GPU-Accelerated Mathematical Illustration

An Introduction to Shader Programming

Steve Trettel

November 2025

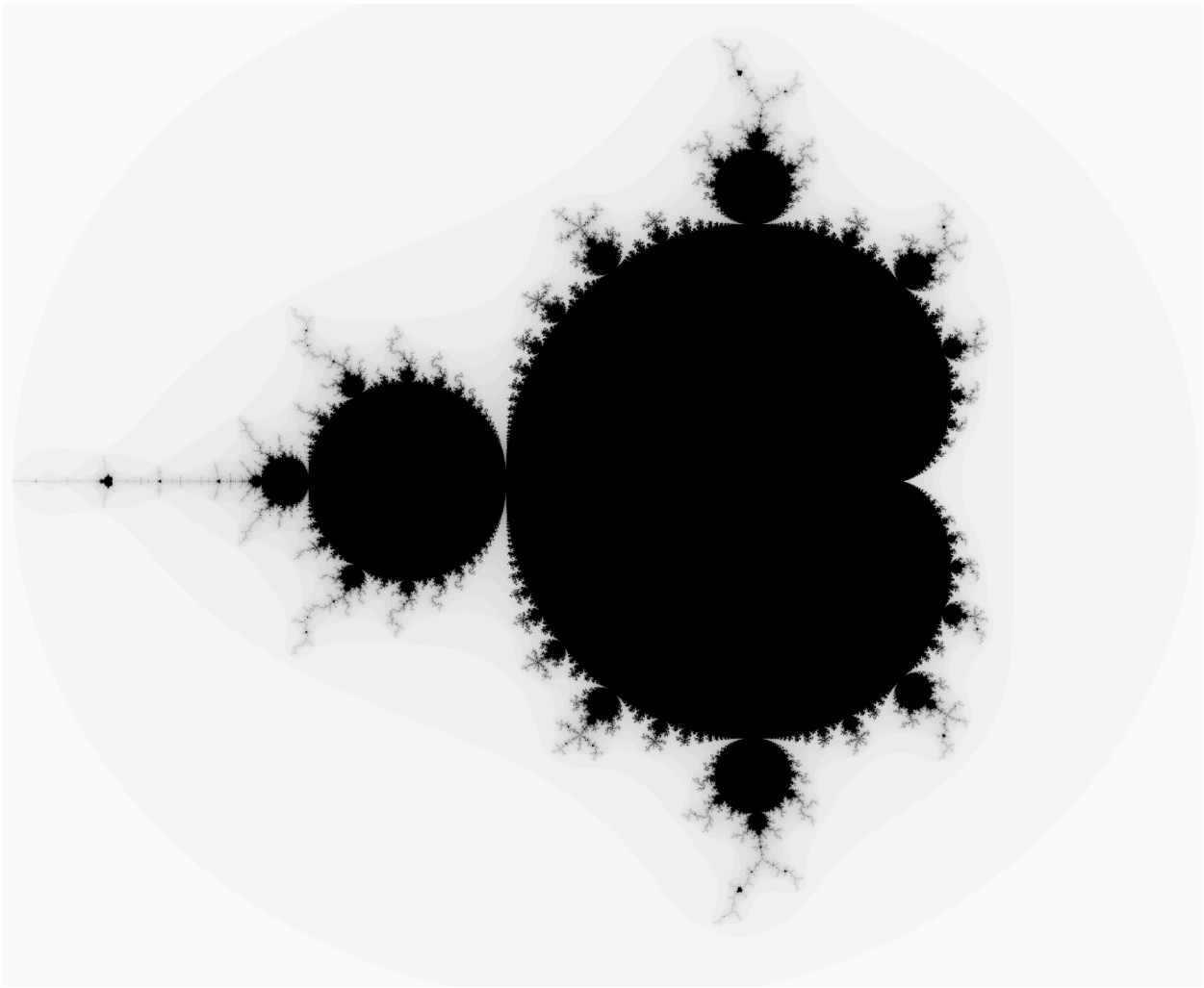
Table of contents

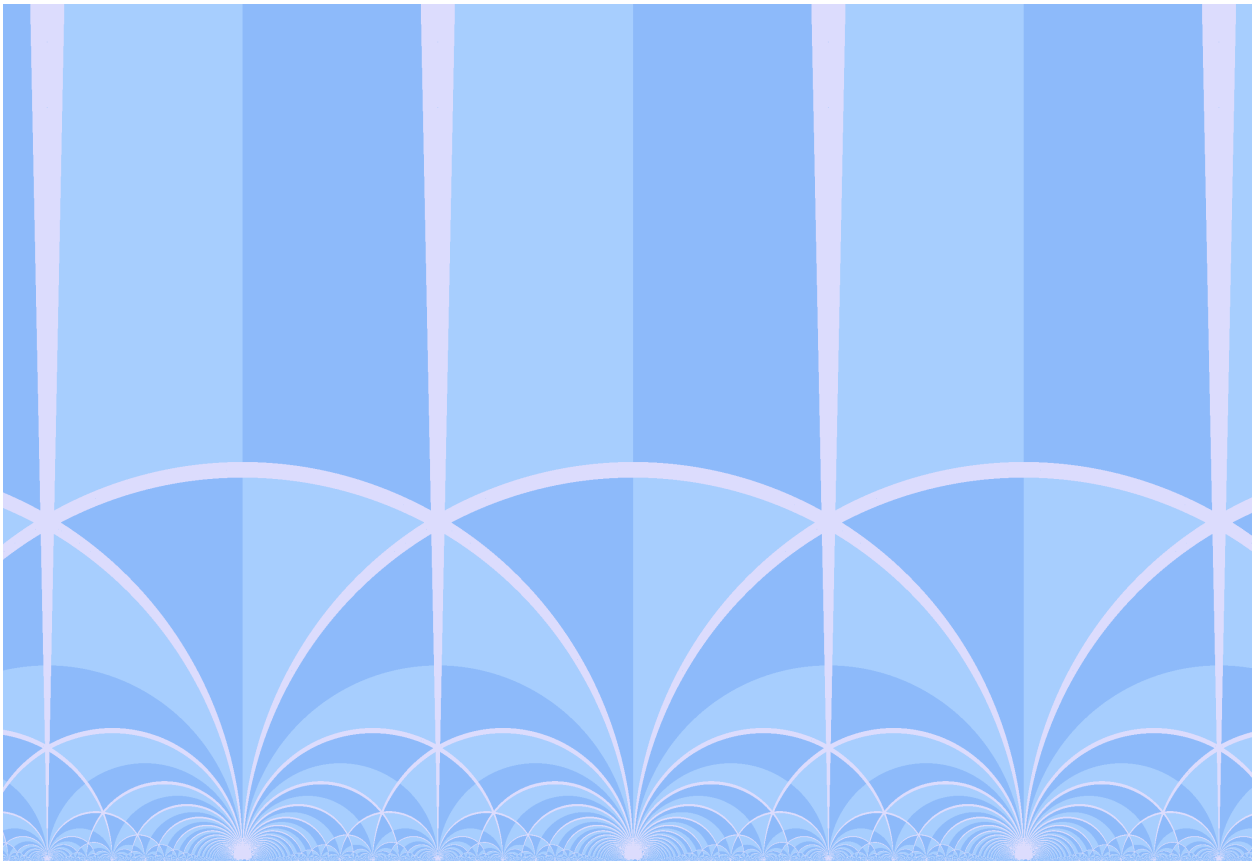
About

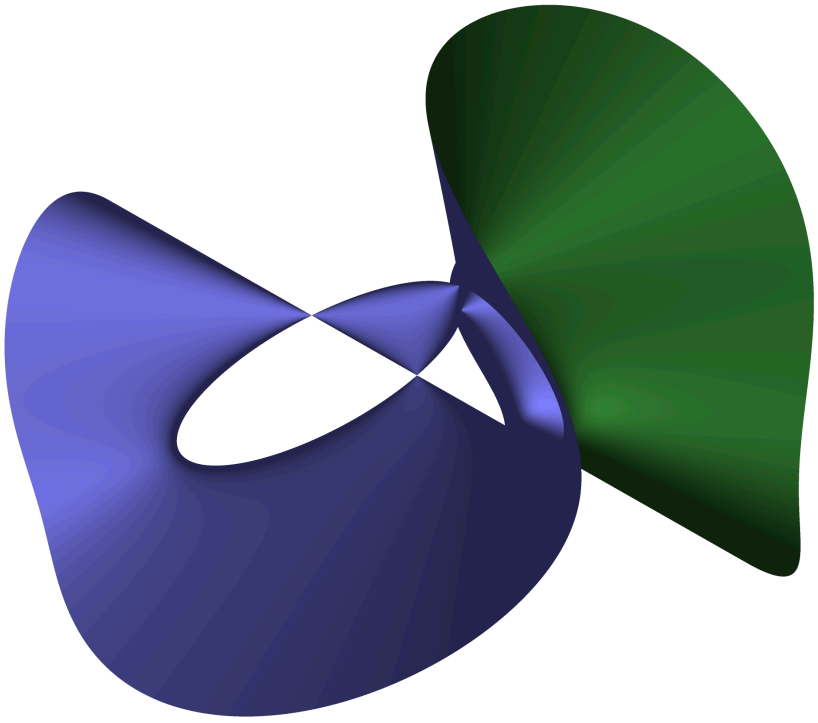
This mini-course introduces shader programming as a tool for mathematical illustration and exploration. Shaders are programs that run in parallel on the GPU, making them exceptionally fast for visualization tasks. We'll learn to write code that "reads like mathematics" using Shadertoy, a beginner-friendly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







Outline

Course Overview

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.

