Mandelbrot Set

Introduction

The Mandelbrot set is a mathematical set defined from a function, similar to and related to the Julia set. However, the Mandelbrot set is more widely known.

Finding Points in the Mandelbrot Set

For our purposes, this is a good enough definition of the Mandelbrot set for a function \((x', y') = f(x, y, a, b)\), where \((x, y), (a, b)\) and \((x', y')\) are the coordinates of points in the 2 dimensional plane.

Take a point \((a, b)\). Using \(x_0 = 0, y_0 = 0, a, b\) as input to \(f(x, y, a, b)\), we receive a new point from the function.

\[
x_0, y_0 = f(x_0, y_0, a, b)
\]

Repeat that process by using the output of the previous call of the function as the input to the current call. We could repeat up to \(n\) times:

\[
x_1, y_1 = f(x_0, y_0, a, b) \\
x_2, y_2 = f(x_1, y_1, a, b) \\
x_3, y_3 = f(x_2, y_2, a, b) \\
x_4, y_4 = f(x_3, y_3, a, b) \\
...  \\
x_{n-1}, y_{n-1} = f(x_{n-2}, y_{n-2}, a, b) \\
x_n, y_n = f(x_{n-1}, y_{n-1}, a, b)
\]

For large enough values of \(n\), the resulting point at \(x_n, y_n\) will fall into one of two categories:

1- The point will still be close to the origin \((0,0)\) of the plane. We define close to mean a distance less than or equal to 2.

2- The point will be far from the origin. We define far to mean a distance greater than 2.

If the point is close to the origin, it is part of the Mandelbrot set for \(f(x,y,a,b)\). Otherwise it is not part of the Mandelbrot set. A point that is not part of the Mandelbrot set “escapes” at the first iteration where the new \((x,y)\) point is a distance of more than 2 from the origin. For points that escape, we will want to remember the “escape count”, or which iteration it escaped.

From Functions to Images

So, where do the interesting pictures come from?

We choose an actual function for \(f(x, y, a, b)\), then color each point in the plane based on how far it is from being in the Mandelbrot set for the selected function. “How far” is the “escape count”.

Actually, there are infinitely many points in the plane, so we can’t do every point. Instead, we select a regular grid of points each representing the points near it in the plane. For each point in the grid:

- Calculate its \((a, b)\) value, from its position in the grid.
- Use \(x_0 = 0\) and \(y_0 = 0\).
- Iterate the application of \(f(x, y, a, b)\) up to \(n\) times.
- If the result becomes far away, remember the iteration number when it first escaped (became far away from the origin). We call this the “escape value”. Points with lower escape values are further from the Mandelbrot set than those with higher escape values.
- If the result doesn’t become far away (escape) within \(n\) iterations, assume it did not escape and remember \(n\) as the escape value. These points are assumed to be part of the Mandelbrot set for \(f(x, y, a, b)\).

Now, we have a set of integers (the escape values) that represent how close each of the points in the grid are to the Mandelbrot set for \(f(x, y, a, b)\). Larger numbers mean they are closer to the Mandelbrot set.

Since the points selected are in a regular grid, we can pair each grid point with a pixel in a rectangular image. We assign the color of pixel based on the escape value of the corresponding grid point. All pixels
associated with the same escape value will have the same color.

**Defining a Regular Grid**

See the description in the Julia set assignment.

**Our Function f(x,y,a,b)**

For our function $f(x,y,a,b)$ we will use this definition:

\[
\begin{align*}
x' &= x^2 - y^2 + a \\
y' &= 2xy + b
\end{align*}
\]

where, $a$ and $b$ are the coordinates of the original point in the plane. Note this is similar to, but different from the Julia set, where $a$ and $b$ where fixed values for any entire image calculation. Here, they depend on the point whose escape count you are calculating.

**Assignment**

In this assignment you will create a class to create and store a Mandelbrot set’s escape values. Most of the work has already been done in the [NumberGrid](#) and [JuliaSet](#) classes. In this assignment, you will create a new class [ComplexFractal](#) that inherits from [NumberGrid](#). Most of the functionality of the [JuliaSet](#) class will be moved into the [ComplexFractal](#) class, and [JuliaSet](#) will change to inherit from [ComplexFractal](#). Finally, the [MandelbrotSet](#) class will be created, by inheriting from [ComplexFractal](#) and adding a few methods specific to the calculation of Mandelbrot sets.

You will also extend the [ppm_menu](#) program to add a few new commands.

