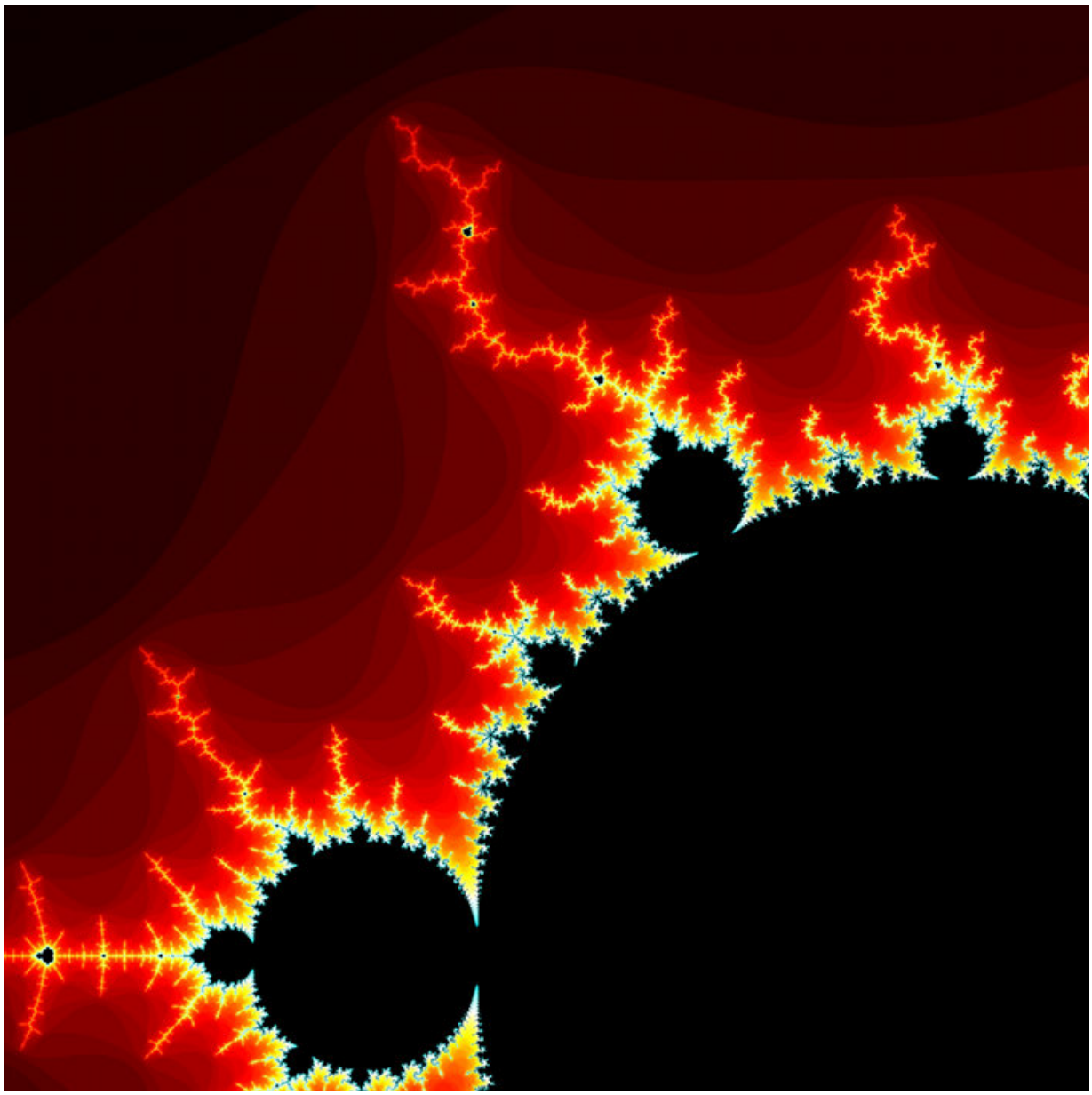
COMP1511: Assignment - 01, Mandelbrot

This is a draft version, it may change. You should regularly check here if there are any changes, in the change log at the top of this page. We will post a notice on the class webpage if there are any major changes.