Format: Five days, each with one hour of lecture and approximately 1.5 hours of required homework, additional hours of optional homework (for those looking to really develop some of the skills, either during the course or after)

Prerequisites: Strong foundation in undergraduate mathematics; no prior experience with shaders or GLSL required

Day 1: Introduction to Shader Programming

Learning Objectives

- Understand the mathematical model of shader programming (function from pixels to colors)
- Learn basic GLSL syntax and conventions
- Master coordinate system setup and distance calculations
- Create simple geometric shapes and patterns

In-Class Content

Mathematical Introduction

- What is a shader? Framing as a function: `color = f(x, y, time, ...)`
- Why GPUs? Parallelism means computing ALL pixels simultaneously
- Shadertoy overview: available uniforms (`iResolution`, `iTime`, `iMouse`)

First Shader: Solid Colors

- Basic shader structure: `void mainImage(out vec4 fragColor, in vec2 fragCoord)`
- Setting `fragColor = vec4(1.0, 0.0, 0.0, 1.0)` for a red screen
- GLSL syntax basics: semicolons, vector types, swizzling
- Animating colors with `iTime`

Coordinate System Setup

- Converting `fragCoord` to centered, normalized coordinates
- Standard boilerplate for coordinate transformation
- Handling aspect ratio correctly

Half-Plane Coloring

- Boolean expressions: `x < 0.0`
- Conditional coloring with ternary operator or `step()` function
- Generalizing to arbitrary lines: `ax + by < 0`

Circles and Distance Fields

- Computing distance to center with `length(p)`
- Filled circle: `length(p) < radius`
- Circle outline: `abs(length(p) - radius) < thickness`
- Optional: Color gradients based on distance

Grids and Repetition

- Using `mod(p, spacing)` to create repeating cells
- Creating a grid of circles
- Alternating patterns with `mod(floor(p), 2.0)`

Homework

Required: Parabola Graphing Calculator

Create a shader that draws a parabola with customizable coefficients: - Draw x and y axes (thick lines at x=0 and y=0) - Define variables: `float a = 1.0; float b = 0.0; float c = 0.0;` - Plot the curve `y = ax² + bx + c` as a thick tube - Should handle any hardcoded values of a, b, c

Optional #1: Animated Curve Family

- Use `iTime` to vary parameters and animate a family of curves
- Suggestions: elliptic fibration, Lissajous curves, morphing shapes

Optional #2: Beautiful Tiling Pattern

- Design a pattern within a fundamental square
- Use `mod()` to tile it across the screen
- Focus on aesthetic appeal and mathematical structure

Day 2: Complex Dynamics and Euclidean Geometry

Learning Objectives

- Implement complex number arithmetic in GLSL
- Understand and render the Mandelbrot and Julia sets
- Create geometric tilings using mathematical transformations
- Learn to use structs for organizing data

In-Class Content

Complex Numbers in GLSL

- Representing complex numbers as `vec2`
- Implementing complex multiplication
- Complex addition, conjugation, and magnitude

The Mandelbrot Set

- Mathematical definition: iterating $z \to z^2 + c$ with $z = 0$
- Escape-time algorithm
- Coloring schemes based on iteration count
- Discussing convergence and divergence

Julia Sets

- Fixing `c` and varying initial `z`
- Relationship to Mandelbrot set
- Creating visually interesting Julia sets

Introduction to Structs

- Defining struct types in GLSL
- Use case: organizing geometric data
- Example: storing triangle vertices or transformation data

Euclidean Triangle Tiling

- Fundamental domain for triangular tiling
- Reflection across edges to create periodic patterns
- Symmetry groups and transformations

Homework

Required: Circle Inversion

- Implement circle inversion as an operation on the plane
- For a circle of radius `R` centered at origin: `p' = R² * p / |p|²`
- For a circle centered at `c` with radius `R`: translate, invert, translate back
- Visualize by applying inversion to a grid or pattern
- Demonstrate that circles through the inversion center become lines, and vice versa

Optional #1: Apollonian Gasket

- Use your circle inversion implementation
- Start with three mutually tangent circles
- Repeatedly invert through each circle
- Color by iteration count or which circle was inverted through
- Explore the fractal structure created by nested circles

Day 3: Fractals and Hyperbolic Geometry

Learning Objectives

- Extend folding techniques to box fractals
- Understand hyperbolic geometry models and their properties
- Implement computations in the upper half-plane model
- Create hyperbolic tilings using inversions and Möbius transformations
- Convert between different models of hyperbolic geometry

In-Class Content

Review: Sierpinski Triangle via Folding

- Generating fractals through iterated reflections
- Using `abs()` for geometric folding
- Scaling and iteration
- Coloring by iteration depth or distance

Introduction to Hyperbolic Geometry

- Why hyperbolic space? Negative curvature vs. Euclidean geometry
- Three primary models: Poincaré disk, upper half-plane, and band model
- How models are related via Möbius transformations
- Properties: geodesics, distance, angle measurement

Computations in the Upper Half-Plane Model

- Definition: points `{z : Im(z) > 0}`
- Geodesics: vertical lines and semicircles perpendicular to real axis
- Distance formula in the upper half-plane
- Möbius transformations as isometries: `z → (az + b)/(cz + d)` with `ad - bc = 1`
- Matrix representation of transformations

Hyperbolic Triangle Tiling

- Setting up a hyperbolic triangle (angles sum to $<$)
- Computing geodesics as circular arcs
- Implementing circle inversion for reflections across geodesics
- Folding points into fundamental domain
- Iterating to create the full tiling
- Coloring by domain or iteration behavior

Drawing in Different Models

- Converting between Poincaré disk and upper half-plane
- Möbius transformation: `w = i(1-z)/(1+z)` (disk to half-plane)
- Band model via additional Möbius transformation
- Visualizing the same tiling in multiple models simultaneously

Homework

Required: Sierpinski Carpet

- Implement the 2D Sierpinski carpet using box folding
- Use `abs()` to create 4-fold symmetry
- Scale by 3 at each iteration, removing middle square
- Color by iteration depth or distance
- Experiment with different iteration counts

Required: Model Conversion and Möbius Transformation

- Convert your hyperbolic tiling from upper half-plane to Poincaré disk (or vice versa)
- Apply a Möbius transformation to your tiling (choose your own or use a suggested one)
- Render both the original and transformed tiling
- Observe how the transformation affects the visual appearance

Optional: Advanced Explorations

1. **Another Triangle Group:** Implement a different hyperbolic triangle group (e.g., (2,3,7), (2,4,6))
2. **Klein Model:** Convert to Klein model via Cayley transform
3. **Conformal Art:** Use complex analysis to map to creative domains
4. **Decorated Tiles:** Create Escher-style decorated tiles

Day 4: Introduction to 3D Rendering

Learning Objectives

- Understand ray setup and camera models
- Implement analytical ray-object intersection
- Learn the raymarching algorithm and signed distance functions
- Apply basic lighting models (diffuse shading)

In-Class Content

Ray Setup and Camera Model

- Defining ray origin and direction from pixel coordinates
- Simple camera model: position, look-at, up vector
- Field of view and perspective projection

Analytical Ray-Sphere Intersection

- Deriving the intersection equation (quadratic)
- Solving for intersection parameter `t`
- Computing surface normal at intersection point
- Rendering the sphere with flat color

Ray-Torus Intersection

- Implicit equation for a torus
- Computing gradient for surface normal
- Discussion: analytical methods become complex quickly

Motivation for Raymarching

- Combining multiple objects is difficult with analytical methods
- Boolean operations (union, intersection) are hard
- Arbitrary implicit surfaces require root-finding

Signed Distance Functions (SDFs)

- Mathematical definition: minimum distance to surface
- SDFs for basic primitives: sphere, box, plane, torus, cylinder
- Properties: Lipschitz continuity and safe marching

The Raymarching Algorithm

- Sphere tracing: march along ray by the distance to nearest surface
- Stopping conditions: hit surface, max iterations, or exit bounds
- Estimating normals via gradient of the SDF

Basic Lighting

- Computing surface normal from SDF gradient
- Diffuse shading: dot product with light direction
- Simple Lambertian lighting model

