Exploring Fractals

CS410P

Ben Truong Ka Hoo Chow

6/14/23

Table of contents

Recursion - The Pythagoras Christmas Tree Fractal

Design Paradigm & Mathematical Description

L-System - Autumn Tree

Figure 1: Autumn Tree

Design Paradigm & Mathematical Description

IFS - Dragon

Design Paradigm & Mathematical Description

Complex number - Mandelbrot Zoom

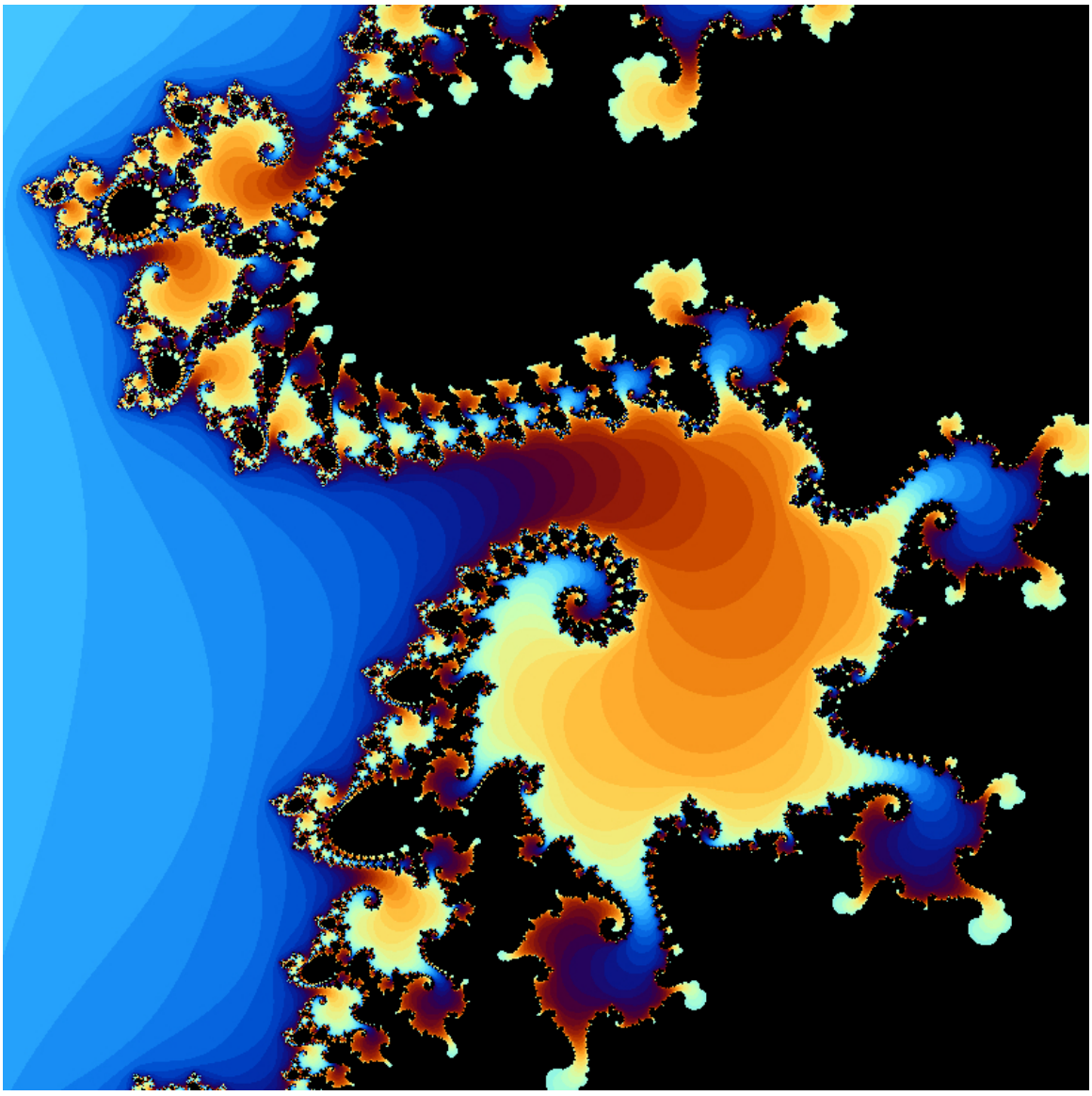

Figure 2: Mandelbrot

Design Paradigm & Mathematical Description

asdfjafndafa asdlfkjaslfka asdlfkj

Julia Circle

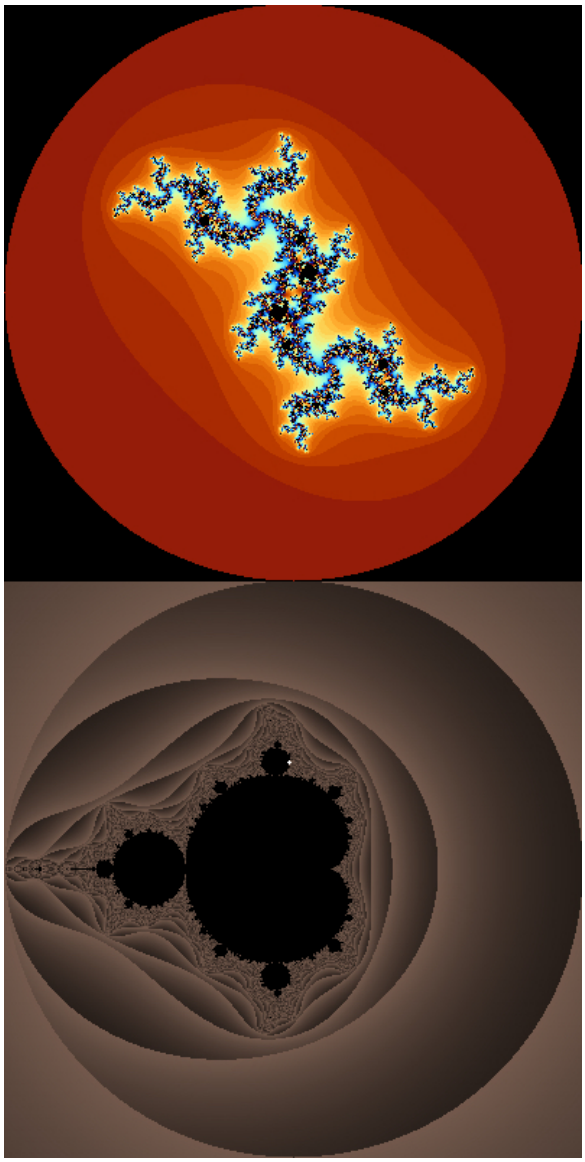

Figure 3: JULIA

Design Paradigm & Mathematical Description

The "check" and "julia_check" functions determine whether a complex number diverges or stays bounded by applying the iterative equation. They return a complex number if divergent and zero if convergent.

The "mandel" function generates a Julia set fractal by iterating over a grid of complex numbers and checking their divergence. It uses the "check" function to determine the divergence behavior and assigns colors based on the result.

The "julia" function generates a variation of the Julia set fractal by iterating over a grid of complex numbers and checking their divergence based on a specific constant 'c.' It assigns colors to the points based on the number of iterations required for divergence.

The "julia_set_circle" function generates a series of Julia set fractals by varying the 'c' value in a circular pattern.

Complex numbers, represented as $z = a + bi$, where a is the real part and b is the imaginary part, form the foundation of the Julia set fractal. By repeatedly applying an iterative equation of the form $z = z^2 + c$, where z is the current complex number and c is a constant value, the fractal pattern emerges. The divergence test determines whether a complex number is part of the Julia set by checking if it diverges or remains bounded. By assessing the magnitude of z after each iteration, if it exceeds a certain threshold, the number is considered to have diverged. To enhance the visual representation, color coding is applied, assigning different colors or shading to diverging points based on the number of iterations it takes for them to diverge. This combination of mathematical concepts results in captivating and intricate Julia set fractal images

Artistic Description:

In the code, the lines:

```
point[0] = cos(angle*M_PI/180) * 0.75;
point[1] = sin(angle*M_PI/180) * 0.75;
```

We used trigonometric functions (cosine and sine) to calculate the x and y coordinates of a point on a circle based on the given angle. We then scaled down the coordinates to fit within a smaller circle with a radius of 0.75. As you can see in the video, this results in a Julia set for every coordinate returned by the calculations.

