
Mandelbrot Metal

Mandelbrot Metal

Executive Summary

Mandelbrot Metal is my high-performance fractal renderer and explorer for iPhone and iPad. It delivers smooth, real-time exploration of the Mandelbrot set with adaptive precision (FP32 → FP64 → double-double), continuous coloring, super-sampling anti-aliasing, a flexible palette system with gamma-aware interpolation, contrast control, and high-resolution export. The renderer prioritizes interactivity during navigation and automatically raises quality when you pause, producing studio-grade images without manual tweaking.

Overview

I built Mandelbrot Metal to make deep fractal exploration fast, precise, and beautiful. It runs a Metal compute kernel that evaluates the escape-time algorithm in parallel, maps continuous iteration values to color via GPU LUTs, and presents the result with minimal CPU–GPU overhead. My goals were simple: instantaneous feel while moving, and uncompromised fidelity when still.

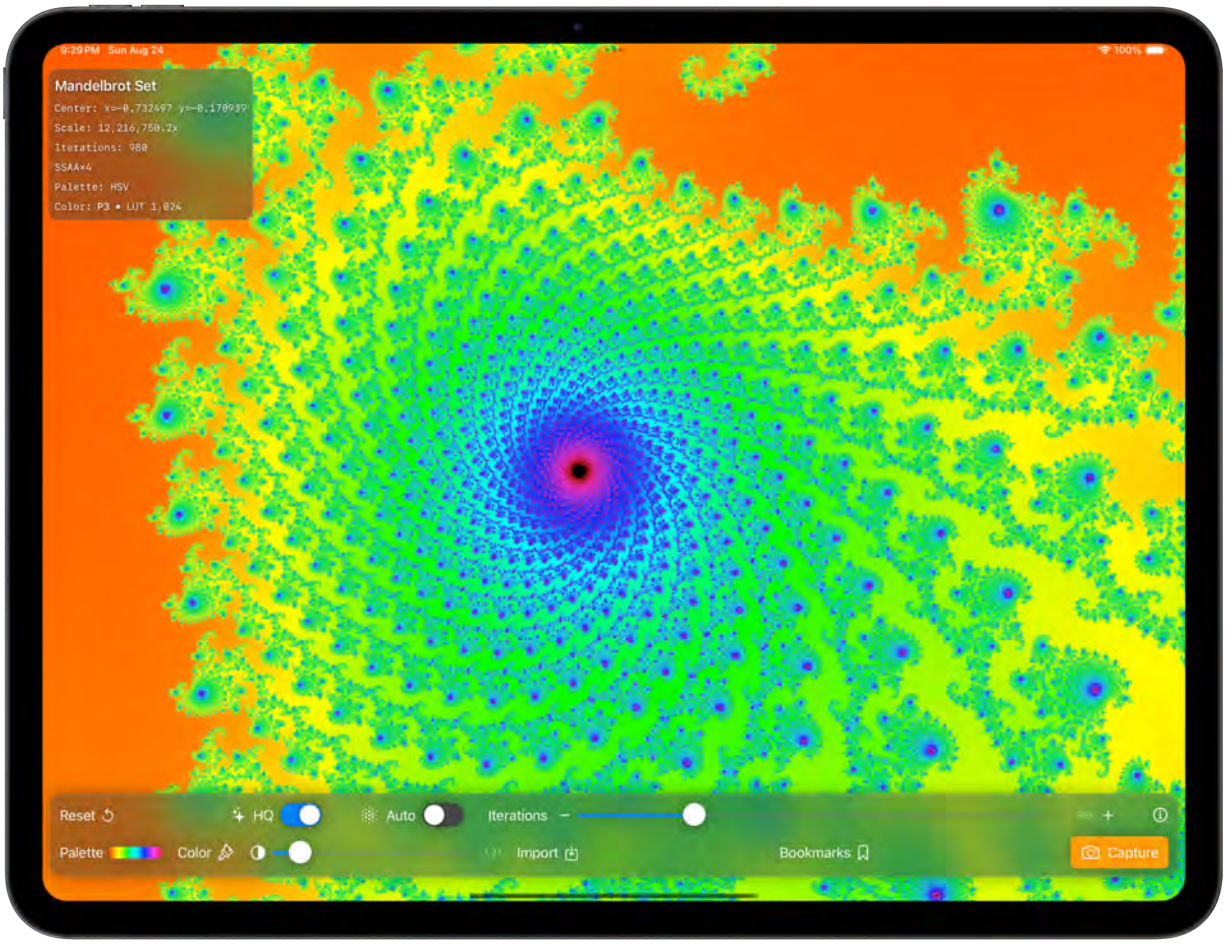


Figure 1 – Mandelbrot Metal on the iPad Pro

Key Features

- Real-time GPU rendering of the Mandelbrot set
- 40+ built-in scientific and artistic color palettes
- Palette Editor with gradient stops and live preview
- Gradient Import from any image with gamma-aware sampling
- Lookup table (LUT) palettes (1×N textures) with configurable resolution (256/1024)
- Instant palette swapping (coloring decoupled from iteration work)
- Nonlinear Iteration Slider for fine and wide-range control
- Auto Iteration scaling with zoom depth
- Adaptive Precision (FP32 → FP64 → double-double)
- HQ Idle mode with SSAA and higher precision
- Deterministic capture and bookmarking

Figure 2 – Mandelbrot Metal on the iPhone 16 Pro

The Mandelbrot Set: An Overview

Fractals

A fractal is a mathematical set or geometric structure that exhibits self-similarity across scales and contains detail at arbitrarily fine resolutions. Formally, fractals are often characterized by having a Hausdorff (or fractal) dimension that exceeds their topological dimension, reflecting their intrinsic complexity.

