CGIP PROJECT

FRACTAL SCALING

Group Members:

1. Abhinav A.V. (Roll No. 1)
2. Johann Luke John (Roll No. 27)
3. Kevin Tom Varghese (Roll No. 29)
4. Thriambak S. (Roll No. 58)

1. ABSTRACT

This program is designed to render images that exhibit fractal properties, with an emphasis on well-known structures such as Barnsley Ferns. Utilizing algorithms grounded in fractal geometry, it generates images that capture the self-similarity and recursive characteristics inherent in these mathematical objects. The implementation leverages iterative methods and recursive transformations to produce complex, self-replicating patterns derived from fundamental equations. By applying affine transformations, scaling, and rotation techniques, the program accurately models the fractal's geometric properties. Through these processes, the program visually demonstrates the recursive depth and intricate detail of fractals, illustrating both their aesthetic and mathematical underpinnings.

2. INTRODUCTION

Fractals are complex geometric objects that exhibit self-similarity, meaning their structure remains consistent across different levels of magnification. Unlike traditional Euclidean shapes, fractals possess fractional dimensions, often characterized by intricate patterns that emerge through recursive or iterative processes. One of the most well-known classes of fractals is the Iterated Function System (IFS), which generates self-replicating structures through repeated transformations of an initial point set. These transformations typically consist of affine mappings, which include scaling, rotation, translation, and shearing operations.

The Barnsley Fern, introduced by Michael Barnsley in 1988, is a prime example of an IFS-based fractal. It is defined by four specific affine transformations, each applied with a given probability to iteratively construct the image of a natural-looking fern. The four transformations that define the Barnsley Fern can be expressed mathematically as follows:

- **Transformation 1 (Stem)**: $(x',y')=(0,0.16)(x,y)$ with a probability of 0.01.
- **Transformation 2 (Successive Leaflets)**: $(x',y')=(0.85,-0.04)(x,y)+(0,1.6)$ with a probability of 0.85.
- **Transformation 3 (Left Leaflet)**: $(x',y')=(0.2,0.23)(x,y)+(0,1.6)$ with a probability of 0.07.
- **Transformation 4 (Right Leaflet)**: $(x',y')=(-0.15,0.26)(x,y)+(0,0.44)$ with a probability of 0.07.

These transformations create a fractal structure that closely resembles the natural form of a fern, demonstrating how simple mathematical rules can lead to complex and aesthetically pleasing results. The self-similarity of the Barnsley Fern is evident as one zooms in on different sections of the fractal, revealing smaller fern-like structures that mirror the overall shape. The Barnsley Fern also exhibits a fractal dimension, which quantifies its complexity. The fractal dimension of the fern is approximately 2.5, indicating that it occupies a space between a two-dimensional plane and a three-dimensional object. This property is a hallmark of fractals, as they often possess non-integer dimensions that reflect their intricate and detailed nature.

3. IMPLEMENTATION OVERVIEW

In this project, we create a program in C that utilizes the SDL2 library to generate and display the Barnsley Fern in an interactive window. The process is based on an iterative approach, where each point of the fractal is calculated from the previous one.

We begin by initializing the SDL2 library, which manages the window and rendering, facilitating the visualization of the fractal. This involves setting up the main window where the fern will be displayed, as well as initializing the rendering context to handle graphics output.

Next, we define functions to apply the four affine transformations that shape the Barnsley Fern. Each transformation is applied according to its associated probability, allowing us to generate new points that mimic the natural growth of a fern.

The program then enters a loop where it calculates a specified number of points for the fractal. Each point is generated by randomly selecting one of the transformations based on its probability and applying it to the current point. The newly calculated points are then drawn directly onto the window, creating a dynamic and engaging visualization.