Homework

Required: Algebraic Variety Rendering

- Choose a polynomial implicit surface (degree 3 or 4)
- Implement root-finding algorithm (bisection, Newton's method, etc.)
- Use gradient for directional derivative to estimate distance
- Optimization: use sphere bounding box (outside sphere $\rightarrow$ return sphere SDF, inside $\rightarrow$ compute polynomial distance)

Optional: Advanced Lighting and Transformations

1. **Specular Lighting:** Implement Phong or Blinn-Phong model
2. **Transformations:** Use rotation matrices to orient objects in the scene
3. **Complex Scene:** Combine multiple transformed objects with analytical intersections

---

Day 5: Choose Your Adventure

The final day will be determined based on pacing, student interest, and energy levels. Most likely we will actually use the day to slow down as previous topics or questions took more time. If we do have the available time for new content, then there will be two choices for the class:

Option A: Advanced Raymarching Techniques

Learning Objectives

- Master domain operations for efficient complex scenes
- Understand and apply boolean operations on SDFs
- Create 3D fractals via iterated folding
- Build sophisticated mathematical visualizations

In-Class Content

Domain Operations

- **Repetition:** Using `mod(p, spacing)` for infinite grids of objects
- **Symmetry:** Using `abs()` for mirror planes
- **Polar repetition:** Radial patterns around an axis
- **Computational advantage:** Zero cost for infinite complexity

Boolean Operations on SDFs

- **Union:** `min(d1, d2)`
- **Intersection:** `max(d1, d2)`
- **Subtraction:** `max(d1, -d2)`
- **Smooth minimum:** `smin()` for organic blending
- Building complex shapes from primitive combinations

The Menger Sponge

- Box folding in 3D with axis-aligned planes
- Iterated subdivision pattern
- Scaling and repetition
- Connection to Day 2's Sierpinski carpet

Advanced Examples

- Architectural structures via boolean operations
- Infinite repeated patterns via domain operations
- Combining techniques for rich scenes

Projects (not homework, as class is over!)

Creative Scene Building

- Build a complex scene using domain operations and boolean combinations
- Experiment with different SDFs and transformations
- Focus on mathematical or aesthetic interest

Sierpinski Tetrahedron

- Implement 3D Sierpinski tetrahedron via folding
- Reflect across four planes (non-axis-aligned)
- Connection to Day 2's triangle folding in higher dimension

Option B: Buffers and Temporal Dynamics

Learning Objectives

- Understand buffer-based computation in Shadertoy
- Implement differential operators (Laplacian)
- Solve partial differential equations on the GPU
- Create dynamic, evolving mathematical systems

In-Class Content

Introduction to Buffers

- Reading from previous frame: `texture(iChannel0, uv)`
- Multi-pass rendering in Shadertoy
- Simple example: reading buffer and applying conditional coloring (bright → yellow, dark → blue)

Edge Detection and the Laplacian

- Discrete Laplacian stencil (5-point or 9-point)
- Sampling neighboring pixels
- Visualizing edges in imagery
- Introduction to spatial derivatives on discrete grids

The Heat Equation

- Mathematical formulation: `u_t =  ²u`
- Applying the Laplacian stencil for diffusion
- Time-stepping: `u_new = u_old + dt *  * laplacian(u_old)`
- Initial conditions: heat distribution in a fractal or Julia set
- Watching the pattern blur and diffuse

Boundary Conditions

- Zero boundary conditions (edges set to 0)
- Avoiding wrap-around artifacts
- Discussion of periodic boundaries (if time)

Timestep Stability

- CFL condition (briefly mentioned)
- Providing a stable `dt` value
- Warning: don't make timestep too large!

Projects

Interactive Heat Equation or Reaction-Diffusion

1. **Interactive Heat Source:** Add heat at mouse position, watch it diffuse
2. **Gray-Scott Reaction-Diffusion:** Implement pattern formation (spots, stripes, etc.) - store U and V in different color channels

Wave Equation

- Requires two buffers (current and previous state)
- Implement `u_tt = c² ²u`
- Initial conditions: pluck a "string" or create a disturbance
- Watch waves propagate and reflect

---

1. Day 1: Introduction to Shader Programming

1.1. Overview

Today we introduce the fundamental concept of shader programming: computing a function from pixel coordinates to colors, executed in parallel across the entire image. We'll learn basic GLSL syntax, set up a coordinate system, and create simple geometric shapes.

By the end of today, you'll be able to render implicit curves, distance-based coloring, and repeating patterns—all computed in real-time on the GPU.

1.2. What is a Shader?

1.2.1. Mathematical Perspective

A shader is fundamentally a function

$$f : \mathbb{R}^2 \times \mathbb{R} \times \cdots \to \mathbb{R}^4$$

that maps pixel coordinates (and potentially time, mouse position, etc.) to color values. For each pixel on the screen, we evaluate this function to determine what color to display.

The key insight: modern GPUs can evaluate this function for **all pixels simultaneously**. If your screen has 1920×1080 pixels, that's over 2 million function evaluations happening in parallel, typically 60 times per second.

This parallelism is what makes shaders extraordinarily fast for mathematical visualization—we're not looping over pixels sequentially, we're computing them all at once.

1.2.2. Why Shadertoy?

Shadertoy is a web-based platform that abstracts away the complexities of GPU programming (setting up OpenGL contexts, managing buffers, compiling shaders, etc.). You write a single function, and Shadertoy handles everything else.

The platform provides several built-in **uniforms** (read-only global variables): - `iResolution`: screen resolution as a `vec3` (width, height, pixel aspect ratio) - `iTime`: elapsed time in seconds - `iMouse`: mouse position and click state as a `vec4`

We'll use these throughout the week.

1.3. First Shader: Solid Colors

1.3.1. Basic Structure

Every Shadertoy shader has the same entry point:

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

Parameters: - `fragCoord`: the pixel coordinate we're currently computing, as a `vec2` (x, y) - `fragColor`: the output color we need to set, as a `vec4` (red, green, blue, alpha)

Colors are represented in RGBA format with values in $[0, 1]$. So `vec4(1.0, 0.0, 0.0, 1.0)` represents opaque red, while `vec4(0.5, 0.5, 0.5, 1.0)` is middle gray.

1.3.2. Example: Red Screen

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

This sets every pixel to red. The function is evaluated once per pixel, but since the output doesn't depend on `fragCoord`, every pixel gets the same value.

1.3.3. GLSL Syntax Basics

A few essential points about the GLSL language:

Semicolons are required. Every statement must end with a semicolon. This is not Python!

Vector types: GLSL has built-in types `vec2`, `vec3`, `vec4` for 2D, 3D, and 4D vectors. You can construct them with:

```
vec2 v = vec2(1.0, 2.0);
vec3 w = vec3(1.0, 2.0, 3.0);
vec4 color = vec4(v, 0.0, 1.0);  // Can combine vectors and scalars
```

Swizzling: You can access components by name: `v.x`, `v.y` or `v.r`, `v.g` (same thing, different naming convention). We'll cover more syntax in the appendix.

Floating point literals: Write `1.0` not `1` for floating point values. GLSL is picky about types.

1.3.4. Animating with Time

Let's make something that changes:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    float red = 0.5 + 0.5 * sin(iTime);
    fragColor = vec4(red, 0.0, 0.0, 1.0);
}
```

Here `iTime` grows continuously, `sin(iTime)` oscillates between -1 and 1, and we remap to $[0, 1]$ with the affine transformation $x \mapsto \frac{1}{2}(1 + x)$. The screen now pulses between black and red.

1.4. Coordinate Systems

1.4.1. Raw Coordinates

By default, `fragCoord` gives pixel coordinates with: - Origin $(0, 0)$ at the bottom-left - x increases rightward to `iResolution.x` - y increases upward to `iResolution.y`

For mathematical work, we want: - Origin at the center - Coordinates normalized (not in pixels) - Aspect ratio handled correctly

1.4.2. Centered, Normalized Coordinates

Here's a standard transformation:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Normalize to [0,1]
    vec2 uv = fragCoord / iResolution.xy;

    // Center at origin: [-0.5, 0.5]
    uv = uv - 0.5;

    // Scale to account for aspect ratio
    uv.x *= iResolution.x / iResolution.y;

    // Now uv is centered and aspect-corrected
    // Scale to desired viewing window (e.g., [-2, 2] on x-axis)
    vec2 p = uv * 4.0;  // Now p is in [-2, 2] × [-h, h] where h depends on aspect ratio

    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

From now on, we'll assume this coordinate setup is done at the start of every shader, storing the result in a variable `p` for "position."

1.4.3. Visualizing Coordinates

Let's color pixels by their position:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // [Coordinate setup as above, resulting in p]

    // Map x coordinate to red, y to green
    vec2 color_rg = p * 0.5 + 0.5;  // Remap to [0, 1]
    fragColor = vec4(color_rg, 0.0, 1.0);
}
```

You should see a smooth gradient: red increases rightward, green increases upward.