Fractals are typically generated by iterative or recursive processes, where the repeated application of simple rules produces structures of great intricacy. The Mandelbrot set is a canonical example: its boundary is infinitely complex, and magnification reveals patterns that echo the structure of the whole, a hallmark of self-similarity. There are other canonical fractals including the Julia Set and the Burning Ship. I will be adding these to the app in the future.

Introduction to the Mandelbrot Set

At the heart of the complex plane lies the enigmatic fractal known as the Mandelbrot set, a mathematical marvel distinguished by its intricate boundaries and infinite depth. When a specific operation is repeatedly applied to complex numbers, those outside the set diverge toward infinity, while those within remain, tracing subtle and mesmerizing patterns. The boundary itself is a stage for endlessly detailed wanderings, revealing astonishing variety and beauty.

Figure 3 – The complete Mandelbrot Set as captured in Mandelbrot Metal

Origins and Historical Context

Named for Benoit B. Mandelbrot, a research fellow at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, the set emerged from his

pioneering work in geometric forms during the mid-1970s. Mandelbrot's explorations gave rise to fractal geometry, a field dedicated to studying shapes with fractional dimensions.

Exploring the Set

The boundary of the Mandelbrot set exemplifies a fractal, presenting a unique subject for mathematical exploration. Through specialized software, computers become virtual microscopes, revealing ever more detailed structures at increasing magnification. From afar, the set appears as a squat, wart-covered figure eight on its side, with a shadowy black interior surrounded by vibrant halos of electric blue, deep yellow, red, green, and more. This visual complexity invites endless curiosity and study.

Complex Numbers and the Complex Plane

A complex number is of the form: *a + bi*, where:

- *a* is the real part
- *b* is the imaginary part
- *i* is the imaginary unit ($\sqrt{-1}$)

Complex numbers can be plotted on a set of coordinates in which the horizontal axis the real part and the vertical axis is the imaginary part.

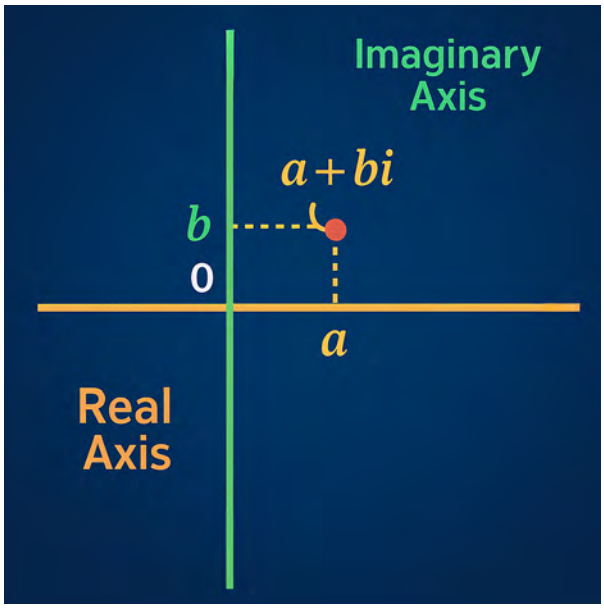


Figure 4 – The complex plane with its real and imaginary axes showing the coordinates of the complex number a + bi.