The planning blog section of this assignment is due on
Sunday 26th August (end of week 5),
at **11:59:59pm** (Sydney time).

The remaining sections of this assignment are due on
Sunday 9th September (end of week 7),
at **11:59:59pm** (Sydney time).

The Mandelbrot set, named after its inventor Benoit B. Mandelbrot is astonishingly complex and detailed at all levels of abstraction.



It has breathtaking beauty. It always reminds me of the glorious complexity and ultimate unknowability of the universe. It will probably always remind you of Assignment 1 in your first computing course.

Objectives

- You produce a working correct Mandelbrot image server which you can use to explore the Mandelbrot set.
- You successfully manage this multi-part project.
- You write clear, high-quality, correct code.
- You use your code to produce your own striking and impressive images of the Mandelbrot set and share these with others.
    - To do this you will need to explore and discover/select interesting regions of the set.
    - Then you will need to design an effective color map to display the features of the region in the most effective way you can, to produce the most beautiful images you can.

Summary

Step 1: You are to write functions related to generating the Mandelbrot set, which (when combined with the provided server code) will generate and serve images of small regions of the Mandelbrot set (given a location and the level of zoom to use).

Step 2: You will then use this tool to explore the Mandelbrot set and produce images of your most interesting journeys and most beautiful discoveries.

In this task we expect you to:

- create a **high level plan** for your solution to the problem,
- produce **correct**, **clearly-written**, and **rigorously tested** C code, and
- for this to be **delivered on time**.

We are not only interested in the software you write, but also **how you use it to solve a problem**: to **explore** and **document** interesting parts of the Mandelbrot set.

We will be assessing the following aspects of your assignment:

- your ability to plan your approach to a programming project;
- the **correctness** of your code;
- your ability to produce code that implements a **particular specification**; and
- your **programming style**;

Before you begin...

Making a plan

You should first document how you plan to approach the task.

In your plan, you should take care to:

- provide an outline for *when* you will have each section of the assignment complete,
- show *how* you will know that you have completed each section, and
- outline what strategies you will use to keep to your schedule.

The blog post doesn't have to be long (it should be about 200 - 500 word) and having a list of bullet points is appropriate.

You must also make sure that your blog is on WebCMS3 and is completed **before** you begin work on your assignment. Your peers should not be able to read your blog post (set it to viewable by Course Staff).

> The blog section of this assignment is due on
> **Sunday 25th August**,
> at **11:59:59pm** (Sydney time).

Understanding the problem

It's important to make sure you've fully understood the problem before you try to start writing a solution.

Get the files

You are provided with two header files for this assignment:

- `mandelbrot.h`
  - defines constants `TILE_SIZE`, `MAX_STEPS`, and `NO_ESCAPE`.
  - defines structures `struct pixel` and `struct complex`.
  - declares prototypes for `serveMandelbrot`, `drawMandelbrot`, `escapeSteps`.
- `pixelColor.h`
  - defines prototype for `pixelColor`.

You are also provided with several stub files for this assignment:

- `mandelbrot.c`
  - This should contain all of your Mandelbrot-related code including implementations for `serveMandelbrot`, `drawMandelbrot`, `escapeSteps`.
- `pixelColor.c`
  - This should contain all of your color-related functions including implementations of `pixelColor`.
- `server.c`
  - This contains all of the server-related code, including the `main` function for the server.
- `drawTile.c`
  - This contains a simple program that creates single tiles from the command line.

All of these .c files should `#include "mandelbrot.h"`.

In addition to this, `mandelbrot.c` and `pixelColor.c` should also `#include "pixelColor.h"`.

Implement the function prototypes from the .h files in the correspondingly named .c files. Make any additional helper functions you write `static`. Don't change the .h files: we'll use our own copy of the .h files when marking -- you'll only submit the two .c files (mandelbrot.c and pixelColor.c).

More Details: The Mandelbrot Set

The Mandelbrot Set is a complex system, that arises from a very simple algorithm.

Any point in 2-dimensional complex space (any *complex number*) is either *in* the Mandelbrot Set or *not in* the Mandelbrot Set.

