Exploring Fractals

CS410P

Ben Truong Ka Hoo Chow

6/15/23

Table of contents

Recursion - The Pythagoras Christmas Tree Fractal

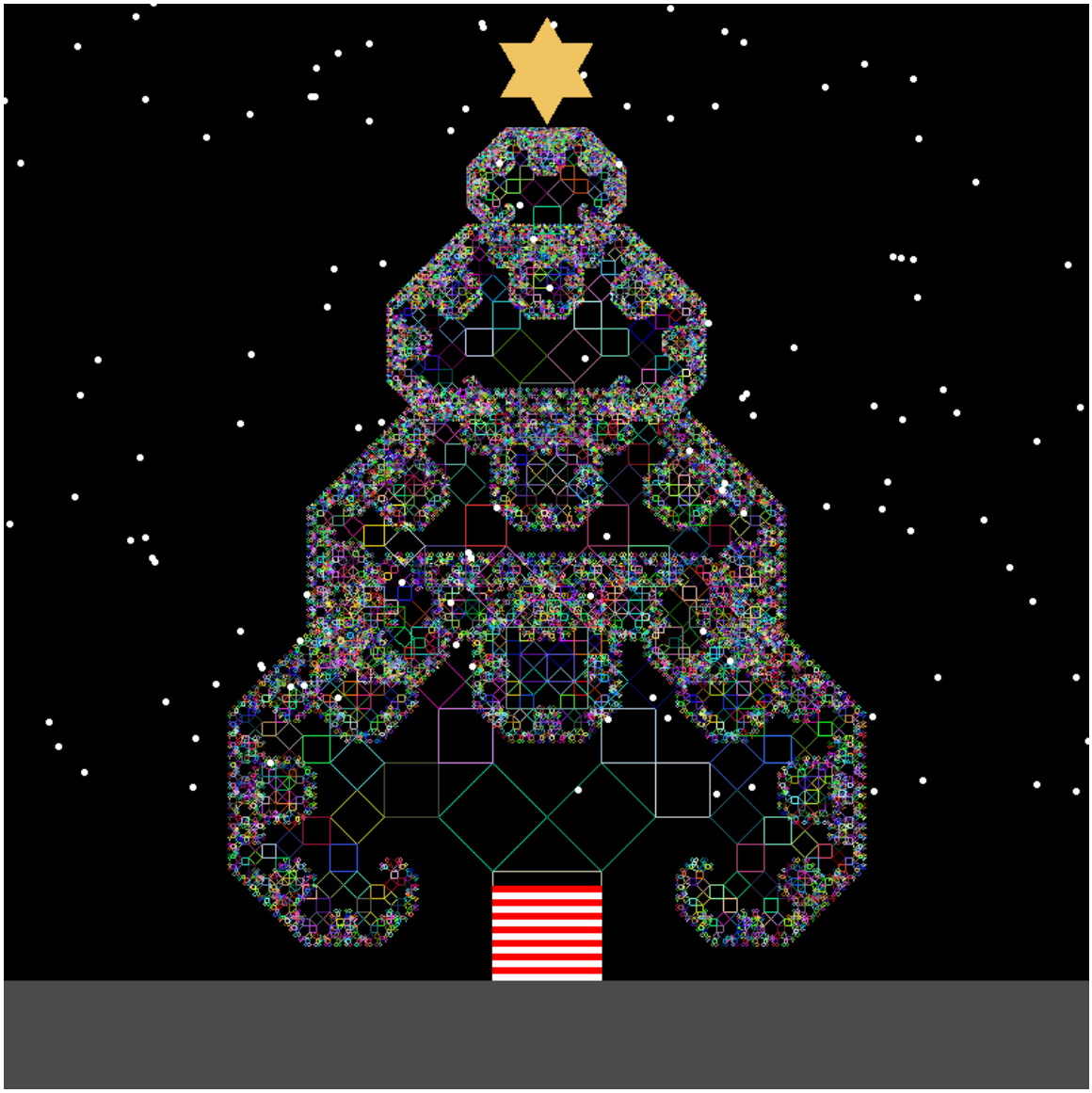


Figure 1: Christmas Tree (LINK TO VIDEO)

