GPU-Accelerated Mathematical Illustration

An Introduction to Shader Programming

Steve Trettel

December 2025

Table of contents

About

This mini-course introduces shader programming as a tool for mathematical illustration and exploration. Shaders are programs that run in parallel on the GPU, making them exceptionally fast for visualization tasks. We'll learn to write code that "reads like mathematics" using Shadertoy, a beginner-friendly web-based platform that handles all the low-level programming complexities.

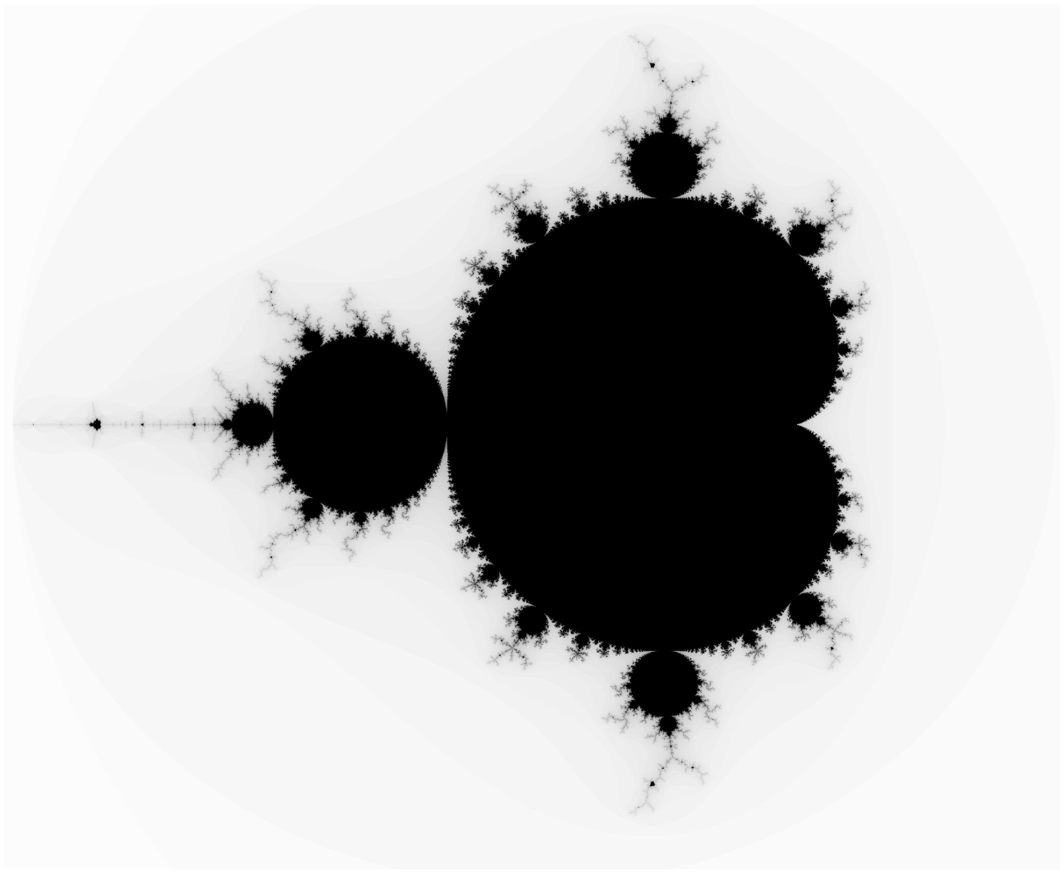
We'll progress from 2D foundations (curves, tilings, fractals) to 3D rendering via raymarching. Along the way, we will implement classic examples like the Mandelbrot set, hyperbolic tessellations, and implicit surface renderers. The final day will explore either advanced geometric techniques (domain operations, 3D fractals) or temporal simulation methods (PDEs, buffer-based dynamics), depending on the group's interests.

No prior experience with shaders or GLSL is required—only a strong foundation in undergraduate mathematics and willingness to work hard and experiment with code through daily homework exercises. Here are some examples of things we will make:

The code

and the demo

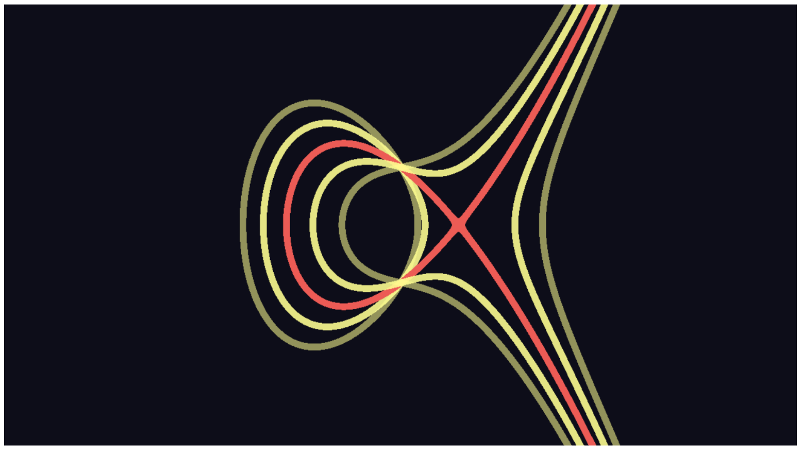
and a still pic for testing



1 Day 1: Introduction to Shader Programming

1.1 Overview

By the end of today, you'll be able to create this:



A family of elliptic curves $y^2 = x^3 + ax + b$, drawn for several values of a simultaneously, with b varying across the screen. The curves shift in brightness to show the family structure, and you can watch singularities appear and disappear along the discriminant locus.

This image is computed in real time, every pixel evaluated independently on the GPU. To get here, we'll learn:

- What a shader is: a function from coordinates to colors, evaluated in parallel
- How to set up a coordinate system for mathematical visualization
- How to draw shapes using distance functions
- How to render implicit curves $F(x, y) = 0$ with uniform thickness
- How to add interactivity with mouse input

Let's begin.

1.2 What is a Shader?

We want to draw images on a screen.

Mathematically, an image is a function from a region $S \subset \mathbb{R}^2$ to the space of visible colors $\mathscr{C}$. This color space is three-dimensional, spanned by the responses of the three types of cone cells in our eyes. A convenient basis, roughly aligned with these responses, is red, green, and blue.

To realize this on a computer, we discretize. A screen is a grid of *pixels*: X pixels wide, Y pixels tall. Each pixel is a point in the integer lattice

$$\{0, 1, \ldots, X-1\} \times \{0, 1, \ldots, Y-1\}.$$

Colors are represented as RGB triples: red, green, and blue intensities, each in $[0, 1]$. The constraint to $[0, 1]$ reflects physical reality—a pixel has a maximum brightness it can display. (We can't draw the sun.) So an image is a function

$$f\colon \{0, \ldots, X-1\} \times \{0, \ldots, Y-1\} \to [0, 1]^3$$

$$(i, j) \mapsto (r, g, b).$$

In practice, we add a fourth component: *alpha*, representing transparency. This matters when compositing multiple layers (we won't use it in this course, but the machinery expects it). So our shader computes

$$f\colon (i, j) \mapsto (r, g, b, 1).$$

This is what a shader is. You write a function that takes pixel coordinates and returns an RGBA color. The GPU evaluates your function at every pixel to produce the image.

1.2.1 Parallelism

A 1920×1080 display has over two million pixels. How do we evaluate f at all of them fast enough to animate at 60 frames per second?

The answer is parallelism. A GPU contains thousands of cores, and it evaluates f at all pixels *simultaneously*. There's no loop over pixels in your code—you write f, and the hardware handles the rest.

The tradeoff: each pixel's computation must be *independent*. Pixel $(100, 200)$ cannot ask what color pixel $(100, 199)$ received. Every pixel sees the same global inputs—coordinates, time, mouse position—and must determine its color from those alone. Learning to think within this constraint is what shader programming is about.

Why "shader"?

The name comes from 3D graphics, where these programs computed *shading*—how light interacts with surfaces. It stuck even though we now use shaders for fractals, simulations, and mathematical visualization.

1.2.2 Why Shadertoy?

Shader programming normally requires substantial setup: OpenGL contexts, buffer management, compilation, render loops. Shadertoy abstracts all of this—you write one function, press play, and see results. We'll use it throughout the course.

1.3 First Shaders: Colors and Syntax

1.3.1 The mainImage Function

In Shadertoy, your shader is a function called `mainImage`:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // your code here
}
```

This function is called once per pixel, every frame. The inputs and outputs:

- `fragCoord` — the pixel coordinates, passed *in* to your function
- `fragColor` — the RGBA color, which you write *out*

The `in` and `out` keywords are explicit about data flow: `fragCoord` is read-only input, `fragColor` is where you write your result. The function returns `void` because the output goes through `fragColor`, not a return value.

1.3.2 Hello World: A Solid Color

The simplest shader: make every pixel red.

The `vec4(1.0, 0.0, 0.0, 1.0)` constructs a 4-component vector: red=1, green=0, blue=0, alpha=1. Every pixel receives the same color, so the screen fills with red.

1.3.3 GLSL Syntax Essentials

GLSL (OpenGL Shading Language) will feel familiar if you've seen C-like syntax, but a few things are worth noting upfront.

Semicolons are required at the end of each statement.

Floats must include a decimal point. Write `1.0`, not `1`. The integer `1` and the float `1.0` are different types, and GLSL is strict about this.

Vector types are built in: `vec2`, `vec3`, `vec4` for 2, 3, and 4 component vectors. Construct them with:

```
vec2 p = vec2(3.0, 4.0);
vec3 color = vec3(1.0, 0.5, 0.0);
vec4 rgba = vec4(1.0, 0.0, 0.0, 1.0);
```

Arithmetic is component-wise. Adding two vectors adds their components:

```
vec2(1.0, 2.0) + vec2(3.0, 4.0)  // = vec2(4.0, 6.0)
```

Scalar-vector operations apply the scalar to each component:

```
2.0 * vec2(1.0, 3.0)  // = vec2(2.0, 6.0)
```

Accessing components uses `.x`, `.y`, `.z`, `.w`:

```
vec2 p = vec2(3.0, 4.0);
float a = p.x;  // 3.0
float b = p.y;  // 4.0
```

For colors, `.r`, `.g`, `.b`, `.a` are synonyms—`color.r` is the same as `color.x`.

Common math functions work as expected: `sin`, `cos`, `abs`, `min`, `max`, `sqrt`, `pow`. These operate on floats, and apply component-wise to vectors:

```
sin(vec2(0.0, 3.14159))  // = vec2(0.0, ~0.0)
```

For loops work as you'd expect:

