Mandelbrot

Description

This program is made to draw the mandelbrot fractal.

Colors are calculated using normalized iteration count; histogram coloring, logarithmic smoothing, and linear regression are combined for smooth color transitions.

To speed up render times, it utilizes multiple threads to render multiple sections of the image at once.

When a thread is done, it will communicate with other threads to share the remaining load. This improves speed by an average of 53%.

The thread is set up for polymorphism; it's easy to redefine the rendering method while keeping synchronization.

Usage and Defaults

Program options and defaults can be shown by passing "-h".

See image folder for default render result.

```
Usage: ./mandelbrot [options]
Options:
    -h    this cruft
    -w    image width                              default: 1920
    -H    image height                             default: 1080
    -o    image output path                        default: out.png
    -j    jobs -- set this to your corecount       default: 1
    -c    complex bottom border     default: (-0.74364389269000009,0.13182587270999999)
    -C    complex top border        default: (-0.74364383269000001,0.13182593271000001)
    -i    fractal iterations                       default: 50000
    -I    bailout value                            default: 256

FOR COMPLEX NUMBERS: if you want to input, say, 2-3i, your option argument will be "(2,-3)".
```

Design

The following is a list of steps the program takes.

Main

Parse user options, change values if necessary
Allocate large arrays needed to hold values for mandelbrot
Create a mthread object for each job/thread
Call thread.spawn_render()
Periodically read progress variable and show percentage
When progress is full, for each mthread, call mthread.join()
Colorize photo using values obtained by mthreads

Mthread::spawn_thread() creates a thread executing mthread's main procedure.

Mthread procedure

Set synchronization flags to zero
For each row: (location 1)
    Add progress
    Calculate and set rows remaining (row_load)
    If syn flag set: Communicate with thread as shown by diagram below
    For each pixel:
        Find mandelbrot value (more details in citation), add to double array
        Using calculated value as index, add 1 to element in histogram array.
Set searching = true and load_finished = true
Obtain list of other threads, sort by largest load
count how many have load_finished set into loads_finished
If loads_finished == jobs - 1: return
For each thread, starting with the largest load:
    Double check thread status and communicate as shown by diagram below
Go to location 1