Design Paradigm & Mathematical Description

We have used recursion to generate the Pythagoras tree, which is a fractal composed of right-angled triangles. Each function call inherits two points (p1 and p2) and generates two new points, p3 and p4 that create a square. p3 and p4 are both generated by perpendicular movement from segment p1-p2. P3 and p4 then form the hypotenuse of a right triangle, whose vertex is a point, p5. The code then recursively calls two children that pass one of the children points p4 and p5 and the other passes p5 and p3.

```
How points p3 and p4 are completed, using right angle movement:
    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);


How point p5 is computed to form a 45-degree right triangle:

    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;
```

Artistic Description:

For artistic enhancements, and for the picture above and video of snow falling with a Christmas tree in a winter night:

We called the tree function 4 times with different scales and positioned them where the latter is smaller and higher than the previous one, creating an image of a Christmas tree where its branches get smaller as the tree goes higher. To define the scaling of the trees, we use a ratio in relation with screen width, as we want to eventually place the trees in the middle of the screen, and the width of the trees get smaller for each tree. Here is how we defined the parameters for the 4 trees:

```
    // 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;
```

The trees are generated with random rgb colored lines to create the illusion of them having twinkling lights.

For the tree trunk, we use a loop that draws a group of horizontal lines (alternating between red and white) with the length of the original points of the first tree, going as high, to draw the red and white ribbons around the trunk.

To create the star on top of the we use sines and cosines of points on a unit circle, to generate 2 equilateral triangles that is one is up-side-down as the other. These 2 angles are filled with a nice yellow-ish color using G-fill-triangle. Then scaled and translated to be on top of the highest tree.

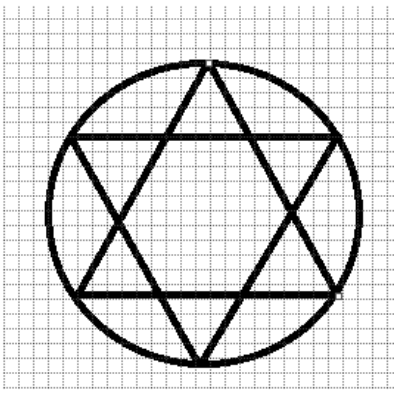


Figure 2: Star

For the snow, we generate 2 arrays doubles of size 1000, to maintain coordinates of 1000 snowflakes. We then would give the x coordinates random values ranging from 0 to screen width. Y coordinates are random values ranging from screen height to 6 times screen height. This will create an illusion of snow only starting to fall that night, and they come from outside of the screen. At every frame, the y coordinates of the snow will each drop by 2 pixels.

```
// 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;
}
```

And lastly, we draw a rectangle spanning the screen width and the height is from the bottom of the screen to the bottom of the lowest tree. The color of this rectangle is fading from black to white as the frame goes higher. This was done by setting all rgb the same, and as the loop goes to generate a new image each frame, this rgb value is set by iteration i divided by number of total frames.

L-System - Autumn Tree

Figure 3: Autumn Tree

Design Paradigm & Mathematical Description

The L-system (Lindenmayer system) is a string rewriting system commonly used to generate complex and self-similar structures, such as plants. It was introduced by the biologist Aristid Lindenmayer in 1968. L-systems consist of an initial string (often called the axiom) and a set of production rules that define how to rewrite or replace specific symbols in the string. By iteratively applying these rules, complex and visually interesting patterns can be generated. (@wikipedia2021)

For this entry, we used the language provided by our professor at PSU, David Ely, which has the following definitions:

```
Axiom: A
Production rules:

    A = "F-[[A]+A]+F[+FA]-A"
    F = "FF"

Character meanings:
  + : increment the current angle by a certain degree`
  - : decrement the current angle by a certain degree
  [ : saves the angle and coordinates of the current location into a stack
  ] : retrieve (peek) the angle and coordinates of the top of the stack and remove
      this from the top of the stack
A/F : draw a line moving forward a certain distance

  Degree to increment/decrement: 22.5
  Initial angle to start drawing: 90 (facing up)
  Distance to move forward each time: 0.5
  Level of iteration to generate the final string: 9
```

