Assignment 3A: Fractals

Assignment by Chris Gregg, based on an assignment by Marty Stepp and Victoria Kirst. Originally based on a problem by Julie Zelenski and Jerry Cain.

JANUARY 27, 2017

Outline:

Description Style Extras FAQ

This problem focuses on recursion.

It is fine to write **"helper" functions** to assist you in implementing the recursive algorithms for any part of the assignment. Some parts of the assignment essentially *require* a helper to implement them properly. It is up to you to decide which parts should have a helper, what parameter(s) the helpers should accept, and so on. You can declare function prototypes for any such helper functions near the top of your **.cpp** file. (Don't modify the provided **.h** files to add your prototypes; put them in your own **.cpp** file.)

Files and Links:

Project Starter ZIP (open **Fractals.pro**)

Turn in: **fractals.cpp**

Demo JAR

We provide a GUI for you that helps you run and test your code.

Problem Description:

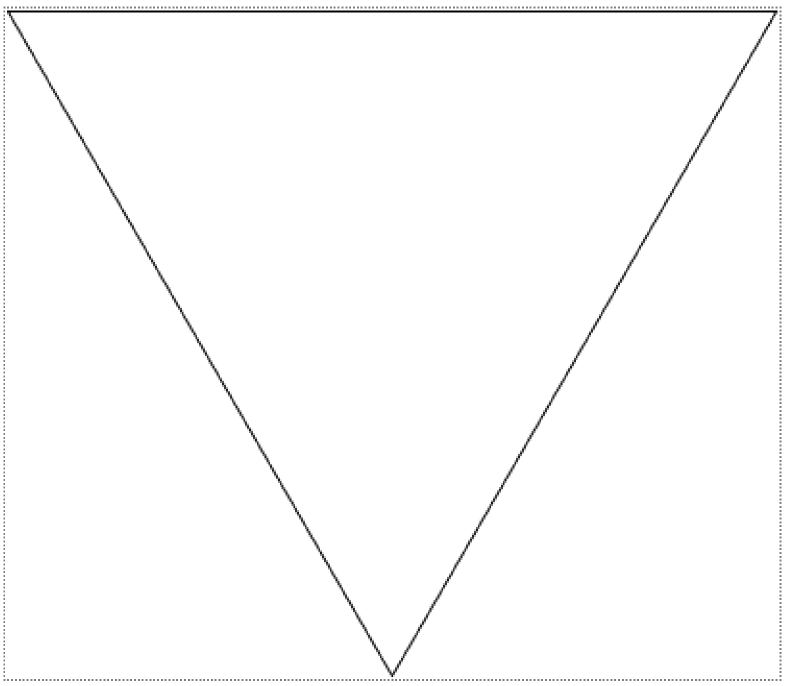
In this problem you will write several recursive functions related to drawing graphics. Recursive graphical patterns are also called *fractals*.

Sierpinski Triangle

If you search the web for fractal designs, you will find many intricate wonders beyond the Koch snowflake illustrated in Chapter 8. One of these is the Sierpinski Triangle, named after its inventor, the Polish mathematician Waclaw Sierpinski (1882-1969). The order-1 Sierpinski Triangle is an equilateral triangle, as shown in the diagram below.

For this problem, you will write a recursive function that draws the Sierpinski triangle fractal image. Your solution should not use any loops; you must use recursion. Do not use any data structures in your solution such as a `Vector`, `Map`, arrays, etc.

```
void drawSierpinskiTriangle(GWindow& gw, double x, double y, double siz
```



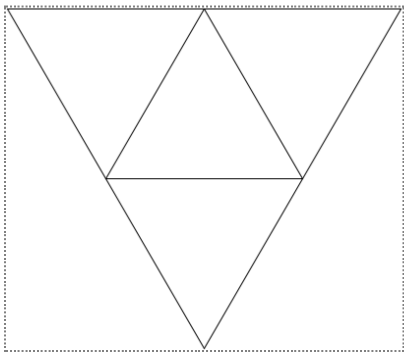
Order 1 Sierpinski triangle

To create an order-*K* Sierpinski Triangle, you draw three Sierpinski Triangles of order *K*-1, each of which has half the edge length of the original. Those three triangles are positioned in what would be the corners of the larger triangle; together they combine to form the larger triangle itself. Take a look at the Order-2 Sierpinski triangle below to get the idea.