Complex Number Addition

Two complex numbers *a + bi* and *c + di* add as follows:

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

Complex Number Multiplication

Two complex numbers *a + bi* and *c + di* multiply as follows:

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$$

Theory of Operation

Mapping the Complex Number Plan to Display Screen Coordinates

The Mandelbrot set is defined as the set of all complex numbers c for which the recurrence:

$$z_0 = 0, \quad z_{n+1} = z_n^2 + c$$

remains bounded as n approaches infinity.

If $|z_n|$ (here the absolute value means the complex modulus) never exceeds 2 within a practical iteration budget, I treat c as in-set; otherwise, it escapes. The boundary is fractal—infinitely detailed and self-similar in structure. This is the region of interest.

Why the bailout is 2: If $|z| > 2$ at any step, $|z_{n+1}| = |z^2 + c| \geq |z|^2 - |c|$, which grows without bound.

If $|z_n|$ never exceeds a bailout radius (I use 2) within a maximum iteration budget, I treat c as inside the Mandelbrot set; otherwise, it's outside. The boundary exhibits infinite, nontrivial structure at every scale.

Visually, the Mandelbrot set is a map of stability in the complex plane—black areas remain bounded, while colored areas diverge, revealing intricate self-similar patterns.

I map each display pixel (x, y) to a complex coordinate via a linear transform. Each pixel (x, y) on the screen corresponds to a complex number c in the plane. As you zoom and pan, the app recalculates the mapping so that:

$$\mathrm{Im}(c) = y_{\min} + \frac{y}{H}(y_{\max} - y_{\min}), \quad \mathrm{Re}(c) = x_{\min} + \frac{x}{W}(x_{\max} - x_{\min})$$

Zooming in or out changes these bounds—not the underlying math.

Escape-Time Core Algorithm

For each pixel's c:

1. Set $z = 0$
2. Repeat $z \leftarrow z^2 + c$ until up to a max iteration count
3. If $|z| > 2$, the point has escaped — color is based on iteration count at escape
4. If never escaped, color it black (inside set)

This is known as the escape-time algorithm.

Coloring the Mandelbrot Set

Interior (the set itself)

Points c that do not escape—meaning $|z_n|$ never exceeds the escape radius (typically 2) within the maximum allowed iterations—are considered part of the Mandelbrot set. These are colored black in Mandelbrot Metal.

Exterior (escape points)

Points c for which $|z_n|$ exceeds the escape radius are classified as outside the set. For these points, we record the iteration count at which escape occurs. This value is then converted using a smooth coloring function (such as the continuous escape-time formula μ) and finally mapped to a color via a 1×N lookup table (LUT).

This means:

- Slow escapes (low iteration counts) map to one part of the palette.
- Fast escapes (high iteration counts, near the boundary) map to another.
- Smooth interpolation between these values avoids visible bands.

Result:

The black regions form the infinitely detailed Mandelbrot set itself, while the rich gradients and colors are applied to the escaping points, revealing the filaments, spirals, and self-similar structures around the boundary.

Auto Iteration and Manual Control

Rendering fidelity in the Mandelbrot set depends critically on the maximum iteration count. Shallow zooms require only modest iteration budgets to resolve detail, but deeper zooms demand exponentially more iterations to prevent premature escape detection and banding artifacts.

Auto Iteration

To balance speed and accuracy, I use an auto iteration function that increases the iteration cap as you zoom in. The Auto Iteration controller scales the maximum iteration budget with zoom depth using a polynomial in log scale:

$$n_{\max} \approx n_0 + a \log_{10}(\rho) + b\,[\log_{10}(\rho)]^2 + c$$

where:

- $n_{\max}$ is the iteration limit for the current zoom
- ρ is the zoom factor (relative to baseline scale)
- n_0, a, b, c are tuned coefficients chosen empirically for smooth progression

This scaling ensures that as zoom depth increases, iteration budgets rise automatically, keeping the image smooth and free of noise without user intervention.

Manual Slider (Nonlinear Response)

In addition to auto iteration, the app provides a manual iteration slider for fine-tuning. The slider is nonlinear, meaning its position is mapped to iteration count via an exponential or logarithmic curve rather than linearly. This design has two benefits:

1. Precision at low values — users can make small adjustments where low iteration counts are sensitive to change.
2. Reach at high values — the slider still allows access to very high iteration counts (thousands+) without requiring impractically long slider travel.