Artistic Description:

For artistic purposes the picture of the plant above was added with leaves and coloring to make it look more like a tree in the Fall, when its leaves' color are being changed to more yellow and red. To accomplish this, we implement an extra step for the l-system to add coloring by `G_point` whenever there is a '] ' at that location. The idea is that we would want the color ("leaves") to only be about where the branches don't extend further, so we don't have colors at strange locations on the plant.

Initial rgb values for the plant and its leaves:

```
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;
```

IFS - Dragon

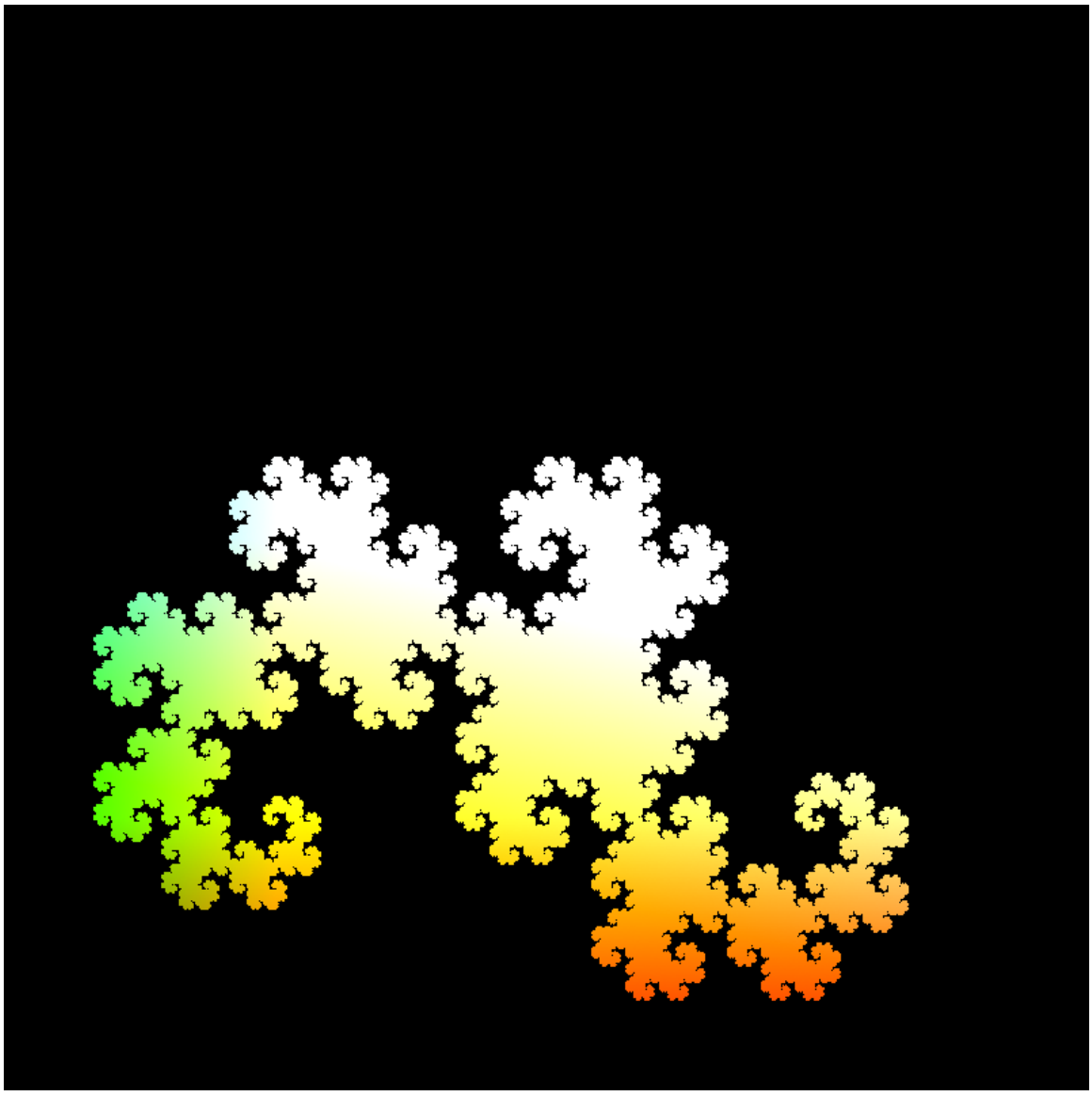


Figure 4: Dragon

Design Paradigm & Mathematical Description

We used this transformation matrix to rotate and translate our fractal, it results in the same rotation equation taught in class:

