* *Due: Nov 30, 2012, 11:59pm*

**Introduction** In this assignment we will use the *QDisplay* graphics API and the *gthreads* threading library to compute and display a visual image of the *Mandelbrot Set*. Once the set is displayed, we will use the mouse to select a square region from the displayed set, and recompute the set using only the selected region of the complex plane. Details of the math to compute the *Mandelbrot Set* and how to use multiple threads are all given in the paragraphs below.

**The Mandelbrot Set** The *Mandelbrot Set* is defined as the set of points in the complex plane that satisfy

$$M = \{c \in C | \lim_{n\to\infty} Z_n \neq \infty\}$$

where

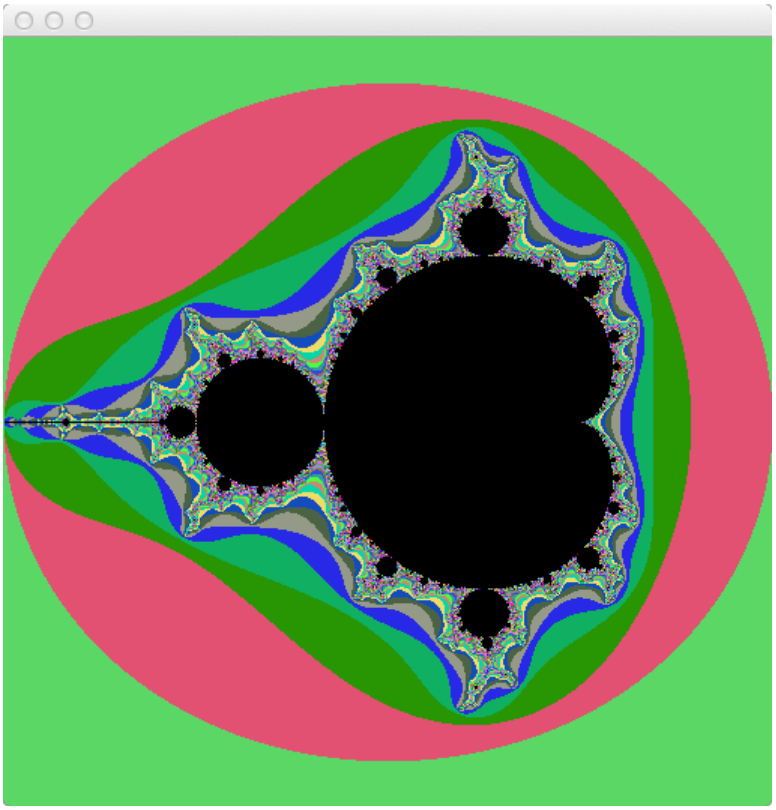
$M$ is the set of all complex numbers **in** the *Mandelbrot Set*.
$C$ is the set of all complex numbers in the complex plane,
$Z_0 = c$,
$Z_{n+1} = Z_n^2 + c$

From this description it appears we have to iterate $n$ over all values from 0 to $\infty$, which would take quite a bit of computation. Luckily, we can make two simplifying assumptions.

1. If the magnitude of any $Z_n$ is greater than $2.0$, then it can be proved that Z will eventually reach infinity, so if we ever get a $Z_n$ for which the magnitude is greater then $2.0$ we can stop iterating and claim that the point $c$ is *not* in the *Mandelbrot Set*.

2. If we have iterated 2,000 times and still not found a $Z_n$ with a magnitude greater than $2.0$, we can again stop iterating, but this time claim the $c$ *is* in the *Mandelbrot Set*.

**Displaying the *Mandelbrot Set*** Start by creating a display window of 512 by 512 pixels. Then compute and display the *Mandelbrot Set*. Unfortunately, it appears in the above discussion of the math to compute the *Mandelbrot Set*, that we have to iterate all possible values of $c \in C$. Again we are lucky that we can limit the possible range of the $c$ value as follows:
$(-2.0, -1.5) < c < (1.0, 1.5)$
This still seems to be an infinite number of possible $c$ values. To again avoid infinite processing, simply select 512 discrete $c$ values in both the real and imaginary ranges above, resulting in 512 * 512 total unique $c$ values. For each $c$, simply iterate the $Z$ values as shown above. If the $c$ is in the *Mandelbrot Set*, display a black pixel at the appropriate point in the display window, otherwise display a colored pixel. The correct color for each of the iteration counts is given in the skeleton code provided. Your result should be identical to that below.



**Selecting the pixel colors** As discussed above, we iterate each point $Z$ using the equations shown above. If the magnitude of $Z_n$ is greater than 2.0 at any point during the iteration, can immediately claim the point is **NOT** in the *Mandelbrot Set*. The color to use for that point is a function of the iteration count where the $Z_n > 2.0$ was detected. The skeleton code has a pre-defined array called `pixelColors`, indexed by the iteration count.

If you iterate 2000 times and still have not found a $Z_n > 2.0$, then paint the pixel black.

**The *QDisplay* Graphics API** We will again use the *QDisplay* graphics API that we used in the prior PathLoss assignment, with some minor differences. In the PathLoss assignments, we were not interested in either mouse clicks or keyboard entries, but we need both of those for this assignment. Using the *QDisplay* mouse functions (described below) enable the "selection" of a square region in the displayed image. While the mouse is moving display a red square to indicate the selected region. When the mouse button is released re-compute the image using the bounds selected with the mouse **IMPORTANT. The selected region should always be square, not rectangular.** When outlining the selected region in red, adjust the mouse x, y location ot make the region square even if it is not.

The *QDisplay* class has several *pure virtual functions* as follows:

```
// Pure virtual functions for subclasses
virtual void MousePressed (int x, int y) = 0;
virtual void MouseMoved   (int x, int y) = 0;
virtual void MouseReleased(int x, int y) = 0;
virtual void KeyPressed   (char ch)      = 0;
```

This means you must implement a subclass of *QDisplay*. You can call this subclass anything you want, but *MyDisplay* seems reasonable. The mouse and keyboard functions above are self-explanatory, but note that the `MouseMoved` function is only called when the mouse moves with the left button depressed.

For the KeyPressed implementation, if the user presses lower case 'q', (quit) simply exit the program. **10 point bonus.** Implement a "history" of the displayed images, and go *back* if the user persses lower case 'b'.

**Using threads** If you simply implement the steps above, you will find it takes way too long to compute and display the *Mandelbrot Set*. This is unacceptably long, so we will solve the problem by throwing more hardware at the problem. To speed up the computation and display of the *Mandelbrot Set*, simply use 16 threads each of which calculate the *Mandelbrot Set* iteration values for a unique part of the image. This is nearly identical to the approach we used in the PathLoss assignment.

**Copying the Project Skeletons**

1. Log into `jinx-login.cc` using `ssh` and your prism log-in name.
2. Copy the files from the ECE2036 user account using the following command:

   ```
   /usr/bin/rsync -avu /nethome/ECE2036/MBSet .
   ```

   Be sure to notice the period at the end of the above command.
3. Change your working directory to `MBSet`

   ```
   cd MBSet
   ```

4. Copy the provided `MBSet-skeleton.cc` to `MBSet.cc` as follows:

   ```
   cp MBSet-skeleton.cc MBSet.cc
   ```

5. Then edit `MBSet.cc` to create your code for the project.

**Turning in your Project** Turn in your assignment by the due date using the `riley-turnin` procedure as always.