```
for (int i = 0; i < 5; i++) {
    // body executes with i = 0, 1, 2, 3, 4
}
```

The loop variable is an `int`. Note that some older GPUs require the loop bounds to be constants known at compile time—you can't always loop up to a variable. We'll use loops extensively starting tomorrow.

1.3.4 Uniforms: Global Inputs

Shadertoy provides *uniforms*—global values that are constant across all pixels. Unlike `fragCoord`, which takes a different value at each pixel, a uniform has the same value everywhere. They're how external information (time, screen size, mouse position) gets into your shader.

Uniform	Type	Description
`iResolution`	`vec3`	Viewport size: `(width, height, pixel_aspect_ratio)`
`iTime`	`float`	Seconds since the shader started
`iMouse`	`vec4`	Mouse position and click state

We'll use `iResolution` constantly (for coordinate transforms) and `iTime` for animation.

1.3.5 Animation: Using iTime

Let's make the red channel pulse:

Since `sin(iTime)` oscillates between -1 and 1, the expression `0.5 + 0.5 * sin(iTime)` oscillates between 0 and 1. The screen pulses from black to red.

This is our first *animated* shader—the output depends on time.

1.4 Coordinate Systems

1.4.1 Pixel Coordinates

The input `fragCoord` gives the pixel coordinates of the current pixel. The coordinate system:

- Origin at the **bottom-left** corner
- `fragCoord.x` increases to the right
- `fragCoord.y` increases upward

- Ranges from $(0, 0)$ to (X, Y) where $X \times Y$ is the screen resolution

This is workable, but inconvenient for mathematics. We'd prefer coordinates centered at the origin with a reasonable scale. Let's build up a transformation step by step.

1.4.2 Step 1: Normalize to $[0, 1]^2$

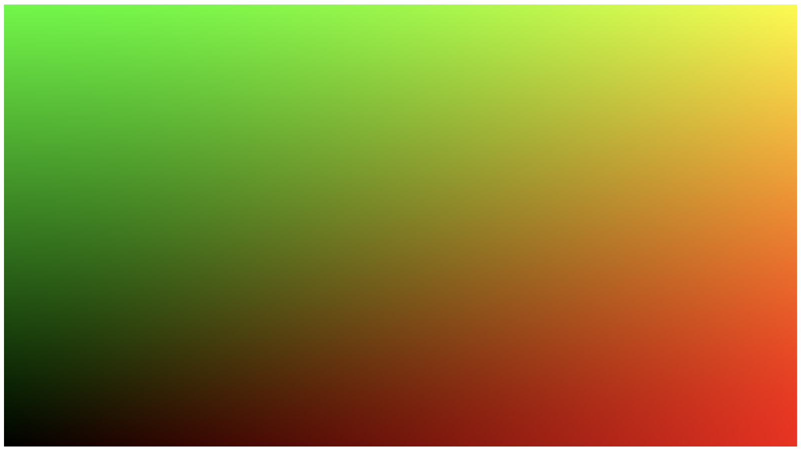
Divide by the resolution to map pixel coordinates to the unit square:

```
vec2 uv = fragCoord / iResolution.xy;
```

Now `uv` ranges from $(0, 0)$ at bottom-left to $(1, 1)$ at top-right.

Since both coordinates are in $[0, 1]$, we can visualize them directly as color:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 uv = fragCoord / iResolution.xy;
    fragColor = vec4(uv.x, uv.y, 0.0, 1.0);
}
```



Black at bottom-left (0,0), red at bottom-right (1,0), green at top-left (0,1), yellow at top-right (1,1).

1.4.3 Step 2: Center the Origin

Subtract $(0.5, 0.5)$ to center the origin:

```
uv = uv - vec2(0.5, 0.5);
```

Now `uv` ranges from $(-0.5, -0.5)$ to $(0.5, 0.5)$, with $(0, 0)$ at the screen center.

1.4.4 Step 3: Aspect Ratio Correction

We've mapped a rectangle of pixels ($X \times Y$) to the square $[-0.5, 0.5]^2$. This is an affine transformation, not a similarity—it distorts shapes. A circle in our coordinates would render as an ellipse on screen.

To fix this, we scale the x-coordinate by the aspect ratio:

```
uv.x *= iResolution.x / iResolution.y;
```

Now a circle in our coordinates appears as a circle on screen. (When we draw shapes later, try commenting out this line to see the distortion.)

1.4.5 Step 4: Scale to a Useful Range

Finally, scale to a convenient window:

```
vec2 p = uv * 4.0;
```

With a scale factor of 4, our coordinates range roughly from -2 to 2—a good default for visualizing mathematical objects.

1.4.6 The Standard Boilerplate

Putting it together, here's the coordinate setup we'll use throughout the course:

```
vec2 uv = fragCoord / iResolution.xy;    // normalize to [0,1]
uv = uv - vec2(0.5, 0.5);                // center origin
uv.x *= iResolution.x / iResolution.y;   // aspect correction
vec2 p = uv * 4.0;                       // scale
```

From here on, `p` is our mathematical coordinate, centered at the origin, aspect-corrected, with a reasonable range.

1.5 Drawing with Distance

So far we've colored every pixel the same, or colored based on position as a gradient. Now we want to *draw*: to render a shape on screen.

What does it mean to draw a shape? For a simple filled region, we need a rule that tells us, for each pixel: are you inside the shape or not? When inside, we do one thing (say, color yellow). When outside, we do another (color blue). The boundary of the shape is where we switch.

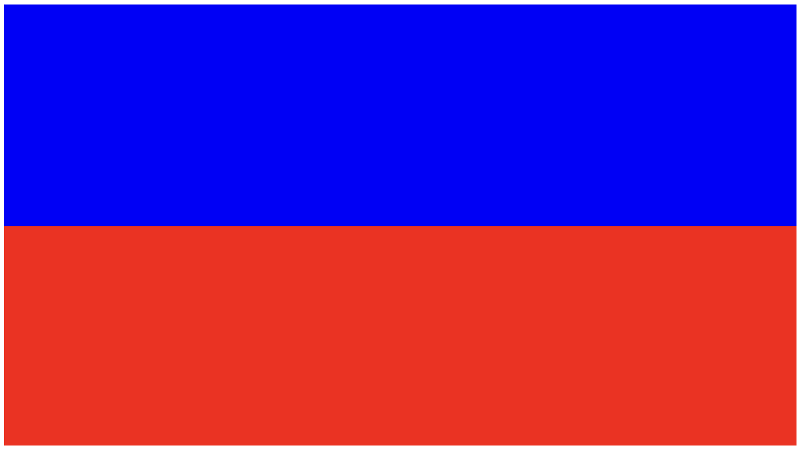
1.5.1 Half-Planes

The simplest shape is a half-plane. Consider the rule: is the y-coordinate greater than 0? This divides the plane into two regions—above and below the x-axis.

```
float L = p.y;

vec3 color;
if (L < 0.0) {
    color = vec3(1.0, 0.0, 0.0);  // red below
} else {
    color = vec3(0.0, 0.0, 1.0);  // blue above
}

fragColor = vec4(color, 1.0);
```



To color left versus right instead, use `p.x` in place of `p.y`.

More generally, a line in the plane has the form $ax + by + c = 0$. This divides the plane into two half-planes: where $ax + by + c < 0$ and where $ax + by + c > 0$.

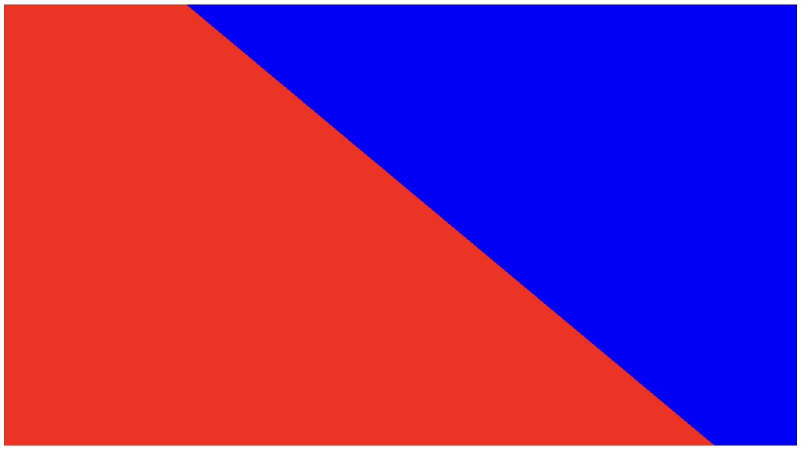
```
float a = 1.0, b = 1.0, c = 0.0;
float L = a * p.x + b * p.y + c;

vec3 color;
if (L < 0.0) {
    color = vec3(1.0, 0.0, 0.0);  // red
} else {
    color = vec3(0.0, 0.0, 1.0);  // blue
}

fragColor = vec4(color, 1.0);
```

Recall that (a, b) is the normal vector to the line, and c is an offset. Since these are just variables, we can animate them to move the line around:

```
float a = cos(iTime);
float b = sin(iTime);
float c = 0.5 * sin(iTime * 0.7);
```



1.5.2 Circles

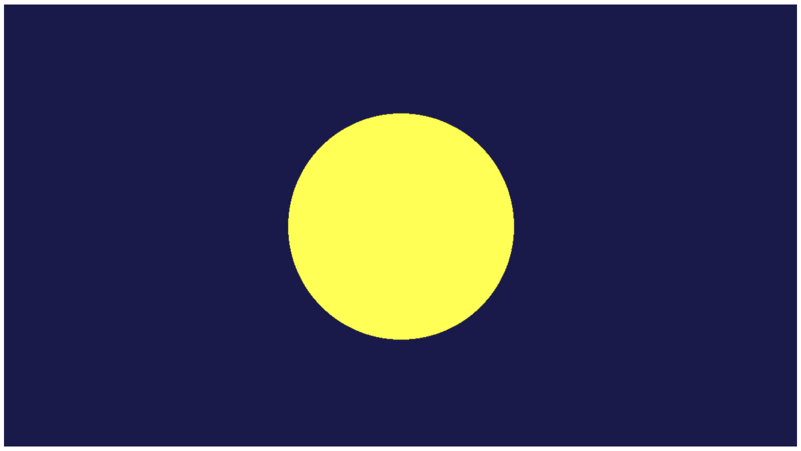
Now consider the function $d(p) = |p|$, the distance from the origin. Geometrically, the graph of this function is a cone—zero at the origin, increasing linearly in all directions.

