GPU-Accelerated Mathematical Illustration

An Introduction to Shader Programming

Steve Trettel

December 2025

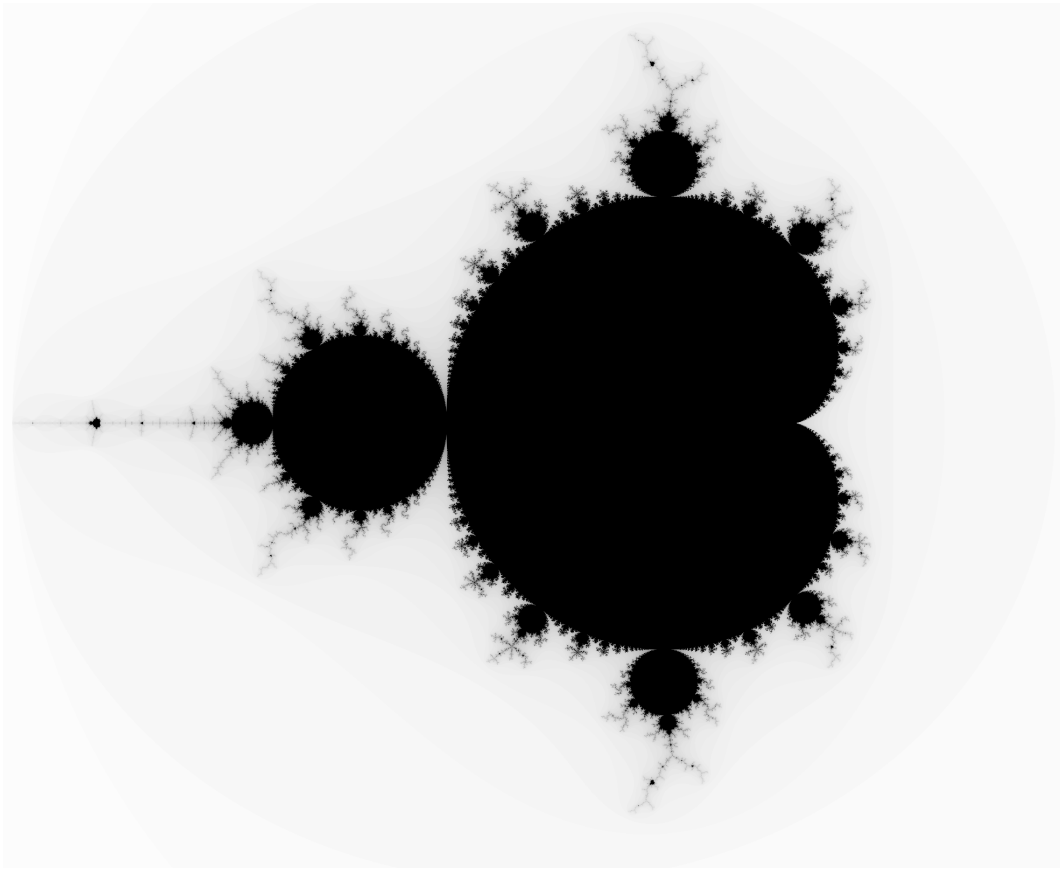
Table of contents

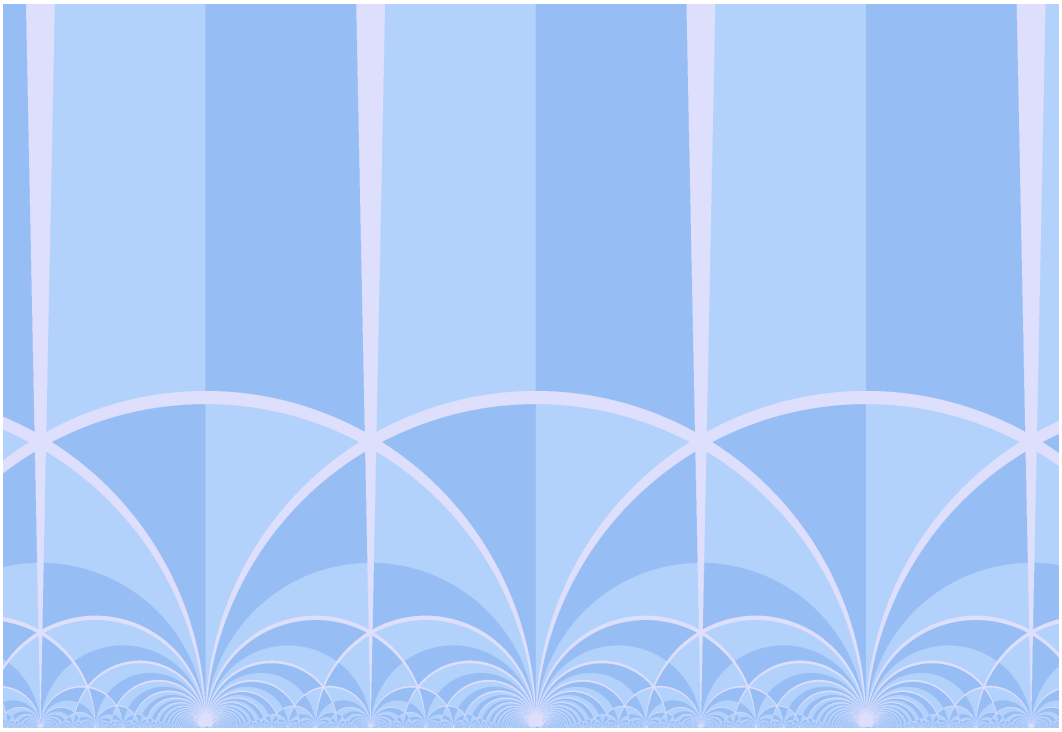
About

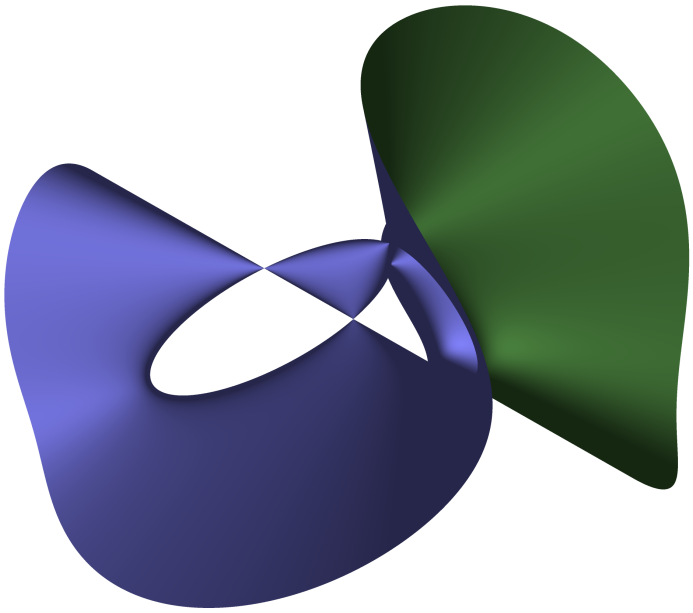
This mini-course introduces shader programming as a tool for mathematical illustration and exploration. Shaders are programs that run in parallel on the GPU, making them exceptionally fast for visualization tasks. We'll learn to write code that "reads like mathematics" using Shadertoy, a beginner-friendly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







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 homework

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

Audience: Graduate students, postdocs, and faculty in mathematics

---

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 implicit curves

In-Class Content

- **Mathematical framing**: Shaders as parallel functions computing colors for all pixels simultaneously
- **GLSL basics**: Syntax, vector types, built-in functions
- **Coordinate systems**: Centering, normalizing, aspect ratio correction
- **Conditional coloring**: Half-planes and regions defined by inequalities
- **Distance fields**: Circles, filled and outlined
- **Repetition**: Using `mod()` for grids and patterns
- **Implicit curves**: Rendering curves defined by $F(x, y) = 0$