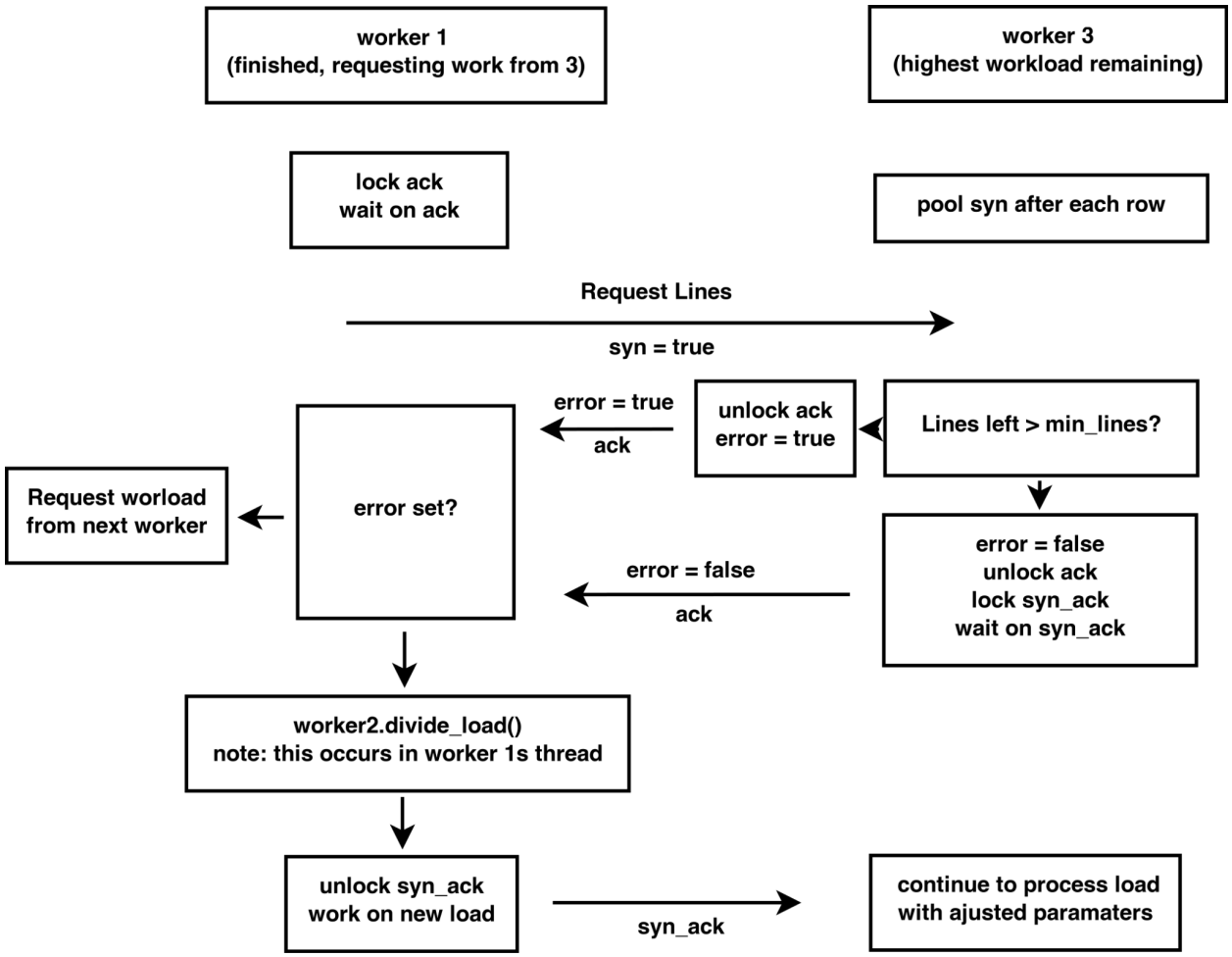
Known bugs:

If the coordinates supplied differ from the images aspect ratio, renders are distorted to fit the given coordinates.

It may be possible to optimize thread locking to decrease code complexity.

It would be faster to use a GPU for rendering; it wasn't possible to learn OpenCL or Cuda in the allocated time.

The following diagram shows communication during request to divide workload:



To determine a pixel's color, there's three major arrays: the histogram and value map, which are calculated by each thread, and the hue map, which is calculated after.

The histogram contains how often each floored value in the value map is hit.
The value map contains a calculated double for each pixel.

Colors are generated using the following procedure:

1. Get sum of all values in histogram
2. For each histogram element:
   Current saturation += histogram element / histogram sum
   Add curent_saturation to hue array
3. For each pixel:
   Use floored value as index in hue array
   Record hue of current and next index
   Find the midpoint of the two colors using value found in value map

Roadmap

Feature	Size
Figure out libpng, create png object	**Large**
Figure out std threads, get mthread independently threading	**Medium**
Draw and save basic fractal with multiple threads	**Small**
Finish fractal coloring shading, coloring, etc	**Large**
Learn more about std synchronization primitives; plan thread synchronization after workload of a thread is complete	**Large**
Implement threads, test for race conditions, debug any issues	**Extra large**
Refactor old code	**Medium**
Delete and replace constants with getopt options	**Medium**
Find some cool coordinates for screenshots and default options	**Medium**

Postmortem

I panicked a bit about race conditions and synchronization, but after I took time to understand and test concepts such as condition variables it was easier to implement than expected. I timed the program before and after I enabled idle threads to share loads, and it was about 53% faster; distributing the workloads paid off. It was very exciting to create the images that are included in the screenshots, even if I wish I could have more time to tweak the coloring off of a simple hue ramp.

In the end, the synchronization code became very disorganized. Due to a lack of time, I didn't have time to fully think through all possible race conditions. As a result, some of the code may be a bit slow; I lock all publicly shared data at once, and threads needing to access it wait until it's available, regardless of whether or not the data it needs is about to be modified. Additionally, I'm not 100% sure if race conditions are completely eliminated… I had a single segfault, and no matter how hard I tried, I could not reproduce it.

