https://github.com/karthikbanjan/fractalflames

Karthik Banjan, Hanumanth Padmanabhan

# Fractal Generation Algorithms in Rust

## Fractal Flames

# Table of contents


1 Introduction

2 Algorithm and Sequential Implementation

3 Images and Graphs

4 Parallelization Strategies

5 Future Work

6 Appendix

# Fractals

- Complex geometric shapes that possess self-similarity

# Fractals

- Complex geometric shapes that possess self-similarity
- Each part of a fractal pattern is a scaled-down replica of the whole structure

# Fractals

- Complex geometric shapes that possess self-similarity
- Each part of a fractal pattern is a scaled-down replica of the whole structure
- Unlike conventional Euclidean geometric objects, fractals have non-integer (fractal) dimensions.

# Example: Mandelbrot Set

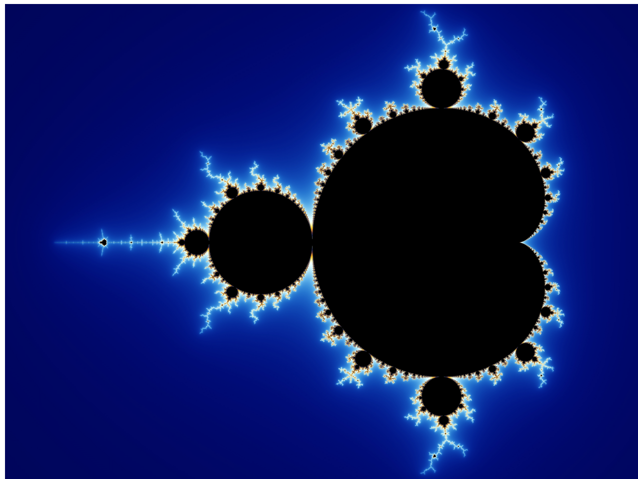


Image source: https://en.wikipedia.org/wiki/Mandelbrot_set

# Example: Barnsley Fern

Image source: Smithsonian Magazine

# Fractal Flames

- Attractors of probabilistic iterative function systems (IFS)

# Fractal Flames

- Attractors of probabilistic iterative function systems (IFS)
- Color pixels by structure of the IFS and density of attractor

# Fractal Flames

- Attractors of probabilistic iterative function systems (IFS)
- Color pixels by structure of the IFS and density of attractor
- Can be approximated by iterating the IFS on random initial point(s)

# Example: Fractal Flame

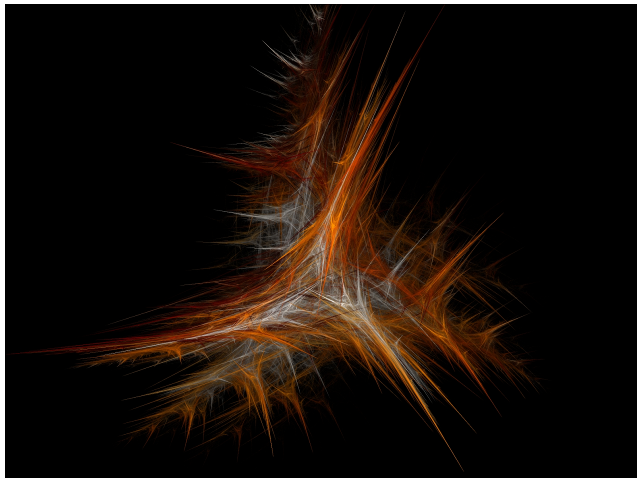


Image source: https://en.wikipedia.org/wiki/Fractal_flame

# Problem

- Large number of iterations required to properly resolve the structure

# Problem Visualized

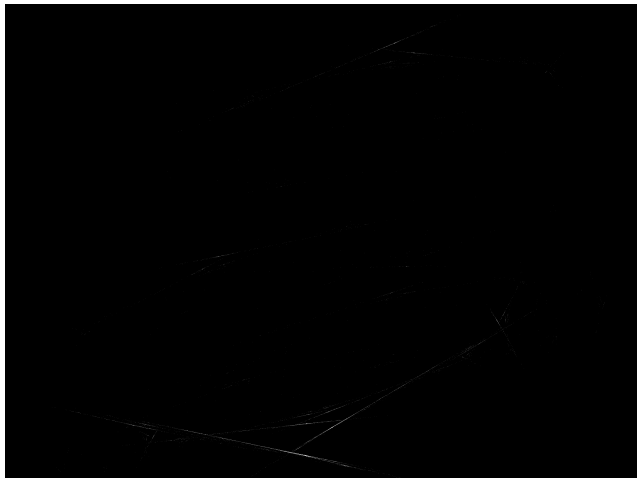


Figure: $2^{15}$ *iterations*

# Problem Visualized

Figure: $2^{25}$ *iterations*

# Outline

1 Introduction

2 Algorithm and Sequential Implementation

3 Images and Graphs

4 Parallelization Strategies

5 Future Work

6 Appendix

## Main Transformation

Rust

```rust
struct AffineTransform {
    a: f64,
    b: f64,
    c: f64,
    d: f64,
    e: f64,
    f: f64,
    weight: f64,
    variation: Variation,
    color: (f64, f64, f64),
}
```