Complex number Recap

All you need to know about complex number for this assignment is the following rules:

- A complex number is the sum of two components, one *real* component and one *imaginary*.
- The imaginary component is some *real* multiple of $\mathbf{i}$, where $\mathbf{i} = \sqrt{-1}$.
- An complex number with a real part of x and imaginary part of y could be described as $(x + y \times \mathbf{i})$.
- Addition of two complex numbers:

$$(a + b\mathbf{i}) + (c + d\mathbf{i}) = ((a + c) + (b + d)\mathbf{i})$$

- Multiplication of two complex numbers:

$$\begin{aligned}&(a + b\mathbf{i}) \times (c + d\mathbf{i})\\ &= ac + ad\mathbf{i} + bc\mathbf{i} + bd\mathbf{i}^2\\ &= ac + (ad + bc)\mathbf{i} + bd\sqrt{-1}^2\\ &= ((ac - bd) + (ad + bc)\mathbf{i})\end{aligned}$$

- The *modulus* of a complex number $\mathbf{z} = (x + y\mathbf{i})$:

$$|\mathbf{z}| = |(x + y\mathbf{i})| = \sqrt{x^2 + y^2}$$

Calculating the Mandelbrot Set

To determine if a complex number $\mathbf{z}$ is in the Mandelbrot set we can use the following loop:

- Let $\mathbf{z}$ be the complex number to test.
- Let $\mathbf{w}$ be the complex number $(0 + 0\mathbf{i})$.
- While $|\mathbf{w}| < 2$
    - Change $\mathbf{w}$ to equal $(\mathbf{w} \times \mathbf{w}) + \mathbf{z}$.

If the loop ever exits, then the complex number $\mathbf{z}$ is not in the Mandelbrot Set.

This issue with this algorithm is that it can loop infinitely and it's hard to tell if a number is in the set or not without waiting for an infinite amount of time.

Instead, we *approximate* the Mandelbrot Set by only looping a certain number of times. If, after that number of times, $|\mathbf{w}| < 2$ is still true, we assume it is in the set.

Another interesting property of the Mandelbrot Set occurs when you take the number of loops it takes for a given point to show that it is not in the set. That is, how many times did the loop have to run before $|\mathbf{w}| < 2$ was false. If we take that number and plot it as an image where each pixel is turned into a complex number we get something similar to what is shown on almondbread.

Summary

For this assignment, you are required to complete the following 3 functions within a C program.

```
int escapeSteps(struct complex c);
```

escapeSteps

`escapeSteps` takes a complex number as represented in a `struct complex` and returns the number of iterations in the loop described above were needed to show that it was not in the Mandelbrot set, or that it was assumed to be *in* the set.

drawMandelbrot

```
void drawMandelbrot(
    struct pixel pixels[TILE_SIZE][TILE_SIZE],
    struct complex center,
    int z
);
```

`drawMandelbrot` takes a 2-dimensional array of `struct pixel` and draws on it a single *tile* of the mandelbrot set. The argument `center` is a complex number which describes the point at the center of the *tile*. The argument `z` is the *zoom* level of the tile. This describes the distance between the center of each pixel. Each pixel will be 2^{-z} apart from each other.

pixelColor

```
pixel pixelColor(int steps);
```

`pixelColor` takes the number of steps that were taken to determine that a complex number was not in the Mandelbrot set. For values that are assumed to be in the Mandelbrot set, it takes the special value `NO_ESCAPE`. This function must then return a color for that number of steps as a `struct pixel`. You can use whatever scheme you want to map steps to colors, but try to make sure that it's clear which values are in the set and which are not.

Viewing your Mandelbrot set

To view your Mandelbrot set, you can compile your solution together with `server.c` with the following command in `dcc`.

```
$ dcc -o server server.c mandelbrot.c pixelColor.c
```

You can then run the `server` program which will give you an address you can use to view your Mandelbrot set. The URL that you see for your Mandelbrot may change and will not be the same as anyone else's. When you start the server, you should see something like the following.

```
$ ./server
[SERVER] Starting mandelbrot server 3.00
[SERVER] Serving fractals since 2011
[SERVER] Access this server at http://weber.cse.unsw.edu.au:12345/
[SERVER] Waiting for requests...
[SERVER] Served 0 requests
```