In practice, this nonlinear control makes manual tuning feel natural, letting you glide smoothly between quick, coarse previews and high-quality, detail-preserving renders.

Coloring Smoothing Algorithm

To avoid aliasing, I compute a smooth iteration value, μ, where:

$$\mu = n + 1 - \frac{\ln(\ln|z|)}{\ln 2}$$

I then map μ into a palette—built-in or lookup table (LUT)—with linear interpolation between gradient stops. Contrast is applied in a perceptual space for predictable mid-tone control.

This produces gradients instead of harsh bands. Palettes are applied by mapping μ to a color gradient. Colors are averaged in linear space, then re-encoded for display.

Color Palettes

Mandelbrot Metal features a large palette library

- **Built-in Library** – >40 professionally crafted palettes spanning scientific, artistic, and creative styles.
- **Favorites** – Mark your favorite palettes and they will appear at the top of the library with a ★.
- **Palette Editor** – Create fully custom palettes by defining and adjusting gradient stops, with live preview and precise color stop positioning.
- **LUT-Based Implementation** – Palettes are stored as 1×N lookup textures in either sRGB or Display P3 color space. Linear interpolation between stops is performed in a gamma-aware manner for perceptually accurate color blending.
- **Configurable Resolution** – LUT width can be switched between 256 and 1024 samples, allowing a trade-off between gradient smoothness and memory usage.
- **Real-Time Swapping** – Palette changes apply instantly without triggering a re-iteration pass; coloring is decoupled from the iteration workload.
- **GPU-Optimized** – All palettes are designed for efficient GPU sampling, with minimal overhead in both deep zoom and high-iteration scenarios.
- **Import Gradient** – Sample images in the your photo library as a source for custom gradients to fine-tune and then save as to the palette library for reuse.

Figure 5 – Palette Library in Mandelbrot Metal

- **Per-Palette Metadata** – Each palette can store metadata such as name, category, description, and favorite status, making it easy to organize large collections.

The palette library includes a range from scientific to artistic schemes, all GPU-optimized. You can create your own palettes by editing gradient stops.

GPU Pipeline (Metal)

1. The CPU computes viewport parameters (center, scale), selects precision and iteration budget, and uploads small uniforms and palette LUTs
2. The GPU compute kernel (one thread per pixel/subsample) performs the escape loop, computes smooth μ, and looks up colors in linear space before re-encoding for display.
3. Color mapping with gamma-aware interpolation.

4. Presentation writes directly into a renderable texture bound to the drawable for low-latency display.

The kernel is designed to minimize branching, leverages fused multiply-add (FMA) operations where supported, and keeps palette data resident in fast memory (threadgroup or constant) to reduce fetch latency.

Super-Sampling Anti-Aliasing (SSAA)

General SSAA (unweighted box filter)

For a pixel centered at (x, y) and N sub-pixel samples at offsets $(\delta x, \delta y)$:

$$\mathbf{C}(x,y) = \frac{1}{N}\sum_{k=1}^{N} \mathbf{L}(x+\delta x_k,\ y+\delta y_k)$$

L returns linear-space RGB—or intensity—values.

Weighted form (any filter):

$$\mathbf{C}(x,y) = \sum_{k=1}^{N} w_k\,\mathbf{L}(x+\delta x_k,\ y+\delta y_k),\quad \sum_{k=1}^{N} w_k = 1,\ w_k \geq 0$$

Sample positions for SSAA×m (m × m grid)

Let $\mu, \nu \in (0, \ldots, m\text{-}1)$ and $M = m^2$. A canonical grid about the pixel center:

$$\delta x_{uv} = \frac{u+\frac{1}{2}-\frac{m}{2}}{m},\quad \delta y_{uv} = \frac{v+\frac{1}{2}-\frac{m}{2}}{m},\quad u,v \in \{0,\ldots,m-1\},\ N = m^2$$

Color-management note (average in linear space)

If Γ represents the display's transfer function (e.g., sRGB or Display P3), I perform the averaging in linear color space, then re-encode the result using Γ:

$$\mathbf{C}_{\text{display}} = \Gamma\left(\frac{1}{N}\sum_{k=1}^{N} \Gamma^{-1}(\mathbf{C}_k)\right)$$