## Post Transformation

Rust

```rust
struct PostTransform {
    a: f64,
    b: f64,
    c: f64,
    d: f64,
    e: f64,
    f: f64,
}
```

# Transformation Formulas

$$F_i(x, y) = (a_i x + b_i y + c_i, d_i x + e_i y + f_i)$$

# Transformation Formulas

$$F_i(x, y) = (a_i x + b_i y + c_i, d_i x + e_i y + f_i)$$

$$F_i(x, y) = \sum_j V_j(a_i x + b_i y + c_i, d_i x + e_i y + f_i)$$

# Transformation Formulas

$$F_i(x, y) = (a_i x + b_i y + c_i, d_i x + e_i y + f_i)$$

$$F_i(x, y) = \sum_j V_j(a_i x + b_i y + c_i, d_i x + e_i y + f_i)$$

$$F_i(x, y) = P_i(\sum_j V_j(a_i x + b_i y + c_i, d_i x + e_i y + f_i))$$

# Variation Formula Examples

$$V_0(x, y) = (x, y)$$

# Variation Formula Examples

$$V_0(x, y) = (x, y)$$

$$V_1(x, y) = (sinx, siny)$$

# Chaos game

Rust

```rust
let mut rng = rand::thread_rng();
let mut x = rng.gen_range(-1.0..1.0);
let mut y = rng.gen_range(-1.0..1.0);
let mut points = Vec::new();
let weights: Vec<f64> = self.transforms.iter().map(|t| t.weight).collect();
let dist = WeightedIndex::new(&weights).unwrap();
for i in 0..iterations {
    let transform_index = dist.sample(&mut rng);
    let transform = &self.transforms[transform_index];
    (x, y) = transform.apply(x, y);
    if i >= 20 {
        points.push(((x, y), transform_index));
    }
}
```

## Moving all points to first quadrant

Rust

```
points.into_iter()
            .map(|((x, y), index)| (post_transform.apply(x, y), index))
            .collect()
```

# Transforming coord points to pixels

Rust

```rust
let min_x = points.iter().map(|((x, _), _)| *x).fold(f64::INFINITY, f64::min);
let max_x = points.iter().map(|((x, _), _)| *x).fold(f64::NEG_INFINITY, f64::max);
let min_y = points.iter().map(|((_, y), _)| *y).fold(f64::INFINITY, f64::min);
let max_y = points.iter().map(|((_, y), _)| *y).fold(f64::NEG_INFINITY, f64::max);

points.into_iter().map(|((x, y), index)| {
  let pixel_x = ((x - min_x) / (max_x - min_x) * (width as f64)).round() as i32;
  let pixel_y = ((y - min_y) / (max_y - min_y) * (height as f64)).round() as i32;
  ((pixel_x, height as i32 - pixel_y), index)
}).collect()
```

## Histogram

Rust

```rust
for &((x, y), index) in pixel_points {
    let transform_color = self.transforms[index].color;
    let entry = histogram.entry((x, y)).or_insert((transform_color, 0));
    entry.1 += 1;
    if entry.1 > 1 {
        entry.0.0 = (entry.0.0 + transform_color.0) / 2.0;
        entry.0.1 = (entry.0.1 + transform_color.1) / 2.0;
        entry.0.2 = (entry.0.2 + transform_color.2) / 2.0;
    } else {
        entry.0.0 = (c.0 + transform_color.0) / 2.0;
        entry.0.1 = (c.1 + transform_color.1) / 2.0;
        entry.0.2 = (c.2 + transform_color.2) / 2.0;
    }
}
```

## Plotting (Without color (Not the same commit as previous codes - Only Log-Density))

Rust

```rust
let max_hits = *histogram.values().max().unwrap_or(&1) as f64;

for (&(x, y), &count) in &histogram {
    let intensity = (count as f64).ln_1p() / (max_hits.ln_1p());
    let shade = (intensity * 255.0).round() as u8;
    let color = RGBColor(shade, shade, shade);
    root.draw_pixel((x, y), &color)?;
}
```