To draw a filled disk of radius r, we could threshold on $d < r$ versus $d \geq r$. But it's cleaner to define $f(p) = |p| - r$. This function is negative inside the circle (where $d < r$) and positive outside (where $d > r$). The circle itself is the level set $f = 0$.

```
float d = length(p);
float r = 1.0;
float f = d - r;

vec3 color;
if (f < 0.0) {
    color = vec3(1.0, 1.0, 0.0);  // yellow inside
} else {
    color = vec3(0.1, 0.1, 0.3);  // dark blue outside
}

fragColor = vec4(color, 1.0);
```



Try commenting out the aspect ratio correction (`uv.x *= ...`) to see the distortion—the circle becomes an ellipse.

To center the circle at a point c instead of the origin, compute distance from c:

```
vec2 center = vec2(1.0, 0.5);
float d = length(p - center);
```

Since `center` and `r` are variables, you can animate them with `iTime` to create moving, pulsing circles.

1.5.3 Drawing a Ring

Our function $f = d - r$ is negative inside the circle and positive outside. To draw a filled disk, we colored based on the sign of f.

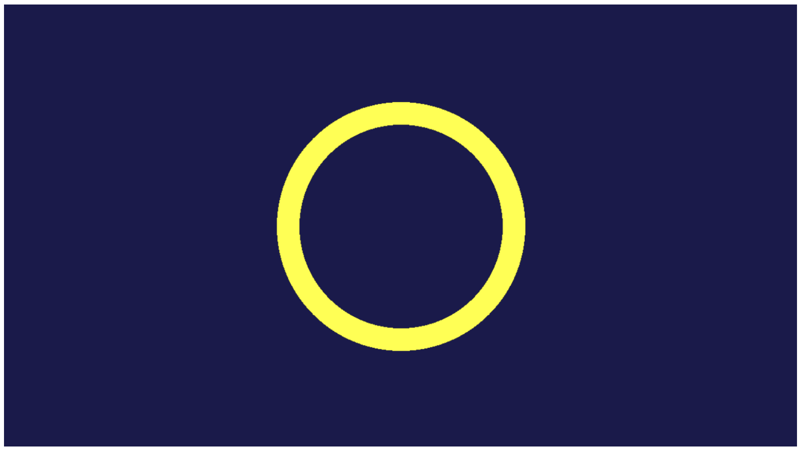
But what if we want just the boundary—a ring of some thickness? We want to color one way when f is small in absolute value (near the circle), and a different way when $|f|$ is large (far from the circle).

So we look at $|f| = |d - r|$ and ask: is this less than some threshold ε, or greater? Equivalently, is $|d - r| - \varepsilon$ negative or positive?

```
float d = length(p);
float r = 1.0;
float eps = 0.1;
float f = abs(d - r) - eps;

vec3 color;
if (f < 0.0) {
    color = vec3(1.0, 1.0, 1.0);  // white ring
} else {
    color = vec3(0.1, 0.1, 0.3);  // dark background
}

fragColor = vec4(color, 1.0);
```



1.6 Implicit Curves

We've drawn circles using the distance function $|p| - r$. But circles are just one example of curves defined by an equation. Any equation $F(x, y) = 0$ defines a curve—the set of points satisfying that equation. We can draw it the same way: threshold on $|F|$.

1.6.1 A First Example: The Parabola

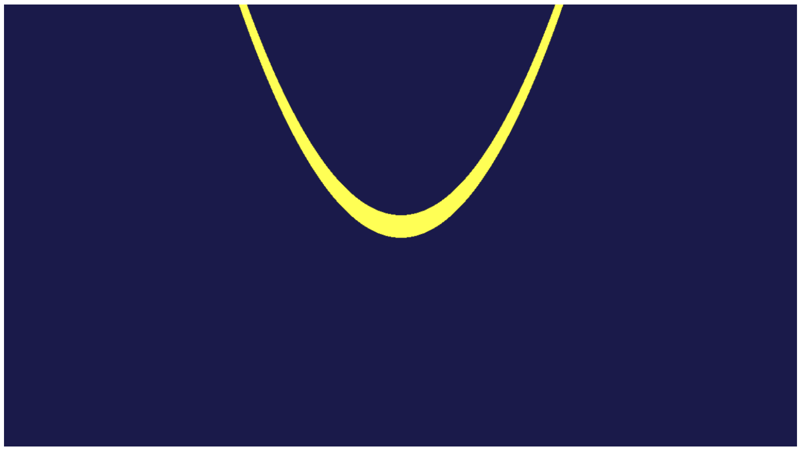
Consider $F(x, y) = y - x^2$. The curve $F = 0$ is the parabola $y = x^2$. Points where $F < 0$ lie below the parabola; points where $F > 0$ lie above.

To draw the curve itself, we color pixels where $|F|$ is small:

```
float F = p.y - p.x * p.x;
float eps = 0.1;

vec3 color;
if (abs(F) < eps) {
    color = vec3(1.0, 1.0, 0.0);  // yellow curve
} else {
    color = vec3(0.1, 0.1, 0.3);  // dark background
}

fragColor = vec4(color, 1.0);
```



1.6.2 More Examples

An ellipse: $F(x, y) = \frac{x^2}{a^2} + \frac{y^2}{b^2} - 1$

```
float a = 2.0, b = 1.0;
float F = (p.x*p.x)/(a*a) + (p.y*p.y)/(b*b) - 1.0;
```

A hyperbola: $F(x, y) = \frac{x^2}{a^2} - \frac{y^2}{b^2} - 1$

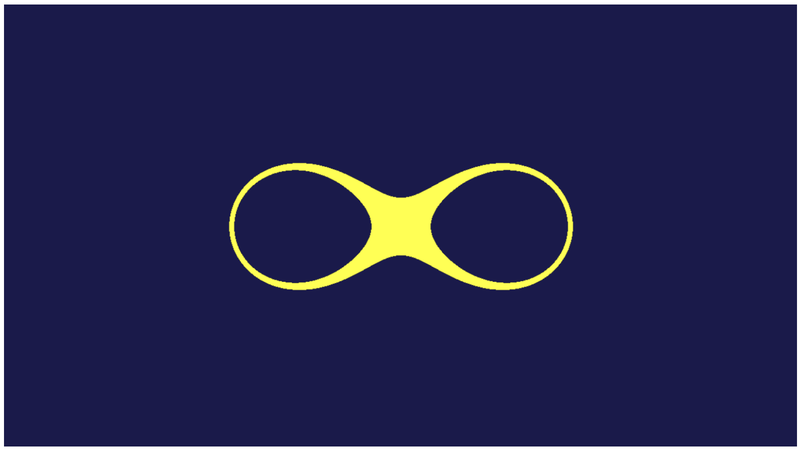
```
float a = 1.0, b = 1.0;
float F = (p.x*p.x)/(a*a) - (p.y*p.y)/(b*b) - 1.0;
```

The lemniscate of Bernoulli: $(x^2 + y^2)^2 = a^2(x^2 - y^2)$, or $F = (x^2 + y^2)^2 - a^2(x^2 - y^2)$

```
float a = 1.5;
float r2 = dot(p, p);  // x² + y²
float F = r2 * r2 - a * a * (p.x * p.x - p.y * p.y);
```

1.6.3 The Thickness Problem

Look carefully at the parabola. The rendered thickness isn't uniform—it's thinner where the curve is steep, thicker where it's flat. The problem gets worse with more complicated curves, especially those with singularities. Here's the lemniscate:



Notice how the thickness blows up near the origin, where the curve crosses itself.

Why does this happen? The set $|F| < \varepsilon$ contains all points within ε of zero *in the F direction*. But F doesn't measure distance to the curve—it's just some function that happens to be zero on the curve. Where $|\nabla F|$ is large, F changes rapidly, so the band $|F| < \varepsilon$ is narrow. Where $|\nabla F|$ is small, F changes slowly, so the band is wide. At the singular point, $\nabla F = 0$, and the band becomes infinitely wide.

1.6.4 Why Circles Worked

For the circle, we used $f(p) = |p| - r$. This is the *signed distance function*: it measures actual geometric distance to the curve. The gradient of a distance function has magnitude 1 everywhere (it points toward or away from the curve at unit rate). So $|f| < \varepsilon$ really does capture points within distance ε, giving uniform thickness.

This is a fact from differential geometry: $|\nabla d| = 1$ for a distance function d. When we use an arbitrary implicit equation $F = 0$, we lose this property.

1.6.5 Gradient Correction

We can fix the non-uniform thickness by dividing by the gradient magnitude. Instead of thresholding $|F| < \varepsilon$, we threshold

$$\frac{|F|}{|\nabla F|} < \varepsilon.$$

This approximates the signed distance to the curve. The intuition: $|F|/|\nabla F|$ estimates how far you'd need to travel (in the direction F changes fastest) to reach the curve.

For the lemniscate, we compute the gradient analytically:

$$\nabla F = \left(4x(x^2 + y^2) - 2a^2x,\ 4y(x^2 + y^2) + 2a^2y\right)$$

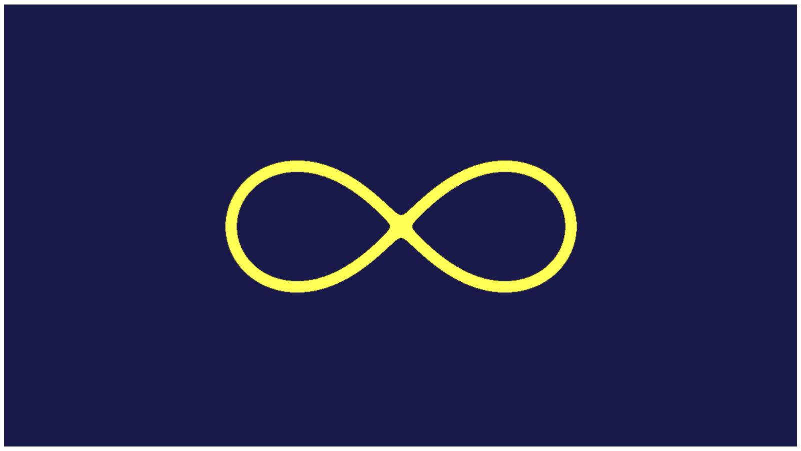
```
float a = 1.5;
float r2 = dot(p, p);
float F = r2 * r2 - a * a * (p.x * p.x - p.y * p.y);

vec2 grad = vec2(
    4.0 * p.x * r2 - 2.0 * a * a * p.x,
    4.0 * p.y * r2 + 2.0 * a * a * p.y
);

float dist = abs(F) / max(length(grad), 0.01);  // avoid division by zero
float eps = 0.05;

vec3 color;
if (dist < eps) {
    color = vec3(1.0, 1.0, 0.0);
} else {
    color = vec3(0.1, 0.1, 0.3);
}

fragColor = vec4(color, 1.0);
```



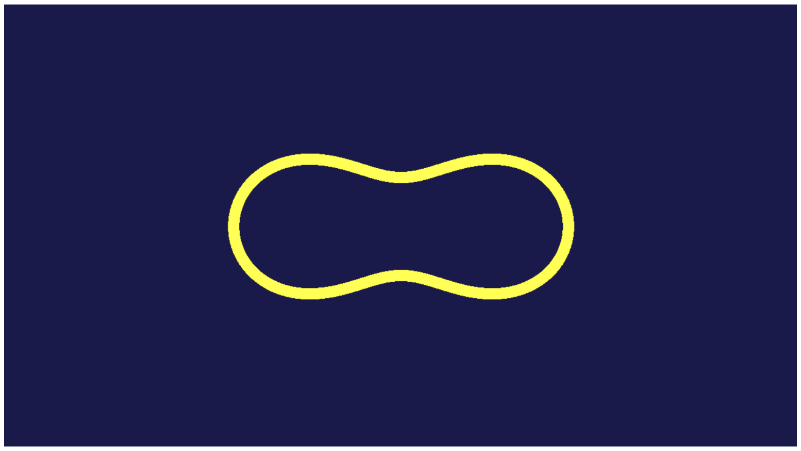
Compare with the naive version above to see the difference in thickness uniformity.

