Exploring Fractals

CS410P

Ben Truong Ka Hoo Chow

6/15/23

Table of contents

Recursion - The Pythagoras Christmas Tree Fractal

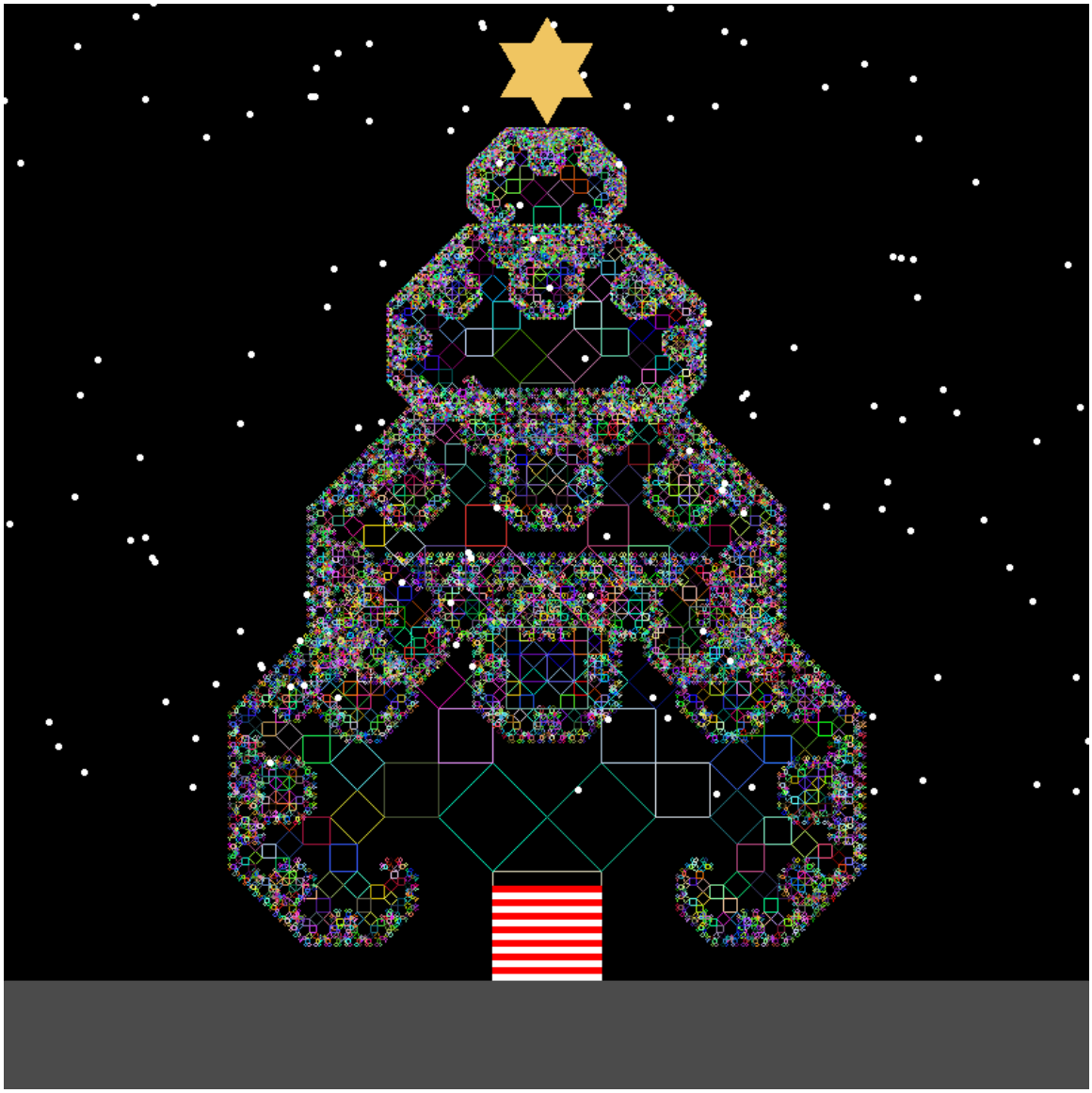

Figure 1: Christmas Tree

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

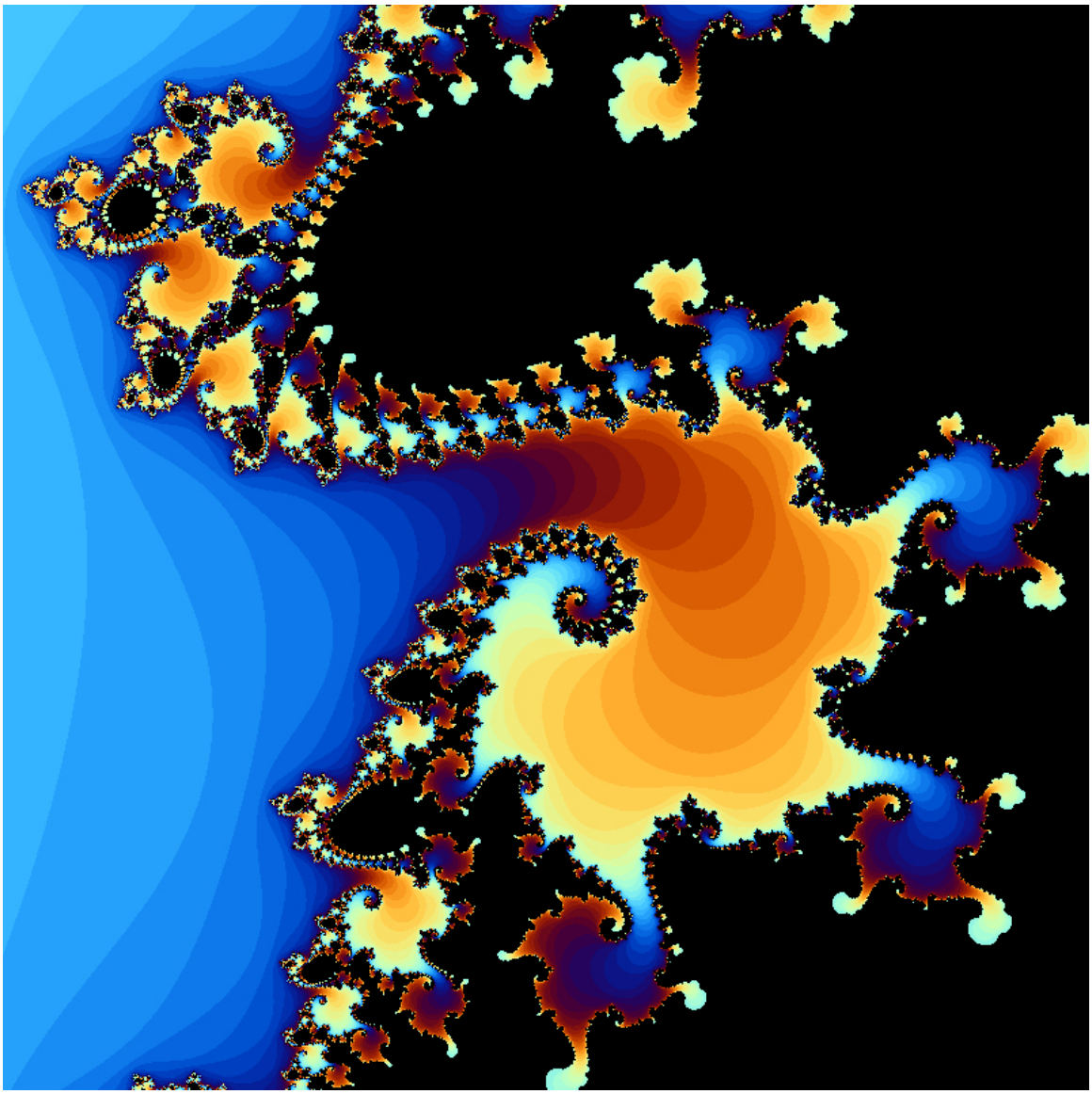

Figure 3: Mandelbrot

Design Paradigm & Mathematical Description

asdfjafndafa asdlfkjaslfka asdlfkj

Julia Circle

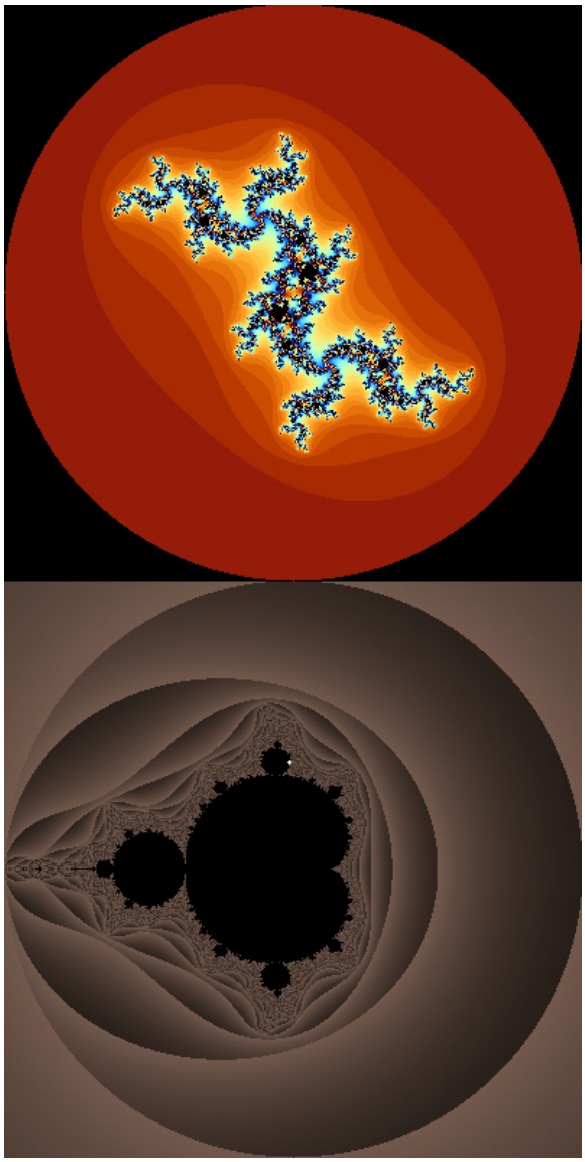

Figure 4: Julia Circle

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;

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

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

    // parameters for tree 1
    point p1, p2;
    p1.x = swidth *3.6/8;
    p1.y = sheight/10;
    p2.x = swidth - p1.x;
    p2.y = p1.y;

    // parameters for tree 2
    point p3, p4;
    p3.x = swidth *3.7/8;
    p3.y = p1.y + 200;