# Outline

1 Introduction

2 Algorithm and Sequential Implementation

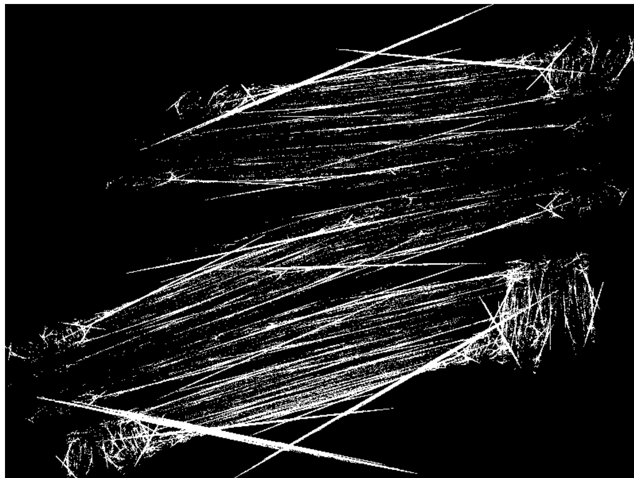
3 Images and Graphs

4 Parallelization Strategies

5 Future Work

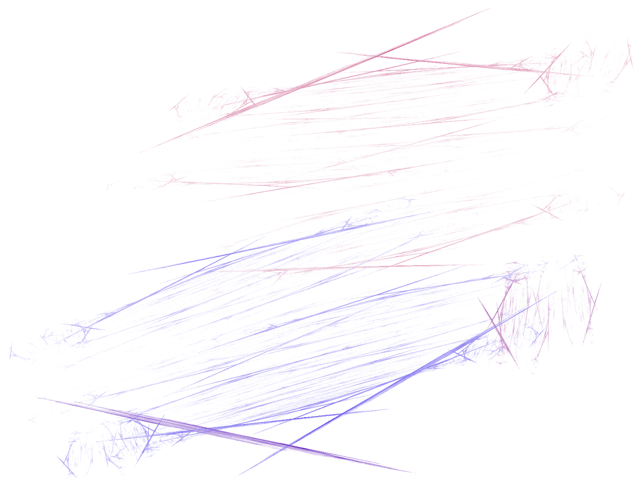
6 Appendix

# Direct Rendering

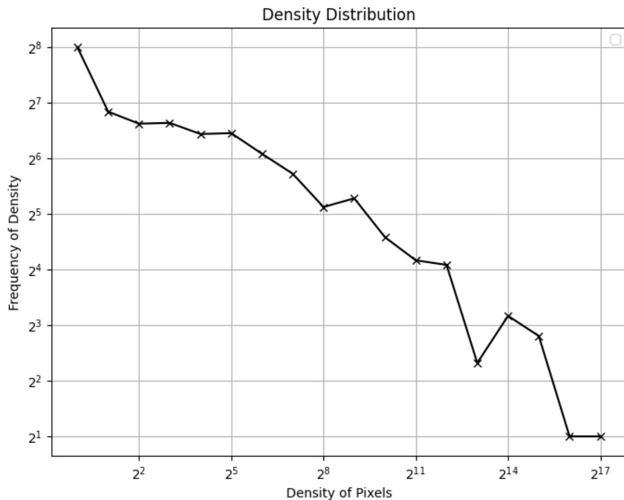


# Log-Density Rendering

# Log-Density Rendering and Coloration



# Density Distribution



# Outline


1 Introduction

2 Algorithm and Sequential Implementation

3 Images and Graphs

4 Parallelization Strategies

5 Future Work

6 Appendix

# Strategy 1

- Distribution of Iterations - Single Starting Point
  - Static Load Balancing - Pre-assign Workload
  - Dynamic Load Balancing - Master-Worker Model

# Strategy 2

- Distribution of Iterations - Multiple Starting Points
  - Static Load Balancing - Pre-assign Workload
  - Dynamic Load Balancing - Master-Worker Model

# Outline

1 Introduction

2 Algorithm and Sequential Implementation

3 Images and Graphs

4 Parallelization Strategies

5 Future Work

6 Appendix

# Future Work

- Sequential
  - Satisfactory coloring logic

# Future Work

- Sequential
  - Satisfactory coloring logic
  - Possibly Gamma Correction, Anti-Aliasing, Motion Blur

# Future Work

- Sequential
  - Satisfactory coloring logic
  - Possibly Gamma Correction, Anti-Aliasing, Motion Blur
  - Check for sections of code that can be further improved after benchmarking

# Future Work

- Parallel
  - Brainstorm more parallelization strategies

# Future Work

- Parallel
  - Brainstorm more parallelization strategies
  - Implement parallelization using MPI

# Future Work

- Parallel
  - Brainstorm more parallelization strategies
  - Implement parallelization using MPI
  - Benchmarking and Improving

# Outline

1 Introduction

2 Algorithm and Sequential Implementation

3 Images and Graphs

4 Parallelization Strategies

5 Future Work

6 Appendix

# References

Draves, Scott and Erik Reckase. “The fractal flame algorithm”. In: *Citeseerx. Recuperado de http://citeseerx. ist. psu. edu/viewdoc/summary* (2008).

Hou, Hao. *Rust Plotters Library*. 2024. URL: https://docs.rs/plotters/latest/plotters/.

Rust-Foundation. *Rust Standard Library*. 2024. URL: https://doc.rust-lang.org/std/index.html.

Softology. *Visions of Chaos*. 2024. URL: https://softology.pro/voc.htm.