1.5. Conditional Coloring: Half-Planes

1.5.1. The Concept

Given a linear function $L(x, y) = ax + by$, we want to color pixels differently depending on whether $L(p) < 0$ or $L(p) \geq 0$. This divides the plane into two half-planes.

1.5.2. Implementation

GLSL provides a conditional operator (ternary operator) just like C:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // [Coordinate setup, resulting in p]

    float L = p.x;  // The function L(x,y) = x

    vec3 color = (L < 0.0) ? vec3(1.0, 0.0, 0.0) : vec3(0.0, 0.0, 1.0);
    fragColor = vec4(color, 1.0);
}
```

Left half-plane is red, right half-plane is blue.

1.5.3. The Step Function

GLSL also provides `step(edge, x)` which returns 0 if $x <$ edge and 1 otherwise. This is useful for smooth code:

```
float s = step(0.0, p.x);  // 0 on left, 1 on right
vec3 color = mix(vec3(1.0, 0.0, 0.0), vec3(0.0, 0.0, 1.0), s);
```

Here `mix(a, b, t)` performs linear interpolation: $(1-t)a + tb$.

1.5.4. Arbitrary Half-Planes

For a general line $ax + by = 0$:

```
float a = 1.0, b = 1.0;
float L = a * p.x + b * p.y;
vec3 color = (L < 0.0) ? vec3(1.0, 0.0, 0.0) : vec3(0.0, 0.0, 1.0);
fragColor = vec4(color, 1.0);
```

Try different values of a and b to see different line orientations.

1.6. Distance Fields and Circles

1.6.1. Distance to Center

The distance from a point $p = (x, y)$ to the origin is simply

$$d = \|p\| = \sqrt{x^2 + y^2}$$

In GLSL:

```
float d = length(p);
```

The `length()` function computes the Euclidean norm of a vector.

1.6.2. Filled Circle

A circle of radius r centered at the origin is the set $\{p : \|p\| < r\}$. We can color inside vs. outside:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // [Coordinate setup]

    float d = length(p);
    float r = 1.0;

    vec3 color = (d < r) ? vec3(1.0, 1.0, 0.0) : vec3(0.1, 0.1, 0.3);
    fragColor = vec4(color, 1.0);
}
```

This renders a yellow disk on a dark blue background.

1.6.3. Distance-Based Coloring

We can create gradients by mapping distance to color:

```
float d = length(p);
float intensity = 1.0 - d / 2.0;  // Fades out with distance
intensity = clamp(intensity, 0.0, 1.0);  // Keep in [0, 1]
vec3 color = vec3(intensity);
fragColor = vec4(color, 1.0);
```

1.6.4. Circle Outline

To draw just the boundary of a circle (an annulus of small thickness), we check if d is approximately equal to r:

```
float d = length(p);
float r = 1.0;
float thickness = 0.05;

float circle_mask = abs(d - r) < thickness ? 1.0 : 0.0;
vec3 color = vec3(circle_mask);
fragColor = vec4(color, 1.0);
```

Mathematically, we're coloring the set $\{p : |d(p) - r| < \epsilon\}$ where ϵ is our thickness parameter.

For a smoother edge, we can use `smoothstep`:

```
float circle_mask = 1.0 - smoothstep(r - thickness, r + thickness, d);
```

The `smoothstep(a, b, x)` function performs smooth Hermite interpolation between a and b.

1.7. Grids and Repetition

1.7.1. Modular Arithmetic

The modulo operation creates periodic repetition. For a period T, the function $p \mapsto (p \bmod T) - T/2$ maps $\mathbb{R}$ to $[-T/2, T/2]$ repeatedly.

In GLSL, `mod(x, T)` computes $x \bmod T$.

1.7.2. Creating a Grid

To create a grid of repeated cells:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // [Coordinate setup, resulting in p]

    float spacing = 1.0;
    vec2 cell_p = mod(p + spacing/2.0, spacing) - spacing/2.0;

    // Now cell_p repeats every spacing units
    // Draw a circle in each cell
    float d = length(cell_p);
    float r = 0.3;

    vec3 color = (d < r) ? vec3(1.0, 1.0, 0.0) : vec3(0.1, 0.1, 0.3);
    fragColor = vec4(color, 1.0);
}
```

This creates an infinite grid of yellow circles!

1.7.3. Alternating Pattern

We can create checkerboard-like patterns using the cell index:

```
vec2 cell_id = floor(p / spacing);
float checker = mod(cell_id.x + cell_id.y, 2.0);

vec3 color_a = vec3(1.0, 0.0, 0.0);
vec3 color_b = vec3(0.0, 0.0, 1.0);
vec3 bg_color = mix(color_a, color_b, checker);
```

Here `floor(p / spacing)` gives us integer grid indices, and we alternate colors based on the parity of $i + j$.

1.7.4. Combining with Circles

Put it all together for a grid of circles on an alternating background:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // [Coordinate setup]

    float spacing = 1.0;
    vec2 cell_id = floor(p / spacing);
    vec2 cell_p = mod(p + spacing/2.0, spacing) - spacing/2.0;

    // Checkerboard background
    float checker = mod(cell_id.x + cell_id.y, 2.0);
    vec3 bg_color = mix(vec3(0.2, 0.2, 0.3), vec3(0.3, 0.2, 0.2), checker);

    // Circle in each cell
    float d = length(cell_p);
    float r = 0.3;
    vec3 circle_color = vec3(1.0, 1.0, 0.0);

    vec3 color = (d < r) ? circle_color : bg_color;
    fragColor = vec4(color, 1.0);
}
```

1.8. Implicit Curves

1.8.1. General Principle

An implicit curve is defined by an equation $F(x, y) = 0$. To render it, we compute $F(p)$ for each pixel and color based on proximity to zero:

```
float F = [some function of p.x and p.y];
float thickness = 0.05;
float curve_mask = abs(F) < thickness ? 1.0 : 0.0;
vec3 color = mix(background, curve_color, curve_mask);
```

1.8.2. Example: Parabola

The parabola $y = x^2$ can be written implicitly as $F(x, y) = y - x^2 = 0$:

```
float F = p.y - p.x * p.x;
float thickness = 0.1;
float curve_mask = abs(F) < thickness ? 1.0 : 0.0;

vec3 color = mix(vec3(0.1, 0.1, 0.3), vec3(1.0, 1.0, 0.0), curve_mask);
fragColor = vec4(color, 1.0);
```

1.8.3. Example: Circle (Implicit Form)

We've been using $\|p\| < r$ for filled circles, but we can also write the circle implicitly as $x^2+y^2-r^2 = 0$:

```
float r = 1.0;
float F = dot(p, p) - r * r;  // dot(p,p) = x² + y²
float thickness = 0.1;
float curve_mask = abs(F) < thickness ? 1.0 : 0.0;
```

This is mathematically equivalent to our earlier approach but demonstrates the general implicit curve technique.

1.9. Summary

Today we've learned:

1. **Shaders as parallel functions**: Every pixel evaluates $f(x, y, t, ...) \rightarrow$ color simultaneously
2. **GLSL basics**: Syntax, vector types, and built-in functions
3. **Coordinate systems**: Centering, normalizing, and scaling for mathematical work
4. **Conditional coloring**: Using boolean expressions and `step()` for discrete color regions
5. **Distance fields**: Using `length()` to create circles and radial patterns
6. **Modular arithmetic**: Creating grids and repeating patterns with `mod()`
7. **Implicit curves**: Rendering curves defined by $F(x, y) = 0$

With these tools, you can already create a wide variety of mathematical visualizations!

1.10. Homework

1.10.1. Required: Parabola Graphing Calculator

Create a shader that draws a customizable parabola $y = ax^2 + bx + c$ along with coordinate axes.

Requirements: - Define variables `a`, `b`, `c` at the top of your shader (hardcoded values are fine) - Draw the x-axis and y-axis as thick lines (use the implicit line technique: $|y| < \epsilon$ for x-axis, $|x| < \epsilon$ for y-axis) - Plot the parabola $y = ax^2 + bx + c$ as a thick curve - Use distinct colors for axes and parabola - The visualization should work for any reasonable values of a, b, c

Suggested approach:

```
// Define parameters
float a = 1.0;
float b = 0.0;
float c = 0.0;

// Axes
float x_axis_mask = abs(p.y) < 0.05 ? 1.0 : 0.0;
float y_axis_mask = abs(p.x) < 0.05 ? 1.0 : 0.0;

// Parabola: F(x,y) = y - (ax² + bx + c) = 0
float F = p.y - (a * p.x * p.x + b * p.x + c);
float parabola_mask = abs(F) < 0.1 ? 1.0 : 0.0;

// Combine
vec3 color = background;
color = mix(color, axis_color, max(x_axis_mask, y_axis_mask));
color = mix(color, parabola_color, parabola_mask);
```

Try different values of a, b, c and verify your grapher works correctly!

