



Exploring Fractals

CS410P

Ben Truong

Ka Hoo Chow

6/15/23

Table of contents

Recursion - The Pythagoras Christmas Tree Fractal	2
Design Paradigm & Mathematical Description	3
L-System - Autumn Tree	4
Design Paradigm & Mathematical Description	4
IFS - Dragon	5
Design Paradigm & Mathematical Description	5
Complex number - Mandelbrot Zoom	6
Design Paradigm & Mathematical Description	7
Julia Circle	8
Design Paradigm & Mathematical Description	9
Artistic Description:	9
Appendix	10
References:	10
Christmas Tree Code:	11
Autumn Tree Code:	14
Mandelbrot Zoom Code:	17

Recursion - The Pythagoras Christmas Tree Fractal

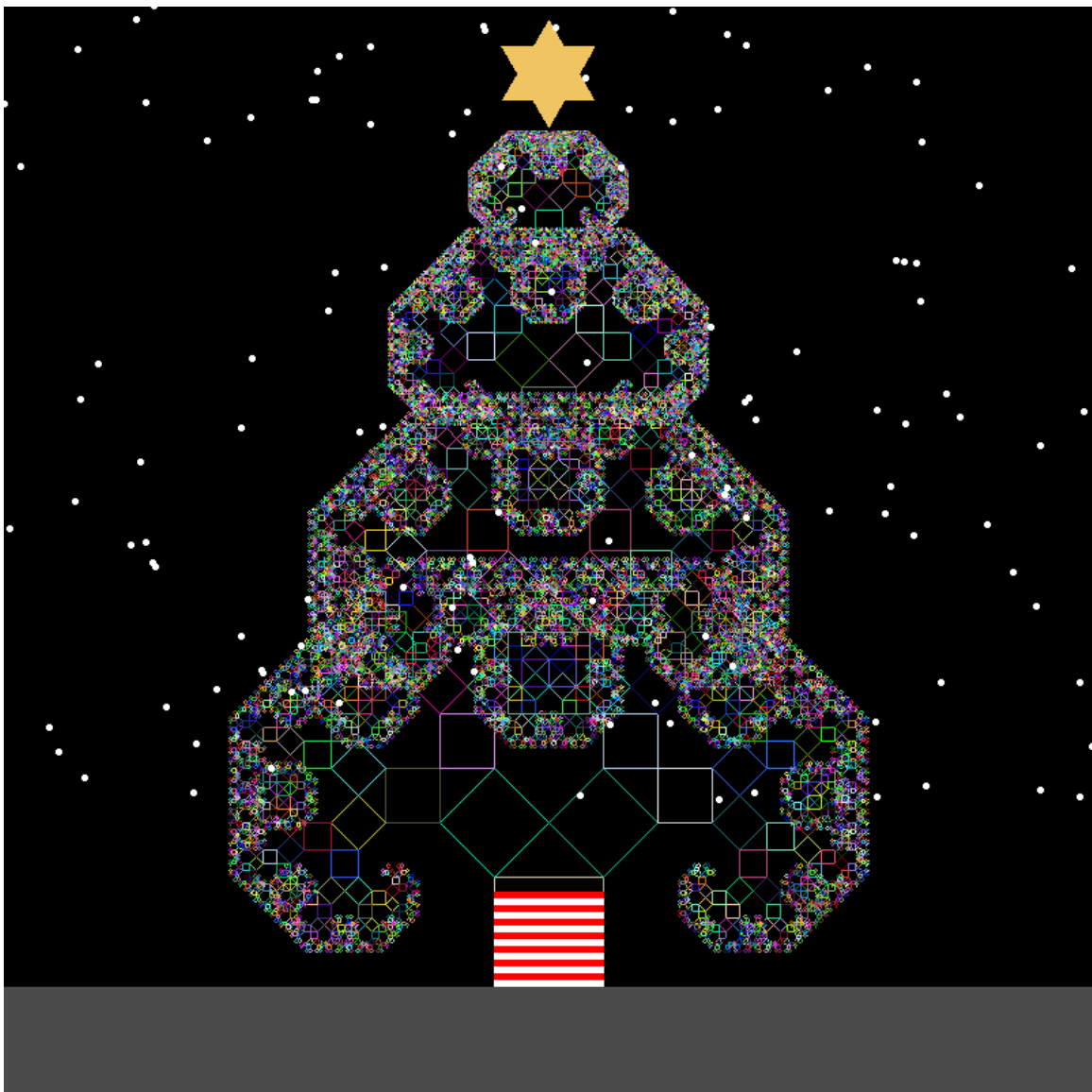


Figure 1: [Christmas Tree](#)