Mandelbrot-specific instantiation

Map each sub-sample to the complex plane C_k, run escape-time, compute smooth iteration μ_k, palette in linear space, then average:

$$\mu_k = n_k + 1 - \frac{\ln(\ln|z_k|)}{\ln 2}$$

$$\mathbf{c}_{\text{display}} = \Gamma\left(\frac{1}{N}\sum_{k=1}^{N}\mathbf{c}_k\right)$$

If not converting to linear, I will replace C_k with:

$$\mathbf{c}_k = \text{Palette}_{\text{lin}}(\mu_k)$$

At deep zooms, the boundary's high spatial frequency can alias. When HQ idle is active—and only when interaction pauses—I switch to SSAA:

- For SSAA×2, ×3, ×4 I sample a rotated grid of 4/9/16 sub-pixels per pixel.
- Each sub-sample runs a full escape test; colors are averaged in registers and written once.
- Sample patterns are chosen to suppress moiré; the palette lookup is done per sub-sample to keep gradients smooth.

Auto Iterations & Manual Slider

Detail demands grow with magnification. I compute an iteration target from the current scale relative to a baseline:

$$n_{\text{max}} \approx n_{\text{base}} + a\log_{10}(\rho) + b\,[\log_{10}(\rho)]^2 + c$$

where ρ is the scale ratio and a, b, and c are tuned constants. This keeps fine filaments crisp without unnecessary work at wide views.

Manual control is provided by a nonlinear slider mapping slider position s ∈ [0,1] to iteration count:

$$n(s) = \lfloor n_{\text{min}}\,(\frac{n_{\text{max}}}{n_{\text{min}}})^{s^\gamma} \rfloor, \quad s \in [0,1], \;\; \gamma > 0$$

Adaptive Precision & Double-Double Precision

The renderer dynamically promotes numeric precision based on zoom depth and pixel scale. FP32 is used for shallow zooms, FP64 for deeper views, and double-double arithmetic (~106-bit mantissa) for extreme zooms and export-quality stills. Double-double is implemented as:

1. FP32 (single) for wide views and moderate zooms—the fastest.
2. FP64 (double) once pixel size in the complex plane falls below a threshold (e.g., $<10^{-8}$), preventing coordinate collapse.

3. Double-Double (DD) for extreme zooms: I represent a value *x* as *x = x_hi + x_lo*, where each term is a 64-bit double. With Dekker-Veltkamp-style error-free transforms, I implement add/multiply with explicit error correction:

    - Two-Sum (error-free addition of doubles)
    - Two-Prod-FMA (error-captured multiplication using FMA)
    - Complex multiply uses DD operations on both real and imaginary parts, preserving ~106 bits (~32 decimal digits) of precision.

When I switch: I monitor the world-space delta between adjacent pixel centers and compare it against the effective mantissa of the current mode (including accumulated rounding in the escape loop). If precision margin drops below a safety factor, I promote the kernel to the next mode. Promotion occurs during HQ idle, so interaction stays snappy. Demotion occurs when zooming back out.

(Later, I may implement perturbation rendering. I will then evaluate *z* as a series around a high-precision reference orbit and ship only small deltas per pixel. My framework already has a fallback hook for that.)

Gradient Import

Gradient Import lets you turn any image into a palette. The app samples a row, column, or averaged stripe, converts the colors to linear space, smooths them, and stores them as a 1×N LUT. You can then edit stops, adjust contrast, and save the palette with metadata. This enables artistic exploration using palettes derived from nature, artwork, or photographs.

- LUT = Look-Up Table. In graphics, this usually means a precomputed array of colors that can be indexed quickly.
- 1×N = a 1-pixel-tall, N-pixel-wide texture.
    - The "1" means it's just a single row (one dimension of variation).
    - The "N" means it has *N* samples (e.g., 256 or 1024 pixels wide).

So essentially, it's a strip of colors laid out horizontally.

- Each entry corresponds to a palette "stop."
- When rendering, the shader takes the normalized smooth iteration value (say between 0 and 1), multiplies by N, and samples from this texture.
- Because it's a GPU texture, the hardware automatically interpolates between samples (linear filtering), so colors blend smoothly.
- User may toggle Exact LUT on/off. With Exact LUT disabled, the colors do not blend smoothly and are allowed to band.

Image Capture

The capture path renders at the exact canvas resolution to avoid tiling seams, then optionally scales to a target (Canvas, 4K, 6K, 8K, or custom) using aspect-fit. Export is PNG for lossless quality and correct color management. Before capture I commit any in-flight gestures and sync the renderer, so framing is exact.