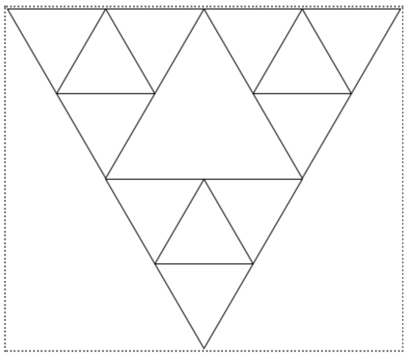
Your function should draw a black outlined Sierpinski triangle when passed a reference to a graphical window, the x/y coordinates of the top/left of the triangle, the length of each side of the triangle, and the order of the figure to draw (such as 1 for Order-1, etc.). The provided code already contains a **`main`** function that pops up a graphical user interface to allow the user to choose various x/y positions, sizes, and orders. When the user clicks the appropriate button, the GUI will call your function and pass it the relevant parameters as entered by the user. The rest is up to you.

If the order passed is 0, your function should not draw anything. If the x, y, order, or size passed is negative, your function should throw a string **exception**. Otherwise you may assume that the window passed is large enough to draw the figure at the given position and size.

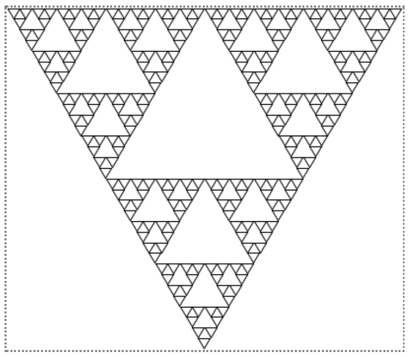
Some students mistakenly think that some levels of Sierpinski triangles are to be drawn pointing upward and others downward; this is incorrect. The upward-pointing triangle in the middle of the Order-2 figure is not drawn explicitly, but is instead formed by the sides of the other three downward-pointing triangles that are drawn. That area, moreover, is not recursively subdivided and will remain unchanged at every order of the fractal decomposition. Thus, the Order-3 Sierpinski Triangle has the same open area in the middle.