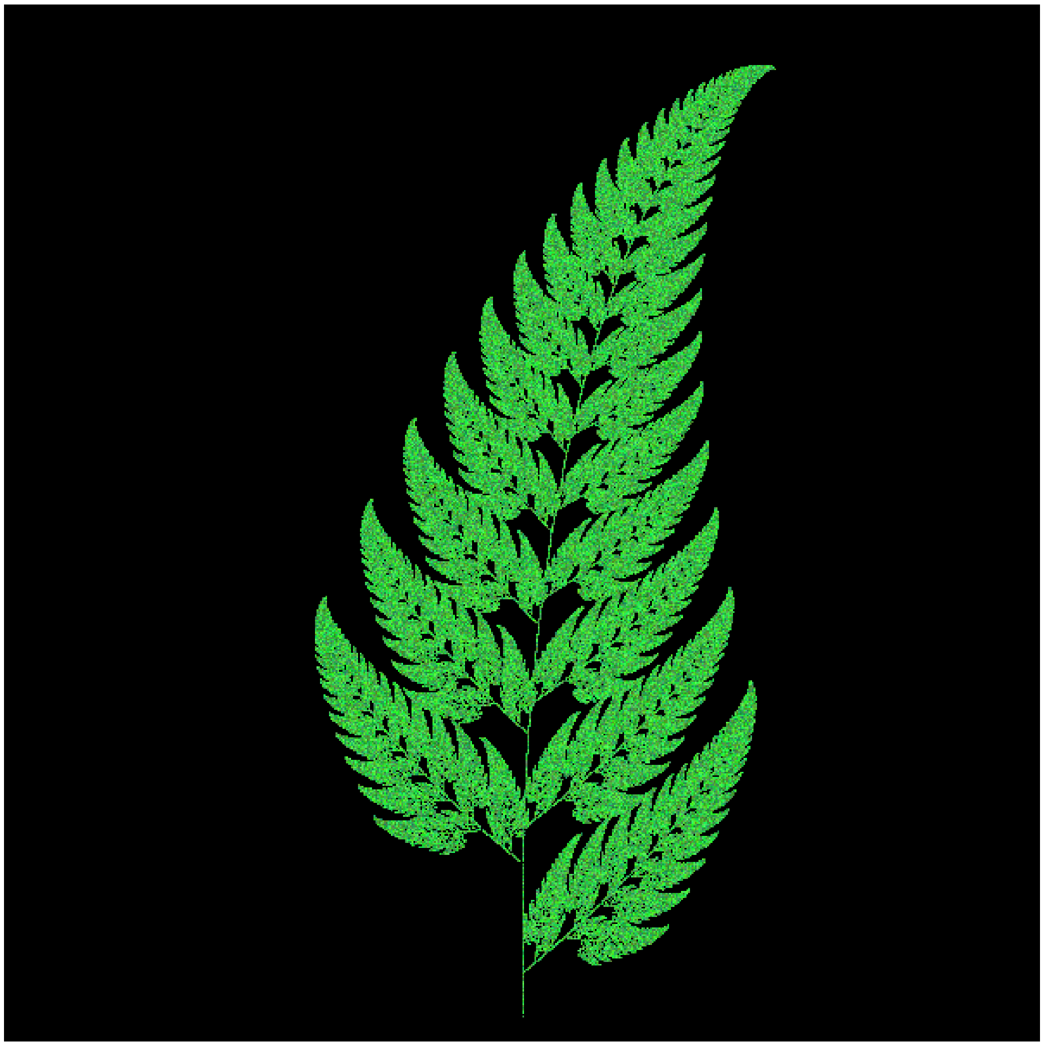
Using SDL2, we efficiently render the points on the screen, enabling users to interact with the window and observe the fractal coming to life in real time. The program can be designed to allow users to reset the fern, change the number of iterations, or even adjust the transformation probabilities, enhancing the interactive experience. Additionally, the program can include features such as zooming in and out or panning across the fractal, providing a deeper exploration of its intricate details.