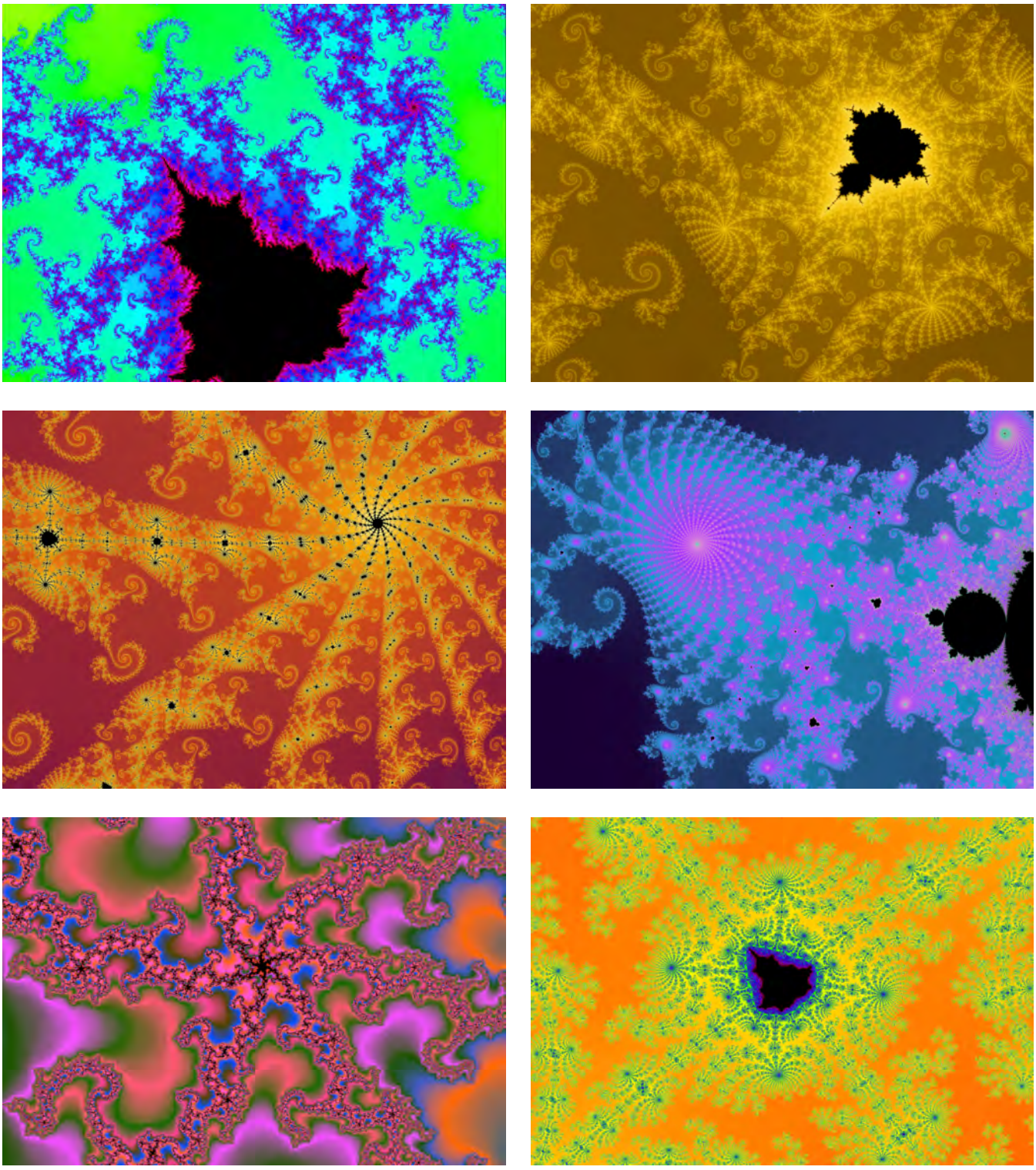


Figure 6 – Sample rendered Mandelbrot set images in Mandelbrot Metal, showing detail and smooth coloring at deep zoom levels. Each point represents a complex number c used in the Mandelbrot formula. Zoomed regions reveal repeating patterns.