Order-2



Order-3



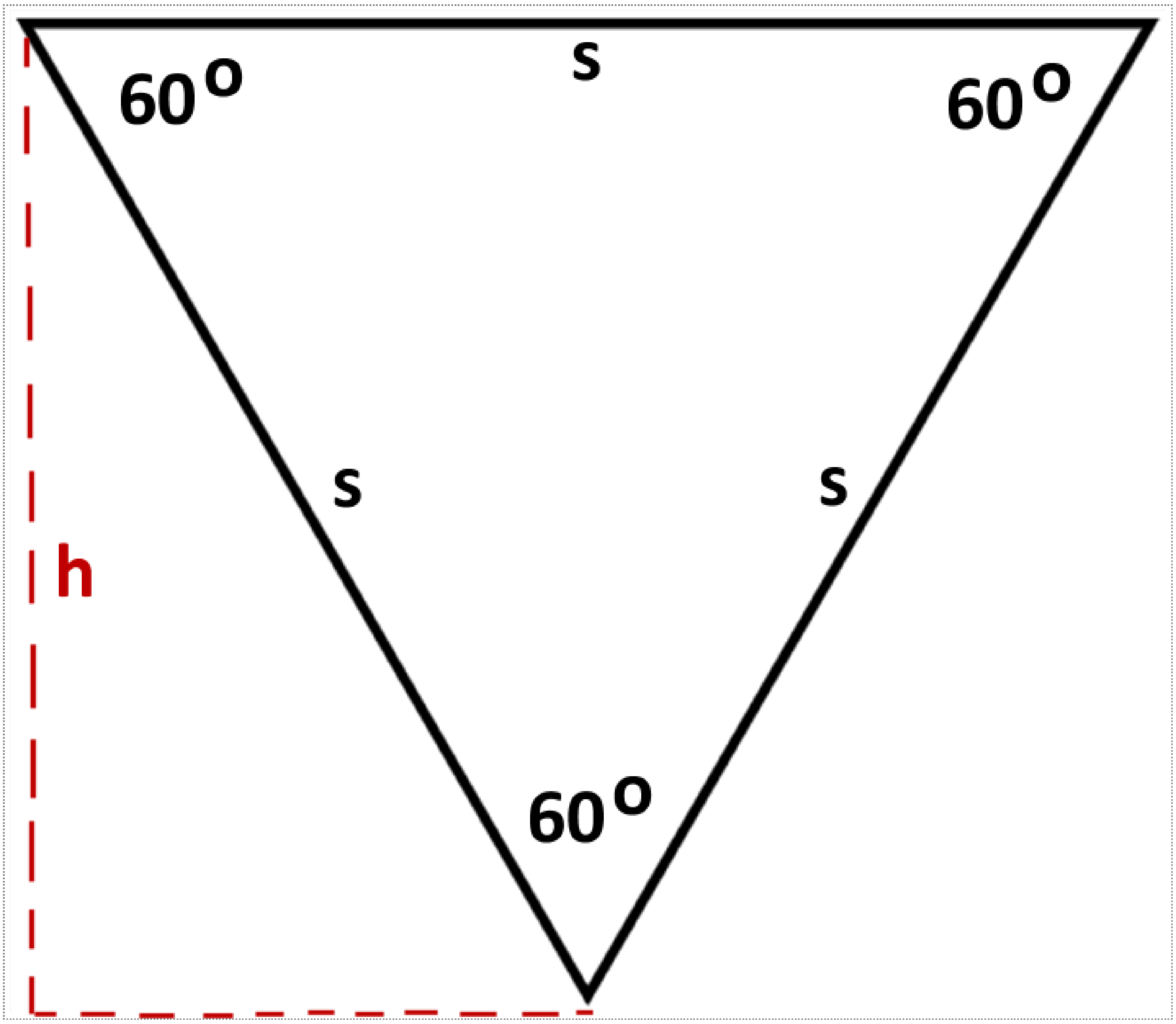
... Order-6

We have not really learned much about the **`GWindow`** class or drawing graphics, but you do not need to know much about it to solve this problem. The only member function you will need from the **`GWindow`** is its **`drawLine`** function. (complete **`GWindow`** documentation):

Member	**Description**
`gw`**`.drawLine(`*****`x1, y1, x2, y2`*****`);`**	draws a line from point (*x*1, *y*1) to point (*x*2, *y*2)

Note: You may find yourself needing to compute the height of a given triangle so you can pass the right x/y coordinates to your function or to the drawing functions. Keep in mind that the height *h* of an equilateral triangle is not the same as its side length *s*. The diagram below shows the relationship between the triangle and its height. You may want to look at information about equilateral triangles on Wikipedia and/or refresh your trigonometry.

- http://en.wikipedia.org/wiki/Equilateral_triangle



Computing an equilateral triangle's height from its side length

A particular style of solution we want you to avoid is the "**pair of functions**" solution, where you write one function to draw "downward-pointing" triangles and another to draw "upward-pointing" triangles, and each one calls the other in an alternating fashion. That is a poor solution that does not capture the self-similarity inherent in this fractal figure.

Another thing you should avoid is **re-drawing** the same line multiple times. If your code is structured poorly, you end up drawing a line again (or part of a line again) that was already drawn, which is unnecessary and inefficient. If you aren't sure whether your solution is redrawing lines, try making a counter variable that is incremented each time you draw a line and checking its value.

If the order passed is 0, your function should not draw anything. If the x, y, order, or size passed is negative, your function should throw a string **exception**. Otherwise you may assume that the window passed is large enough to draw the figure at the given position and size.

Expected output: You can compare your graphical output against the following image files, which are already packed into the starter code and can be compared against by clicking the "compare output" icon in the provided GUI, as shown at right. 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.

