Homework 2

Complex Numbers and Fractal Generation

Due Monday October 17th by 11:59 pm

100 points

This project is divided into two components:

1. Implement a new data type named `Complex` that defines a *complex number* of the form *a* + *bi*.
2. Use this new type to generate a fractal PNG image of some arbitrary width x height dimension.

Creating a new data type in C++ can be done using either a `struct` or `class` object. Using *operator overloading*, you will need to define some basic mathematical operations on your type implementation, which are described in detail in the background section below. Your implementation will be run through a series of *unit tests* to confirm that all operations and methods are numerically correct. As such, make certain you carefully following the naming conventions outlined below.

Once a complex number type is defined, it's actually quite easy to generate very beautiful fractal images using *Julia sets*. One simple approach is to recursively compute a complex point for every pixel coordinate in an image until the distance between that point and the origin exceeds some radius (typically 2). This iterative process (described in detail in the background section below) provides a way to assign a single integer score to each pixel in an image and generate the types of fractal images seen below.

In order to generate these images, you will need to allocate a dynamic array of some arbitrary size (specified at run time) and compute pixel color values for each (x,y) coordinate in that array. You will be provided with some external code that provides functionality for exporting a 1D array of RGBA pixels (4 unsigned chars per pixel) to a PNG image file. You are also provided with code that maps a single integer value from the range [0…255] to a continuous RGB color value. You may use this color map to compute color values for each pixel.

You program should accept 5 command line arguments of the form:

```
$ ./fractal <width > <height> <output_png> OPTIONAL: <ca> <cb>
```

Where `<width>` and `<height>` specify the PNG's dimensions and `<output_png>` is the name of the PNG file you wish to write. The last 2 arguments, `<ca>` and `<cb>`, are the floating point numbers used for the complex number constant used in fractal generation. These values are *optional*, meaning you should define some default values to use if they aren't provided on the command line. Image dimensions and output filenames, however, are mandatory and should **not** be hardcoded into the program. See the background section on parsing command line arguments for more information.

Naming Conventions:

1. You must name your complex number implementation `complex.h` / `complex.cpp` otherwise unit testing will fail.
2. If you write your own makefile, it must be named `Makefile` (no extension!)

Code Requirements:

To receive full credit you must implement all features as outlined above, as well as fulfill the following code specific guidelines:

- Implement a complex number primitive (named `Complex`) in a separate header and cpp file.
- Define arithmetic operators (i.e., `+-*/`) for this Complex data type using operator overloading. **See definitions below.**
- Define a single member function named `magnitude2` that returns a complex number's squared magnitude (also called absolute value). **See definition below.**
- Import the provided `lodepng.h` and `rgb.h` headers.
- Parse command line arguments specifying PNG dimensions and output filename.
- Implement a function named `julia` that, given an (x,y) input and max_itr value, generates a single integer value in the range [0… max_itr].
- Create a 1D array of RGBA pixels (4 unsigned char per pixel) and write a fractal image PNG to disk using the lodepng library.

Grading:

This is a rough outline on how points will be assigned. Remember, if code doesn't compile on the CSG machines or a makefile isn't included additional points will be taken off.

20 points:	Complex number implementation: member variables `+-*/` operators `magnitude2` member function
5 points:	Correctly import and use `lodgepng` and `rgb` dependancies
10 points:	Dynamically allocate an array of unsigned char for use in generating your PNG output file.
10 points:	Julia set function implementation.
5 points:	Read all arguments by the command line.
5 points:	All code is well commented and syntax style is consistent and easy to follow.
5 points:	Advanced makefile is included in your zip file.

Example Command Line Usage:

```
$ ./fractal 800 600 foobar.png
$ ./fractal 800 600 foobar.png -0.7 0.278
```

Background Information:

Complex numbers

As per Wikipedia, a complex number is any number that is expressed in the form $a + bi$, where a and b are real numbers and i is the imaginary unit. Representing complex numbers amounts to storing the values of a (the real part) and b (the imaginary part) and defining the behavior of mathematical operations on complex numbers. The definitions you are required to implement are defined below:

Addition

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

Subtraction

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

Multiplication

$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i$$

Division

$$\frac{(a + bi)}{(c + di)} = \left(\frac{ac + bd}{c^2 + d^2}\right) + \left(\frac{bc - ad}{c^2 + d^2}\right)i$$

Magnitude

$$\sqrt{a^2 + b^2}$$

More information can be found at: http://en.wikipedia.org/wiki/Complex_number

Lodepng

Lodepng is a self-contained C++ library for writing reading and writing PNG files. The general usage form for writing files is the following function call:

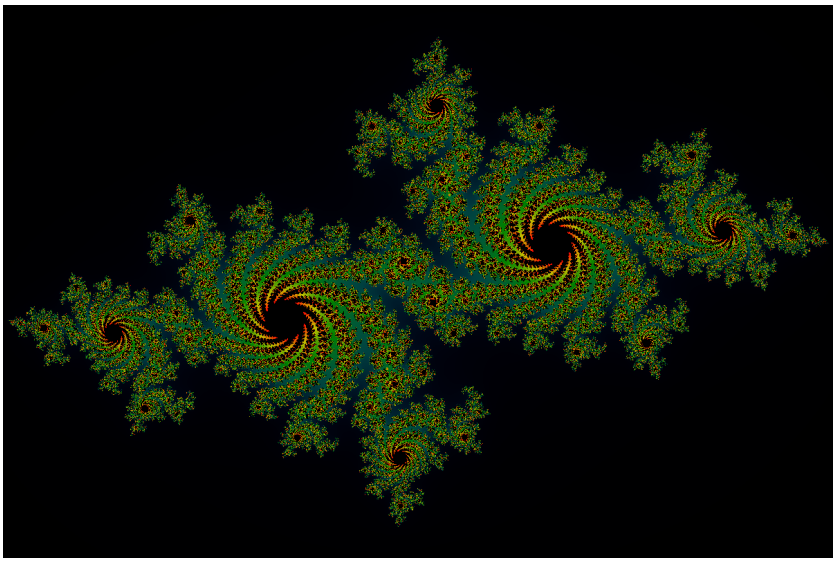
```
lodepng::encode(filename, image, width, height);
```