You can also use the command line tool `drawTile` to create individual tiles with your solution. To compile, use the following command.

```
$ dcc -o drawTile drawTile.c mandelbrot.c pixelColor.c
```

Then you can use `drawTile` to create a specific tile.

```
$ ./drawTile
Enter x co-ordinate: -0.3
Enter y co-ordinate: 0
Enter zoom level: 7
Wrote tile to 'tile.-0.300000_0.000000_7.bmp'.
$ ./drawTile -0.3 0 7
Wrote tile to 'tile.-0.300000_0.000000_7.bmp'.
```

Completing the assignment

Step 1: Generating Tiles

Make sure to also check out the how to approach this task page! The content below is just a high-level overview of the task.

We have provided you with an HTTP server to serve image tiles showing a part of the Mandelbrot set.

Your server is to serve up 512 x 512 pixel BMP images. Given the x, y co-ordinates (doubles) and a zoom level (an int) the server returns a 512x512 pixel tile of the region of the Mandelbrot set centered at that point.

The desired `x`, `y`, and `zoom` will be encoded in the URL of the tile file requested from the server. The *hostname* and *port* of the server will change depending on the machine you use and who runs it. The `server` program will tell you what to use when it starts.

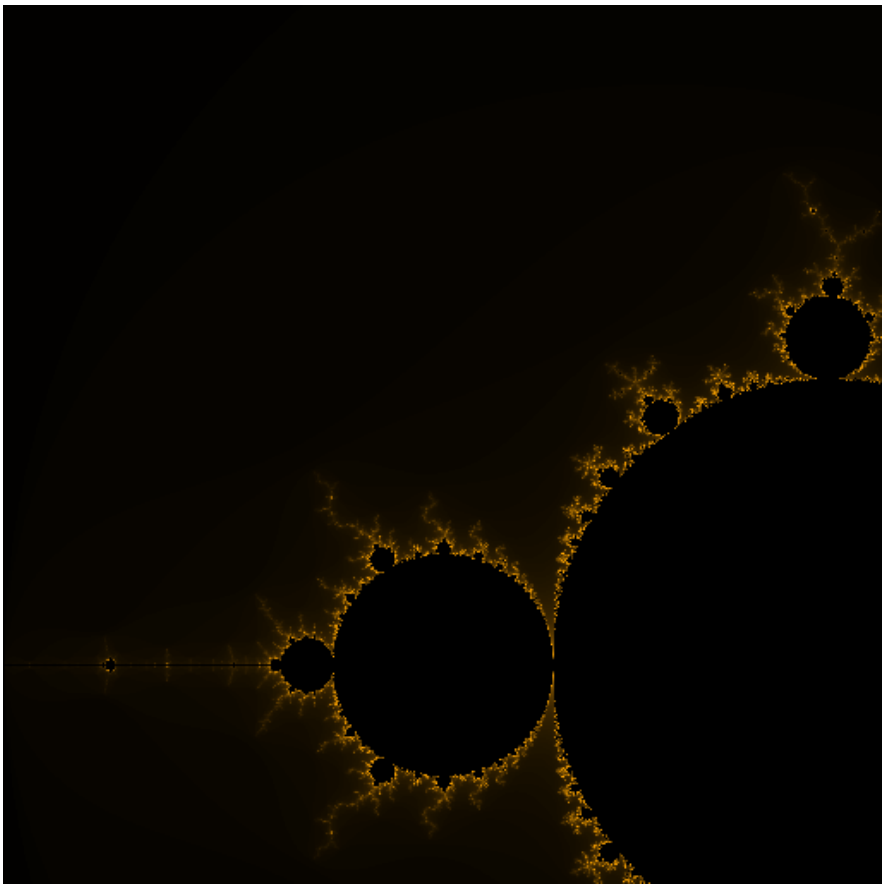
The URL is structured:

`http://hostname:port/mandelbrot/2/zoom/x-coord/y-coord/tile.bmp`

For example, if your server was running at `vx1.cse.unsw.edu.au:12345` you typed:

`http://vx1.cse.unsw.edu.au:12345/mandelbrot/2/8/-1.0/0.5/tile.bmp`

in your web browser, then the server should return an HTTP response containing an image of the region of the Mandelbrot set with center $(-1.0, 0.5)$ and zoom level 8 (see below for how to interpret the zoom level). That particular tile should look something like this:



And if you typed

`http://vx1.cse.unsw.edu.au:12345/`

without any file name at all, your server should just respond with the HTML code:

```
<script src="http://almondbread.cse.unsw.edu.au/tiles.js"></script>
```

This code will cause your browser to download and display a funky viewer (like this!) with which to navigate your beautiful Mandelbrot set.