4. CODE

```c
#include <SDL2/SDL.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <pthread.h>

#define WINDOW_WIDTH 800
#define WINDOW_HEIGHT 800
#define NUM_THREADS 4

typedef struct {
    double scale;
    double offsetX;
    double offsetY;
    SDL_Renderer *renderer;
    unsigned long iter_per_thread;
    unsigned int seed;
    SDL_mutex *render_mutex;
    SDL_Surface *surface;
} ThreadData;

typedef struct {
    double scale;
    double offsetX;
    double offsetY;
} FractalView;

void* barnsleyFernThread(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    double x0 = 0, y0 = 0, x1, y1;

    int diceThrow;
    int centerX = WINDOW_WIDTH / 2;
    int centerY = WINDOW_HEIGHT / 2;
    unsigned int seed = data->seed;

    Uint32 green = SDL_MapRGB(data->surface->format, 0, 255, 0);

    for (unsigned long i = 0; i < data->iter_per_thread; i++) {
        diceThrow = rand_r(&seed) % 100;
        if (diceThrow == 0) {
            x1 = 0;
            y1 = 0.16 * y0;
        }
        else if (diceThrow >= 1 && diceThrow <= 7) {
            x1 = -0.15 * x0 + 0.28 * y0;
            y1 = 0.26 * x0 + 0.24 * y0 + 0.44;
        }
        else if (diceThrow >= 8 && diceThrow <= 15) {
            x1 = 0.2 * x0 - 0.26 * y0;
            y1 = 0.23 * x0 + 0.22 * y0 + 1.6;
        }
        else {
            x1 = 0.85 * x0 + 0.04 * y0;
            y1 = -0.04 * x0 + 0.85 * y0 + 1.6;
        }

        int screenX = (int)(data->scale *x1 + data->offsetX + centerX);
        int screenY = (int)(data->scale *y1 + data->offsetY + centerY);

        if (screenX >= 0 && screenX < WINDOW_WIDTH && screenY >= 0 &&
 screenY < WINDOW_HEIGHT) {
            SDL_LockMutex(data->render_mutex);
            Uint32 *pixels = (Uint32 *)data->surface->pixels;
            pixels[screenY * data->surface->w + screenX] = green;
            SDL_UnlockMutex(data->render_mutex);
        }
        x0 = x1;
        y0 = y1;
    }
    return NULL;
}

void barnsleyFern(SDL_Renderer *renderer, unsigned long iter,
 FractalView *view) {
    pthread_t threads[NUM_THREADS];
    ThreadData threadData[NUM_THREADS];
    unsigned long iter_per_thread = iter / NUM_THREADS;
    unsigned int seed = (unsigned int)time(NULL);

    SDL_Surface *surface = SDL_CreateRGBSurface(0, WINDOW_WIDTH,
 WINDOW_HEIGHT, 32, 0x00FF0000, 0x0000FF00, 0x000000FF, 0xFF000000);

    if (!surface) {
        printf("Surface creation failed: %s\n", SDL_GetError());
        return;
    }

    SDL_FillRect(surface, NULL, SDL_MapRGB(surface->format, 0, 0, 0));
    SDL_mutex *render_mutex = SDL_CreateMutex();
    if (!render_mutex) {
        printf("Mutex creation failed: %s\n", SDL_GetError());
        SDL_FreeSurface(surface);
        return;
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        threadData[i].renderer = renderer;
        threadData[i].scale = view->scale;
        threadData[i].offsetX = view->offsetX;
        threadData[i].offsetY = view->offsetY;
        threadData[i].iter_per_thread = iter_per_thread;
        threadData[i].seed = seed + i;
        threadData[i].render_mutex = render_mutex;
        threadData[i].surface = surface;

        if (i == NUM_THREADS - 1) {
            threadData[i].iter_per_thread += iter % NUM_THREADS;
        }

        if (pthread_create(&threads[i], NULL, barnsleyFernThread,
 &threadData[i])) {
            printf("Thread creation failed!\n");
            SDL_DestroyMutex(render_mutex);
            SDL_FreeSurface(surface);
            return;
        }
    }

    for (int i = 0; i < NUM_THREADS; i++)
        pthread_join(threads[i], NULL);