1.6.6 Animated Curve Families

The lemniscate is part of a one-parameter family called the Cassini ovals, defined by the product of distances from two foci being constant:

$$(x^2 + y^2)^2 - 2c^2(x^2 - y^2) = a^4 - c^4$$

As the parameter a varies relative to the fixed focal distance c, the topology changes: two separate loops when $a < c$, a lemniscate when $a = c$, a single oval when $a > c$.



1.7 Interactivity and Abstraction

So far our shaders respond to time (`iTime`) but not to user input. Shadertoy provides `iMouse` for mouse interaction.

1.7.1 The iMouse Uniform

`iMouse` is a `vec4`:

- `iMouse.xy` — current mouse position (in pixels)
- `iMouse.zw` — position where the mouse was last clicked

For now we'll focus on `iMouse.xy`.

1.7.2 Dragging a Circle

Let's draw a circle centered at the mouse position. Since `iMouse.xy` is in pixel coordinates, we need to normalize it the same way we normalize `fragCoord`:

```
// Normalize fragment coordinate
vec2 uv = fragCoord / iResolution.xy;
uv = uv - vec2(0.5, 0.5);
uv.x *= iResolution.x / iResolution.y;
vec2 p = uv * 4.0;

// Normalize mouse coordinate the same way
vec2 mouse = iMouse.xy / iResolution.xy;
mouse = mouse - vec2(0.5, 0.5);
mouse.x *= iResolution.x / iResolution.y;
mouse = mouse * 4.0;

// Circle centered at mouse
float d = length(p - mouse);
float r = 0.5;

vec3 color;
if (d < r) {
    color = vec3(1.0, 0.9, 0.2);  // yellow
} else {
    color = vec3(0.1, 0.1, 0.3);
}

fragColor = vec4(color, 1.0);
```

Click and drag to move the circle.

1.7.3 Writing a Helper Function

We just wrote the same four lines of coordinate normalization twice. This is a sign we should write a function.

A GLSL function declares its return type, then the function name, then its parameters with their types:

```
vec2 normalize_coord(vec2 coord) {
    vec2 uv = coord / iResolution.xy;
    uv = uv - vec2(0.5, 0.5);
    uv.x *= iResolution.x / iResolution.y;
    return uv * 4.0;
}
```

Functions must be defined before they're used, so they go above `mainImage`. Here's the overall structure:

```
vec2 normalize_coord(vec2 coord) {
    // normalization logic here
}

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    vec2 p = normalize_coord(fragCoord);
    vec2 mouse = normalize_coord(iMouse.xy);

    // code using p and mouse
}
```

Now our shader is cleaner, and we won't make mistakes copying the normalization code.

1.7.4 Combining iMouse and iTime: Sun and Earth

Let's make a circle orbit around the mouse position:

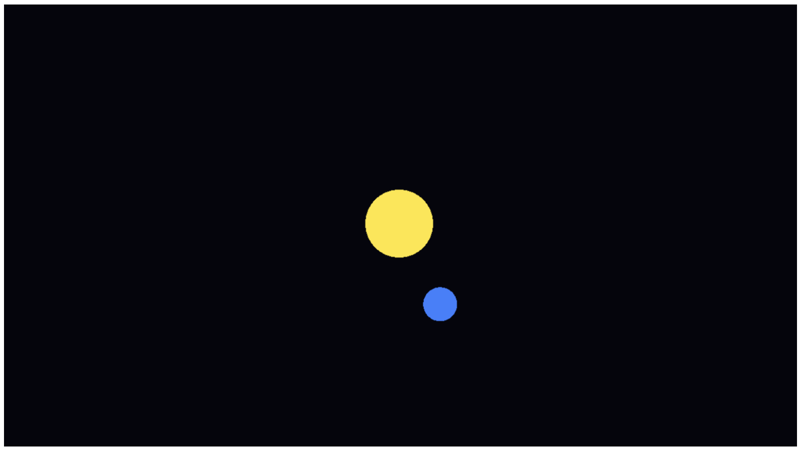
```
vec2 p = normalize_coord(fragCoord);
vec2 sun = normalize_coord(iMouse.xy);

// Earth orbits the sun
float orbit_radius = 0.8;
vec2 earth = sun + orbit_radius * vec2(cos(iTime), sin(iTime));

// Draw sun (larger, yellow)
float d_sun = length(p - sun);
// Draw earth (smaller, blue)
float d_earth = length(p - earth);

vec3 color = vec3(0.02, 0.02, 0.05);  // dark background
if (d_sun < 0.3) {
    color = vec3(1.0, 0.9, 0.2);  // yellow sun
}
if (d_earth < 0.15) {
    color = vec3(0.2, 0.5, 1.0);  // blue earth
}

fragColor = vec4(color, 1.0);
```



Drag to move the sun; the earth follows in orbit. (Exercise: add a moon orbiting the earth!)

1.7.5 Mouse as Parameter

The mouse doesn't have to control position—it can control any parameter. A useful pattern: map `iMouse.x` to a parameter range and drag across the screen to explore a family of curves.

The folium of Descartes is the curve $x^3 + y^3 = 3axy$. We can explore its level sets by drawing $x^3 + y^3 - 3axy = c$ for different values of c:

```
vec2 p = normalize_coord(fragCoord);

// Fixed parameter a
float a = 1.5;

// Map mouse x to level set value c in [-2, 2]
float c = mix(-2.0, 2.0, iMouse.x / iResolution.x);

// Folium of Descartes: x³ + y³ - 3axy = c
float F = p.x*p.x*p.x + p.y*p.y*p.y - 3.0*a*p.x*p.y - c;

// Gradient: ∇F = (3x² - 3ay, 3y² - 3ax)
vec2 grad = vec2(3.0*p.x*p.x - 3.0*a*p.y, 3.0*p.y*p.y - 3.0*a*p.x);
float dist = abs(F) / max(length(grad), 0.01);

vec3 color;
if (dist < 0.05) {
    color = vec3(1.0, 1.0, 0.0);
} else {
    color = vec3(0.1, 0.1, 0.3);
}

fragColor = vec4(color, 1.0);
```

Drag left and right to sweep through the level sets and watch the curve topology change.

1.8 Exercises

Homework is organized into four types:

Checkpoints — Short exercises to verify you understood the lecture material. Required for anyone new to shader programming.

Explorations — Open-ended problems that extend the lecture topics. Pick the ones that interest you. If you can do several of these, you're right on track with the course.

Challenges — Problems that may require learning new concepts beyond what was covered in lecture. Attempt these if you skipped the checkpoints and found an exploration or two too easy.

Project — An extended project for someone familiar with shader basics, to make an artwork.

1.8.1 Checkpoints

C1. Solid Colors. Modify the red screen shader to display: (a) green, (b) cyan, (c) a color of your choice using all three RGB channels.

C2. Vertical Split. Modify the half-plane shader to divide the screen into left (red) and right (blue) instead of top and bottom.

C3. Off-Center Circle. Draw a filled circle of radius 0.5 centered at the point $(1, 1)$ instead of the origin.

C4. Pulsing Circle. Make a circle whose radius oscillates between 0.5 and 1.5 over time using `iTime`.

C5. Ring Thickness. Draw a ring (circle outline) centered at the origin. Experiment with different values of `eps` to understand how it controls thickness.

1.8.2 Explorations

E1. Concentric Rings. Draw several concentric rings (circles of different radii, all centered at the origin). Can you color alternate rings differently?

E2. Moon Orbit. Extend the sun-earth shader to add a moon that orbits the earth. The moon should be smaller than the earth and orbit faster.

E3. Your Favorite Curve. Pick an implicit curve from your mathematical experience (or find one online) and render it. Some suggestions: the cardioid $(x^2 + y^2 - ax)^2 = a^2(x^2 + y^2)$, the astroid $x^{2/3} + y^{2/3} = a^{2/3}$, or a rose curve in implicit form. Apply gradient correction for uniform thickness.

E4. Curve Explorer. Take any one-parameter family of curves and build a mouse-controlled explorer (like the folium example). Map `iMouse.x` to the parameter and drag to explore the family.

E5. Two Circles. Draw two filled circles at different positions. What happens when they overlap? Can you make one "in front of" the other? Can you make the intersection a different color, like a Venn diagram?

1.8.3 Challenges

H1. Parabola Graphing Calculator. Build an interactive graphing calculator for the parabola $y = ax^2 + bx + c$. Requirements: - Draw coordinate axes (the lines $x = 0$ and $y = 0$) - Draw the parabola using implicit curve techniques - Find the roots (where $y = 0$) and draw small circles around them - Use mouse position to control two of the coefficients (e.g., a and b, with c fixed, or b and c with a fixed)

As you drag the mouse, the parabola should reshape and the root indicators should move (or appear/disappear as roots become real or complex).

H2. Elliptic Curve Explorer. Elliptic curves in Weierstrass form are $y^2 = x^3 + ax + b$. Build a shader where the mouse position controls (a, b). Use gradient correction for uniform thickness. The *discriminant* $\Delta = 4a^3 + 27b^2$ determines whether the curve is smooth ($\Delta \neq 0$) or singular ($\Delta = 0$). Can you display the current value of Δ somehow, or change the curve's color when it becomes singular?

