Fractals Everywhere: 2D and 3D Procedural Generation and Rendering

Kristian Praizner, Nick Gilligan

December 26, 2025

Abstract

Fractals are geometric structures that exhibit self-similarity across scales and arise from simple iterative rules. This project presents *Fractal IFS Explorer*, a real-time fractal visualization system implemented in the browser using WebGL, Three.js, and custom GLSL shaders. The system supports both 2D Iterated Function Systems (IFS), rendered via a two-pass GPU accumulation pipeline, and 3D fractals defined by signed and unsigned distance fields and rendered by raymarching. A full-featured user interface enables real-time parameter adjustment, animation, and artistic control. Additionally, an offline video-rendering subsystem uses FFmpeg.wasm to generate high-quality MP4 exports at arbitrary resolutions and framerates.

1 Introduction and Motivation

Fractals appear ubiquitously in mathematics, computer graphics, and the natural world. Their structure emerges from iterated transformations, yet the resulting forms display complexity rivaling natural phenomena such as ferns, coastlines, coral, and turbulent flows. This duality of simple rule and complex outcome makes fractals a rich domain for both scientific study and computational art.

The goal of this project is to construct a real-time, browser-based system for rendering and exploring fractals in both two and three dimensions. Unlike traditional offline fractal renderers, our system emphasizes interactive control, GPU acceleration, and high-quality offline video export, enabling both live creative exploration and cinematic production workflows. The implementation combines React, TypeScript, WebGL, Three.js, and custom GLSL shaders to deliver a fast and expressive environment for fractal visualization.

1.1 Background and Influences

This work is primarily inspired by three major sources. First, Barnsley's theory of Iterated Function Systems formalizes the construction of 2D fractals via weighted affine contractions. Classic examples such as the Barnsley fern and Sierpiński triangle arise from compact parameter sets and stochastic iteration. Second, Scott Draves' Fractal Flame algorithm introduced nonlinear variations, density estimation, and sophisticated color-mapping strategies that strongly influenced the design of our 2D accumulation and tone-mapping pipeline [2]. Finally, natural fractal structures such as branching plants, mountain ridges, lightning, and turbulent flows motivate the inclusion of both mathematically precise models and visually expressive shader-based extensions.

2 2D Fractals via Iterated Function Systems

2.1 Affine Iterated Function Systems

We consider a finite set of affine transformations acting on the plane:

$$\mathbf{p}' = M_i\mathbf{p} + \mathbf{t}_i, \tag{1}$$

where

$$M_i = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix}, \qquad \mathbf{t}_i = \begin{bmatrix} e_i \\ f_i \end{bmatrix}. \tag{2}$$

Each transformation $T_i = (M_i, \mathbf{t}_i)$ is selected with probability p_i. When the transformations are contractive, Barnsley's theorem guarantees the existence of a unique attractor invariant under the IFS. Different parameter choices generate drastically different fractal structures. For example, the Sierpiński triangle arises from three uniform contractions toward the triangle vertices, while the Barnsley fern emerges from four asymmetric transformations modeling biological growth. Our system provides a preset library containing these and additional spiral and dragon-type systems.

2.2 Chaos Game Algorithm

Before introducing the procedural steps, we briefly motivate why the chaos game is such a powerful and surprising tool in fractal generation. Even though an IFS is defined by deterministic affine maps, directly iterating all possible compositions quickly becomes infeasible due to exponential growth. The chaos game provides an elegant stochastic alternative: by repeatedly applying randomly chosen maps according to their probabilities, the sequence of points almost surely converges onto the fractal's invariant set. This probabilistic approach yields the full structure of the attractor without ever computing its geometry explicitly, making it ideally suited for high-throughput GPU rendering and interactive exploration.

The chaos game is implemented as follows:

1. Initialize a random point $\mathbf{p}_0$ in a bounding region.
2. Iterate the system for N_{burn} steps without plotting to allow convergence to the attractor.
3. For each subsequent iteration:
   (a) Sample a transformation index i according to probabilities $\{p_i\}$.
   (b) Apply $\mathbf{p}_{k+1} = M_i \mathbf{p}_k + \mathbf{t}_i$.
   (c) Store the resulting point and associated color.

For performance, large batches of generated points and their attributes are stored in `Float32Array` buffers before being uploaded to the GPU. Optional coloring schemes based on transformation index, iteration depth, and density are supported, along with morphing between IFS presets for animation.

2.3 2D GPU Rendering Pipeline

The 2D system separates computation into a two-pass GPU pipeline:

1. **Point generation (CPU).** The chaos game iteratively generates large batches of 2D fractal sample points.
2. **Accumulation buffer (GPU).** Points are rasterized with additive blending into a floating-point framebuffer, building a density histogram. Zoom, pan, rotation, and adaptive level-of-detail are handled at this stage.
3. **Tone mapping and color sampling.** A fullscreen quad samples the density texture and applies logarithmic tone mapping followed by palette- or gradient-based colorization.
4. **Post-processing.** Additional screen-space effects such as bloom, chromatic aberration, kaleidoscope symmetries, trails, and rotation are applied.