$$\begin{bmatrix} cos\ \theta & -sin\ \theta \\ sin\ \theta & cos\ \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\ cos\ \theta - y\ sin\ \theta \\ x\ sin\ \theta + y\ cos\ \theta \end{bmatrix}$$

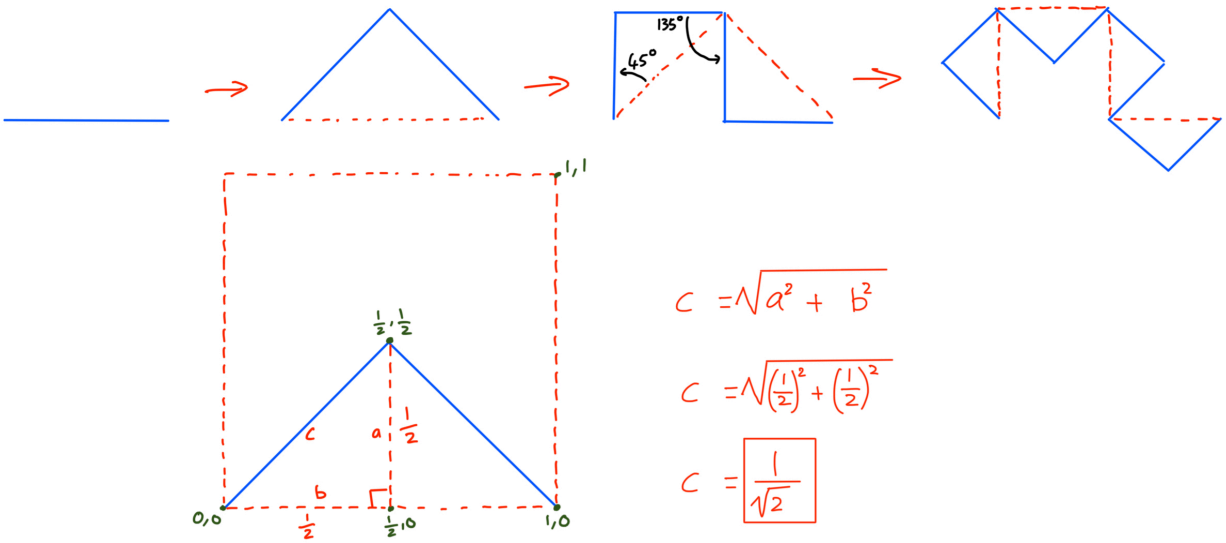
The first transformation, denoted as $f_1(x, y)$, rotates the coordinates by 45 degrees counterclockwise and scales them by a factor of $\frac{1}{\sqrt{2}}$.

This transformation is defined as: $f_1(x, y) = \frac{1}{\sqrt{2}} \begin{bmatrix} cos\ 45^\circ & -sin\ 45^\circ \\ sin\ 45^\circ & cos\ 45^\circ \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

The second transformation, denoted as $f_2(x, y)$, rotates the coordinates by 135 degrees counterclockwise, scales them by a factor of $\frac{1}{\sqrt{2}}$, and translates them horizontally by 1 unit.