H3. Signed Distance Functions. For a filled circle, $f(p) = |p| - r$ is the *signed* distance function: negative inside, positive outside, with $|f|$ giving the actual distance to the boundary. What is the signed distance function for a half-plane? For an axis-aligned rectangle? Implement both and draw them with uniform-thickness boundaries. Note: when you have the true signed distance function, you don't need the gradient correction trick—that's the payoff for computing the right thing from the start!

H4. Smooth Blending. When two circles overlap, we currently just draw one on top of the other. Research *smooth minimum* functions (e.g., `smin`) that blend distance fields smoothly. Draw two circles that "melt together" where they meet.

H5. Inversion. Circle inversion is the map $p \mapsto p/|p|^2$. Apply this transformation to your coordinate p before drawing a shape. What happens to a line? What happens to a circle not passing through the origin? Experiment with different shapes.

1.8.4 Project: Grid Patterns

This extended project introduces a powerful technique—using modular arithmetic to repeat patterns across the plane. We'll build up the machinery carefully, since we'll use it again in Day 2 to create grids of Julia sets.

1.8.4.1 Part 1: Setting Up a Grid of Square Cells

We want to tile the screen with square cells—say, 4 cells across. The challenge: the screen isn't square, so we need to handle the aspect ratio.

Let's say we want `N` columns of cells. Each cell has width $L = \text{screen\_width}/N$ in pixels, and since cells are square, height L as well. The number of rows depends on the screen's aspect ratio.

Working in our normalized coordinates (after aspect correction), the screen spans roughly $[-2 \cdot \text{aspect}, 2 \cdot \text{aspect}]$ in x and $[-2, 2]$ in y. If we want cells of side length L in these coordinates:

```glsl
float aspect = iResolution.x / iResolution.y;
float N = 5.0;  // number of columns
float L = (4.0 * aspect) / N;  // cell size in our coordinate system
```

Now each cell is an $L \times L$ square.

1.8.4.2 Part 2: Cell Coordinates and Identity

For each pixel, we want two things:

1. **Which cell are we in?** Integer coordinates (i, j) identifying the cell.
2. **Where in the cell are we?** Local coordinates ranging from $-L/2$ to $L/2$, with $(0, 0)$ at the cell center.

```glsl
vec2 cell_id = floor(p / L);
vec2 cell_p = mod(p + vec2(L/2.0, L/2.0), L) - vec2(L/2.0, L/2.0);
```

The `cell_id` tells us which cell; the `cell_p` gives local coordinates within that cell.

If we want local coordinates normalized to $[-1, 1]$ (useful for drawing things at a standard scale), we can rescale:

```glsl
vec2 local = cell_p / (L / 2.0);  // now in [-1, 1] x [-1, 1]
```

This is exactly the setup we'll need for Day 2, where each cell will contain a Julia set with its own coordinate system.

1.8.4.3 Part 3: Drawing in Each Cell

Now draw something using the local coordinates. A filled circle at the center of each cell:

```
float d = length(cell_p);
float r = L * 0.4;  // radius relative to cell size

vec3 color;
if (d < r) {
    color = vec3(1.0, 1.0, 0.0);
} else {
    color = vec3(0.1, 0.1, 0.3);
}
```

Try changing `N` to get more or fewer columns. The cells stay square regardless of screen shape.

1.8.4.4 Part 4: Varying by Cell

The `cell_id` lets each cell behave differently. Some ideas:

Checkerboard background:

```
float checker = mod(cell_id.x + cell_id.y, 2.0);
vec3 bg = mix(vec3(0.2, 0.2, 0.3), vec3(0.3, 0.2, 0.2), checker);
```

Radius varying by cell:

```
float r = L * (0.2 + 0.15 * mod(cell_id.x + cell_id.y, 3.0));
```

Wave animation:

```
float cell_dist = length(cell_id);
float r = L * (0.3 + 0.1 * sin(iTime * 2.0 - cell_dist * 0.5));
```

1.8.4.5 Part 5: Design Challenge

Design a grid-based pattern that you find visually interesting. Some directions:

Connecting shapes: Draw shapes that connect across cell boundaries. Quarter-circles in each corner create a continuous network. What implicit curves tile seamlessly?

Alternating motifs: Use `cell_id` to alternate between different shapes—circles in some cells, rings in others, or different orientations.

Color fields: Map `cell_id` to colors using distance from origin, stripes, or a palette.

Phase shifts: Animate cells with different phase offsets to create waves or ripples.

Using local coordinates: Draw something more complex in each cell using the $[-1, 1]$ local coordinate system—perhaps a small implicit curve, or a pattern that changes based on `cell_id`.

The goal is to produce an image you'd be happy to hang on a wall.

1.8.5 Project: Fourier Epicycles

This project builds a visualization of Fourier series using epicycles—circles whose centers sit on the circumferences of other circles. This is how Ptolemy modeled planetary motion, and it turns out to be exactly how Fourier series work geometrically.

1.8.5.1 Part 1: The Idea

Any periodic function can be written as a sum of sines and cosines. Geometrically, $\sin(n\omega t)$ and $\cos(n\omega t)$ describe a point moving around a circle of frequency $n\omega$. Adding these components corresponds to stacking circles: each circle's center rides on the previous circle's edge.

For example, the square wave has Fourier series:

$$f(t) = \sum_{n=1,3,5,\ldots} \frac{1}{n} \sin(n\omega t)$$

This means circles with: - Radii: $1, \frac{1}{3}, \frac{1}{5}, \frac{1}{7}, \ldots$ - Frequencies: $\omega, 3\omega, 5\omega, 7\omega, \ldots$

The more terms we add, the closer the final point's y-coordinate approximates a square wave.

1.8.5.2 Part 2: Drawing Circles

Start by drawing a chain of circles. Each circle is centered at the current position, and the next position is computed by moving along the circle:

```
vec2 pos = vec2(0.0, 0.0);  // start at origin

for (int i = 0; i < N; i++) {
    int n = 2 * i + 1;  // 1, 3, 5, 7, ...
    float r = scale / float(n);
    float freq = float(n) * omega;

    // Draw circle at current position
    float d_circle = abs(length(p - pos) - r);
    if (d_circle < 0.02) {
        // color the circle
    }

    // Move to next position
    pos = pos + r * vec2(cos(freq * iTime), sin(freq * iTime));
}

// Draw final point
float d_point = length(p - pos);
if (d_point < 0.08) {
    // bright color
}
```

Try this with `N = 1`, then `N = 3`, then `N = 7`. Watch how more circles create more complex motion.

1.8.5.3 Part 3: The Line Segment SDF

To draw the arms connecting circle centers, we need the signed distance function for a line segment. Given endpoints a and b, the distance from point p to the segment is:

```
float sd_segment(vec2 p, vec2 a, vec2 b) {
    vec2 pa = p - a;
    vec2 ba = b - a;
    float t = clamp(dot(pa, ba) / dot(ba, ba), 0.0, 1.0);
    return length(pa - ba * t);
}
```

The math: we project $p - a$ onto the line direction $b - a$, clamp to $[0, 1]$ to stay within the segment, then measure the distance to that closest point.

1.8.5.4 Part 4: Connecting the Arms

Now modify your loop to also draw line segments:

```
vec2 pos = vec2(0.0, 0.0);

for (int i = 0; i < N; i++) {
    int n = 2 * i + 1;
    float r = scale / float(n);
    float freq = float(n) * omega;

    vec2 next_pos = pos + r * vec2(cos(freq * iTime), sin(freq * iTime));

    // Draw circle
    float d_circle = abs(length(p - pos) - r);
    if (d_circle < 0.02) {
        // faint circle color
    }

    // Draw arm from pos to next_pos
    float d_arm = sd_segment(p, pos, next_pos);
    if (d_arm < 0.015) {
        // arm color
    }

    pos = next_pos;
}
```

1.8.5.5 Part 5: Polish and Explore

Now make it beautiful:

Fading circles: Later circles are smaller and less important. Fade their brightness:

```
float fade = 1.0 - float(i) / float(N);
```

Color variation: Color circles differently based on their index, or based on their frequency.

Different waves: The square wave uses odd harmonics with $1/n$ coefficients. Try: - Triangle wave: odd harmonics with $1/n^2$ coefficients (alternating signs) - Sawtooth wave: all harmonics with $1/n$ coefficients

Mouse control: Map `iMouse.x` to the number of terms, so dragging adds or removes circles.

The goal: create a mesmerizing animation that reveals the geometry hidden inside Fourier series.

2 Day 2: Complex Dynamics and Circle Inversion

2.1 Overview

By the end of today, you'll understand how simple iteration rules generate infinite complexity—and have the tools to render it in real time.

Missing Demo

Shader demo `day2/mandelbrot-zoom` not found.

A zoom into the Mandelbrot set, colored by escape time. The entire structure emerges from iterating one formula: $z \mapsto z^2 + c$.

Today we explore two kinds of iterative systems:

1. **Complex dynamics**: Iterating holomorphic maps gives us the Mandelbrot set, Julia sets, and their cousins
2. **Circle inversion**: Iterating geometric transformations gives us the Apollonian gasket

Both share the same GPU-friendly structure: each pixel asks "what happens when I iterate from here?" No pixel depends on any other—perfect for parallel computation.

Along the way, we'll learn to implement complex arithmetic in GLSL and organize geometric data using structs.

2.2 Complex Numbers in GLSL

The complex numbers $\mathbb{C}$ are the plane equipped with a multiplication operation. Today we implement that algebra in GLSL.

2.2.1 Representation

A complex number $z = a + bi$ is naturally represented as a 2D vector:

```glsl
vec2 z = vec2(a, b);  // represents a + bi
```

We'll consistently use the convention that `z.x` is the real part and `z.y` is the imaginary part.

2.2.2 Arithmetic

Addition of complex numbers is componentwise—exactly what GLSL's built-in `+` does for vectors. No helper function needed.

Multiplication is more interesting. In GLSL, the `*` operator on vectors is componentwise: `vec2(a,b) * vec2(c,d)` gives `vec2(a*c, b*d)`. This is *not* complex multiplication! We need to implement the correct formula ourselves.

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

```
vec2 cmul(vec2 z, vec2 w) {
    return vec2(
        z.x * w.x - z.y * w.y,
        z.x * w.y + z.y * w.x
    );
}
```

This is the FOIL pattern with $i^2 = -1$ giving the minus sign in the real part.