Appendix

References

Include all bibliographical references used here. See this link to learn about adding references to the document.

Mandelbrot Zoom

```
#include <stdio.h>
#include <complex.h>
#include "FPToolkit.c"

int Wsize = 800;
int Hsize = 800;

int check (complex c) {
    // return a complex if diverge - return 0 if converge
    complex z = 0 + 0*I;
    for (int i = 0; i < 100; ++i) {
        z = z*z + c;
        if (cabs(z) > 2)
            // { printf ("Diverge\n");
            return i;
    }
    // printf ("Converge\n");
    return 0;
}

void mandel_zoom (double a, double b, int times) {
    double range = 4.0;
    for (int i = 0; i < times; ++i) {
    double unit = range / Wsize;
    // a is the real part, b is the coefficient
        for (double a_sub = a - range/2; a_sub < a + range/2; a_sub += unit) {
            for (double b_sub = b - range/2; b_sub < b + range/2; b_sub += unit) {
                int i = check(a_sub + b_sub*I);
                if (i == 0) {
                    G_rgb(0,0,0);
                } else {
                    double t = (double)i / 100.0; // Normalize iteration count
                    // Define the color gradient
```

```
                    double red = sin(5 * M_PI * t);
                    double green = sin(5 * M_PI * (t + 1.0 / 3.0));
                    double blue = sin(5 * M_PI * (t + 2.0 / 3.0));
                    G_rgb((red + 1.0) / 2.0, (green + 1.0) / 2.0, (blue + 1.0) / 2.0);
                }
                G_point((a_sub - a + range/2)*Wsize/range, (b_sub - b + range/2)*Wsize/range);
            }
        }
            // Change this number for zoom factor
            range -= 0.01*range;
            // uncomment/comment this for live view of code
            for (int i = 0; i < 500000; ++i) {
                if (i % 100000 == 0)    {
                    G_display_image();
                    usleep(100);
                }
            }

            // uncomment/comment this for making movie bmp
            // char fname[400];
            // sprintf(fname, "img%04d.bmp", i);
            // G_save_to_bmp_file(fname);

    }

}

int main () {

    G_init_graphics (Wsize, Wsize) ;  // interactive graphics
    // clear the screen in a given color
    G_rgb (0, 0, 0) ; // black screen
    G_clear () ;

    // set the desired coordinates to zoom into and the number of clicks on that point.
    int click = 1000;
    double zoomx, zoomy;
    zoomx =-0.743643135;
    zoomy = 0.131825963;
    mandel_zoom(zoomx, zoomy, click);

    // mandel_zoom(-0.730000, 0.280000, 800);

    return 0;
}
```