Here `image` is a 1D array of type `unsigned char` (this is just an unsigned integer type). This will write a PNG file to the path provided as the `filename` argument.

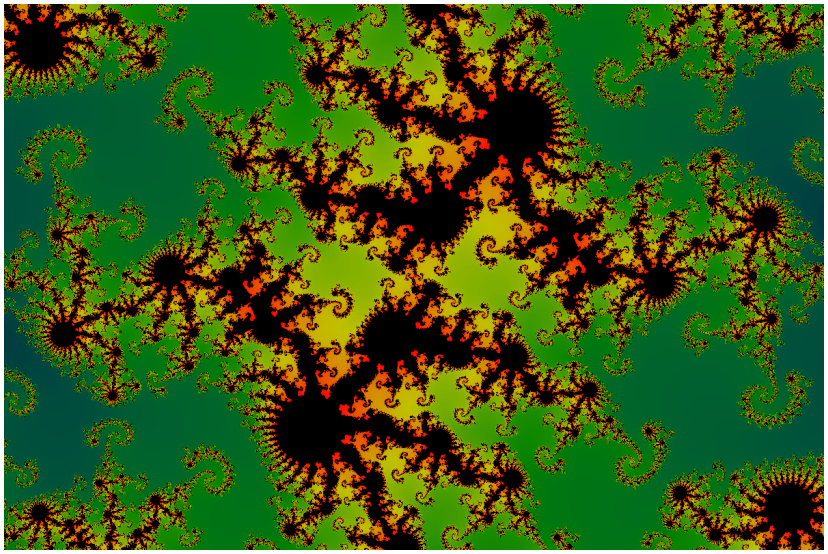
NOTE: PNGs define their pixel color space using 4 bytes (Red, Blue, Green, Alpha). For this project, you can always set the alpha channel to 255.

More information and coding examples may be found at: http://lodev.org/lodepng/

Julia Fractals



Example 1: Julia set fractal 1200 x 800, c = $(-0.7 + 0.27i)$ zoom = 1.0



Example 2: Julia set fractal 1200 x 800, c = $(-0.8 + 0.156i)$ zoom = 40.0

The simplest way to generate a fractal from the Julia set is to use the **Escape Time Algorithm**. This simple approach recursively calculates a complex point for every pixel in your target image until the distance of that point from the origin is greater than some threshold. The number of iterations this takes is your *escape time* and determines the color of the target pixel. Formally this is defined as follows: take a complex point

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

where c is some fixed complex constant that you specify a priori and z_n is assigned using the pixel coordinate in question, For example, given an (x,y) coordinate of (100,20) we first transform x and y to fall within the range [-1, 1]. This transformation is dependent upon the width and height of your target image and is defined as follows:

tx = **1.5 *** (x - width * 0.5) / (0.5 * zoom * width)
ty = (y - height * 0.5) / (0.5 * zoom * height)

(Note the **1.5** is a fudge factor to better center the image in the PNG.)

This transformed (tx,ty) coordinate can be then be used as the real valued components (a and b) in our complex number z_n. The distance of z_n to the origin is simply the complex number's *magnitude*, defined as $\sqrt{a^2 + b^2}$. We want to recursively compute z_{n+1} (using our last value z_n) until it falls outside of a circle with radius 2, i.e., magnitude > 2. By counting the number of iterations this process takes (bounded by some maximum iteration value) we can compute a single integer score *m*, in the range [0… MAX_ITR], for every image coordinate (x,y). By setting all color channels (i.e., red, green, blue) in a pixel to `m % 256`, we can compute a grayscale intensity value for all pixels in an image.

Alternatively, you may use the provided linear color map, which takes a single value and maps it to a continuous RGB color for you. Please refer to the provided `rgb.h` *code for more details.*

See http://en.wikipedia.org/wiki/Julia_set for more information.

Command line arguments

The example code files we've looked at in class use this parameter list for main:

```
int main(int argc, const char * argv[])
```

While we haven't discussed this (and indeed your main function requires no arguments to compile correctly), these two function parameters contain the results of information passed to the program executable from the command line. `argc` is the length of the argument array `argv`. By iterating over this array data using a simple for loop, you gain access to any text information typed immediately following the executable name. For example,

```
./fractal 800 600 foobar.png
```

results in an array of length 4 with these contents:

```
[ "<current_path>/fractal", "800", "600", "foobar.png" ]
```

therefore you can uses array values like `argv[3]` (i.e., "foobar.png") as input to functions that require filenames or need similar user specified values. Note that the numeric values are stored (initially) as c-style strings.

Converting strings to integer values

You may find it useful to convert a c-style sting to its integer value using `atoi`. For floating point numbers you can use `atof`.

```
std::string token = "123";
int foo = atoi(token.c_str());
std::cout << foo << std::endl;

>> 123
```