2.2.3 Magnitude

The magnitude of $z = a + bi$ is the distance from the origin:

$$|z| = \sqrt{a^2 + b^2}$$

We can implement this directly:

```
float cabs(vec2 z) {
    return sqrt(z.x * z.x + z.y * z.y);
}
```

But consider: we'll often have conditions like "is $|z|$ bigger than 2?" rather than needing the actual magnitude. In these cases, we can check $|z|^2 > 4$ instead of $|z| > 2$—same answer, but no square root. When you're doing this check millions of times per frame (once per pixel, 60 frames per second), avoiding unnecessary square roots adds up.

So we define the squared magnitude:

```
float cabs2(vec2 z) {
    return z.x * z.x + z.y * z.y;
}
```

If we want to be even more efficient, we can use GLSL's built-in dot product, which computes exactly this sum of products:

```
float cabs2(vec2 z) {
    return dot(z, z);  // a² + b²
}
```

2.3 The Mandelbrot Set

The Mandelbrot set is perhaps the most iconic fractal—its shape is instantly recognizable, and its discovery in 1980 helped launch the era of computer-generated mathematical visualization. The definition is remarkably simple.

2.3.1 Definition

Fix a complex number c. Starting from $z_0 = 0$, define a sequence by iterating:

$$z_{n+1} = z_n^2 + c$$

For some values of c, this sequence stays bounded forever. For others, it escapes to infinity. The **Mandelbrot set** $\mathscr{M}$ is the set of all c for which the sequence remains bounded.

That's it! One quadratic formula, iterated. The intricate structure of the Mandelbrot set emerges entirely from this simple rule.

2.3.2 Rendering Strategy

To draw the Mandelbrot set, we test each pixel: is this value of c in $\mathscr{M}$ or not?

This is exactly the kind of question shaders excel at. Each pixel performs its own independent calculation—no pixel needs information from any other pixel. The entire image can be computed in parallel, with thousands of GPU cores each testing their own value of c simultaneously.

But there's a problem: the definition involves iterating "forever" and checking if the sequence "stays bounded." We can't iterate infinitely, and we can't wait forever to decide. We need a practical criterion for when to stop.

2.3.3 The Escape Radius

We need two facts that make efficient rendering possible.

Fact 1. The Mandelbrot set is contained in the disk of radius 2. That is, if $|c| > 2$, then $c \notin \mathscr{M}$.

Proof. [TODO] □

This tells us what region to display: we only need to look at $|c| \leq 2$.

Fact 2. If $|c| \leq 2$ and $|z_n| > 2$ for some n, then the orbit escapes to infinity (so $c \notin \mathscr{M}$).

Proof. [TODO] □

This gives us a stopping criterion: once $|z_n| > 2$, we know c is not in the Mandelbrot set. We don't need to keep iterating.

Together, these facts justify the **escape-time algorithm**: iterate until either $|z_n| > 2$ (escaped, not in $\mathscr{M}$) or we hit a maximum iteration count (probably in $\mathscr{M}$).

2.3.4 The Escape-Time Algorithm

These two facts give us our algorithm:

1. For each pixel, let c be the corresponding complex number
2. Start with $z = 0$
3. Iterate $z \mapsto z^2 + c$
4. If $|z| > 2$, stop—this point escapes (not in $\mathscr{M}$)
5. If we reach a maximum iteration count without escaping, assume bounded (in $\mathscr{M}$)

The core of this is a loop:

```
vec2 z = vec2(0.0, 0.0);
int max_iter = 100;
int iter;

for (iter = 0; iter < max_iter; iter++) {
    if (cabs2(z) > 4.0) break;  // |z|² > 4 means |z| > 2
    z = cmul(z, z) + c;
}
```

After this loop, `iter` tells us what happened: if `iter == max_iter`, we never escaped (probably in $\mathscr{M}$). Otherwise, we escaped on iteration `iter`.

2.3.5 Binary Coloring

The simplest approach: color points black if they're in the set, white if they escaped.

```
vec3 color;
if (iter == max_iter) {
    color = vec3(0.0);  // In the set: black
} else {
    color = vec3(1.0);  // Escaped: white
}
```

2.3.6 Full Implementation

Putting it together with our coordinate setup:

```
vec2 cmul(vec2 z, vec2 w) {
    return vec2(z.x * w.x - z.y * w.y, z.x * w.y + z.y * w.x);
}

float cabs2(vec2 z) {
    return dot(z, z);
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup
    vec2 uv = fragCoord / iResolution.xy;
    uv = uv - vec2(0.5, 0.5);
    uv.x *= iResolution.x / iResolution.y;

    // Scale and center to show the Mandelbrot set
    // By Fact 1, the set lies in |c| ≤ 2
    vec2 c = uv * 4.0;
    c.x -= 0.5;  // shift left to center the interesting part

    // Mandelbrot iteration
    vec2 z = vec2(0.0, 0.0);
    int max_iter = 100;
    int iter;

    for (iter = 0; iter < max_iter; iter++) {
        if (cabs2(z) > 4.0) break;
        z = cmul(z, z) + c;
    }

    // Binary coloring
    vec3 color;
    if (iter == max_iter) {
        color = vec3(0.0);  // In the set: black
    } else {
        color = vec3(1.0);  // Escaped: white
    }

    fragColor = vec4(color, 1.0);
}
```

Missing Demo

Shader demo `day2/mandelbrot-bw` not found.

There it is—the Mandelbrot set in black and white!

2.3.7 Coloring by Iteration Count

Black and white shows the set, but we're throwing away information. The number of iterations before escape tells us how "close" a point is to the boundary—points that escape after 5 iterations are different from points that escape after 50.

Let's use `iter` to create a gradient:

```
vec3 color;
if (iter == max_iter) {
    color = vec3(0.0);
} else {
    float t = float(iter) / float(max_iter);
    color = vec3(t);
}
```

Missing Demo

Shader demo `day2/mandelbrot-gray` not found.

Now we see structure! The boundary reveals intricate detail—tendrils, spirals, bulbs. Points near the boundary take many iterations to escape (bright), while points far away escape quickly (dark).

2.3.8 Color Palettes

Grayscale works, but we can do better. A common technique uses cosines to create smooth, cycling color palettes:

```
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.00, 0.33, 0.67);
    return a + b * cos(6.28318 * (c * t + d));
}
```

The parameters `a`, `b`, `c`, `d` control the palette's character. The vector `d` shifts the phase of each color channel, creating different hues. Try `d = vec3(0.00, 0.10, 0.20)` for blues and purples, or `d = vec3(0.30, 0.20, 0.20)` for warmer tones.

Missing Demo

Shader demo `day2/mandelbrot-color` not found.

The color bands correspond to iteration counts—regions of the same color escaped after the same number of iterations. You'll notice the bands have sharp edges. In the exercises, we'll show you a technique called *smooth coloring* that interpolates between iteration counts, eliminating the banding for even smoother gradients.

2.4 Other Escape-Time Fractals

The Mandelbrot set is one example of an escape-time fractal, but the same algorithm works for many other iterated systems. We just swap out the iteration formula (and sometimes the escape condition). Let's explore a few.

2.4.1 Julia Sets

The Mandelbrot set asks: for which values of c does the orbit of 0 stay bounded? We can ask a different question: for a *fixed* c, which *starting points* z_0 have bounded orbits?

Fix a complex number c. The **filled Julia set** K_c is the set of all starting points z_0 for which the iteration

$$z_{n+1} = z_n^2 + c$$

remains bounded.

Same iteration, different question. For the Mandelbrot set, we vary c and always start at $z_0 = 0$. For a Julia set, we fix c and vary z_0.

The code change is minimal:

```
// Mandelbrot: c varies, z starts at 0
vec2 c = p;
vec2 z = vec2(0.0, 0.0);

// Julia: c is fixed, z starts at pixel position
vec2 c = vec2(-0.7, 0.27015);  // fixed parameter
vec2 z = p;
```

Missing Demo

Shader demo `day2/julia-static` not found.

Different values of c produce dramatically different Julia sets. In the exercises, you'll build an interactive explorer that lets you click anywhere on the Mandelbrot set to see the corresponding Julia set:

Missing Demo

Shader demo `day2/julia-explorer` not found.

2.4.2 The Burning Ship

The **Burning Ship fractal** modifies the Mandelbrot iteration by taking the absolute value of the imaginary part after each squaring:

$$z_{n+1} = (\mathrm{Re}(z_n^2) + i|\mathrm{Im}(z_n^2)|) + c$$

In code:

```
// Inside the loop:
z = cmul(z, z);
z.y = abs(z.y);  // Take absolute value of imaginary part
z = z + c;
```

Missing Demo

Shader demo `day2/burning-ship` not found.

The absolute value breaks the symmetry, creating an asymmetric fractal that (with some imagination) resembles a burning ship.

2.4.3 Higher Powers

The Mandelbrot set uses $z^2 + c$. What about $z^3 + c$? Or $z^4 + c$?

For z^3, we need to implement cubing:

```
vec2 ccube(vec2 z) {
    // z³ = z · z · z
    return cmul(cmul(z, z), z);
}
```

Or we can derive it directly: $(a + bi)^3 = a^3 + 3a^2(bi) + 3a(bi)^2 + (bi)^3 = (a^3 - 3ab^2) + (3a^2b - b^3)i$

```
vec2 ccube(vec2 z) {
    float a = z.x, b = z.y;
    return vec2(
        a*a*a - 3.0*a*b*b,
        3.0*a*a*b - b*b*b
    );
}
```

Then the iteration becomes:

```
z = ccube(z) + c;
```

Missing Demo

Shader demo `day2/mandelbrot-cubic` not found.

Higher powers give higher-order rotational symmetry: $z^3 + c$ has 3-fold symmetry, $z^4 + c$ has 4-fold symmetry, and so on.

2.4.4 The Pattern

All these fractals share the same structure:

1. **Iterate** some function $f(z, c)$
2. **Check escape**: has $|z|$ exceeded some threshold?
3. **Color** based on iteration count

Changing the iteration function changes the fractal. The exercises include more variations for you to try.

2.5 Circle Inversion

We shift gears from complex dynamics to geometric transformations. Circle inversion is a classical operation that turns circles into circles (or lines), preserves angles, and creates beautiful fractal patterns when iterated.

2.5.1 Definition

Inversion in the unit circle maps a point $\mathbf{p}$ to:

$$\text{inv}(\mathbf{p}) = \frac{\mathbf{p}}{|\mathbf{p}|^2}$$

What does this do geometrically? The inverted point lies on the same ray from the origin as $\mathbf{p}$, but at distance $1/r$ instead of r.