1.10.2. Optional #1: Animated Curve Family

Create a shader that animates through a family of curves.

Suggestions: - **Elliptic fibration**: Fix a cubic polynomial and vary a parameter: $y^2 = x^3 + ax + b$ where a or b varies with `iTime` - **Lissajous curves**: $x = A\sin(at + \delta)$, $y = B\sin(bt)$, animate δ or the frequency ratio

Use `iTime` creatively to create a compelling animation. The goal is to explore how continuous parameter variation produces interesting mathematical families.

1.10.3. Optional #2: Beautiful Tiling Pattern

Design an aesthetically pleasing tiling pattern using the `mod()` technique.

Requirements: - Create a non-trivial pattern within a fundamental domain (a single tile) - Use `mod()` to repeat it across the plane - The pattern should tile seamlessly (edges should match up)

Ideas: - Geometric patterns: nested circles, polygons, stars - Color gradients that vary by tile position - Combinations of implicit curves within each tile - Symmetry: use `abs()` to create reflections within tiles

Challenge: Can you create a pattern that has different symmetries in different tiles? (For example, alternating rotational symmetry using the checkerboard `cell_id` technique.)

1.11. Looking Ahead

Tomorrow we'll use these techniques to explore **complex dynamics** (the Mandelbrot and Julia sets) and **geometric tilings** (including fractals via iterated folding). The coordinate system and implicit curve techniques you've learned today will be the foundation for everything to come.

Make sure you're comfortable with: - Setting up coordinates - Computing distances and implicit functions
- Using `mod()` for repetition - Conditionally coloring based on mathematical expressions

2. Day 2: Complex Dynamics and Euclidean Geometry

2.1. Overview

Today we explore the power of iteration and geometric transformations. We'll implement complex arithmetic in GLSL, use it to render the iconic Mandelbrot and Julia sets, and create geometric tilings through reflection. We conclude by building our first fractal via iterated folding—the Sierpinski triangle.

By the end of today, you'll understand how simple iterative processes can generate intricate mathematical structures, how to organize geometric data using structs, and how folding operations can create self-similar fractals.

2.2. Complex Numbers in GLSL

2.2.1. Representation

A complex number $z = a + bi$ can be represented as a 2D vector with real part a and imaginary part b. In GLSL:

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

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

2.2.2. Complex Arithmetic

Let $z = a + bi$ and $w = c + di$. We need to implement the basic operations:

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

```
vec2 cadd(vec2 z, vec2 w) {
    return z + w;  // Vector addition is sufficient!
}
```

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

```
vec2 cmul(vec2 z, vec2 w) {
    return vec2(
        z.x * w.x - z.y * w.y,  // Real part: ac - bd
        z.x * w.y + z.y * w.x   // Imaginary part: ad + bc
    );
}
```

Magnitude squared: $|z|^2 = a^2 + b^2$

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

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

```
float cabs(vec2 z) {
    return length(z);
}
```

Conjugate: $\overline{z} = a - bi$

```
vec2 cconj(vec2 z) {
    return vec2(z.x, -z.y);
}
```

These are the building blocks we need for complex dynamics.

2.3. The Mandelbrot Set

2.3.1. Definition

The Mandelbrot set $\mathcal{M}$ is defined as the set of complex numbers c for which the iteration

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

remains bounded as $n \to \infty$.

In practice, we: 1. Start with $z_0 = 0$ 2. Iterate $z_{n+1} = z_n^2 + c$ for a fixed number of iterations (say, 100) 3. Check if $|z_n|$ has escaped some large radius (typically $R = 2$)

Points that escape quickly are definitely not in $\mathcal{M}$. Points that remain bounded after many iterations are likely in $\mathcal{M}$.

2.3.2. Implementation

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup: center at origin, scale to show interesting region
    vec2 uv = fragCoord / iResolution.xy;
    uv = uv - 0.5;
    uv.x *= iResolution.x / iResolution.y;

    // Scale to view the Mandelbrot set (roughly [-2.5, 1] × [-1.25, 1.25])
    vec2 c = uv * 3.5;
    c.x -= 0.5;  // Center on the interesting part

    // Mandelbrot iteration
    vec2 z = vec2(0.0, 0.0);  // z_0 = 0
    int max_iter = 100;
    int iter;

    for(iter = 0; iter < max_iter; iter++) {
        // Check if escaped
        if(cabs2(z) > 4.0) break;  // |z| > 2, so |z|² > 4

        // z_{n+1} = z_n² + c
        z = cmul(z, z) + c;
    }

    // Color based on iteration count
    float t = float(iter) / float(max_iter);
    vec3 color = vec3(t);  // Grayscale for now

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

2.3.3. Coloring Schemes

The grayscale rendering shows structure but isn't very exciting. We can create better colormaps:

Smooth coloring using escape time:

```
if(iter < max_iter) {
    // Smooth iteration count (accounts for continuous escape)
    float log_zn = log(cabs2(z)) / 2.0;
    float nu = log(log_zn / log(2.0)) / log(2.0);
    float smooth_iter = float(iter) + 1.0 - nu;

    float t = smooth_iter / float(max_iter);
    // Use a color palette (see below)
    vec3 color = palette(t);
```

```
} else {
    // Inside the set: black
    vec3 color = vec3(0.0);
}
```

Simple color palette:

```
vec3 palette(float t) {
    // Create a cyclic color palette
    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(0.0, 0.33, 0.67);

    return a + b * cos(6.28318 * (c * t + d));
}
```

This uses a cosine-based palette function that creates smooth, cyclic colors. Play with the parameters `a`, `b`, `c`, `d` to get different color schemes!

2.4. Julia Sets

2.4.1. Definition

For a fixed complex parameter c, the filled Julia set $\mathcal{K}_c$ consists of points z_0 for which the iteration

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

remains bounded.

Key difference from Mandelbrot: Here c is fixed and we vary the initial point z_0 (which comes from the pixel position). In Mandelbrot, $z_0 = 0$ and c varies with pixel position.

2.4.2. Relationship to Mandelbrot

There's a beautiful connection: the Mandelbrot set is essentially a "parameter space" for Julia sets. Each point c in the complex plane has an associated Julia set $\mathcal{K}_c$: - If $c \in \mathcal{M}$, then $\mathcal{K}_c$ is connected - If $c \notin \mathcal{M}$, then $\mathcal{K}_c$ is a Cantor dust (totally disconnected)

2.4.3. Implementation

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup
    vec2 uv = fragCoord / iResolution.xy;
    uv = uv - 0.5;
    uv.x *= iResolution.x / iResolution.y;
    vec2 z = uv * 3.0;  // Initial point z_0 comes from pixel position

    // Fix c to an interesting value
    vec2 c = vec2(-0.7, 0.27015);  // A classic choice
    // Try: vec2(-0.4, 0.6), vec2(0.285, 0.01), vec2(-0.8, 0.156)

    // Iterate z_{n+1} = z_n² + c
    int max_iter = 100;
    int iter;

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

    // Color (same as Mandelbrot)
    float t = float(iter) / float(max_iter);
    vec3 color = palette(t);
    if(iter == max_iter) color = vec3(0.0);

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

Try different values of c to explore the incredible variety of Julia sets! You can even animate c with time:

```
vec2 c = vec2(0.7 * cos(iTime * 0.3), 0.7 * sin(iTime * 0.3));
```

2.5. Structs in GLSL

2.5.1. Motivation

As we build more complex geometric objects, we need to organize related data. GLSL provides **structs** (similar to C structs or simple classes without methods).

2.5.2. Defining a Struct

```
struct Triangle {
    vec2 v0;  // First vertex
    vec2 v1;  // Second vertex
    vec2 v2;  // Third vertex
};
```

2.5.3. Using Structs

```
Triangle tri;
tri.v0 = vec2(0.0, 1.0);
tri.v1 = vec2(-0.866, -0.5);
tri.v2 = vec2(0.866, -0.5);

// Access fields with dot notation
vec2 centroid = (tri.v0 + tri.v1 + tri.v2) / 3.0;
```

2.5.4. Why Structs?

For geometric transformations (reflections, rotations), we'll need to pass around geometric data. Structs make the code cleaner and more mathematical. For example, we can write:

```
vec2 reflect(vec2 p, Triangle tri) {
    // Reflect point p across an edge of tri
    // [Implementation details]
}
```

This is more readable than passing six individual float parameters for the triangle vertices.