All rendering parameters, including iterations per frame, brightness, bloom thresholds, color palettes, decay, psychedelic modes, and morphing, are exposed via a real-time Leva control panel.

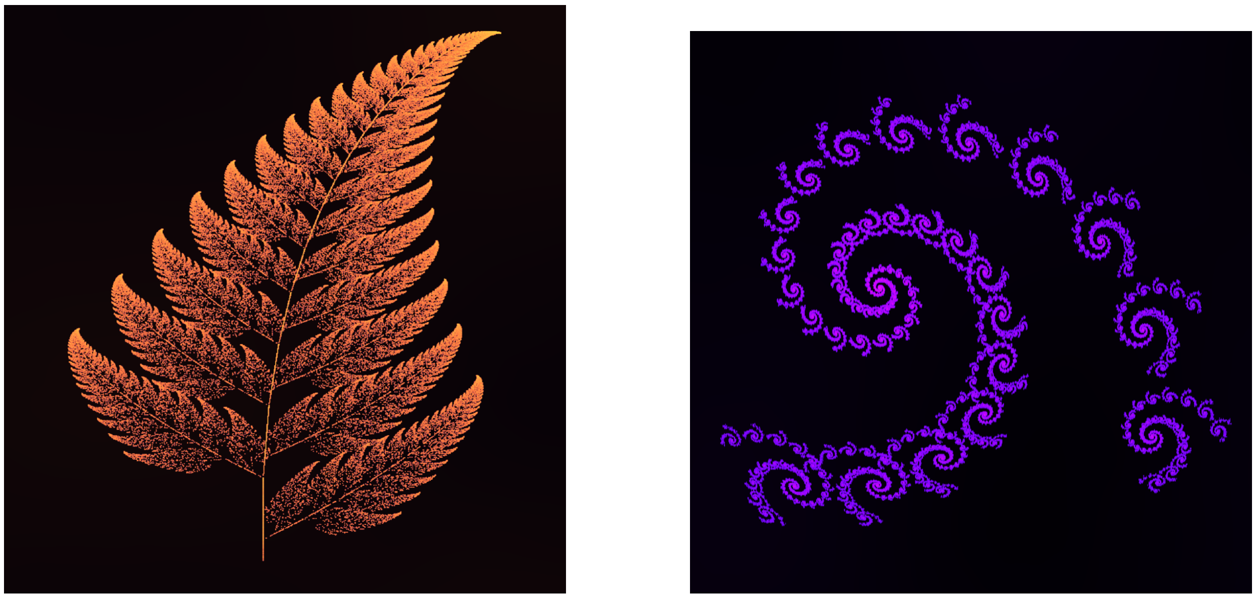

Figure 1: Example 2D fractals generated by the affine chaos game and rendered with the 2D pipeline (fern and spiral shapes).

3 3D Fractals: Mandelbulb Power Iteration

3.1 Mandelbulb Formulation

The Mandelbulb generalizes the complex quadratic iteration $z \mapsto z^p + c$ to three dimensions using spherical coordinates. Given a point $\mathbf{x} \in \mathbb{R}^3$, we:

1. Convert $\mathbf{x}$ to spherical coordinates (r, θ, φ).
2. Apply the power p:
$$r \leftarrow r^p, \qquad \theta \leftarrow p\theta, \qquad \varphi \leftarrow p\varphi.$$
3. Convert back to Cartesian coordinates and add a constant vector $\mathbf{c}$.

Repeated iteration generates a 3D fractal set whose symmetry and complexity are controlled by the power parameter p.

3.2 Distance Estimation and Raymarching

Rendering is performed using a signed distance estimator
$$DE(\mathbf{x}) \approx \frac{1}{2} \frac{r \log r}{dr}, \tag{3}$$
where dr is a derivative term updated during iteration. A sphere-tracing raymarcher advances rays by the estimated distance until a surface hit or miss condition is reached. Surface normals are approximated using finite differences, enabling physically motivated lighting including ambient, diffuse, and specular components, along with simple ambient occlusion, glow, and background gradients.

3.3 3D Rendering Pipeline

The full 3D pipeline consists of:

1. Construction of distance fields for Mandelbulb, Mandelbox, Menger sponge, and 3D Julia sets.
2. Per-pixel raymarching on a fullscreen quad using GLSL fragment shaders.
3. Normal estimation, lighting, shading, and optional animation.

Uniform parameters control animation modes such as parameter morphing, escape-time slicing, and animated folding, driven by a global time variable. Interactive camera orbit and zoom are supported in the UI.