- sierpinski at x=10, y=30, size=300, order=1
- sierpinski at x=10, y=30, size=300, order=2
- sierpinski at x=10, y=30, size=300, order=3
- sierpinski at x=10, y=30, size=300, order=4
- sierpinski at x=10, y=30, size=300, order=5
- sierpinski at x=10, y=30, size=300, order=6

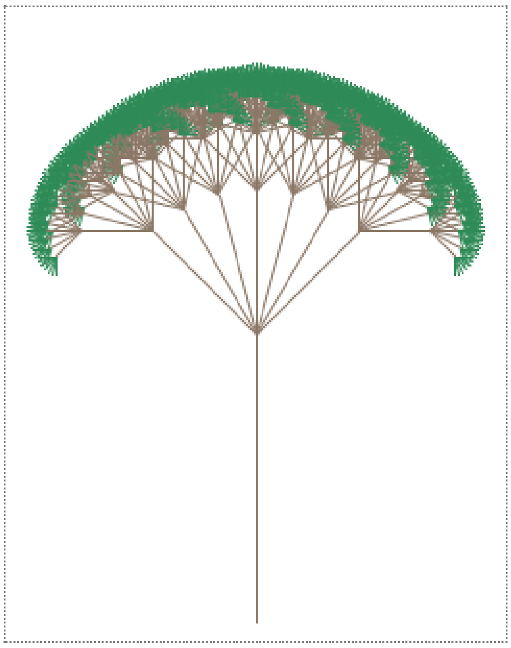
Recursive Tree

For this problem, write a recursive function that draws a recursive tree fractal image as specified. Your solution is **allowed to use loops** if they are useful to help remove redundancy, but your overall approach to drawing nested levels of the figure must still be recursive. Do not use any data structures in your solution such as a **`Vector`**, **`Map`**, arrays, etc.

```
void drawTree(GWindow& gw, double x, double y, double size, int order)
```

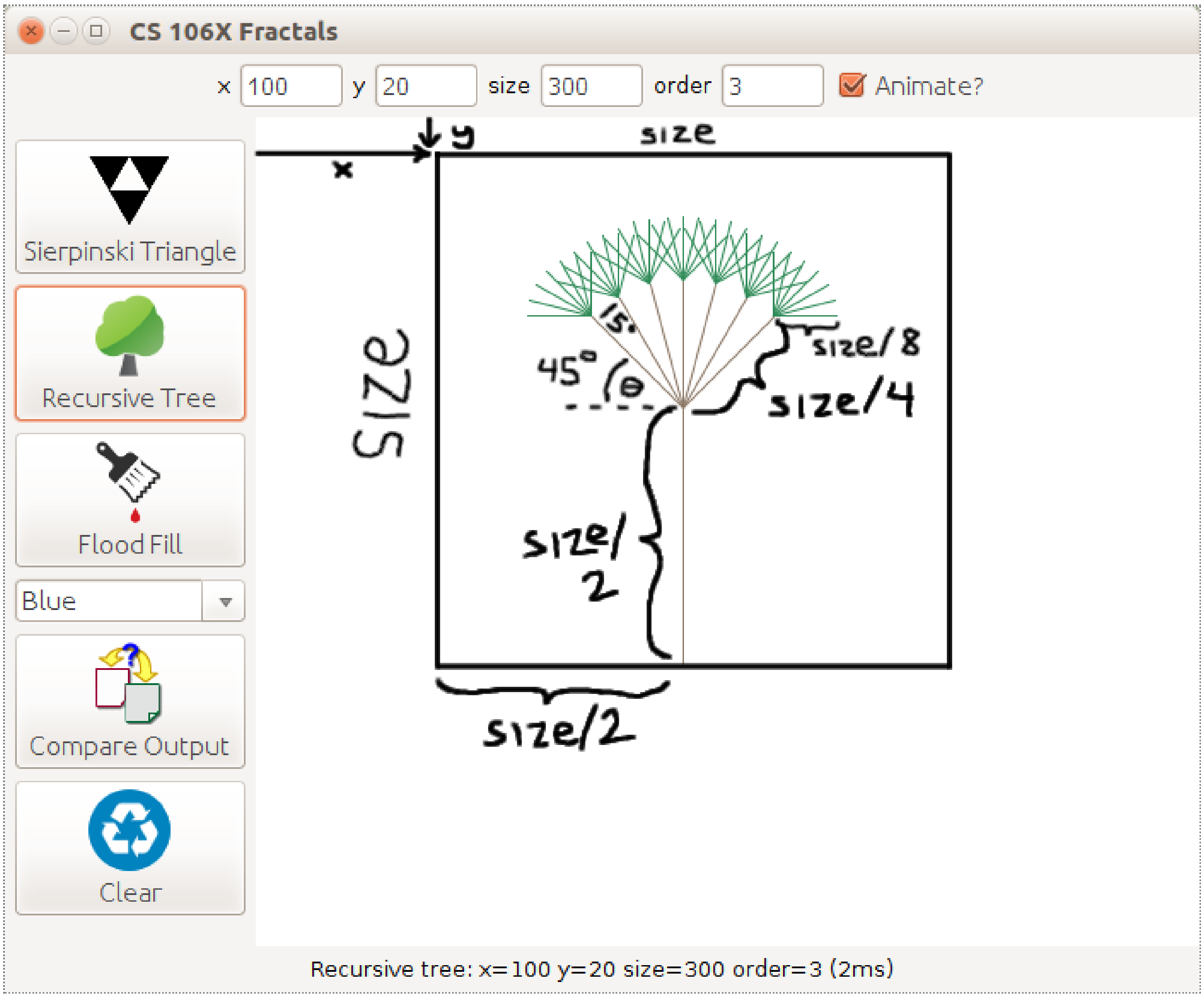
Our tree fractal contains a trunk that is drawn from the bottom center of the applicable area (*x*, *y*) through (*x* + *size*, y + *size*). The trunk extends straight up through a distance that is exactly half as many pixels as *size*.