2.6. Euclidean Triangle Tiling

2.6.1. The Fundamental Domain

An equilateral triangle tiles the Euclidean plane. Given an equilateral triangle, we can: 1. Reflect across its three edges 2. Repeat this process on the resulting triangles 3. Fill the entire plane with copies of the original triangle

2.6.2. Setting Up the Triangle

Let's work with an equilateral triangle with vertices at:

$$v_0 = (0, 1), \quad v_1 = (-\frac{\sqrt{3}}{2}, -\frac{1}{2}), \quad v_2 = (\frac{\sqrt{3}}{2}, -\frac{1}{2})$$

This triangle has side length $\sqrt{3}$ and is centered at the origin.

```
// Define triangle vertices
vec2 v0 = vec2(0.0, 1.0);
vec2 v1 = vec2(-0.866, -0.5);  // -sqrt(3)/2   -0.866
vec2 v2 = vec2(0.866, -0.5);
```

2.6.3. Reflection Across a Line

To reflect a point p across a line through the origin with unit normal $\mathbf{n}$, we use:

$$p' = p - 2(\mathbf{n} \cdot p)\mathbf{n}$$

For a line through two points (an edge of our triangle), we need to: 1. Compute the perpendicular direction (normal to the edge) 2. Determine which side of the line we're on 3. Reflect if necessary

Placeholder for specific implementation: The exact formulas depend on how we set up our edges. Here's the structure:

```
// Reflect across edge v0-v1
vec2 edge = v1 - v0;
vec2 normal = normalize(vec2(-edge.y, edge.x));  // Perpendicular to edge
float dist = dot(p - v0, normal);  // Signed distance to line
if(dist < 0.0) {
    p = p - 2.0 * dist * normal;  // Reflect if on wrong side
}
```

2.6.4. Iterative Folding

The key insight: we repeatedly reflect p across the three edges of the triangle until it lands inside the fundamental domain.

```
vec2 foldToTriangle(vec2 p, int iterations) {
    // Define the three edges and their normals
    // [Edge definitions here]

    for(int i = 0; i < iterations; i++) {
        // Reflect across each edge if necessary
        // [Reflection code for edge 0]
        // [Reflection code for edge 1]
        // [Reflection code for edge 2]
```

```
    }

    return p;
}
```

After folding, all points in the plane map to the interior of our triangle. We can then color based on: - The final position within the triangle - The number of reflections needed - Which edge was crossed most recently

2.6.5. Visualization

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup
    vec2 p = [coordinate setup as usual];

    // Fold to fundamental domain
    vec2 p_folded = foldToTriangle(p, 10);

    // Color based on position in triangle
    // Could use barycentric coordinates, distance to edges, etc.
    vec3 color = vec3(p_folded * 0.5 + 0.5, 0.5);

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

Note: The exact implementation requires careful handling of the geometry. The key mathematical ideas are: 1. Reflection formula: $p' = p - 2(p \cdot n)n$ for a line with normal n through origin 2. Iterative folding brings any point into the fundamental domain 3. Track which reflections occur for interesting coloring

---

2.7. Sierpinski Triangle via Folding

2.7.1. The Concept

The Sierpinski triangle is a fractal that can be generated by: 1. Starting with an equilateral triangle 2. Removing the middle triangle (connecting midpoints) 3. Repeating on each remaining sub-triangle

Equivalently, we can generate it by **iterated folding with scaling**.

2.7.2. Folding Algorithm

At each iteration: 1. Reflect p across the three edges of the triangle (fold it inside) 2. Scale toward the center 3. Repeat

Mathematically, after n iterations, we've zoomed in by a factor of 2^n and applied n reflections.

2.7.3. Implementation Sketch

```
vec2 sierpinskiFold(vec2 p, int iterations) {
    // Triangle vertices (equilateral)
    vec2 v0 = vec2(0.0, 1.0);
    vec2 v1 = vec2(-0.866, -0.5);
    vec2 v2 = vec2(0.866, -0.5);

    float scale = 1.0;

    for(int i = 0; i < iterations; i++) {
        // Reflect across each edge (fold into triangle)
        // [Reflection code similar to triangle tiling]

        // After folding, scale and translate
        p = p * 2.0;  // Zoom in by factor of 2
        scale *= 2.0;

        // [Additional centering/translation may be needed]
    }

    return p;
}
```

2.7.4. Coloring by Iteration Depth

We can track how many times we hit certain conditions during folding:

```
int orbit = 0;  // Track some property during iteration

for(int i = 0; i < max_iter; i++) {
    // Folding operations

    // Track orbit behavior
    if([some condition]) orbit++;
}

// Color based on orbit count
float t = float(orbit) / float(max_iter);
vec3 color = palette(t);
```

The specific conditions to track depend on the geometric setup, but typically we color based on: - Which edge we reflected across most recently - How many reflections were needed - Distance from the center after folding

2.7.5. Visualization

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 p = [coordinate setup];

    // Apply Sierpinski folding
    vec2 p_folded = sierpinskiFold(p, 8);

    // Color based on the folded position
    float d = length(p_folded);
    vec3 color = d < 0.1 ? vec3(1.0) : vec3(0.0);

    // OR: color based on iteration behavior
    // [More sophisticated coloring]

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

The Sierpinski triangle should emerge as a self-similar fractal pattern!

2.8. Summary

Today we covered:

1. **Complex arithmetic in GLSL**: Representing complex numbers as `vec2` and implementing multiplication
2. **Mandelbrot set**: Iterating $z_{n+1} = z_n^2 + c$ with $z_0 = 0$, coloring by escape time
3. **Julia sets**: Fixing c and varying initial point z_0, exploring the parameter space
4. **Structs**: Organizing geometric data for cleaner code
5. **Triangle tiling**: Using reflection to fold the plane into a fundamental domain
6. **Sierpinski triangle**: Generating fractals through iterated folding and scaling

The key themes: - Simple iterations create complex behavior (chaos and fractals) - Geometric transformations (reflections) can tile or generate fractals - Tracking iteration behavior gives us rich information for coloring - Folding operations create self-similar structures

Tomorrow we'll extend these ideas: more fractals via folding, then hyperbolic geometry where the same techniques create entirely different tilings.

2.9. Homework

2.9.1. Required: Interactive Julia Sets

Add mouse control to the julia sets: take iMouse and read it out as a complex number, then draw the julia set for that number.

2.9.2. Required: Circle Inversion

Circle inversion is a fundamental transformation in geometry that will be the basis for many interesting visualizations. Your task is to implement it as a general operation on the plane.

Mathematical Background:

Inversion with respect to a circle of radius R centered at the origin maps a point $p \neq 0$ to:

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

Geometrically, this maps the inside of the circle to the outside and vice versa, with points on the circle remaining fixed. Key properties: - Circles passing through the center become lines (and vice versa) - Circles not passing through the center remain circles - Angles are preserved (it's a conformal map)

For a circle centered at c with radius R, we: 1. Translate to center the circle at origin: $p' = p - c$ 2. Apply inversion: $p'' = R^2 \frac{p'}{|p'|^2}$ 3. Translate back: result $= p'' + c$

Implementation Tasks:

1. **Basic inversion centered at origin:**

```
vec2 invert(vec2 p, float R) {
    float r2 = dot(p, p);
    return R * R * p / r2;
}
```

2. **Inversion centered at arbitrary point:**

```
vec2 invertCircle(vec2 p, vec2 center, float radius) {
    // Translate, invert, translate back
    // [Implement this]
}
```

3. **Visualization:** Create a shader that applies circle inversion to a pattern. Suggestions:
   - Start with a grid and show its image under inversion
   - Draw several circles and show how they transform
   - Apply inversion to implicit curves from Day 1
   - Make the inversion center follow the mouse position using `iMouse.xy`

4. **Demonstrate properties:** Show that:

- A line through the inversion center maps to itself
- A line not through the center maps to a circle through the center
- A circle through the center maps to a line

Expected Output: A shader that clearly demonstrates circle inversion operating on the plane, with visual evidence of its geometric properties.

2.9.3. Optional #1: Apollonian Gasket

Now that you've implemented circle inversion, use it to create the beautiful Apollonian gasket fractal.

The Construction:

Start with three mutually tangent circles. The Apollonian gasket is generated by: 1. Finding the fourth circle tangent to all three (Apollonius's problem—there are generally two solutions) 2. Recursively filling gaps with new tangent circles 3. Continuing indefinitely

Simplified Approach via Inversion:

Rather than solving for tangent circles explicitly, we can use inversions: 1. Start with a configuration of circles (e.g., three circles tangent at a point) 2. Define inversions with respect to each circle 3. Apply these inversions repeatedly to generate the fractal pattern 4. Color based on iteration count or which circle was most recently inverted through

Challenge: Setting up the initial configuration and inversion circles requires working out the tangency conditions. One standard setup: - Three circles of radius 1 centered at the vertices of an equilateral triangle of side length 2 - These are mutually tangent

Visualization: The resulting fractal should show nested circles filling the gaps in a self-similar pattern, creating a beautiful packing.

2.9.3.1. Optional: Grid of Julia Sets!

Divide the plane into a small rectangular grid, inside each grid draw the julia set for the complex number at its center.

2.10. Looking Ahead

Tomorrow we'll continue with **fractals and hyperbolic geometry**. We'll start by extending our folding techniques to create the Sierpinski carpet, then dive into hyperbolic space with multiple models (Poincaré disk, upper half-plane, band model) and create beautiful tilings in non-Euclidean geometry.