    p4.x = swidth - p3.x;
    p4.y = p3.y;

    // parameters for tree 3
    point p5, p6;
    p5.x = swidth *3.8/8;
    p5.y = p1.y + 400;
    p6.x = swidth - p5.x;
    p6.y = p5.y;

    // parameters for tree 4
    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);
       }

       // snow falling down 2 pixels each frame
       for (int i = 0; i < 1000; ++i) {
            y[i] -= 2;
       }

        // drawing the ground that is fading from black to white
        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);
            }
        }

        // saving a bmp file for this frame
        char fname[100];
        sprintf(fname, "./xmas%04d.bmp", i);
        G_save_to_bmp_file(fname);

    }
    return 0;
}
```

Autumn Tree Code:

plant.c:

```
#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);


   G_save_to_bmp_file("fall-plant.bmp") ;

   return 0;
}

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 < strleng) {
         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) / strleng / 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;
      }
}


string_builder.c :

char a [19] = "F-[[A]+A]+F[+FA]-A";
char f [3] = "FF";

#include <string.h>
#include <stdio.h>


int replace(char str [], int i);

int string_builder (char str []) {

    // setting Axiom
   strcpy(str, "A");

   int level;
   printf("Please enter level: ");
   scanf("%d", &level);

   for (int i = 1; i <= level; ++i) {
      int k = 0;
      while (k < strlen(str)) {
    if (str[k] != '-' && str[k] != '+' && str[k] != '[' && str[k] != ']')
       k += replace(str, k);
    else
       ++k;
      }
      printf("Level %d: %s \n", i, str);
   }
   return 0;
};

int string_builder_level (char str [], int level) {

   strcpy(str, "A");