IMPORTANT NOTE: The viewer code will not work unless your code can successfully serve images by browsing directly to them. So do this step only after you've gotten the previous step working!

The Region

The x, y co-ordinates give the center of the image. The zoom level z gives the distance between pixels as follows:

distance between pixels = 2^{-z}

So, given $z = 8$, $x = -1.0$, and $y = 0.5$, the distance between adjacent pixels in the image returned will be $2^{-8} = 1/256 = 0.00390625$, and the requested image is for a region of the set 1.9960938 wide by 1.9960938 high (as $511 \times 0.00390625 = 1.9960938$), centered at $(-1.0, 0.5) = (-1.0 + 0.5\mathbf{i})$.

Pixel iterations, without complex numbers

To generate an image of part of the Mandelbrot set follow the following process for each pixel in the image: find the (x, y) co-ordinates of the pixel's position in the region, and then iterate the Mandelbrot equation:

$$\text{Point}_n = (\text{Point}_{n-1})^2 + (x, y)$$

starting with $\text{Point}_0 = (0, 0)$ until you generate a point Point_n which is 2 or more away from the origin, or until you have performed 256 iterations.

Pixel iterations, with complex numbers

If you'd prefer a more complex solution, try this: pretend the region is an Argand diagram, a diagram of the complex plane.

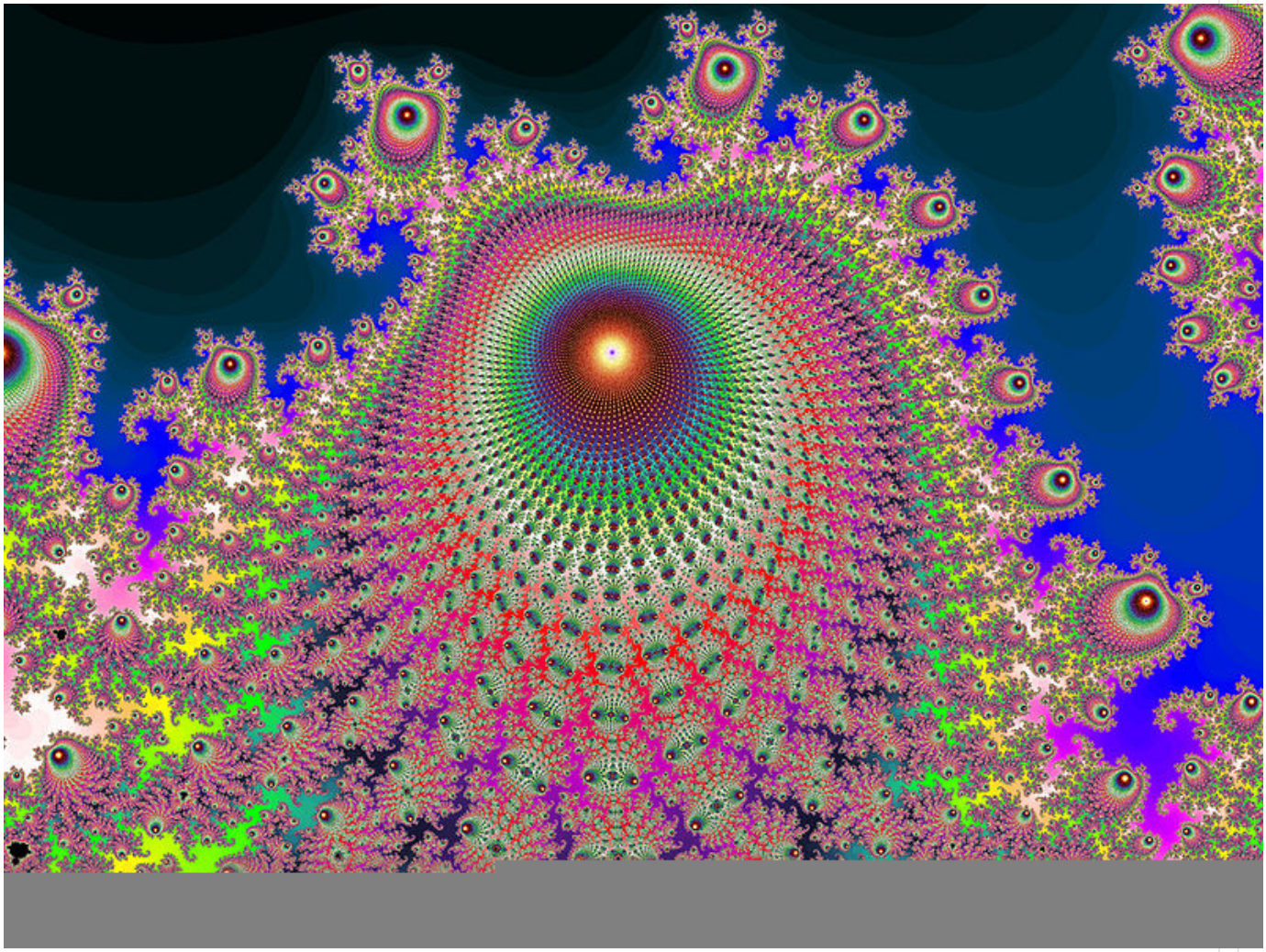
As before, when we find the (x, y) co-ordinates of the pixel's position in the region, we've actually found a complex number, $\mu = x + y\mathbf{i}$. Now, for any integer $n \geq 1$,