I also feel like I didn't *really* understand the logarithmic aspect of the smooth shading. While it only contributes to smoothing out the histogram coloring method, I wish I could optimize it; with finals in other classes coming up I couldn't afford the time.

Next time I'd give myself more time to plan the synchronization aspect, and fully understand the logarithmic smoothing.

Citations

Condition variable: https://cplusplus.com/reference/condition_variable/condition_variable/

Mandelbrot shading: (smooth shading section)
https://en.wikipedia.org/wiki/Plotting_algorithms_for_the_Mandelbrot_set#Continuous_(smooth)_coloring

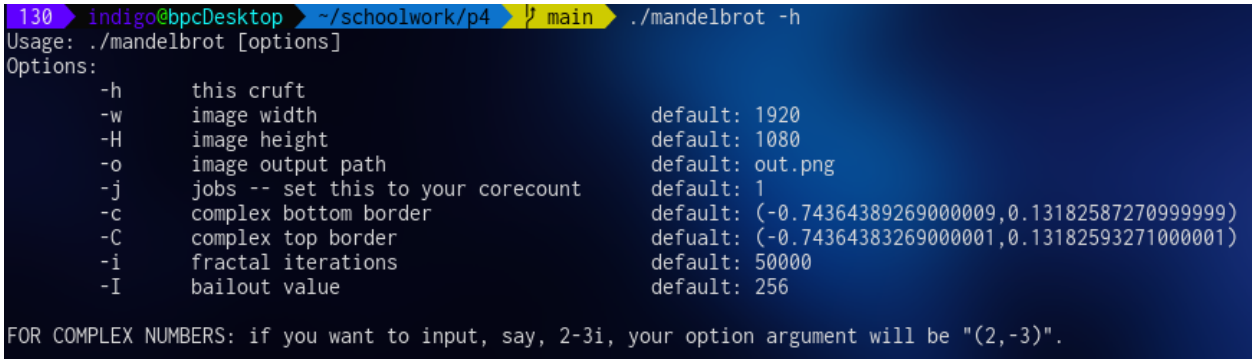
Mandelbrot general concept: https://en.wikipedia.org/wiki/Mandelbrot_set

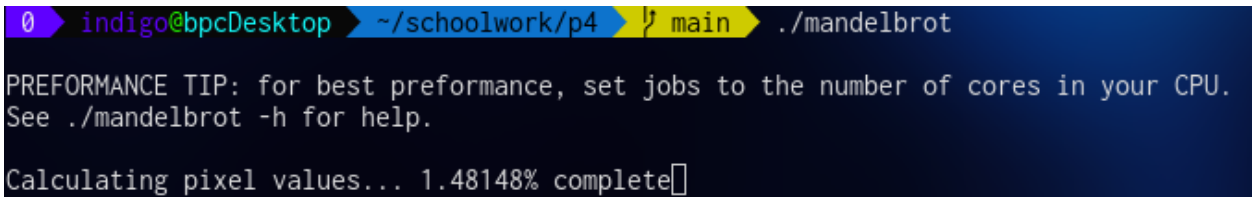
Mutex: https://www.cplusplus.com/reference/mutex/

Starting thread with member function:
https://stackoverflow.com/questions/10673585/start-thread-with-member-function