- Points inside the unit circle map to points outside
- Points outside map to points inside

- Points on the unit circle stay fixed
- The origin maps to “infinity”

2.5.2 Implementation

```
vec2 invert(vec2 p) {
    return p / dot(p, p);
}
```

2.5.3 Visualizing Inversion

To see what inversion does, let’s draw some shapes and their images. We’ll draw the unit circle (gray), plus a vertical line and a circle (yellow). To make it clearer, we’ll toggle between showing the original shapes and their inversions:

```
vec2 invert(vec2 p) {
    return p / dot(p, p);
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 uv = fragCoord / iResolution.xy;
    uv = uv - vec2(0.5, 0.5);
    uv.x *= iResolution.x / iResolution.y;
    vec2 p = uv * 4.0;

    // Compute the inversion of p
    vec2 p_inv = invert(p);
    // Toggle between original and inverted every second
    float time = fract(iTime * 0.5);
    vec2 q;
    if (time < 0.5) {
        q = p;      // original
    } else {
        q = p_inv;  // inverted
    }

    vec3 color = vec3(0.1, 0.1, 0.15);

    // Draw the unit circle (the inversion circle)
    float d_unit = abs(length(p) - 1.0);
    if (d_unit < 0.02) color = vec3(0.5, 0.5, 0.5);

    // Draw a vertical line at x = 2
    if (abs(q.x - 2.0) < 0.02) color = vec3(1.0, 1.0, 0.0);

    // Draw a horizontal line at y = 1.5
    if (abs(q.y - 1.5) < 0.02) color = vec3(1.0, 1.0, 0.0);

    // Draw a circle centered at (2, 0) with radius 0.5
    float d_circle = abs(length(q - vec2(2.0, 0.0)) - 0.5);
    if (d_circle < 0.02) color = vec3(1.0, 1.0, 0.0);

    fragColor = vec4(color, 1.0);
}
```

Missing Demo

Shader demo `day2/inversion-toggle` not found.

Watch the lines become circles! A line not passing through the origin inverts to a circle that *does* pass through the origin. The circle inverts to another circle (with a different center and radius).

GLSL Shortcuts: mix and step

The toggle logic can be written more compactly using built-in functions:

- `step(edge, x)` returns 0 if `x < edge`, otherwise 1
- `mix(a, b, t)` linearly interpolates: returns `a` when `t = 0`, `b` when `t = 1`

```
float t = step(0.5, fract(iTime * 0.5));
vec2 q = mix(p, p_inv, t);
```

This is a common pattern for toggling or smoothly transitioning between states.

2.5.4 Key Properties

Circle inversion maps circles to circles (or to lines, if the circle passes through the center). It's *conformal*—it preserves angles between curves. And it's *involutive*: applying inversion twice returns to the original point.

2.5.5 Inverting a Grid

For a more dramatic visualization, let's invert a whole grid of lines. The function `mod(q, 0.5)` gives the position of `q` within a repeating 0.5×0.5 cell—when either component is near zero, we're on a grid line:

```
vec2 grid = mod(q, 0.5);
if (grid.x < 0.02 || grid.y < 0.02) color = vec3(1.0, 1.0, 0.0);
```

(If you did the grid project from Day 1, this is familiar!)

We can run the same shader as before, just replacing the individual shapes with this grid:

Missing Demo

Shader demo `day2/inversion-grid` not found.

2.6 Structs

So far our `invert` function only works for the unit circle at the origin. What if we want to invert through a different circle?

2.6.1 General Circle Inversion

For a circle with center $\mathbf{c}$ and radius R, inversion maps a point $\mathbf{p}$ to:

$$\mathrm{inv}(\mathbf{p}) = \mathbf{c} + R^2 \frac{\mathbf{p} - \mathbf{c}}{|\mathbf{p} - \mathbf{c}|^2}$$

The idea is the same as before: the inverted point lies on the ray from $\mathbf{c}$ through $\mathbf{p}$, at distance R^2/r from the center (where $r = |\mathbf{p} - \mathbf{c}|$). When $\mathbf{c} = \mathbf{0}$ and $R = 1$, this reduces to our earlier formula $\mathbf{p}/|\mathbf{p}|^2$.

In code:

```
vec2 invertInCircle(vec2 p, vec2 center, float radius) {
    vec2 d = p - center;
    return center + radius * radius * d / dot(d, d);
}
```

This works, but notice we need to pass *two* things (`center` and `radius`) to describe *one* object (a circle). If we're working with multiple circles, every function call needs `center1, radius1, center2, radius2, ...`—it gets verbose and error-prone.

2.6.2 Defining a Struct

GLSL lets us bundle related data into a **struct**:

```
struct Circle {
    vec2 center;
    float radius;
};
```

Now `Circle` is a type, just like `vec2` or `float`. We can create circles and access their fields:

```
Circle c;
c.center = vec2(1.0, 0.5);
c.radius = 0.7;

// Or initialize directly:
Circle c = Circle(vec2(1.0, 0.5), 0.7);
```

2.6.3 Inversion with Structs

Now our inversion function takes a `Circle`:

```
vec2 invert(vec2 p, Circle c) {
    vec2 d = p - c.center;
    return c.center + c.radius * c.radius * d / dot(d, d);
}
```

Much cleaner! And when we're working with multiple circles, we can pass them around as single objects.

2.6.4 Demo: Moving Circle

Let's animate a circle moving around and watch how the inversion changes:

```
struct Circle {
    vec2 center;
    float radius;
};

vec2 invert(vec2 p, Circle c) {
    vec2 d = p - c.center;
    return c.center + c.radius * c.radius * d / dot(d, d);
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 uv = fragCoord / iResolution.xy;
    uv = uv - vec2(0.5, 0.5);
    uv.x *= iResolution.x / iResolution.y;
    vec2 p = uv * 4.0;

    // Animate the inversion circle
    Circle inv_circle;
    inv_circle.center = vec2(sin(iTime) * 0.5, cos(iTime * 0.7) * 0.5);
    inv_circle.radius = 1.0 + 0.3 * sin(iTime * 1.3);

    // Compute inversion
    vec2 p_inv = invert(p, inv_circle);

    vec3 color = vec3(0.1, 0.1, 0.15);

    // Draw the inversion circle
    float d_inv = abs(length(p - inv_circle.center) - inv_circle.radius);
    if (d_inv < 0.02) color = vec3(0.5, 0.5, 0.5);

    // Draw a grid in the inverted space
    vec2 grid = mod(p_inv, 0.5);
    if (grid.x < 0.02 || grid.y < 0.02) color = vec3(1.0, 1.0, 0.0);

    fragColor = vec4(color, 1.0);
}
```

Missing Demo

Shader demo `day2/inversion-moving` not found.

As the circle moves and breathes, the inverted grid warps and flows.

2.7 The Apollonian Gasket

The Apollonian gasket is a classical fractal arising from circle packing. It's named after Apollonius of Perga (~200 BCE), who studied the problem of finding circles tangent to three given circles.

Missing Demo

Shader demo `day2/apollonian-final` not found.

2.7.1 The Setup

Start with four mutually tangent circles: three "inner" circles that touch each other pairwise, all enclosed by one "outer" circle that touches all three.

Let's define these circles using our `Circle` struct. We'll place three circles of radius *r* with centers forming an equilateral triangle, all tangent to each other and to an outer circle:

```
struct Circle {
    vec2 center;
    float radius;
};

// Three inner circles, mutually tangent, plus outer circle
// For circles of radius r to be mutually tangent, their centers
// must be 2r apart. This forms an equilateral triangle with side 2r.
float r = 1.0;
float triSide = 2.0 * r;  // distance between inner circle centers
float circumradius = triSide / sqrt(3.0);  // distance from origin to centers

Circle c1 = Circle(vec2(0.0, circumradius), r);
Circle c2 = Circle(vec2(-circumradius * 0.866, -circumradius * 0.5), r);
Circle c3 = Circle(vec2(circumradius * 0.866, -circumradius * 0.5), r);
Circle outer = Circle(vec2(0.0, 0.0), circumradius + r);
```

To draw these circles, we need a distance function:

```
float distToCircle(vec2 p, Circle c) {
    return abs(length(p - c.center) - c.radius);
}
```

This returns the distance from `p` to the circle's boundary—zero on the circle, positive elsewhere.

Let's draw our starting configuration:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 uv = fragCoord / iResolution.xy;
    uv = uv - vec2(0.5, 0.5);
    uv.x *= iResolution.x / iResolution.y;
    vec2 p = uv * 6.0;

    // Define circles
    float r = 1.0;
    float triSide = 2.0 * r;
    float circumradius = triSide / sqrt(3.0);

    Circle c1 = Circle(vec2(0.0, circumradius), r);
    Circle c2 = Circle(vec2(-circumradius * 0.866, -circumradius * 0.5), r);
    Circle c3 = Circle(vec2(circumradius * 0.866, -circumradius * 0.5), r);
    Circle outer = Circle(vec2(0.0, 0.0), circumradius + r);

    vec3 color = vec3(0.1, 0.1, 0.15);

    // Draw all four circles
    if (distToCircle(p, c1) < 0.03) color = vec3(1.0, 0.3, 0.3);
    if (distToCircle(p, c2) < 0.03) color = vec3(0.3, 1.0, 0.3);
    if (distToCircle(p, c3) < 0.03) color = vec3(0.3, 0.3, 1.0);
    if (distToCircle(p, outer) < 0.03) color = vec3(1.0, 1.0, 1.0);

    fragColor = vec4(color, 1.0);
}
```

Missing Demo

Shader demo `day2/apollonian-setup` not found.

Three colored circles inside a white outer circle, all mutually tangent.

2.7.2 From Drawing to Iteration

The gaps between circles are curvilinear triangles. The Apollonian gasket fills each gap with a circle tangent to its three neighbors, then fills the new gaps, and so on forever.

Here's the key insight: we can generate this structure by *iterating inversions.* If a point is inside one of the inner circles, invert through that circle—this "pushes" it out. If a point is outside the outer circle, invert through the outer circle—this "pulls" it in.

We need to check: - Is **p** inside circle `c1`, `c2`, or `c3`? (distance from center < radius) - Is **p** outside circle `outer`? (distance from center > radius)

```
bool isInside(vec2 p, Circle c) {
    return length(p - c.center) < c.radius;
}

bool isOutside(vec2 p, Circle c) {
    return length(p - c.center) > c.radius;
}
```

The iteration: keep inverting until the point lands in a "gap" (inside outer, outside all inner circles) or we hit a maximum iteration count.

2.7.3 Full Implementation

```
struct Circle {
    vec2 center;
    float radius;
};

vec2 invert(vec2 p, Circle c) {
    vec2 d = p - c.center;
    return c.center + c.radius * c.radius * d / dot(d, d);
}

float distToCircle(vec2 p, Circle c) {
    return abs(length(p - c.center) - c.radius);
```