This transformation is defined as: $f_2(x, y) = \frac{1}{\sqrt{2}} \begin{bmatrix} cos\ 135^\circ & -sin\ 135^\circ \\ sin\ 135^\circ & cos\ 135^\circ \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

The Figure below illustrates how the dragon rotates and how the scale factor $\frac{1}{\sqrt{2}}$ is found using Pythagoras Theorem:



The transformations are defined by the functions `rule1()` and `rule2()`. `rule1()` applies a transformation matrix that rotates the coordinates by 45 degrees counterclockwise and scales them by a factor of $\frac{1}{2}$. `rule2()` applies a transformation matrix that rotates the coordinates by 135 degrees counterclockwise, scales them by a factor of $\frac{1}{2}$, and translates them horizontally by 2 units. These transformations are defined using a transformation matrix and implemented in the `transformation()` function.

The `dragon()` function generates the fractal pattern by repeatedly applying the transformations based on a random number. If the random number is less than $\frac{1}{2}$, `rule1()` is applied; otherwise, `rule2()` is applied. The resulting coordinates are plotted on the canvas using the `G_point()` function. Additionally, the color of each point is determined by modifying the RGB values based on the current iteration count.

Artistic Description:

Red - `x+5/c`: The x coordinate is used as the basis for the red component of the color. By adding 5/c to x, the value is shifted by an amount that depends on the iteration count. This can create variations in the red component of the color as the iteration progresses.

Green - `y+5/c`: The y coordinate is used as the basis for the green component of the color. Similar to the red component, y is shifted by 5/c to introduce variations in the green component based on the iteration count.

Blue - `y+x/c`: Both x and y coordinates are combined and divided by c to determine the blue component of the color. This computation creates a combination of x and y values that is influenced by the iteration count.