Stability and Reproducibility

- Arithmetic pathways are deterministic per device and precision mode; palette LUTs are versioned; bookmarks store center/scale/iterations/contrast/palette name, so scenes are reproducible.
- The HUD exposes center, scale, iteration budget (or auto), SSAA status, and palette name to make captures auditable.

Benchmarks

On recent model iOS and iPadOS devices, the compute kernel sustains interactive rates at 1–3K iterations with SSAA off during interaction, promoting to SSAA×2, SSAA×3, SSAA×4, and higher iterations when idle. Double-double precision is reserved for the deepest zooms and is enabled only in idle to preserve responsiveness.

Preliminary Benchmarks

Device	Precision	Iterations	SSAA	Resolution	Frame rate (interactive)	Idle quality bump
iPad Pro 13″ M4 (Apple silicon)	FP64	1,000	Off	2752 × 2064	~50–60 fps	SSAA×3 @ ~12–18 fps
iPad Pro 13″ M4 (Apple silicon)	Double-Double	1,500	×2	2752 × 2064	~10–14 fps	SSAA×3 @ ~6–9 fps
iPhone 16 Plus (Apple silicon)	FP64	800	Off	2796 × 1290	~45–60 fps	SSAA×2 @ ~14–20 fps

Why This Matters

The Mandelbrot set is the best example I know of emergent complexity from simple rules. With Mandelbrot Metal, you don't just view pictures—you explore a living mathematical landscape at full fidelity, guided by a renderer that adapts its precision and quality to what you're doing.

About the Developer

Greetings! Michael Stebel, here—software developer and amateur physicist. I'm passionate about scientific programming, computational visualization, fractals, and GPU programming. Mandelbrot Metal is the result of my work combining high-performance graphics with the timeless beauty of mathematics.

I am also a semi-retired software industry executive who's spent over four decades steering both public and private companies to growth and success. Along the way, I've also embraced my inner geek as a programmer, dabbled in the wonders of science as an amateur scientist, and kept my eyes on the future as a passionate technologist.

For more information and app support, visit mandelbrotmetal.com.

Appendix

Pseudocode for Concept Clarity

Escape + Smooth Coloring (per sample)

```
z = 0
for n in 0..<n_max {
    // ddMul/add select FP32, FP64 or Double-Double ops
    z = ddMul(z, z) + c
    if dot(z, z) > 4 { break }
}
if n == n_max:
    color = black
else:
    mu = n + 1 - log(log(|z|)) / log(2)
    color = palette(mu)
```

Double-Double building blocks (sketch)

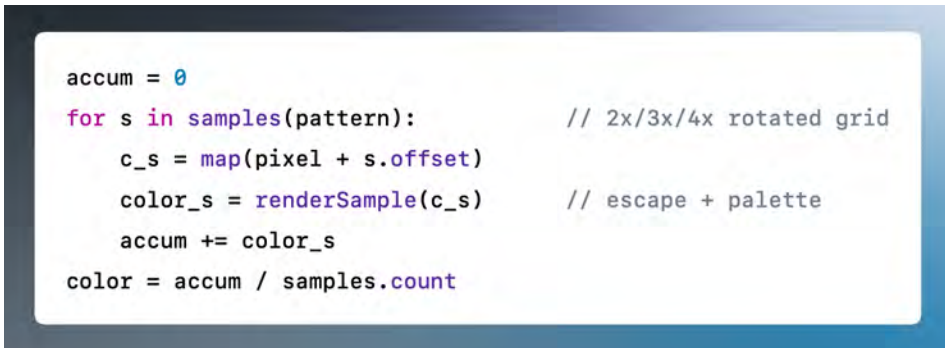
```
struct DD { double hi, lo }

func twoSum(a, b) -> (s, e) { ... }        // error-free add
func twoProdFMA(a, b) -> (p, e) { ... }    // use fma for residual

func ddAdd(x: DD, y: DD) -> DD { ... }
func ddMul(x: DD, y: DD) -> DD { ... }

func cmul(x: DD2, y: DD2) -> DD2 {         // complex DD
    // (xr + i xi) * (yr + i yi)
    rr = ddMul(x.r, y.r) - ddMul(x.i, y.i)
    ri = ddMul(x.r, y.i) + ddMul(x.i, y.r)
    return (rr, ri)
}
```

SSAA (idle only)



```
accum = 0
for s in samples(pattern):            // 2x/3x/4x rotated grid
    c_s = map(pixel + s.offset)
    color_s = renderSample(c_s)       // escape + palette
    accum += color_s
color = accum / samples.count
```