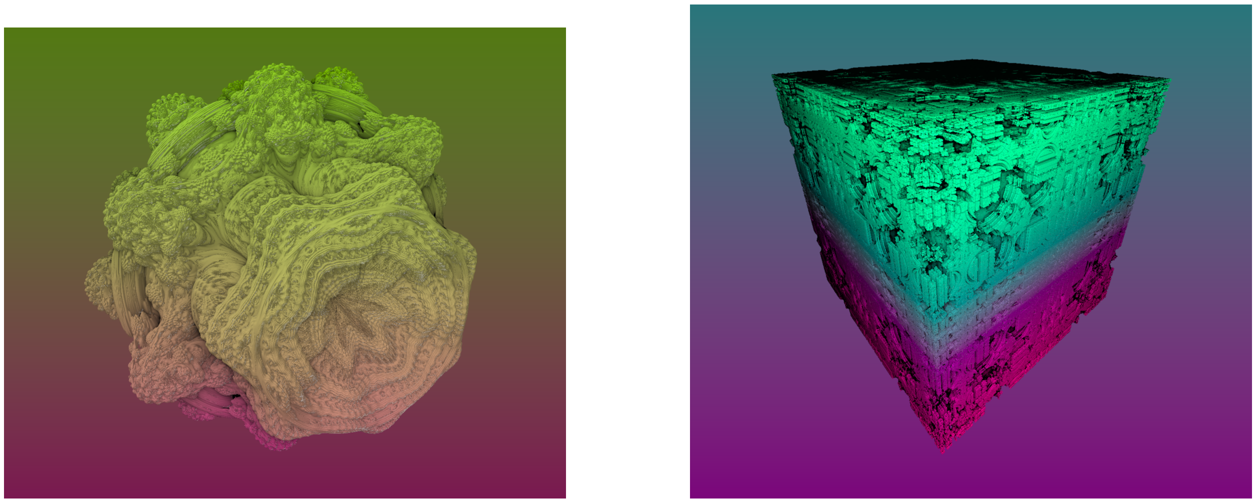

Figure 2: Rendered Mandelbulb and Mandlebox using the SDF-based raymarching pipeline.

4 Frameworks & Tools Summary

The system is implemented as a modern web application that layers a React front end on top of a WebGL-based rendering core. Tables 1–3 summarize the principal technologies and their roles.

4.1 Rendering Stack

Table 1: Rendering frameworks and tools.

Layer	Technology	Purpose
WebGL	Three.js	3D graphics abstraction; manages WebGL.
Shaders	GLSL	Custom vertex/fragment shaders.
Shader import	vite-plugin-glsl	Import `.vert`/`.frag` files as strings.

4.2 UI and State Management

Table 2: UI and application state technologies.

Layer	Technology	Purpose
Framework	React 18	Component architecture.
Control panel	Leva	Real-time parameter tweaking.
Language	TypeScript	Static type checking and tooling.

4.3 Build and Development Tooling

Table 3: Build, deployment, and hosting tools.

Layer	Technology	Purpose
Bundler	Vite	Fast development server.
Hosting	GitHub Pages	Static site hosting at `kristianpraizner.com/fractal`.

4.4 Architecture Flow

At a high level, the application architecture forms a unidirectional control and data flow from the React UI down to the GPU:

React (UI state)
↓
Leva (control panel)
↓
Three.js (WebGL renderer)
↓
GLSL shaders (GPU computation)
↓
HTML canvas (display)

React manages application and animation state; Leva exposes that state as a live control surface. Three.js translates the resulting scene graph and uniforms into WebGL draw calls, while GLSL shaders implement the actual fractal computation and post-processing, with the final image presented in an HTML canvas element.

5 Results and Discussion

The system produces high-quality fractal imagery in both 2D and 3D at interactive framerates on commodity hardware. The two-pass accumulation and tone-mapping pipeline yields smooth, high-dynamic-range density fields, while the 3D raymarcher produces highly detailed implicit surfaces with rich self-similarity. Real-time animation reveals continuous structural transitions in both 2D and 3D fractals. Performance remains interactive due to GPU-centric design, bounded raymarch step counts, and adaptive distance estimator iteration limits. High-resolution still capture and 4K offline rendering are supported.

6 Conclusion and Future Work

We have presented a browser-based, GPU-accelerated fractal visualization platform supporting both classic 2D Iterated Function Systems and modern 3D distance-field fractals. The system integrates real-time rendering, interactive UI control, animation, and offline video export into a unified production environment. The project demonstrates that high-quality procedural fractal rendering can be achieved entirely within a web-based framework.

Future Work. Future work will include the addition of real-time 3D parameter slicing controls, higher-resolution performance optimizations, VR/AR visualization modes, and hybrid neural–fractal rendering techniques for learned distance fields and stylized output.

References

[1] Michael F. Barnsley. *Fractals Everywhere.* Academic Press, 1988.

[2] Scott Draves. The fractal flame algorithm. In *Non-Photorealistic Animation and Rendering*, 2003.