Complex number - Mandelbrot Zoom

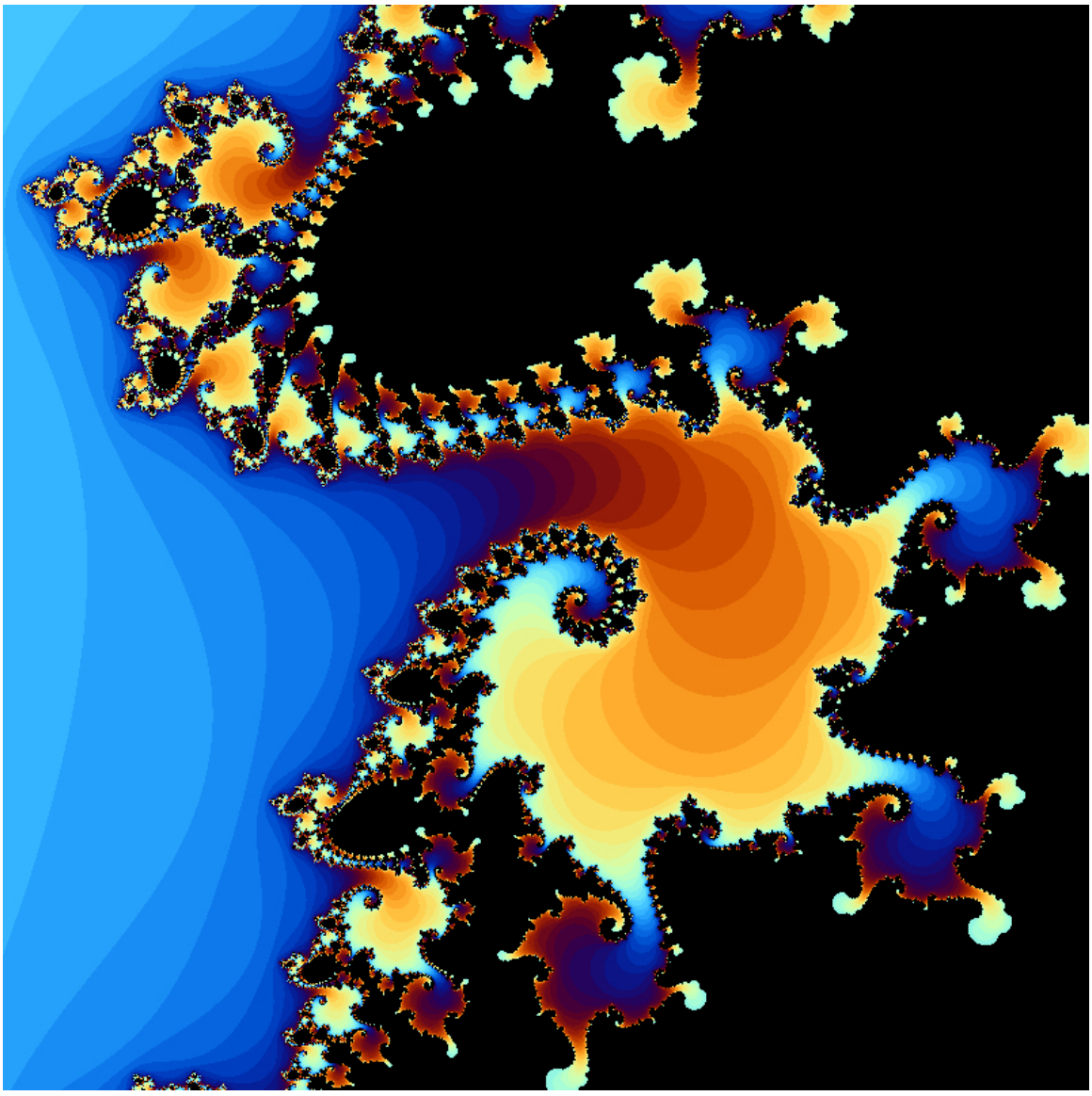


Figure 5: Mandelbrot (LINK TO VIDEO)

Design Paradigm & Mathematical Description

"This set was first defined and drawn by Robert W. Brooks and Peter Matelski in 1978, as part of a study of Kleinian groups.[3] Afterwards, in 1980, Benoit Mandelbrot obtained high-quality visualizations of the set while working at IBM's Thomas J. Watson Research Center in Yorktown Heights, New York."

(1)

A complex number number `c` is iteratively computed using the equation $z = z^2 + c$ starting with `z = 0`. This process is repeated to check to see if the magnitude of `z` exceeds 2 at any point, then `c` is said to be divergent. Otherwise the number is convergent.

In the `check` function, this process is repeated for a maximum of 100 iterations.The function returns the number of iterations taken for divergence. If the iterations complete without divergence, the function returns 0.

The idea behind this entry is that we are going to generate hundreds of frames of the mandelbrot set where at each frame, we are going to zoom in a little further at a predetermined point where the picture would appear with the most interesting patterns.

The `mandel_zoom` function performs the zooming operation on the Mandelbrot fractal. It takes the coordinates `(a, b)` to zoom into and the number of iterations `times` to perform the zooming operation. It initializes a variable `range` that determines the initial size of the region to zoom into. Initial `range` is set at 4 (from -2 to 2) as the original Mandelbrot set image. As we zoom in, this `range` will get smaller and smaller, allowing us to examine finer complex numbers on the same screen resolution (800x800).

The function then iterates over each pixel in the zoomed region. For each pixel, it calculates the corresponding complex number by mapping the pixel's position to the complex plane. It uses the formula

```
a_sub + b_sub * I
```

to construct the complex number. The `check` function is called with this complex number to determine if it diverges or converges.

Based on the result, the function assigns colors to the pixel. If the complex number converges (`i == 0`), the pixel is assigned the color black. Otherwise, a color gradient is defined based on the iteration count `i`, and the pixel is assigned a color based on this gradient.

After processing all pixels, the function reduces the `range`. It then provides options to either display the fractal in real-time or save each frame as a bitmap image file.