    SDL_Texture *texture = SDL_CreateTextureFromSurface(renderer,
 surface);
    if (!texture) {
        printf("Texture creation failed: %s\n", SDL_GetError());
    } else {
        SDL_RenderCopy(renderer, texture, NULL, NULL);
        SDL_DestroyTexture(texture);
    }

    SDL_DestroyMutex(render_mutex);
    SDL_FreeSurface(surface);
}
```

```
int main() {
    unsigned long num = 50000;
    SDL_Event e;
    FractalView view = { 80.0, 0.0, -100.0 };
    int isDragging = 0;
    int prevX, prevY;

    printf("Enter number of iterations (default 50000): ");
    if (scanf("%lu", &num) != 1) {
        num = 50000;
    }

    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
        printf("SDL_Init Error: %s\n", SDL_GetError());
        return 1;
    }

    SDL_Window *window = SDL_CreateWindow("Barnsley Fern",
 SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, WINDOW_WIDTH,
 WINDOW_HEIGHT, SDL_WINDOW_SHOWN);
    if (!window) {
        printf("SDL_CreateWindow Error: %s\n", SDL_GetError());
        SDL_Quit();
        return 1;
    }

    SDL_Renderer *renderer = SDL_CreateRenderer(window, -1,
SDL_RENDERER_PRESENTVSYNC);                             SDL_RENDERER_ACCELERATED |
    if (!renderer) {
        printf("SDL_CreateRenderer Error: %s\n", SDL_GetError());
        SDL_DestroyWindow(window);
        SDL_Quit();
        return 1;
    }

    int quit = 0;
    while (!quit) {
        while (SDL_PollEvent(&e)) {
            if (e.type == SDL_QUIT) {
                quit = 1;
            }
            else if (e.type == SDL_MOUSEWHEEL) {
                double zoomFactor = e.wheel.y > 0 ? 1.1 : 1/1.1;
                view.scale *= zoomFactor;
            }
            else if (e.type == SDL_MOUSEBUTTONDOWN && e.button.button
 == SDL_BUTTON_LEFT) {
                isDragging = 1;
                prevX = e.button.x;
                prevY = e.button.y;
            }
            else if (e.type == SDL_MOUSEBUTTONUP && e.button.button ==
 SDL_BUTTON_LEFT) {
                isDragging = 0;
            }
            else if (e.type == SDL_MOUSEMOTION && isDragging) {
                view.offsetX += e.motion.x - prevX;
                view.offsetY += e.motion.y - prevY;
                prevX = e.motion.x;
                prevY = e.motion.y;
            }
            else if (e.type == SDL_KEYDOWN) {
                if (e.key.keysym.sym == SDLK_r) {
                    view.scale = 80.0;
                    view.offsetX = 0.0;
                    view.offsetY = -100.0;
                }
            }
        }

        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
        SDL_RenderClear(renderer);

        barnsleyFern(renderer, num, &view);

        SDL_RenderPresent(renderer);
        SDL_Delay(16);
    }

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

5. RESULT



6. GITHUB LINK

https://github.com/JohannLuke/Fractal-Scaling

7. CONCLUSION

In this project, we successfully implemented a program to generate the Barnsley Fern fractal using C and the SDL2 library. Through the application of Iterated Function Systems (IFS) and affine transformations, we demonstrated how simple mathematical rules can create complex, natural-looking patterns. The use of multithreading improved performance, allowing for faster rendering of the fractal. Additionally, interactive features like zooming, panning, and resetting enhanced the user experience, providing deeper insight into the fractal's intricate structure. This project not only showcases the beauty of fractals but also highlights the power of computational graphics and algorithmic design.