The new commands required are:

- `julia`: Choose to make a Julia set.
- `mandelbrot`: Choose to make a Mandelbrot set.

**Potential Session**

To run all of the commands from a script, throwing away the prompts

```
$ ./ppm_menu < ppm_menu_assignment_09_sample_session_mandelbrot.txt >> /dev/null
```

```
$ ls -l *.ppm
-rw-rw-r-- 1 cgl cgl 750014 Mar 12 09:17 sample-mandelbrot-image-1.ppm
-rw-rw-r-- 1 cgl cgl 750014 Mar 12 09:17 sample-mandelbrot-image-2.ppm
-rw-rw-r-- 1 cgl cgl 750014 Mar 12 09:17 sample-mandelbrot-image-3.ppm
```

**Programming Requirements**

Below, the functions and methods may have a symbol CG? before them, where ? is a number. This indicates which Code Grinder step requires the function or method to be implemented.

**Create [ComplexFractal.h](#)**

When creating the [ComplexFractal](#) class, remember that it inherits from [NumberGrid](#), and that its functionality should be moved from the [JuliaSet](#) class. Just find all of the methods and data members and move them over. Cut-and-paste might even be appropriate in this case.

The [ComplexFractal](#) class must store the following information:

- minimum and maximum values for $x$ and $y$ to define the plane size
- gap values for $x$ and $y$ to define the grid points in the plane

The following methods must be created in the [ComplexFractal](#) class.

- `ComplexFractal();`
- `ComplexFractal( const int& height, const int& width, const double& min_x, const double& max_x, const double& min_y, const double& max_y );`
- `virtual -ComplexFractal();`
- `double getMinX( ) const;`
- `double getMaxX( ) const;`
- `double getMinY( ) const;`
Create **ComplexFractal.cpp**

The following methods must be implemented in the **ComplexFractal** class. See descriptions in the Julia set assignment. You should be cutting these methods from the **JuliaSet** class and moving them to this class. It should not require rewriting any methods.

```cpp
ComplexFractal();
ComplexFractal(const int& height, const int& width, const double& min_x, const double& max_x, const double& min_y, const double& max_y);
virtual ~ComplexFractal();
double getMinX() const;
double getMaxX() const;
double getMinY() const;
double getMaxY() const;
virtual void setGridSize(const int& height, const int& width);
void setPlaneSize(const double& min_x, const double& max_x, const double& min_y, const double& max_y);
void setDeltas(const double& delta_x, const double& delta_y);
double calculateDeltaX() const;
double calculateDeltaY() const;
double calculatePlaneXFromPixelColumn(const int& column) const;
double calculatePlaneYFromPixelRow(const int& row) const;
void calculatePlaneCoordinatesFromPixelCoordinates(const int& row, const int& column, double& x, double& y) const;
```

Modify **JuliaSet.h**

The **JuliaSet** class should only have the following data members and methods left. If your assignment still has the other data members or methods, you will not receive credit, even if the unit tests pass.

Remaining data members:

* a and b parameter values for the Julia set function

Remaining methods:

```cpp
JuliaSet();
JuliaSet(const int& height, const int& width, const double& min_x, const double& max_x, const double& min_y, const double& max_y, const double& a, const double& b);
virtual ~JuliaSet();
double getA() const;
double getB() const;
void setParameters(const double& a, const double& b);
virtual void calculateNextPoint(const double& x0, const double& y0, double& x1, double& y1) const;
int calculatePlaneEscapeCount(const double& x0, const double& y0) const;
virtual int calculateNumber(const int& row, const int& column) const;
```

Modify **JuliaSet.cpp**

Remove all methods that were moved to the **ComplexFractal** class.
Remaining methods:
- JuliaSet();
- JuliaSet( const int& height, const int& width, const double& min_x, const double& max_x, const double& min_y, const double& max_y, const double& a, const double& b );
- virtual ~JuliaSet();
- double getA() const;
- double getB() const;
- void setParameters( const double& a, const double& b );
- virtual void calculateNextPoint( const double x0, const double y0, double& x1, double& y1 ) const;
- int calculatePlaneEscapeCount( const double& x0, const double& y0 ) const;
- virtual int calculateNumber( const int& row, const int& column ) const;