Design Paradigm & Mathematical Description

L-System - Autumn Tree



Figure 2: Autumn Tree

Design Paradigm & Mathematical Description

IFS - Dragon

Design Paradigm & Mathematical Description

Complex number - Mandelbrot Zoom

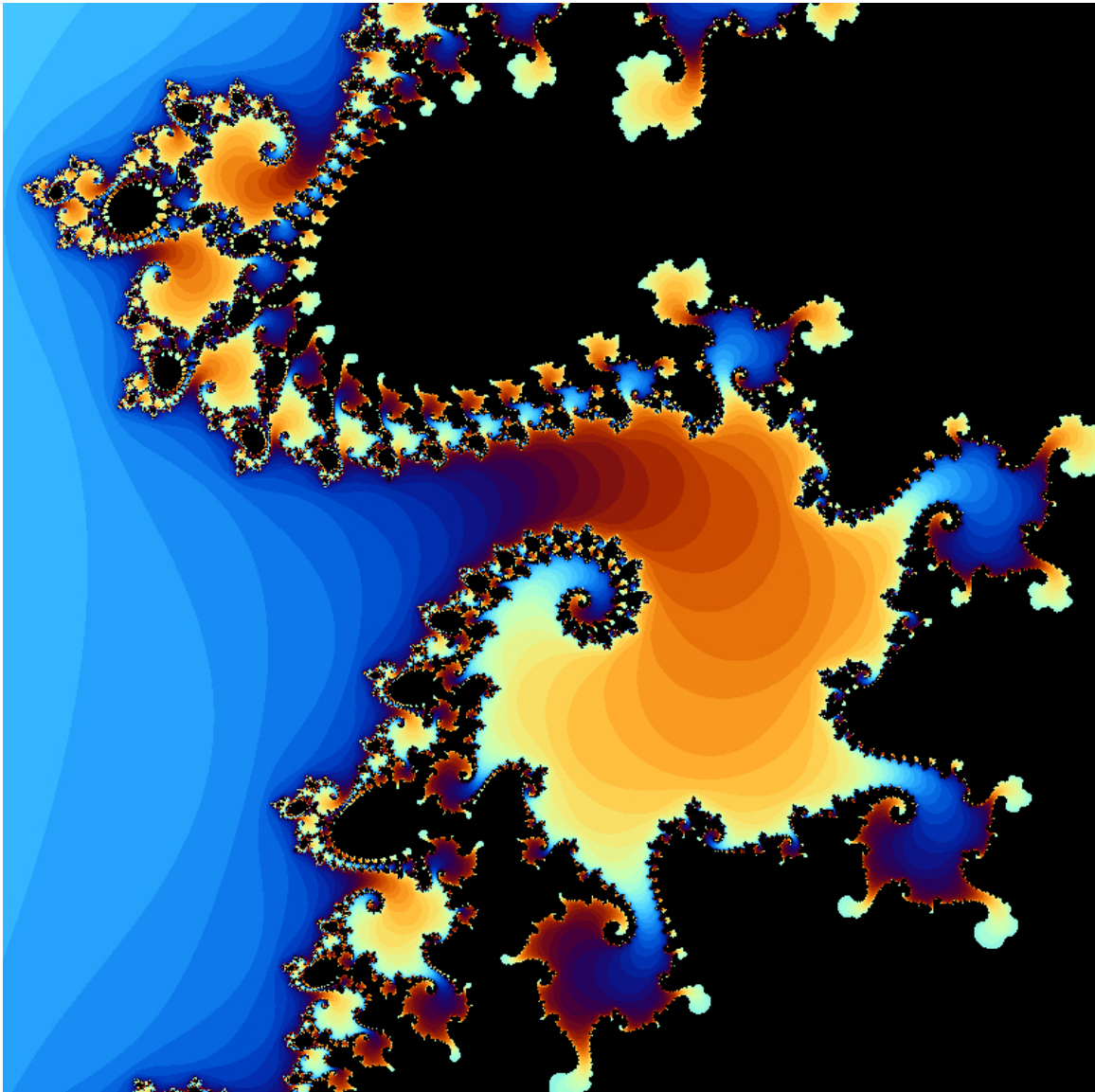


Figure 3: [Mandelbrot](#)