The drawing below is a tree of order 5. Sitting on top of its trunk are seven trees of order 4, each with a base trunk length half as long as the prior order. Each of the order-4 trees is topped off with seven order-3 trees, which are themselves comprised of seven order-2 trees, and so on.


Order-5 tree fractal

The parameters to your function represent, in order: the window on which to draw the figure; the x/y position of the top/left corner of the imaginary bounding box surrounding the area in which to draw the figure; the general width and height (size) of the figure; and the number of levels (order) of the figure.

Some of these parameters are somewhat unintuitive. For example, the x/y coordinates passed are not the x/y coordinates of the tree trunk itself, but instead the x/y coordinates of a bounding box area in which to draw the tree. The diagram below attempts to clarify this.



Recursive tree: x=100 y=20 size=300 order=3 (2ms)

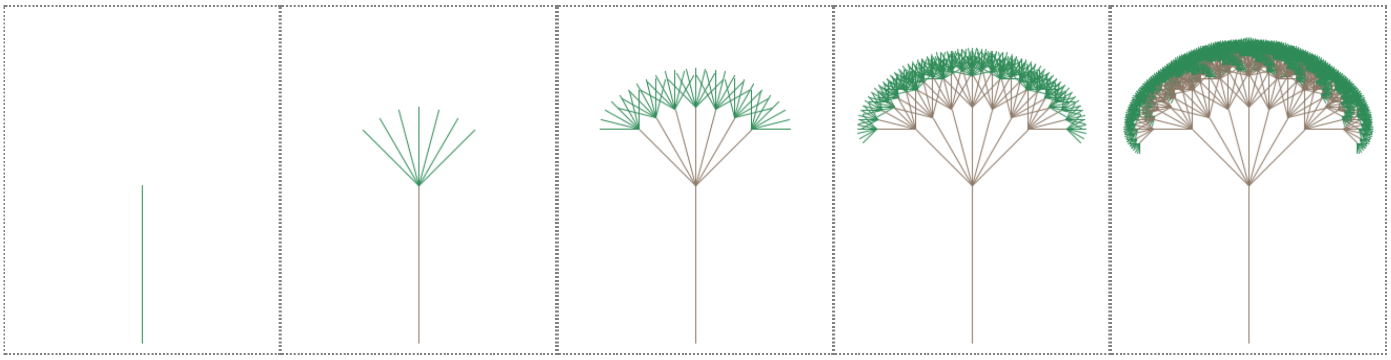
Diagram of **`drawTree(gw, 100, 20, 300, 3);`** call parameters

Lengths: The seven subtrees each have a length that is exactly **half** of their parent branch's length in pixels.