Create **MandelbrotSet.h**

The MandelbrotSet class has no data members. It requires the following methods:
- MandelbrotSet();
- MandelbrotSet( const int& height, const int& width, const double& min_x, const double& max_x, const double& min_y, const double& max_y );
- virtual ~MandelbrotSet();
- virtual void calculateNextPoint( const double x0, const double y0, const double& a, const double& b, double& x1, double& y1 ) const;
- int calculatePlaneEscapeCount( const double& a, const double& b ) const;
- virtual int calculateNumber( const int& row, const int& column ) const;

Create **MandelbrotSet.cpp**

Implement the following methods.
- MandelbrotSet(); The default constructor, chain constructs via the ComplexFractal default constructor.
- MandelbrotSet( const int& height, const int& width, const double& min_x, const double& max_x, const double& min_y, const double& max_y ); Passes arguments on to the ComplexFractal constructor.
- virtual ~MandelbrotSet(); Does nothing, empty code block.
- virtual void calculateNextPoint( const double x0, const double y0, const double& a, const double& b, double& x1, double& y1 ) const; Calculates x1 and y1 using the function described above for the Mandelbrot set.
- int calculatePlaneEscapeCount( const double& a, const double& b ) const; Uses calculateNextPoint to find the escape count for the point at (a, b). The first iteration from x,y = 0,0 to x,y = a,b doesn’t count as an iteration.
- virtual int calculateNumber( const int& row, const int& column ) const; Uses other methods to find the plane coordinate of the pixel at row, column and to calculate the escape count. Returns the count. If row, column isn’t valid, then returns -1.

Update **image_menu.h**

Add the following function declarations to the file. Yes, NumberGrid** is a reference to a pointer to a NumberGrid. This allows the function to modify the pointer.

- void setJuliaFractal( std::istream& is, std::ostream& os, NumberGrid*& grid );
- void setMandelbrotFractal( std::istream& is, std::ostream& os, NumberGrid*& grid );

Update **image_menu.cpp**

This file must include the implementations for the new functions declared in image_menu.h.

- void setJuliaFractal( std::istream& is, std::ostream& os, NumberGrid*& grid ); If grid isn’t 0, delete it. Set grid to be a new JuliaSet object from the heap, default constructed. Doesn’t use is or os.
- void setMandelbrotFractal( std::istream& is, std::ostream& os, NumberGrid*& grid ); If grid isn’t 0, delete it. Set grid to be a new MandelbrotSet object from the heap, default constructed. Doesn’t use is or os.

The following functions will require updates to their functionality and/or declarations.
- void showMenu( std::ostream& os ); Add to the menu to include the following messages: “julia) Choose to make a Julia set.” and “mandelbrot) Choose to make a Mandelbrot set.”.
- void takeAction( std::istream& is, std::ostream& os, const std::string& choice, PPM& input_image1, PPM&
Add clauses to recognize the “julia” and “mandelbrot” commands. However, like with “quit”, do nothing here.

Inside the while loop, after takeAction is called, check if the choice was “julia”, if so, call setJuliaFractal. Do the same for “mandelbrot”. Note that both of these functions will receive grid, not *grid.

Change this dynamic_cast from JuliaSet to ComplexFractal, since setting the plane size is an operation that belongs to that class.

**Update ppm_menu.cpp**

No changes are required for ppm_menu.cpp.

**Update Makefile**

- This file must include the rules to build the program ppm_menu.
- A developer must be able to use the command make to compile all necessary files and link them to the executable program ppm_menu.
- Additionally, add the clean target that has no dependencies, but will remove any .o files and ppm_menu.
- Add automatic source and object file calculation to the Makefile.
- Add automatic dependency calculations to the Makefile.

**Additional Documentation**

- [C++ Reference](#)
- [Examples from class](#)
- [Sample Session Input File](#)
- [Julia set on Wikipedia](#)
- [Mandelbrot set on Wikipedia](#)

**Sample PPM Images**

- [Sample Output1](#)
- [Sample Output2](#)
- [Sample Output3](#)

**Show Off Your Work**

To receive credit for this assignment, you must

- complete the unit tests available in CodeGrinder
- zip the source code (.cpp and .h files) and the Makefile and upload to the Canvas submission system
- use git to add, commit and push your solution to your repository for this class.

Additionally, the program must build, run and give correct output.

**Extra Challenges (Not Required)**

- Create classes that inherit from ComplexFractal that have different calculateNumber() functions for calculating values. Add the ability to use them from the imageMenu().
- Make methods of the ComplexFractal class that allow for zooming in or out in the plane. Add the ability to use them from the imageMenu().
- Try other ways to modify the plane and parameters that would make it easier to create interesting images.