$$z_0 = 0 + 0\mathbf{i}$$
$$z_n = z_{n-1}^2 + \mu$$

If the magnitude of this new value, $|z_n| > 2$, the point has escaped. Rinse and repeat.

`escapeSteps` should then return the last *n* before the value escaped, or 256, whichever is smaller.

Step 2: Discover... and Color!



Source: Wikipedia, original upload 24 November 2005 by David R. Ingham CC-BY-SA 3.0

You may choose how to colour the pixels yourself, using your artistic, analytical and visual design skills. Some suggestions are given below, there are many ideas on the internet, and we'll discuss this more in lectures also.

The basic idea is to colour each pixel based on the number of iterations the above process takes.

The simplest approach is to colour a pixel black if you took 256 iterations, white otherwise. This will give you a black and white image.

A more interesting way of colouring the pixels is to shade them grey -- with intensity depending on the number of iterations.

The most powerful and effective way of colouring the pixels is to use a different RGB colour for each possible number of iterations. That is how the images on this page were generated. The artistry here lies in picking the colours and deciding how they correspond to the different iteration counts.

Put your color decision code into `pixelColor.c` .

If you put any other helper functions into `pixelColor.c` you must make them static. (put the word `static` at the start of the functions' declaration and definition).

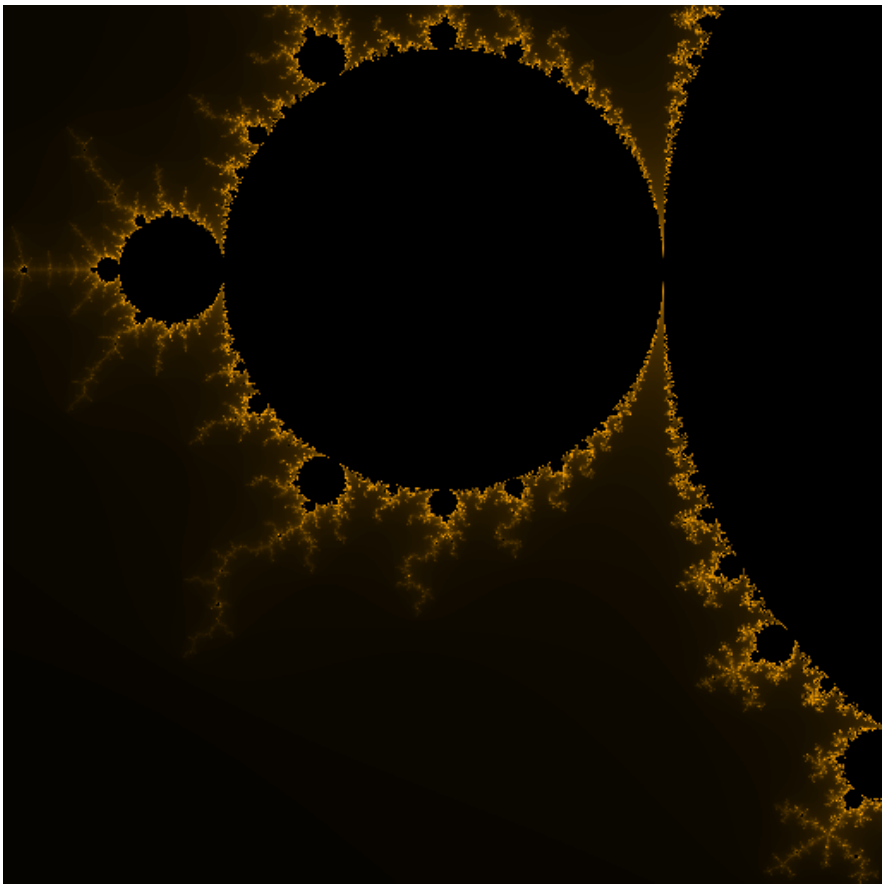
Testing / Debugging

You can use the browser linked from the lab pages to browse the set interactively. To just get single tiles and compare them with the tiles you generate you can ask the sample Mandelbrot server we have left running on the internet. Amusingly it's called the Almond Bread server -- a little jokeule of Julian's (he wrote the first version for us)

For example: to inspect the tile centred on (-1.0, -0.2) at zoom level 9, request the following image in your browser address bar:

http://almondbread.cse.unsw.edu.au/mandelbrot/2/9/-1.0/-0.2/tile.bmp

That url fetches this tile:



Submitting

You should submit `mandelbrot.c` and `pixelColor.c`. You don't need to upload any of the .h files, as these are already provided to the automarker.