Angles: The seven subtrees extend from the tip of the previous tree trunk at relative angles of ±45, ±30, ±15, and 0 degrees relative to their parent branch's angle. For example, if you look at the Order-1 figure, you can think of it as a vertical line being drawn in an upward direction and facing upward, which is a polar angle of 90 degrees. In the Order-2 figure, the seven sub-branches extend at angles of 45, 60, 75, 90, 105, 120, and 135 degrees. And so on.

Colors: Inner branches, ones drawn at order 2 and higher, are drawn in a color of **#8b7765** (r=139, g=119, b=101), and the leafy fringe branches of the tree (branches drawn at level 1) are drawn in a color of **#2e8b57** (r=46, g=139, b=87).

The images below show the progression of each order of the tree fractal figure.



Order-1 *Order-2* *Order-3* *Order-4* *Order-5*

You should use the **`GWindow`** object's **`drawPolarLine`** function to draw lines at various angles and its **`setColor`** function to change drawing colors, as seen in the Koch snowflake example from lecture.

Member	Description
`gw.drawPolarLine(x0, y0, r, theta);` `gw.drawPolarLine(p0, r, theta);`	draws a line the given starting point, of the given length ***r***, at the given angle in degrees ***theta*** relative to the origin
`gw.setColor(color);`	sets color used for future shapes to be drawn, either as a hex string such as **`"#aa00ff"`** or an RGB integer such as **`0xaa00ff`**

If the order passed is 0, your function should not draw anything. If the x, y, order, or size passed is negative, your function should throw a string **exception**. Otherwise you may assume that the window passed is large enough to draw the figure at the given position and size.

Expected output: You can compare your graphical output against the following image files, which are already packed into the starter code and can be compared against by clicking the "compare output" icon in the provided GUI, as shown at right. 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.

- tree at x=100, y=20, size=300, order=1
- tree at x=100, y=20, size=300, order=2
- tree at x=100, y=20, size=300, order=3
- tree at x=100, y=20, size=300, order=4
- tree at x=100, y=20, size=300, order=5
- tree at x=100, y=20, size=300, order=6