In the `main` function, the `mandel_zoom` function is called with the desired zoom coordinates `(zoomx, zoomy)` and number of iterations (400). This generates the Mandelbrot fractal and displays it on the screen.

Artistic Description:

The color gradient in the code is defined based on the iteration count `i` normalized to the range [0, 100]. The purpose of the color gradient is to assign varying colors to the pixels based on the divergence rate of the corresponding complex numbers.

The color gradient is defined using the following equations:

```
Red component   : red   = sin(5 * M_PI * t)
Green component : green = sin(5 * M_PI * (t + 1.0 / 3.0))
Blue component  : blue  = sin(5 * M_PI * (t + 2.0 / 3.0))
```

Here, `t` represents the normalized iteration count, ranging from 0 to 1. The color components are calculated using sine functions with different offsets to achieve a smooth transition of colors across the fractal.

By using sine functions, the color values will oscillate between -1 and 1. Adding 1 and dividing by 2 ensures that the resulting values are in the range [0, 1], which is the expected range for specifying RGB color values. The resulting red, green, and blue values are then used to set the color of the pixels based on the iteration count. (@wikibooks2021)

Julia Circle

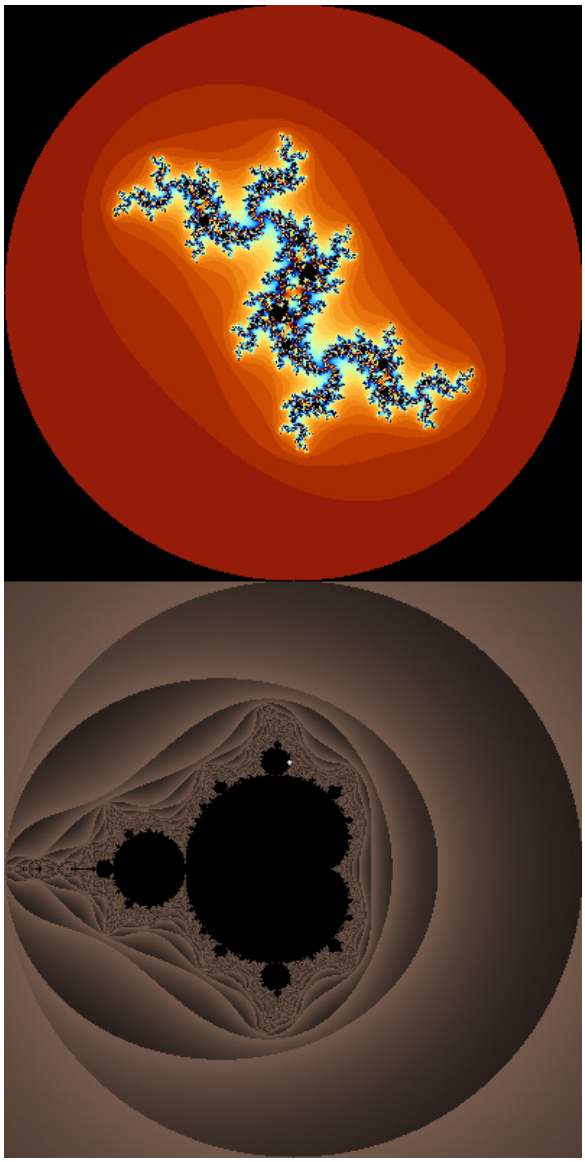


Figure 6: Julia Circle (LINK TO VIDEO)

Design Paradigm & Mathematical Description

Julia set consists of values such that an arbitrarily small perturbation can cause drastic changes in the sequence of iterated function values. Thus the behavior of the function on the Fatou set is "regular", while on the Julia set its behavior is "chaotic". (wikipedia2021a)

We would like to thank Grant VanDomelen, another student in our class that helped us understand the Julia set and approached this entry better.

The math behind generating the Julia image is very similar to the Mandelbrot we did in the previous entry. Using a pixel on the screen combines with another complex number as an argument to finally determine if the chosen pixel on the screen diverges or converges.