The files mandelbrot.c and pixelColor.c should #include *and implement* the provided pixelColor.h and mandelbrot.h exactly.

The deadline for this task is Sunday of Week 7. Submit your two .c files on or before the deadline.

To submit, run the following command for the directory where your code exists.

```
$ 1511 classrun give assign1
```

Assessment

This assignment will contribute 12% to your final mark.

Mark breakdown

The mark breakdown for each component is:

- 20% planning blog
- 60% correctness
- 20% style

Correctness

60% of the marks for this assignment will come from the correctness of your program, i.e. what proportion of the autotests your program passes. These autotests **will not** be available to you during the course of the assignment, and will only be run after the due date of the assignment. Therefore, it is in your best interests to write your own extensive unit tests, so that you can catch any bugs in your program before we run our tests against it.

Style

20% of the marks for this assignment will come from hand-marking of the readability of the code you have written. These marks will be awarded on the basis of clarity, commenting, elegance, and style. In other words, your tutor will assess how easy it is for a human to read and understand your program.

Late Penalty

The assignment is due at 23:59:59 on Sunday 9th September, Australian Eastern Standard Time.

If your assignment is submitted after this date, then for each hour it is late, the maximum mark it can achieve is reduced by 1%. For example, if an assignment worth 74% was submitted 12 hours late (noon on Monday), the late submission would have no effect. If the same assignment was submitted 30 hours late (6am on Tuesday), the mark would be reduced to 70%, the maximum mark it can achieve at this time.

Plagiarism

Penalty	Description
-70%	Knowingly providing your work to anyone and it is subsequently submitted (by anyone).
-70%	Submitting any other person's work. This includes joint work.
0 FL for COMP1511	Paying another person to complete work. Submitting another person's work without their consent.

Resources

- How to approach this task
- Make sure you complete the bmp image activities from previous weeks
- pixelColor.h
- mandelbrot.h