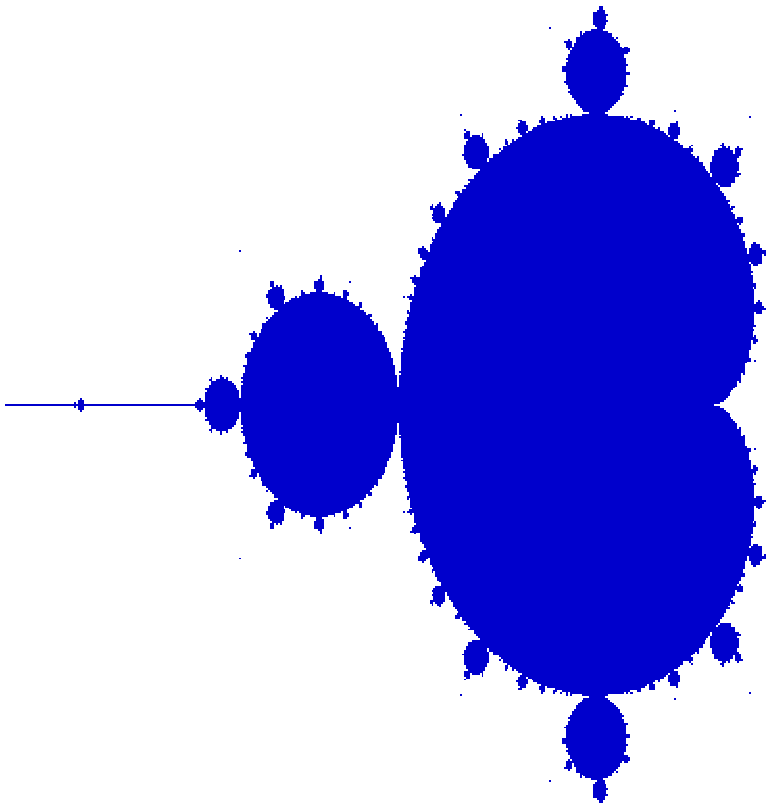
Mandelbrot Set

For this problem, you will write three functions that work together to generate the "Mandelbrot Set" on a graphical window as described below. One of your functions will be recursive. Your solution may use loops to traverse a grid, but you must use recursion to determine if a particular point exists in the Mandelbrot Set.

```
void mandelbrotSet(GWindow& gw, double minX, double incX,
                   double minY, double incY, int maxIterations, int col
int mandelbrotSetIterations(Complex c, int maxIterations)
int mandelbrotSetIterations(Complex z, Complex c, int remainingIteratio
```

The fractal below is a depiction of the "Mandelbrot Set," in tribute to Benoit Mandelbrot, a mathematician who investigated the 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 assignment, you will map coordinates on an image (via a Grid) to the appropriate complex number plane (we have abstracted away many of the 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.
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 has the following functions:

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.realPart()`	returns the real part of the complex number
`cpx.imagPart()`	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

Implementation Details

One of the most interesting aspects of the Mandelbrot Set is that it has an infinitely large self-similarity. I.e., as you zoom into the Mandelbrot Set, you see similar features to an unzoomed Mandelbrot Set. The GUI for this assignment has been set up to allow zooming, based on where the user clicks on the current Mandelbrot Set in the window. The user may click on an area to zoom in, or shift-click on an area to zoom back out. Your **`mandelbrotSet()`** function will be passed in a number of parameters that you will use to determine which pixels in the zoomed window will be in the Mandelbrot Set, the number of iterations you will use in the recursion, and the color(s) of the resulting image. The details of the parameters are described below.

```
void mandelbrotSet(GWindow& gw, double minX, double incX,
                   double minY, double incY, int maxIterations, int col
```

The **`gw`** parameter is the same as in the other two parts. In the starter code, we have already calculated the height and width of the window, created an image for the window, 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.

```
void mandelbrotSet(GWindow& gw, double minX, double incX,
                   double minY, double incY, int maxIterations, int col
```

The **`minX, incX, minY,`** and **`incY`** parameters define the range of values you will inspect for inclusion in the Mandelbrot Set. **`minX`** corresponds to the left-most column in your grid, and it forms the real part of the first point you should check. **`minY`** corresponds to the top-most row in your grid, and it forms the coefficient of the imaginary part of the first point you should check. In other words, the first point you check is going to be the complex number minX + minY*i*, which you could create as follows:

`Complex coord = Complex(minX, minY)`

If you pass this value into your **`mandelbrotSetIterations()`** function, that function will return a number of iterations needed to determine if the coodinate is unbounded. If the function returns maxIterations, we consider the coordinate to be in the Mandelbrot Set. If it returns a number less than maxIterations, then the coordintate is not in the Mandelbrot Set. The reason we do not simply return a **`bool`** is because we can use the number of iterations to provide a range of pretty colors, based on the palette (see the section on color below).

The **`incX`** and **`incY`** parameters have already been scaled to the size of the GWindow, and they provide the increment amount you should use as you travese the pixel grid. E.g., the pixel at row=3, col=5 would have the following coordinate: **`Complex(minX + 5 * incX, minY + 3 * incY);`** If your **`mandelbrotSetIterations()`** function returns maxIterations for that coordinate, it will be considered in the Mandelbrot Set, and should be set to show on the screen in the given color.

```
void mandelbrotSet(GWindow& gw, double minX, double incX,
        double minY, double incY, int maxIterations, int color)
```

You should use the **`maxIterations`** parameter in your **`mandelbrotSetIterations()`** function as one of the base cases for your recursion. If you have recursed **`maxIterations`** number of times and the absolute value of the coordinate remains bounded (less than 4), you should return maxIterations. In the GUI, the "size" value determines the maxIterations, and if you do not enter this value in the GUI, it defaults to a value of 200 iterations. Note: the number of iterations places an O(n^2) complexity on your calculation, so it can be slow. However, the more you zoom, the more iterations you will need to determine whether a point is in the Set, and the more precise your results will be.

```
void mandelbrotSet(GWindow& gw, double minX, double incX,
        double minY, double incY, int maxIterations, int color)
```

The **`color`** parameter is used to determine the color of your picture. If the color that is passed in is non-zero, you can simply use that number in your grid to show that a coordinate is in the Mandelbrot Set. In other words, if your **`mandelbrotSetIterations()`** function returns **`maxIterations`** for a coordinate, then you should set the pixel you are working on to **`color`**.