Threads: https://www.cplusplus.com/reference/thread/thread/

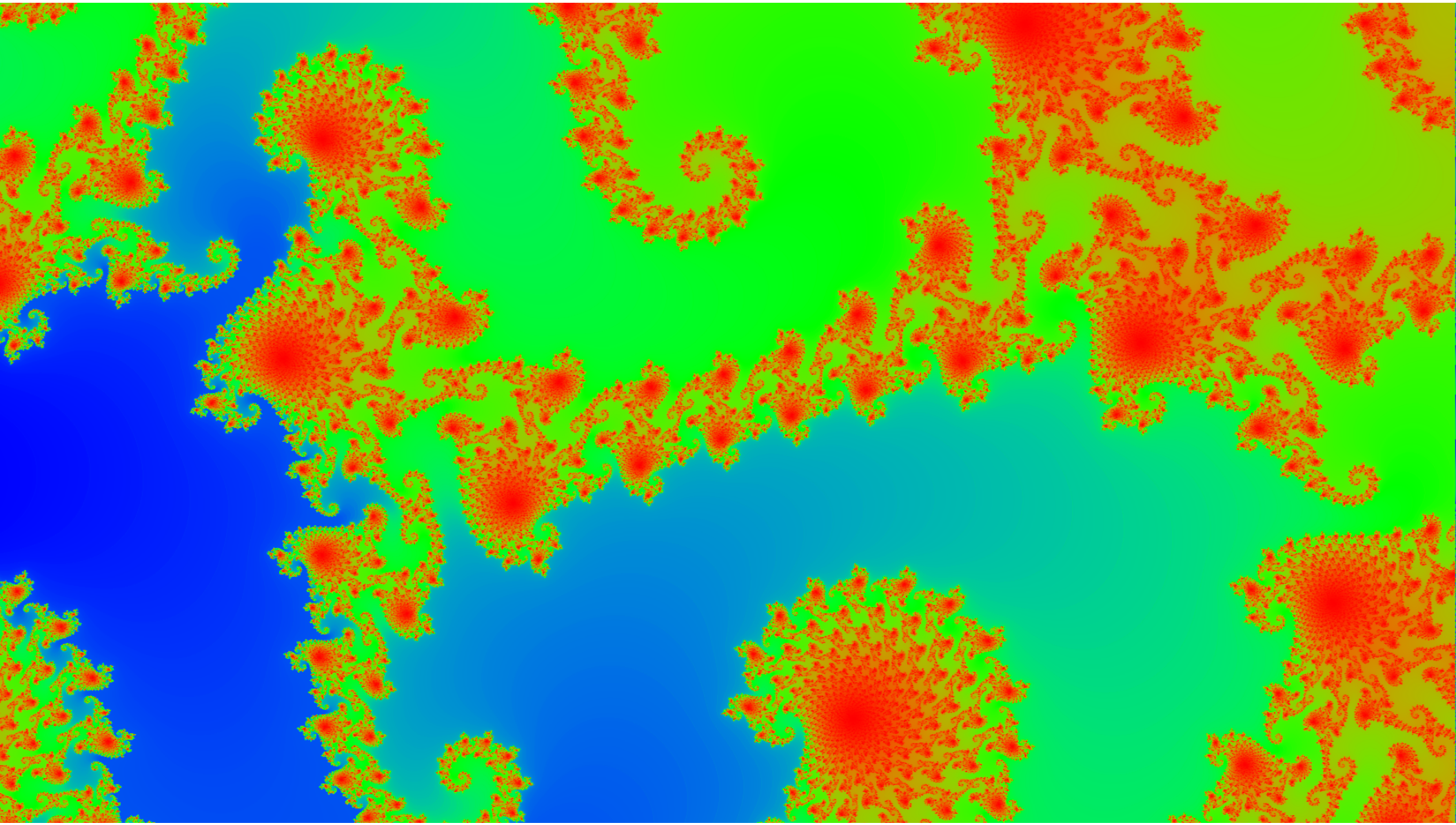
Screenshots



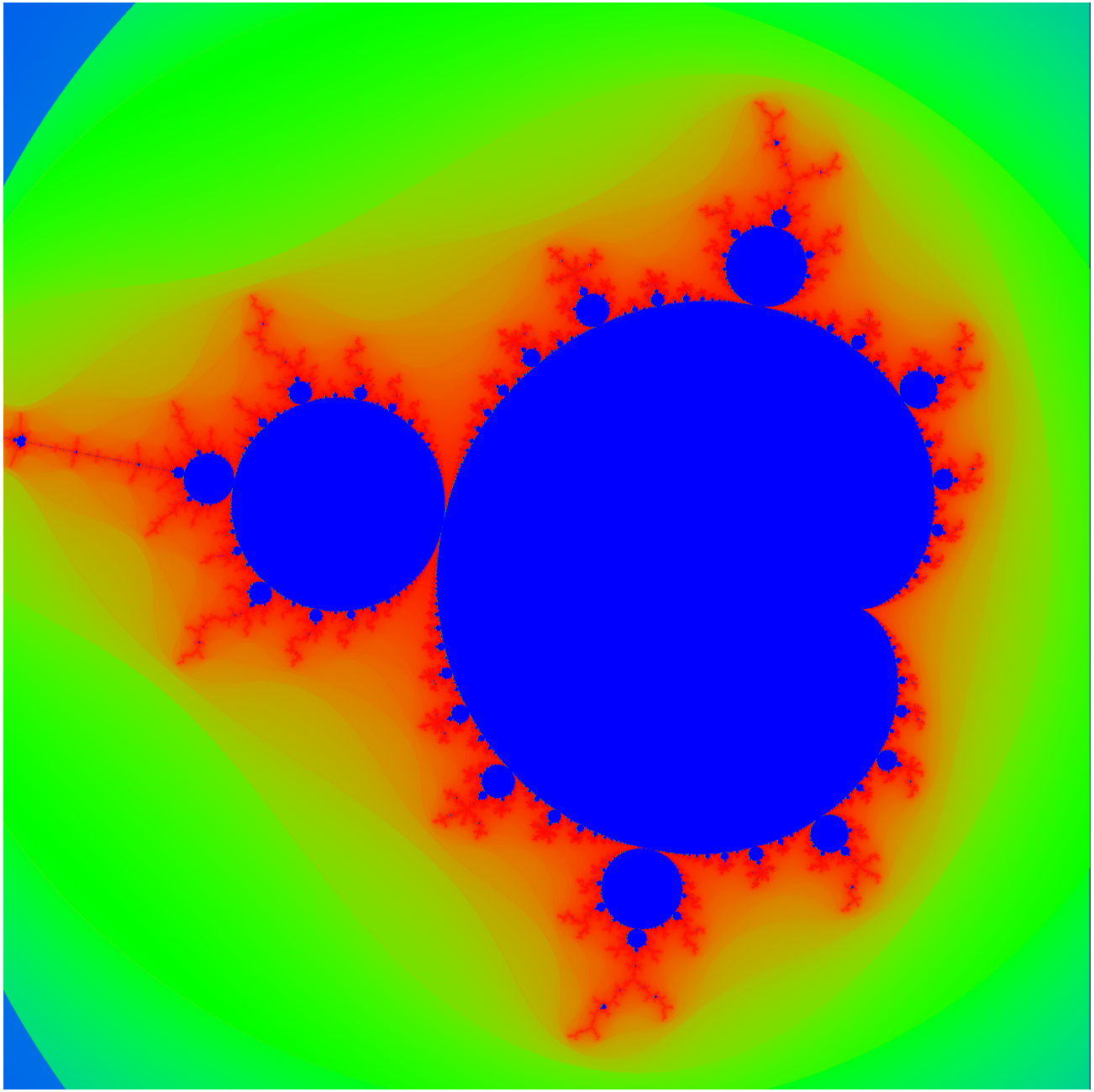


Full scale renders are in docs/images.
Program running is in docs/demo.mp4

./mandelbrot -w 1920 -H 1080 -o demo_2.png -c "(-0.74364387269, 0.13182589271)" -C "(-0.74364385269, 0.13182591271)" -i 50000 -I 256
“Seahorse valley” - higher zoom of last image.



./mandelbrot -w 2000 -H 2000 -o demo_3.png -c "(-2, -1)" -C "(1, 1)" -i 1000 -I 2
All the way zoomed out. Easiest to render.



./mandelbrot -w 1920 -H 1080 -o demo_1.png -c "(-0.74364389269,0.13182587271)" -C "(-0.74364383269,0.13182593271)" -i 50000 -I 256
“Seahorse valley”