Design Paradigm & Mathematical Description

asdfjafndafa asdlfkjaslfka asdlfkj

Julia Circle

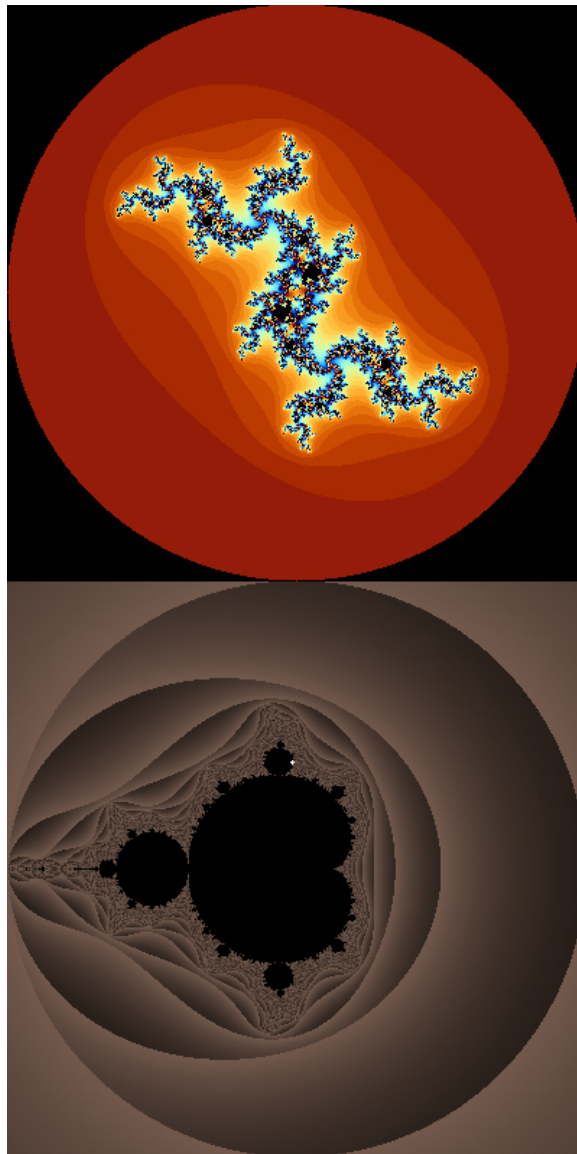


Figure 4: [Julia Circle](#)



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.



Christmas Tree Code:

```
#include "FPToolkit.c"

typedef struct {
    double x,y;
} point;

void tree (point p1, point p2, int level){
    point p3, p4, p5;

    p4.x = p1.x - (p2.y - p1.y);
    p4.y = p1.y - (p1.x - p2.x);

    p3.x = p2.x - (p2.y - p1.y);
    p3.y = p2.y - (p1.x - p2.x);

    p5.x = p3.x + ( p1.x - p2.x - (p2.y - p1.y) ) / 2;
    p5.y = p3.y - ( p1.x - p2.x + p2.y - p1.y ) / 2;

    if(level > 0){
        G_rgb(drand48(), drand48(), drand48());

        G_line(p1.x,p1.y,p2.x,p2.y);
        G_line(p3.x,p3.y,p2.x,p2.y);
        G_line(p3.x,p3.y,p4.x,p4.y);
        G_line(p1.x,p1.y,p4.x,p4.y);

        tree(p4,p5,level-1);
        tree(p5,p3,level-1);
    }
}

void tree_trunk(point p1, point p2) {
    int height = p2.x - p1.x;
    for (int i = 0; i + 10 < height; i += 10) {
        G_rgb (1, 1, 1);
        for (int white = 0; white < 5; ++ white) {
            int y = p1.y + i + white;
            G_line(p1.x, y, p2.x, y);
        }
        G_rgb (1, 0, 0);
        for (int red = 5; red < 10; ++ red) {
            int y = p1.y + i + red;
            G_line(p1.x, y, p2.x, y);
        }
    }
}

void star (double x, double y, double size) {

    G_rgb(0.94,0.77,0.38);
```



```
int a = 30; // angle
double c = cos(a*M_PI/180) * size;
double s = sin(a*M_PI/180) * size;

G_fill_triangle(c + x, s + y, -c + x, s + y, x, y - size);
G_fill_triangle(c + x, -s + y, -c + x, -s + y, x, y + size);
}

int main(){
    int swidth, sheight;
    swidth = 800, sheight = 800;

    G_init_graphics(swidth, sheight);

    G_rgb(0,0,0);
    G_clear();

    point p1, p2;
    p1.x = swidth *3.6/8;
    p1.y = sheight/10;
    p2.x = swidth - p1.x;
    p2.y = p1.y;

    point p3, p4;
    p3.x = swidth *3.7/8;
    p3.y = p1.y + 200;
    p4.x = swidth - p3.x;
    p4.y = p3.y;

    point p5, p6;
    p5.x = swidth *3.8/8;
    p5.y = p1.y + 400;
    p6.x = swidth - p5.x;
    p6.y = p5.y;

    point p7, p8;
    p7.x = swidth *3.9/8;
    p7.y = p1.y + 550;
    p8.x = swidth - p7.x;
    p8.y = p7.y;

    // coordinates for snow
    double x[1000];
    double y[1000];

    // giving snow random initial coordinates but above the screen
    for (int i = 0; i < 1000; ++i) {
        x[i] = swidth * drand48();
        y[i] = sheight * 6 * drand48() + sheight;
    }

    int frames = 1000;
    for (int i = 1; i < frames; ++i) {
```



```
G_rgb(0,0,0);
G_clear();

tree(p1, p2, 12);
tree(p3, p4, 12);
tree(p5, p6, 12);
tree(p7, p8, 12);
tree_trunk(p1, p2);

double star_size = 40.0;
star (swidth/2, p7.y + star_size * 3, star_size);

for (int i = 0; i < 1000; ++i) {
    G_rgb(1,1,1);
    G_fill_circle(x[i], y[i], 2);
}

//
for (int i = 0; i < 1000; ++i) {
    y[i] -= 2;
}

G_rgb((double)i/frames, (double)i/frames, (double)i/frames);
// G_rgb(1,1,1);
G_fill_rectangle(0.0, 0.0, swidth, p1.y);

for (int i = 0; i < 500000; ++i) {
    if (i % 10000 == 0) {
        G_display_image();
        usleep(100);
    }
}
return 0;
}
```