Make sure you're comfortable with: - Complex number iteration and the Mandelbrot/Julia algorithms - Geometric folding operations (reflection via `abs()`, scaling) - Using iteration count for coloring - Circle inversion (from today's homework—this will be crucial tomorrow!)

The circle inversion you implement today will be the foundation for tomorrow's hyperbolic tilings, so take your time understanding how it works geometrically.

3. Day 3: Fractals and Hyperbolic Geometry

3.1. Overview

Today we complete our exploration of 2D fractals and then venture into hyperbolic space. We'll extend yesterday's triangle folding to create the Sierpinski carpet, then implement hyperbolic tilings in the upper half-plane model. Finally, we'll transform our work into the Poincaré disk model to see the same tiling from a different perspective.

By the end of today, you'll have working implementations of hyperbolic geometry computations and be able to create beautiful tilings in non-Euclidean space.

3.2. Sierpinski Carpet via Box Folding

3.2.1. From Triangle to Square

Yesterday we created the Sierpinski triangle by folding across the edges of an equilateral triangle. The Sierpinski carpet applies the same principle to a square, using axis-aligned reflections.

3.2.2. The Algorithm

Starting with a square domain (say $[-1, 1]^2$), we: 1. Fold using `abs()` to create 4-fold symmetry (map all quadrants to first quadrant) 2. Scale by factor of 3 3. Remove the middle square 4. Repeat

After n iterations, we've subdivided the square into a $3^n \times 3^n$ grid and removed all middle squares at every scale.

3.2.3. Implementation

```
vec2 sierpinskiCarpetFold(vec2 p, int iterations, out int removed) {
    removed = 0;

    for(int i = 0; i < iterations; i++) {
        // Fold to first quadrant
        p = abs(p);
```

```
        // Scale by 3, shift to center
        p = p * 3.0 - vec2(1.0);

        // Check if we're in the middle square (to be removed)
        // After scaling and shifting, the middle square is roughly centered
        if(abs(p.x) < 1.0 && abs(p.y) < 1.0) {
            removed = 1;
        }
    }

    return p;
}
```

Note: The exact check for "removed" regions depends on the coordinate setup. After the transformation `p * 3.0 - 1.0`, we're centering each sub-square. The middle third is the region we're removing.

3.2.4. Visualization

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 p = [coordinate setup];

    int removed;
    vec2 p_folded = sierpinskiCarpetFold(p, 6, removed);

    vec3 color = removed == 1 ? vec3(0.0) : vec3(1.0);

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

The characteristic Sierpinski carpet should emerge: a square with self-similar removed regions at all scales.

3.3. Hyperbolic Geometry: Models and Metrics

3.3.1. The Upper Half-Plane Model

The upper half-plane model $\mathbb{H}^2$ consists of complex numbers with positive imaginary part:

$$\mathbb{H}^2 = \{z \in \mathbb{C} : \text{Im}(z) > 0\}$$

The hyperbolic metric is:

$$ds^2 = \frac{dx^2 + dy^2}{y^2} = \frac{|dz|^2}{(\text{Im}(z))^2}$$

This metric "blows up" as we approach the real axis (the boundary at infinity), making the geometry shrink near $y = 0$.

3.3.2. Geodesics in the Upper Half-Plane

Geodesics (hyperbolic "straight lines") in $\mathbb{H}^2$ are: 1. Vertical lines $\{x = c\}$ for constant c 2. Semicircles centered on the real axis, perpendicular to it

For a semicircle of radius R centered at $(c, 0)$, the equation is:

$$(x - c)^2 + y^2 = R^2, \quad y > 0$$

3.3.3. Hyperbolic Distance

The distance between two points $z, w \in \mathbb{H}^2$ is:

$$d(z, w) = \operatorname{arcosh}\left(1 + \frac{|z - w|^2}{2 \cdot \operatorname{Im}(z) \cdot \operatorname{Im}(w)}\right)$$

In GLSL:

```
float hyperbolicDistance(vec2 z, vec2 w) {
    float diff2 = dot(z - w, z - w);  // |z - w|²
    float denom = 2.0 * z.y * w.y;    // 2·Im(z)·Im(w)
    return acosh(1.0 + diff2 / denom);
}
```

Note: GLSL may not have `acosh` built-in. Use: `acosh(x) = log(x + sqrt(x*x - 1.0))`.

3.3.4. The Poincaré Disk Model

The Poincaré disk model consists of the interior of the unit disk:

$$\mathbb{D}^2 = \{z \in \mathbb{C} : |z| < 1\}$$

The metric is:

$$ds^2 = \frac{4(dx^2 + dy^2)}{(1 - |z|^2)^2}$$

Geodesics are circular arcs perpendicular to the unit circle (or diameters).

3.3.5. Converting Between Models

The Cayley transform maps the upper half-plane to the disk:

$$w = \frac{z - i}{z + i}$$

The inverse is:

$$z = i\frac{1 + w}{1 - w}$$

In GLSL (using complex arithmetic from Day 2):

```
vec2 uhpToDisk(vec2 z) {
    // w = (z - i) / (z + i)
    vec2 numerator = z - vec2(0.0, 1.0);      // z - i
    vec2 denominator = z + vec2(0.0, 1.0);    // z + i
    return cdiv(numerator, denominator);
}

vec2 diskToUHP(vec2 w) {
    // z = i(1 + w) / (1 - w)
    vec2 numerator = vec2(0.0, 1.0) + w;      // i(1 + w) = cmul(i, 1+w)
    numerator = vec2(-numerator.y, numerator.x);  // Multiply by i
    vec2 denominator = vec2(1.0, 0.0) - w;    // 1 - w
    return cdiv(numerator, denominator);
}
```

Where `cdiv` is complex division (you'll need to implement this - see homework or use the formula $\frac{a+bi}{c+di} = \frac{(a+bi)(c-di)}{c^2+d^2}$).

3.4. The (2,3,∞) Triangle

3.4.1. Why This Triangle?

The $(2, 3, \infty)$ triangle has angles $\pi/2$, $\pi/3$, and 0 (the "ideal vertex" at infinity). In the upper half-plane, this triangle has particularly nice edges: - One edge is the unit semicircle centered at the origin: $x^2 + y^2 = 1$, $y > 0$ - Two edges are vertical lines at $x = \pm c$ for some constant c

For the $(2, 3, \infty)$ triangle, with the right angle at the origin and the $\pi/3$ angles at the two vertical edges, we have $c = \sqrt{3}$ (this can be derived from hyperbolic trigonometry, but we'll just use it).

Actually, a cleaner setup: let's use the triangle with: - Geodesic from -1 to 1 (unit semicircle centered at origin) - Vertical geodesic at $x = 1$ - Vertical geodesic at $x = -1$

This creates a fundamental domain bounded by these three geodesics. The angles where they meet determine the triangle group.

[Note: Need to work out the exact setup here - which specific triangle gives clean angles? The (2,3,∞) triangle or a different choice like (2,4,∞)? Let me provide the framework and you can adjust the specific parameters.]

3.4.2. Checking if a Point is in the Triangle

For our triangle with: - Bottom edge: unit semicircle - Left edge: vertical line at $x = -1$ - Right edge: vertical line at $x = 1$

```
bool inTriangle(vec2 p) {
    // Above the semicircle: x² + y² > 1
    bool aboveSemicircle = dot(p, p) > 1.0;

    // Between vertical lines: -1 < x < 1
    bool betweenLines = (p.x > -1.0) && (p.x < 1.0);

    return aboveSemicircle && betweenLines;
}
```

3.4.3. Reflection Across Geodesics

Reflection across a vertical line $x = c$:

$$\text{reflect}(x + iy) = (2c - x) + iy$$

```
vec2 reflectVertical(vec2 p, float c) {
    return vec2(2.0 * c - p.x, p.y);
}
```

Reflection across the unit semicircle (circle inversion from yesterday!): Circle inversion with respect to a circle of radius R centered at (c_x, c_y) is:

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

For our unit circle centered at origin:

```
vec2 reflectCircle(vec2 p) {
    return p / dot(p, p);  // Inversion through unit circle
}
```

3.4.4. Folding into the Fundamental Domain

```
vec2 foldToTriangle(vec2 p, int maxIter, out int foldCount) {
    foldCount = 0;

    for(int i = 0; i < maxIter; i++) {
        bool folded = false;

        // Reflect across left vertical line if needed
        if(p.x < -1.0) {
            p = reflectVertical(p, -1.0);
            folded = true;
        }

        // Reflect across right vertical line if needed
        if(p.x > 1.0) {
            p = reflectVertical(p, 1.0);
            folded = true;
        }

        // Reflect across semicircle if needed
        if(dot(p, p) < 1.0) {
            p = reflectCircle(p);
            folded = true;
        }

        if(folded) foldCount++;
        else break;  // In fundamental domain
    }

    return p;
}
```

3.4.5. Visualization: Hyperbolic Tiling

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup - map to upper half-plane
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 4.0;  // Scale to see interesting region
    uv.x *= iResolution.x / iResolution.y;

    // Map to upper half-plane (shift up so y > 0)
    vec2 p = uv + vec2(0.0, 1.5);  // Ensure we're above real axis

    // Fold to fundamental domain
    int foldCount;
```

```
    vec2 p_folded = foldToTriangle(p, 20, foldCount);

    // Color based on fold count
    float t = float(foldCount) / 10.0;
    vec3 color = palette(t);  // Use palette function from Day 2

    // If we're in the fundamental triangle, use different color
    if(inTriangle(p_folded)) {
        color = mix(color, vec3(1.0, 1.0, 1.0), 0.3);
    }

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

The result should be a beautiful hyperbolic tiling - the plane tessellated by copies of our fundamental triangle!