If, however, the color is zero, you should use the palette of colors, based on the result from your **`mandelbrotSetIterations()`** function. The calculation is a simple one:

```
pixels[r][c] = palette[numIterations % palette.size()];
```

```
int mandelbrotSetIterations(Complex c, int maxIterations)
int mandelbrotSetIterations(Complex z, Complex c, int remainingIteratio
```

These two functions have the same name (they are "overloaded"), but they have different parameters. You should plan on having your **`mandelbrotSet()`** function call the first function, which in turn calls the second function to perform the recursion. In this case, we call the second function a "helper" function, because it performs the recursion and in this case is used to pass along the *z* parameter (which, by the recursive definition, starts at zero). The helper function performs the recursion, returns the number of iterations back to the first function, which in turn passes the result back to the original **`mandelbrotSet()`** function. The **`mandelbrotSet()`** function uses this information to color the grid pixel (or not), based on the result.

Expected output: See the example outputs below:

The basic Mandelbrot set in Blue, with a default size (maxIterations) of 200:

- Basic Mandelbrot Set in blue, maxIterations: 200
- Basic Mandelbrot Set in palette, maxIterations: 200
- Basic Mandelbrot Set in red, maxIterations: 200, clicked on (390,60)
- Basic Mandelbrot Set in purple, maxIterations: 200, clicked on (327,367) twice
- Basic Mandelbrot Set in purple, maxIterations: 500, clicked on (285,195) three times

Style Details:

As in other assignments, you should follow our **Style Guide** for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1 and 2 specs, such as the ones about good problem decomposition, parameters, using proper C++ idioms, and commenting. The following are additional points of emphasis and style contraints specific to this problem:

Recursion: Part of your grade will come from appropriately utilizing recursion to implement your algorithm as described previously. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary. Avoid "arm's length" recursion, which is where the true base case is not found and unnecessary code/logic is stuck into the recursive case. **Redundancy** in recursive code is another major grading focus; avoid repeated logic as much as possible. As mentioned previously, it is fine (sometimes necessary) to use "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment.

Variables: While not new to this assignment, we want to stress that you should not make any global or static variables (unless they are constants declared with the **`const`** keyword). Do not use globals as a way of getting around proper recursion and parameter-passing on this assignment.

Collections: Do not use any collections on any of the graphical algorithms in this part of the assignment.

Frequently Asked Questions (FAQ):

For each assignment problem, we receive various frequent student questions. The answers to some of those questions can be found by clicking the link below.

Fractals FAQ (click to show)

Possible Extra Features:

Here are some ideas for extra features that you could add to your program for a very small amount of extra credit:

- **Sierpinski colors:** Make your Sierpinski triangle draw different levels in different colors.
- **Mandelbrot Set Gradient:** The Wikipedia page on the Mandelbrot Set dicusses ways to make a smooth gradient for the images, as opposed to using a palette. Implement a smooth gradient in your code.
- **Mandelbrot Set iterations**: The number of iterations needed to get a clear picture depends on the zoom. In the GUI, you can manually set this value, but you might want to use the minX, incX, and minY, incY parameters to determine the number of iterations, rather than the maxIterations passed into the function.
- **(Advanced) Mandelbrot Parallelism** Every point in the Mandelbrot set is independent of every other point, and thus this part of the assignment can be parallelized. You might do this using PThreads.
- **Add another fractal:** Add another fractal function representing a fractal image you like. You'll have to do some reconstructive surgery on the GUI to achieve this, and/or swap it in place of one of the existing fractal functions (make sure to turn in your non-extra-feature version first, so as not to lose your solution code).
- **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one with the standard file name without any extra features added (or with all necessary features disabled or commented out), and a second one whose file name has the suffix **-extra.cpp** with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously submitted files will not be lost or overwritten.