Autumn Tree Code:

```
#include "FPToolkit.c"
#include "string_builder.c"
#include "stack.c"

int strleng = 2000000;
// set angle to move
double angle = 22.5;

double angle_current = 90.0;

const int Wsize = 800;
// unit to draw
double length = 0.5;
double x_value, y_value;
// start drawing from this point
double x_current = Wsize/2;
double y_current = 50.0;
double y_max = 0.0;

void rotate(double a);
void step();
void values_reset();

void positive_rot() { angle_current += angle; };
void negative_rot() { angle_current -= angle; };

void draw(char * str);

double red_plant = 0.73;
double green_plant = 0.39;
double blue_plant = 0.05;

double red_leaf = 0.64;
double green_leaf = 0.71;
double blue_leaf = 0.17;

int main()
{
    char str [strleng];
    string_builder_level(str, 9);
    strleng = strlen(str);

    // if you want to use external file, comment out the line above and uncomment the line below:
    // scanf("%s", str);

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

    //=====
```



```

    draw(str);

    int key ;
    key = G_wait_key() ; // pause so user can see results

    G_save_to_bmp_file("fall-plant.bmp") ;
}

void rotate(double a) {
    x_value = length*cos(a*M_PI/180);
    y_value = length*sin(a*M_PI/180);
}

void step() {
    values_reset();
    rotate(angle_current);

    G_rgb(red_plant,green_plant,blue_plant) ;

    G_line (x_current, y_current,  x_current+ x_value, y_current + y_value);
    x_current += x_value;
    y_current += y_value;

    values_reset();
}

void values_reset() {
    x_value = length;
    y_value = 0;
}

void draw(char * str) {
    struct stack* stack = createStack(100);
    int i = 0;
    while (str[i] != '\0' && i < strlen) {
        if (str[i] >= 'A' && str[i] <= 'Z') step();
        else if (str[i] == '+') positive_rot();
        else if (str[i] == '-') negative_rot();
        else if (str[i] == '[') {
            struct bracket point;
            point.angle = angle_current;
            point.x = x_current;
            point.y = y_current;
            push(stack, point);
        }
        else if (str[i] == ']') {
            double leaf_shade = (0.1 + i) / strlen / 30000;
            red_leaf += leaf_shade/2.5;
            green_leaf -= leaf_shade/2;
            blue_leaf -= leaf_shade;
            G_rgb(red_leaf, green_leaf, blue_leaf);
            G_fill_circle(x_current,y_current,length) ;
        }
    }
}

```




```
        struct bracket point;  
        point = peek(stack);  
        angle_current = point.angle;  
        x_current = point.x;  
        y_current = point.y;  
        pop(stack);  
    }  
    ++i;  
}  
}
```



Mandelbrot Zoom Code:

```
#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;
}
```