The "check" function behaves the same way as in the previous entry, and is used to present a Mandelbrot set on the bottom screen. The "julia_check" function is a new one that determines whether a combination of 2 complex numbers `z` and `c` would create a pixel that indicates divergence or convergence, by applying the iterative equation. The equation $z = z^2 + c$ is still iteratively computed but instead of starting with `z = 0`, it starts with `z` equals a complex number that represents a point in the range [-2,-2] for both its real and imaginary part. `c` is a complex value given from a Mandelbrot set model that represents the point on the Mandelbrot set we would like to examine. This process is repeated to check to see if the magnitude of `z` exceeds 2 at any point, then `z` is said to be divergent. Otherwise the number is convergent. Return values for this function are similar to the "check" function, and are used to manipulate the coloring.

The "mandel" function generates a fixed Mandelbrot set fractal and is placed on the bottom screen at all times. A white point will be moving around in this set to represent 'c' that we are using to determine convergence/divergence of the Julia set.

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.

Coloring for the Mandelbrot set is reused in this Julia circle illustration.

Appendix

References:

[@wikipedia2021] Wikipedia contributors, “L-system”, Wikipedia, The Free Encyclopedia, 2021, (accessed June 15, 2023).

[@wikibooks2021] Wikibooks contributors, “Color Theory/Color gradient”, Wikibooks, The Free Textbook Project, 2021, (accessed June 15, 2023).

[1] Wikipedia contributors, “Mandelbrot set”, Wikipedia, The Free Encyclopedia, 2021, (accessed June 15, 2023).

[@wikipedia2021a] Wikipedia contributors, “Julia set”, Wikipedia, The Free Encyclopedia, 2021, (accessed June 15, 2023).

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:

```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];
}
```

```
#include "FPToolkit.c"
#include <math.h>

// Canvas size
double canvas = 800.0;
double sheight;
double swidth;

// Starting point
double x = 0.0;
double y = 0.0;

/*
Transformation matrix:

    |a   b| |x|   |e|
    |     | | | + | |
    |c   d| |y|   |f|

    ax + by = x' + e
    cx + dy = y' + f
*/
void transformation(double a, double b, double c, double d, double e, double f){
    double newx = (a*x + b*y) + e;
    double newy = (c*x + d*y) + f;
    x = newx;
    y = newy;
}

void rule1() {
    transformation(1.0/2.0,-1.0/2.0,1.0/2.0,1.0/2.0, 0,0);
}

void rule2() {
    transformation(-1.0/2.0,-1.0/2.0,1.0/2.0,-1.0/2.0, 2.0,0);
}

void dragon(int c){

    int ITER = 10000000;

    for (int i = 0; i < ITER; ++i) {
        double n = drand48();
        if (n < 1.0/2.0) {
            rule1();
        }
        else{
            rule2();
        }

        G_rgb(x+5/c,y+5/c,y+x/c);
        G_point((200*x+200),(200*y+200));
    }
```

```
}

int main() {
    // Canvas height and width
    sheight = canvas;
    swidth = canvas;


    G_init_graphics(sheight, swidth);
    G_rgb(0, 0, 0);
    // G_rgb (0.8, 0.8, 0.8);
    G_clear();

    for (int c = 1; c < 360; ++c) {
    G_rgb(0, 0, 0);  // Set color to green
        G_clear();
        dragon(c);
        for (int i = 0; i < 500000; ++i) {
            if (i % 100000000 == 0) {
                G_display_image();
                usleep(100);
            }
        }
        // char fname[400];
        // sprintf(fname, "img%04d.bmp", c);
        // G_save_to_bmp_file(fname);
    }

    G_wait_key();

    return 0;
}
```

Mandelbrot Zoom:

```c
#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 = 400;
    double zoomx, zoomy;
    zoomx =-0.743643135;
    zoomy = 0.131825963;
    mandel_zoom(zoomx, zoomy, click);

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

    return 0;

}
```

Julia Set 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;

}
```