```
}

bool isInside(vec2 p, Circle c) {
    return length(p - c.center) < c.radius;
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 uv = fragCoord / iResolution.xy;
    uv = uv - vec2(0.5, 0.5);
    uv.x *= iResolution.x / iResolution.y;
    vec2 p = uv * 6.0;

    // Define circles
    float r = 1.0;
    float triSide = 2.0 * r;
    float circumradius = triSide / sqrt(3.0);

    Circle c1 = Circle(vec2(0.0, circumradius), r);
    Circle c2 = Circle(vec2(-circumradius * 0.866, -circumradius * 0.5), r);
    Circle c3 = Circle(vec2(circumradius * 0.866, -circumradius * 0.5), r);
    Circle outer = Circle(vec2(0.0, 0.0), circumradius + r);

    // Iterate inversions
    int max_iter = 50;
    int iter;

    for (iter = 0; iter < max_iter; iter++) {
        if (isInside(p, c1)) {
            p = invert(p, c1);
        } else if (isInside(p, c2)) {
            p = invert(p, c2);
        } else if (isInside(p, c3)) {
            p = invert(p, c3);
        } else if (!isInside(p, outer)) {
            p = invert(p, outer);
        } else {
            break;  // In the gap—done!
        }
    }

    // Color by iteration count
    float t = float(iter) / float(max_iter);
    vec3 color = palette(t);

    // Draw circle boundaries
    float dMin = min(min(distToCircle(p, c1), distToCircle(p, c2)),
                     min(distToCircle(p, c3), distToCircle(p, outer)));
    if (dMin < 0.02) color = vec3(1.0);
    fragColor = vec4(color, 1.0);
}
```

Missing Demo

Shader demo `day2/apollonian-iterated` not found.

2.7.4 Visualizing the Limit Set

The **limit set** of the Apollonian gasket is the fractal boundary—the set of points that never escape to the fundamental domain, no matter how many iterations. Points near the limit set take many iterations before landing in a gap.

We can emphasize the limit set by adjusting our coloring. Instead of using a color palette, we use a nonlinear function that suppresses low iteration counts (the "background") and brightens high iteration counts (near the fractal):

```
float t = float(iter) / float(max_iter);
vec3 color = 30.0 * vec3(pow(t, 2.0));
```

The squaring suppresses points that escape quickly, while the factor of 30 boosts the brightness of points near the limit set.

Missing Demo

Shader demo `day2/apollonian-final` not found.

2.8 Exercises

2.8.1 Checkpoints

C1. Julia Set. Modify the Mandelbrot shader to render a Julia set. Fix `c = vec2(-0.7, 0.27015)` and initialize `z` from the pixel position instead of zero. Verify you get an intricate, connected fractal.

C2. Cubic Mandelbrot. Change the iteration from $z^2 + c$ to $z^3 + c$. You'll need to implement complex cubing:

```
vec2 ccube(vec2 z) {
    float a = z.x, b = z.y;
    return vec2(a*a*a - 3.0*a*b*b, 3.0*a*a*b - b*b*b);
}
```

What symmetry do you observe?

C3. Apollonian Animation. Animate the Apollonian gasket by letting the maximum iteration count grow with time. Use `int max_iter = int(mod(iTime * 5.0, 50.0)) + 1;` so the fractal "builds up" from the four starting circles to the full gasket, then resets. Watch how each iteration reveals a new layer of circles in the gaps.

C4. Colorize a Fractal. Take any of the black-and-white fractals from today (Mandelbrot, Julia, Burning Ship, or Apollonian) and add color using the cosine palette:

```
vec3 palette(float t) {
    vec3 a = vec3(0.5, 0.5, 0.5);
    vec3 b = vec3(0.5, 0.5, 0.5);
    vec3 c = vec3(1.0, 1.0, 1.0);
    vec3 d = vec3(0.00, 0.33, 0.67);
    return a + b * cos(6.28318 * (c * t + d));
}
```

Experiment with different values of `d` to shift the hues.

C5. Circle Art. Create an image with several circles of different sizes scattered across the screen. Color a pixel based on whether it's inside zero, one, two, or more circles. (Hint: use `isInside` and count how many circles contain each point.)

2.8.2 Explorations

E1. Julia Explorer (Mouse). Make the Julia parameter `c` follow the mouse position. Map `iMouse.xy` to a reasonable region of the complex plane (say, $[-2, 2] \times [-2, 2]$). Drag around and watch the Julia set morph!

E2. Julia Animation. Animate the parameter `c` along a path in the complex plane. Try a circle:

```
float angle = iTime * 0.3;
vec2 c = 0.7885 * vec2(cos(angle), sin(angle));
```

Or trace the boundary of the main cardioid of the Mandelbrot set—every point on this curve gives a Julia set with a parabolic fixed point:

```
vec2 cardioid(float t) {
    vec2 eit = vec2(cos(t), sin(t));
    vec2 z = (vec2(2.0, 0.0) - eit) / 4.0;
    return cmul(eit, z);
}

vec2 c = cardioid(iTime * 0.5);
```

Watch the Julia set continuously transform. What happens when `c` crosses from inside to outside the Mandelbrot set?

E3. Other Escape-Time Fractals. Implement one or more of these variations on the Mandelbrot iteration:

- **Burning Ship**: $z \leftarrow z^2$, then $\text{Im}(z) \leftarrow |\text{Im}(z)|$, then $z \leftarrow z + c$

- **Tricorn** (Mandelbar): $z_{n+1} = \bar{z}_n^2 + c$ where $\bar{z}$ is the complex conjugate
- **Celtic**: $z_{n+1} = |\mathrm{Re}(z_n^2)| + i\,\mathrm{Im}(z_n^2) + c$

For each, figure out how to translate the mathematical formula into GLSL. The escape condition ($|z| > 2$) stays the same.

E4. Smooth Coloring. The iteration count is an integer, so coloring by iteration gives discrete bands. But when a point escapes, it doesn't land exactly on the escape radius—it overshoots. We can use *how much* it overshot to interpolate between iteration counts.

The idea: if $|z_n| > 2$, the "true" fractional iteration where $|z| = 2$ is approximately:

$$n_{\text{smooth}} = n - \frac{\log(\log|z_n| / \log 2)}{\log 2}$$

This comes from the fact that near escape, $|z_{n+1}| \approx |z_n|^2$, so $\log|z|$ roughly doubles each iteration.

Implement this for the Mandelbrot set:

```
if (iter < max_iter) {
    float log_zn = log(cabs2(z)) / 2.0;  // log|z|
    float nu = log(log_zn / log(2.0)) / log(2.0);
    float smooth_iter = float(iter) + 1.0 - nu;
    t = smooth_iter / float(max_iter);
}
```

The bands should disappear, replaced by smooth gradients.

E5. Apollonian Coloring. Modify the Apollonian gasket to color by *which circle* was last inverted through, instead of iteration count. Use a different color for each of the four circles. What patterns emerge?

E6. Apollonian Variations. The Apollonian gasket works with *any* four mutually tangent circles—the symmetric configuration we used is just one example. Descartes' Circle Theorem tells us: if four circles are mutually tangent with curvatures k_1, k_2, k_3, k_4 (where curvature = 1/radius, negative for the outer circle), then:

$$(k_1 + k_2 + k_3 + k_4)^2 = 2(k_1^2 + k_2^2 + k_3^2 + k_4^2)$$

Experiment with different configurations: - Change the radii of the three inner circles (they don't have to be equal!) - Use Descartes' theorem to find an outer circle tangent to three given inner circles - What happens if the circles overlap instead of being tangent?

2.8.3 Challenges

H1. Julia Explorer (Full). Build an interactive tool: display the Mandelbrot set, and wherever the user clicks, show the Julia set for that parameter overlaid or side-by-side. This requires: - Rendering Mandelbrot in one region - Reading click position from `iMouse` - Rendering Julia for that `c` in another region (or blended on top)

Missing Demo

Shader demo `day2/julia-explorer` not found.

H2. Newton Fractal. The Newton fractal comes from applying Newton's method to find roots of a polynomial. For $f(z) = z^3 - 1$, Newton's iteration is:

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)} = z_n - \frac{z_n^3 - 1}{3z_n^2} = \frac{2z_n^3 + 1}{3z_n^2}$$

Iterate this and color based on *which root* the orbit converges to (the three cube roots of unity: 1, $e^{2\pi i/3}$, $e^{4\pi i/3}$). Check convergence by testing if $|z^3 - 1| < \epsilon$.

H3. Higher-Power Mandelbrot. Implement $z^n + c$ for general n. Use the polar form: if $z = re^{i\theta}$, then $z^n = r^n e^{in\theta}$. In code:

```
vec2 cpow(vec2 z, float n) {
    float r = length(z);
    float theta = atan(z.y, z.x);
    float rn = pow(r, n);
    return rn * vec2(cos(n * theta), sin(n * theta));
}
```

Try non-integer values of n! What happens at $n = 2.5$?

2.8.4 Projects

Project 1: Grid of Julia Sets

Create a grid where each cell shows the Julia set for a different value of c. The position in the grid determines c—effectively, each cell samples a point in the Mandelbrot parameter space.

When you zoom out, the overall pattern should reveal the Mandelbrot set: cells with connected Julia sets (solid regions) correspond to $c \in \mathcal{M}$, while cells with dust-like Julia sets correspond to $c \notin \mathcal{M}$.

Use the grid technique from Day 1:

```
float grid_size = 8.0;
vec2 cell_id = floor((p + 2.0) * grid_size / 4.0);
vec2 cell_p = fract((p + 2.0) * grid_size / 4.0) * 4.0 - 2.0;

// c comes from which cell we're in
vec2 c = (cell_id / grid_size) * 4.0 - 2.0;

// z starts from position within the cell
vec2 z = cell_p;
```

Project 2: Orbit Visualization

Instead of just coloring by iteration count, visualize the actual orbit of a point. Make the starting point draggable with the mouse:

- Let z_0 be the mouse position (normalized to the complex plane)
- Fix a parameter c (or let it be controllable too)
- Compute the first N iterates: $z_0, z_1, z_2, \dots, z_N$

- Draw small circles at each iterate position
- Connect consecutive iterates with lines (use the segment SDF from Day 1!)
- Color by iteration index (early iterates one color, later iterates another)

This reveals the dynamics directly: bounded orbits stay in a region and may converge to a cycle, while escaping orbits spiral outward. Drag the starting point around and watch how the orbit changes—you'll see the sensitive dependence on initial conditions that makes chaos!

For Julia sets, fix `c` and drag `z_0`. For the Mandelbrot perspective, fix `z_0 = 0` and drag `c`.