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

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.

\[
x_1, y_1 = f(x_0, y_0, a, b)
\]

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:

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

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.

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.

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.

**Assignment**

Create a class named `MandelbrotSet`, to represent a collection of Mandelbrot set calculations.

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

```
\[
x' = x^2 - y^2 + a \\
y' = 2xy + b
\]
```

where, $a$ and $b$ are the coordinates of the original point in the plane.

Since most of this work has already been done in the `JuliaSet` class, and you don’t want to do more work than necessary, create the `ComplexFractal` class as a parent to both `JuliaSet` and `MandelbrotSet`.

**ComplexFractal** Class Programming Requirements

This class must be declared in `ComplexFractal.h` and implemented in `ComplexFractal.cpp`.

Your `ComplexFractal` class must store the following data.

- The width and height (in pixels) of the desired image (`int`s).
- The minimum and maximum values for $x$ and $y$ to define the rectangular space in the plane (`double`s).
- The maximum allowed escape count that corresponds to non-escaping points (`int`).
- The horizontal and vertical gap sizes. These data members must be initialized to 0.01 in the default constructor.
- Vector of all calculated escape counts (`std::vector<int>`). The default constructor should initialize it to have 120000 items.

**Constructor and Getters**

- `ComplexFractal();` Default constructor. Sets up for a 400x300 image using the 4x3 rectangle centered on the origin. Sets maximum escape count to 255.
- `virtual ~ComplexFractal();` Destructor.
- `int getWidth() const;` Returns the pixel width.
- `int getHeight() const;` Returns the pixel height.
- `double getMinX() const;` Returns the plane minimum x value.
- `double getMaxX() const;` Returns the plane maximum x value.
- `double getMinY() const;` Returns the plane minimum y value.
- `double getMaxY() const;` Returns the plane maximum y value.
- `int getMaxEscapeCount() const;` Returns maximum allowed escape count.

**Setters**

- `void setPixelSize(const int& width, const int& height);` Sets the pixel width and height of the desired image. Only makes a change if the width and height are both at least 2.
- `void setPlaneSize(const double& min_x, const double& max_x, const double& min_y, const double& max_y);` Sets the plane boundaries. If the minimum value for a dimension is greater than than maximum value, automatically swap them. Only allows values in the range -2.0 to 2.0 for each. If any is out of range, change nothing. If the minimum value for a dimension is identical to the maximum value for the dimension, then change nothing.
- `void setMaxEscapeCount(const int& count);` Sets the maximum allowed escape count. Only makes the change if count is at least 0.

**Coordinate Methods**

- `double getDeltaX() const;` Returns the horizontal gap value.
- `double getDeltaY() const;` Returns the vertical gap value.
- `void setDeltas(const double& delta_x, const double& delta_y);` Assigns the gaps to data members. Only assigns if both values are positive.
- `double calculateDeltaX() const;` Calculate the horizontal plane distance between neighboring pixel columns. This is the gap value discussed above. Note this method calculates the value and returns it. It does not set the data member.
- `double calculateDeltaY() const;` Calculate the vertical plane distance between neighboring pixel rows. This is the gap value discussed above.
- `double calculatePlaneXFromPixelColumn(const int& column) const;` Calculate the plane x value for a given column. If the column is out of range, return 0.
double calculatePlaneYFromPixelRow( const int& row ) const; Calculate the plane y value for a given row. If the row is out of range, return 0.

void calculatePlaneCoordinatesFromPixelCoordinates( const int& row, const int& column, double& x, double& y ) const; Sets x and y to the plane coordinates for the row and column. If either row or column is out of range, set both x and y to 0. Notice x and y are return by reference.

**Escape Value Methods**

- const std::vector<int>& getEscapeCounts() const; Returns the vector of escape counts.
- int getEscapeCount( const int& row, const int& column ) const; Returns the escape count stored at the row, column position. If the row or column is out of range, return -1.
- void setEscapeCount( const int& row, const int& column, const int& count ); Stores the escape count in the position for row, column. If row or column are out of range, do nothing. If the escape count is out of range, do nothing.
- int calculatePixelEscapeCount( const int& row, const int& column ) const = 0; Pure virtual method causing child classes to override it.
- void calculateAllEscapeCounts(); Calculate escape counts for all pixel locations, storing them.

**Image Methods**

- void setPPM( PPM& ppm ) const; Uses the currently calculated escape values to configure an image in the PPM object. Sets the width and height of the image to match the width and height of the Julia set. Sets the maximum color value to 63. For each pixel in the PPM object, sets the color based on the escape value for the pixel. See the following table:

<table>
<thead>
<tr>
<th>Escape Value</th>
<th>Color (R, G, B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0, 0, 0)</td>
</tr>
<tr>
<td>max escape count</td>
<td>(63, 31, 31)</td>
</tr>
<tr>
<td>count % 8 == 0</td>
<td>(63, 63, 63)</td>
</tr>
<tr>
<td>count % 8 == 1</td>
<td>(63, 31, 31)</td>
</tr>
<tr>
<td>count % 8 == 2</td>
<td>(63, 63, 31)</td>
</tr>
<tr>
<td>count % 8 == 3</td>
<td>(31, 63, 31)</td>
</tr>
<tr>
<td>count % 8 == 4</td>
<td>(0, 0, 0)</td>
</tr>
<tr>
<td>count % 8 == 5</td>
<td>(31, 63, 31)</td>
</tr>
<tr>
<td>count % 8 == 6</td>
<td>(31, 31, 63)</td>
</tr>
<tr>
<td>count % 8 == 7</td>
<td>(63, 31, 63)</td>
</tr>
</tbody>
</table>