3.5. Visualizing in the Poincaré Disk

Now we'll see the same tiling in a different model. The key insight: we can do all our computations in the upper half-plane, then convert the final coordinates to the disk for display.

3.5.1. By Precomposition

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup - map to disk
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 2.0;  // Map to [-1, 1]
    uv.x *= iResolution.x / iResolution.y;

    // Only render inside unit disk
    if(length(uv) >= 1.0) {
        fragColor = vec4(0.0, 0.0, 0.0, 1.0);
        return;
    }

    // Convert disk coordinates to upper half-plane
    vec2 p = diskToUHP(uv);

    // Now do all computations in UHP
    int foldCount;
    vec2 p_folded = foldToTriangle(p, 20, foldCount);
```

```
    // Color based on fold count
    float t = float(foldCount) / 10.0;
    vec3 color = palette(t);

    if(inTriangle(p_folded)) {
        color = mix(color, vec3(1.0, 1.0, 1.0), 0.3);
    }

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

The tiling now appears in the Poincaré disk - the same mathematical object, but displayed in a different model. Notice how the triangles near the boundary appear compressed (they're the same hyperbolic size, but Euclidean distances shrink near the boundary).

3.6. Distance to Geodesics

For homework, you'll want to draw geodesic boundaries. Here's the framework:

3.6.1. Distance to a Vertical Geodesic

For a vertical line $x = c$, the hyperbolic distance from a point (x, y) to the line is:

$$d = \text{arcosh}\left(\frac{|x - c|}{\text{some formula involving } y}\right)$$

[Placeholder: exact formula for distance to vertical geodesic]

3.6.2. Distance to a Semicircular Geodesic

For a semicircle of radius R centered at $(c, 0)$, the distance calculation is more involved. One approach: 1. Find the closest point on the semicircle to p 2. Compute hyperbolic distance between p and that closest point

[Placeholder: geodesic distance formula or algorithm]

Alternatively, you can find formulas in hyperbolic geometry references, or derive them from the metric.

3.7. Summary

Today we covered:

1. **Sierpinski carpet**: Box folding with axis-aligned symmetry creates 2D fractals
2. **Hyperbolic geometry models**: Upper half-plane and Poincaré disk, with metrics and geodesics
3. **Coordinate transformations**: Cayley transform connecting the two models
4. **Hyperbolic triangle tiling**: Using reflections (including circle inversion) to tile hyperbolic space
5. **Multiple representations**: Same tiling visualized in different models

Key insights: - Geometric algorithms transfer to non-Euclidean spaces with the right distance/geodesic formulas - Circle inversion (from Day 2) is fundamental to hyperbolic geometry - The same mathematical object looks different in different models - Folding algorithms work in any geometry with the appropriate reflection operations

---

3.8. Homework

3.8.1. Required: Sierpinski Carpet

Implement the Sierpinski carpet fractal using box folding.

Task:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 p = [coordinate setup];

    int iterations = 6;
    for(int i = 0; i < iterations; i++) {
        // Fold to first quadrant
        p = abs(p);

        // Scale by 3 and recenter
        p = p * 3.0 - vec2(1.0);
    }

    // Check if p is in a "removed" region
    // [Implement this check]

    // Color accordingly
}
```

Expected output: The characteristic Sierpinski carpet pattern with removed squares at all scales.

Experiments: - Different iteration counts (watch detail increase) - Color by iteration depth rather than binary in/out - Animate the zoom level with `iTime`

3.8.2. Required: Drawing Geodesics and Hyperbolic Disks

Implement visualization of basic hyperbolic objects to understand the geometry.

Part 1: Draw several geodesics in the upper half-plane - Vertical lines at various x values (these are easy!) - Semicircles with different centers and radii - Use a thickness threshold on the implicit equation to draw them

Part 2: Draw hyperbolic disks (circles in hyperbolic metric) - Pick a point $z_0 \in \mathbb{H}^2$ - Draw the set $\{z : d_{\text{hyp}}(z, z_0) < R\}$ for some radius R - These appear as Euclidean circles, but positioned/sized according to hyperbolic metric - Draw several disks at different locations

Part 3: Visualize how geometry changes - Draw a grid of hyperbolic disks of the same hyperbolic radius - Observe how they appear smaller (in Euclidean sense) near the boundary - This demonstrates the "shrinking" effect of the hyperbolic metric

Optional enhancement: Repeat in Poincaré disk model by precomposition.

3.8.3. Required: Drawing Triangle Edges and Vertices

Enhance your hyperbolic tiling by drawing the triangle boundaries.

Task: - Compute the hyperbolic distance from each point to the three geodesics forming your triangle boundary - If distance is less than some threshold, color the point as an edge - Similarly, compute distance to the three vertices and draw them as points

Implementation hints:

```
// Distance to vertical line x = c
float distToVertical(vec2 p, float c) {
    // [Implement using hyperbolic distance formula]
}

// Distance to semicircle (center, radius)
float distToSemicircle(vec2 p, float center, float radius) {
    // [Implement - find closest point on semicircle, compute distance]
}

// In main shader:
float d1 = distToVertical(p, -1.0);
float d2 = distToVertical(p, 1.0);
float d3 = distToSemicircle(p, 0.0, 1.0);
```

```
float edgeThickness = 0.05;
bool onEdge = (d1 < edgeThickness) || (d2 < edgeThickness) || (d3 < edgeThickness);
```

Expected output: Your tiling with clearly visible triangle boundaries, making the tessellation structure explicit.

3.8.4. Required: Model Conversion and Möbius Transformation

Work with different representations of hyperbolic space.

Part 1: Convert your tiling to Poincaré disk - You already have `uhpToDisk()` and `diskToUHP()` functions - Create a shader that displays your (2,3,∞) tiling in the disk model - Compare the visual appearance to the upper half-plane version

Part 2: Apply a Möbius transformation - A Möbius transformation has the form $z \mapsto \frac{az+b}{cz+d}$ with $ad - bc = 1$ - Choose an interesting transformation (or try: $z \mapsto z + 1$ to translate, or $z \mapsto 2z$ to scale) - Apply it to your tiling and observe the result - **Key insight:** Möbius transformations are isometries of hyperbolic space, so they permute the tiles but preserve the geometry

Implementation:

```
vec2 mobius(vec2 z, vec2 a, vec2 b, vec2 c, vec2 d) {
    // w = (az + b) / (cz + d)
    vec2 num = cadd(cmul(a, z), b);
    vec2 den = cadd(cmul(c, z), d);
    return cdiv(num, den);
}

// In shader, before folding:
p = mobius(p, vec2(2.0, 0.0), vec2(0.0, 0.0), vec2(0.0, 0.0), vec2(1.0, 0.0));  // Scale by 2
```

3.8.5. Optional: Advanced Hyperbolic Explorations

Choose one or more:

1. **Different triangle groups**: Implement (2,4,6), (3,3,3), or the famous (2,3,7) triangle. Each creates a different tiling pattern.
2. **Klein model**: The Klein model is another representation where geodesics are straight lines. Convert via the Cayley transform from Poincaré disk: $w = \frac{2z}{1+|z|^2}$
3. **Decorated tiles**: Add patterns inside each triangle (like Escher's Circle Limit prints). Use the barycentric coordinates within each fundamental domain.
4. **Conformal mapping art**: Use complex analysis to map the disk to other regions (strip, annulus, etc.) and visualize hyperbolic geometry in these exotic spaces.

3.9. Looking Ahead

Tomorrow we move to 3D! We'll learn raymarching and signed distance functions to render implicit surfaces. The geometric intuition from folding and distance computations will carry over, but now in three dimensions with lighting and shading.

Make sure you're comfortable with: - Distance computations (Euclidean and hyperbolic) - Reflection operations (these generalize to 3D) - Iterative algorithms for geometric structures

The 3D rendering techniques we'll learn are the culmination of everything so far: distance fields, iterative marching, geometric transformations, and real-time GPU computation.

4. Day 4

5. Day 5a

6. Day 5bs

A. GLSL

B. Debugging