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

```
      int k = 0;
      while (k < strlen(str)) {
    if (str[k] != '-' && str[k] != '+' && str[k] != '[' && str[k] != ']')
       k += replace(str, k);
    else
       ++k;
      }
      printf("Level %d: %s \n", i, str);
   }
   return 0;
};

int replace(char str [], int i) {
    char temp [strlen(str) + 10];
    int substr_len;
    if (str[i] == 'A') {
       strcpy(temp, a);
       substr_len = strlen(a);
    }
    else {
       strcpy(temp, f);
       substr_len = strlen(f);
    }

    strcat(temp, &str[i+1]);
    strcpy(&str[i], temp);

    return substr_len;
};


stack.c :
/*
The init_stack() function initializes a stack with the given size.

The push() function pushes the given value onto the stack.

The pop() function pops the top element off the stack and returns its value.

The free_stack() function frees the memory allocated for the stack.
*/
#include <stdio.h>
#include <stdlib.h>

// Define the bracket struct
struct bracket {
    double angle;
    double x;
    double y;
};

// Define the stack struct
struct stack {
```

```
    int top;
    int capacity;
    struct bracket* array;
};

// Create a new stack
struct stack* createStack(int capacity) {
    struct stack* stack = (struct stack*)malloc(sizeof(struct stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (struct bracket*)malloc(stack->capacity * sizeof(struct bracket));
    return stack;
}

// Check if the stack is empty
int isEmpty(struct stack* stack) {
    return stack->top == -1;
}

// Check if the stack is full
int isFull(struct stack* stack) {
    return stack->top == stack->capacity - 1;
}

// Push an element onto the stack
void push(struct stack* stack, struct bracket item) {
    if (isFull(stack)) {
        printf("Error: Stack overflow\n");
        return;
    }
    stack->array[++stack->top] = item;
}

// Pop an element off the stack
struct bracket pop(struct stack* stack) {
    if (isEmpty(stack)) {
        printf("Error: Stack underflow\n");
        exit(EXIT_FAILURE);
    }
    return stack->array[stack->top--];
}

// Get the top element of the stack without popping it
struct bracket peek(struct stack* stack) {
    if (isEmpty(stack)) {
        printf("Error: Stack underflow\n");
        exit(EXIT_FAILURE);
    }
    return stack->array[stack->top];
}
```

Mandelbrot Zoom Code:

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

int Wsize = 800;
int Hsize = 800;

void prcmx (char *control, complex c)
//print complex number
{
  double a,b ;
  a = creal(c) ;
  b = cimag(c) ;

  printf(control,a) ;
  if (b >= 0) {
    printf("+") ;
  } else {
    printf("-") ;
  }
  printf(control,fabs(b)) ;
  printf("I") ;
}

complex 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)
            return cabs(z);
    }
    return 0;
}

int julia_check (complex z, complex c) {
    // return a complex number if diverge - return 0 if converge
    for (int i = 0; i < 100; ++i) {
        if (cabs(z) > 2)
            return i;
        z = z*z + c;
    }
    return 0;
}

void mandel (double a, double b, int click) {
    double range = 4.0 / pow(2, click);
    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) {
            complex i = check(a_sub + b_sub*I);
            if (i == 0) {
                G_rgb(0,0,0);
            } else {
                G_rgb(0.9/creal(i),0.7/creal(i), 0.6/creal(i) );
            }
                G_point ((a_sub - a + range/2)*Wsize/range, (b_sub - b + range/2)*Wsize/range);
        }
    }

}


void julia (double a, double b) {
    double range = 4.0;
    double unit = range / Wsize;

    // a is the real part, b is the coefficient
    complex c = a + b*I;

    for (double a_sub = -2.0; a_sub < 2.0; a_sub += unit) {
        for (double b_sub = -2.0; b_sub < 2.0; b_sub += unit) {
            complex z = a_sub + b_sub*I;
            int result = julia_check(z, c);
            if (result == 0) {
                G_rgb(0,0,0);
            } else {
                double t = (double)result / 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 + 2.0)*Wsize/range, (b_sub + 2.0)*Wsize/range + Hsize/2);
        }
    }

}


void julia_set_circle() {

    Wsize = 400;
    Hsize = Wsize * 2;

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

```
    double point[2];
    point[0] = 0;
    point[1] = 0;

    for (int angle = 0; angle < 360; ++angle) {
        point[0] = cos(angle*M_PI/180) * 0.75;
        point[1] = sin(angle*M_PI/180) * 0.75;
        julia(point[0], point[1]);
        mandel(0.0, 0.0, 0);

        G_rgb(1,1,1);
        G_fill_circle(point[0]*Wsize/4 + Wsize/2, point[1]*Wsize/4 + Wsize/2, 1);

        // printf("Point: %lf, %lf\n", point[0], point[1]);
        // for (int i = 0; i < 500000; ++i) {
        //     if (i % 100000 == 0)    {
        //         G_display_image();
        //         usleep(100);
        //     }
        // }
        char fname[100];
        sprintf(fname, "./julia%04d.bmp", angle);
        G_save_to_bmp_file(fname);

        G_rgb (0, 0, 0) ; // black screen
        G_clear () ;
    }
}

int main () {

    julia_set_circle();

    return 0;

}
```