- void setPPM( PPM& ppm, const ColorTable& colors ) const; Uses the currently calculated escape values to configure an image in the PPM object. Sets the width and height of the image to match the width and height of the pixels. Sets the maximum color value to the maximum color value of any color in the color table. For each pixel in the PPM object, sets the color based on the escape value for the pixel. If the color table does not have at least 3 colors, make no changes to the PPM object. Use the color table item at the last position for any pixels with a maximum escape value. Use the color table item at the next to last position for any pixels with a 0 escape value. For all other escape values, use (the escape value) mod (the size of the color table - 2) as the index into the color table.

**JuliaSet**

Class Programming Requirements

This class must be declared in `JuliaSet.h` and implemented in `JuliaSet.cpp`.

Your `JuliaSet` class must store the following data.

- The two parameters `a` and `b` used in `f(x, y)` above (double's).

Your `JuliaSet` class must have the following methods.

**Constructor and Getters**

- JuliaSet(); Default constructor. Sets `a` to -0.650492, `b` to -0.478235.
- virtual ~JuliaSet(); Destructor.
- double getA() const; Returns the `a` parameter.
- double getB() const; Returns the `b` parameter.

**Setters**

- void setParameters( const double& a, const double& b ); Sets values for `a` and `b`. Only allows values in
the range -2.0 to 2.0 for each. If either is out of range, change nothing.

**Escape Value Methods**

- `void calculateNextPoint( const double x0, const double y0, double& x1, double &y1 ) const;` Calculate the next escape point after x0, y0 and store in x1, y1. Note that x1 and y1 are return by reference.
- `int calculatePlaneEscapeCount( const double& x0, const double& y0 ) const;` Calculate the number of iterations required for x0, y0 to escape. The return value should be in the range 0 to maximum escape count, inclusive. 0 means immediately escaped. Maximum escape count means never escaped, or escaped on the last step. Escape means the distance from the origin is more than 2.
- `virtual int calculatePixelEscapeCount( const int& row, const int& column ) const;` Calculate the number of iterations required for row, column to escape. The return value should be in the range 0 to maximum escape count, inclusive. 0 means immediately escaped. Maximum escape count means never escaped, or escaped on the last step. If row or column is out of range, return -1.

**MandelbrotSet Class Programming Requirements**

This class must be declared in `MandelbrotSet.h` and implemented in `MandelbrotSet.cpp`.

Your `MandelbrotSet` class must store the following data.

- None.

Your `MandelbrotSet` class must have the following methods.

**Constructor and Getters**

- `MandelbrotSet();` Default constructor.
- `virtual ~MandelbrotSet();` Destructor.

**Escape Value Methods**

- `void calculateNextPoint( const double x0, const double y0, const double& a, const double& b, double& x1, double &y1 ) const;` Calculate the next escape point after x0, y0 and store in x1, y1, using the point a, b. Note that x1 and y1 are return by reference.
- `int calculatePlaneEscapeCount( const double& a, const double& b ) const;` Calculate the number of iterations required for x=0, y=0, to escape, using a,b. The return value should be in the range 0 to maximum escape count, inclusive. 0 means immediately escaped. Maximum escape count means never escaped, or escaped on the last step. Escape means the distance from the origin is more than 2. Note that 0,0 will never escape, so `calculateNextPoint()` should be used once, before checking for the 0 escape.
- `virtual int calculatePixelEscapeCount( const int& row, const int& column ) const;` Calculate the number of iterations required for row, column to escape. The return value should be in the range 0 to maximum escape count, inclusive. 0 means immediately escaped. Maximum escape count means never escaped, or escaped on the last step. If row or column is out of range, return -1. Calculates a and b from column and row, then uses `calculatePlaneEscapeCount()` to do the rest of the work.

**Program Programming Requirements**

This program must be written in `fractal.cpp`, and the final program must be named `fractal`.

The program must:

- Use a menu system.
- Menu loops until user chooses quit.
- Options in menu
  - w: set width
  - h: set height
  - x: set minimum x
  - X: set maximum x
  - y: set minimum y
  - Y: set maximum y
  - a: set parameter a
  - b: set parameter b
  - e: set maximum escape value
  - c: calculate all Julia set escape values
  - p: assign colors to the PPM using Julia escape values
  - s: save the PPM to file
S: show Julia set meta data
C: calculate all Mandelbrot escape values
P: assign colors to the PPM using Mandelbrot escape values
M: show Mandelbrot set meta data
q: quit

Example Input

This sample input produces these six images:

- fractal_example01.ppm.
- fractal_example02.ppm.
- fractal_example03.ppm.
- fractal_example11.ppm.
- fractal_example12.ppm.
- fractal_example13.ppm.

This Makefile and fractal.cpp may be used.

Additional Documentation

- Julia set on Wikipedia
- Mandelbrot set on Wikipedia

Show Off Your Work

To receive credit for this assignment, you must complete the unit tests available in CodeGrinder, and you must upload the source code (.cpp and .h files) and the Makefile to the Canvas submission system.

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