Homework

Required: Parabola graphing calculator - Draw coordinate axes - Plot $y = ax^2 + bx + c$ with customizable coefficients - Make it robust for various parameter values

Optional #1: Animated curve family (vary parameters with time)

Optional #2: Beautiful tiling pattern using `mod()`

Day 2: Complex Dynamics and Iterated Inversions

Learning Objectives

- Implement complex number arithmetic in GLSL
- Render the Mandelbrot set through escape-time iteration
- Master circle inversion as a conformal transformation
- Use structs to organize geometric data
- Generate the Apollonian gasket through iterated inversions

In-Class Content

- **Complex arithmetic**: Addition, multiplication, division, conjugation
- **Mandelbrot set**:
  - Iteration $z_{n+1} = z_n^2 + c$ with $z_0 = 0$
  - Escape-time algorithm
  - Smooth coloring and palettes
- **Circle inversion**:
  - Mathematical definition and properties
  - Conformal mapping (preserves angles, maps circles to circles/lines)
  - Implementation and visualization
- **Structs in GLSL**: Organizing circle data (center, radius)
- **Apollonian gasket**:
  - Three mutually tangent circles
  - Iterated inversions generate fractal structure
  - Coloring by escape time or basin of attraction

Homework

Required: Julia sets - Implement for fixed c, varying initial z_0 - Explore parameter space (try different values of c) - Optional: animate c to watch morphing

Optional: Schottky groups - Four or more disjoint circles - Alternating inversion patterns - Create intricate nested structures

---

Day 3: Geometric Tilings in Euclidean and Hyperbolic Space

Learning Objectives

- Create Euclidean triangle tilings through reflection
- Understand hyperbolic geometry models (upper half-plane, Poincaré disk)
- Implement hyperbolic triangle tilings using circle inversion
- Convert between different hyperbolic models

In-Class Content

- **Euclidean triangle tiling**:
  - Fundamental domain (equilateral triangle)
  - Reflection across edges
  - Iterative folding algorithm
  - Coloring by reflection count
- **Hyperbolic geometry introduction**:
  - Upper half-plane model: $\mathbb{H}^2 = \{z : \text{Im}(z) > 0\}$
  - Hyperbolic metric: $ds^2 = \frac{dx^2+dy^2}{y^2}$
  - Geodesics: vertical lines and semicircles
  - Hyperbolic distance formula
- **Poincaré disk model**:
  - Unit disk representation
  - Cayley transform between models
- **Hyperbolic triangle tiling**:
  - $(2, 3, \infty)$ triangle with nice edges
  - Reflection across vertical geodesics (simple)
  - Reflection across circular geodesics (circle inversion!)
  - Folding algorithm
  - Visualization in both models

Homework

Required #1: Draw geodesics and hyperbolic disks - Visualize geodesics in upper half-plane - Draw hyperbolic disks (constant hyperbolic distance) - Observe metric distortion

Required #2: Draw triangle edges and vertices - Compute distance to geodesics - Render triangle boundaries explicitly - Mark vertices

Required #3: Model conversion and Möbius transformations - Convert tiling to Poincaré disk - Apply Möbius transformations (isometries) - Observe how tiling transforms

Optional: - Different triangle groups (e.g., $(2, 3, 7)$ for Escher-like tilings) - Klein model (geodesics become straight lines) - Decorated tiles (Escher-style patterns)

Day 4: Introduction to 3D Rendering

Learning Objectives

- Set up camera and generate rays from pixels
- Implement analytical ray-object intersection
- Learn the raymarching algorithm and signed distance functions
- Apply basic lighting (diffuse shading)

In-Class Content

- **Camera and ray setup**:
  - Pinhole camera model
  - Ray generation from pixel coordinates
  - Field of view control
- **Analytical intersections**:
  - Ray-sphere: solve quadratic equation
  - Compute surface normals analytically
  - Ray-torus: implicit equation and gradient
  - Bisection method for root-finding
- **Lighting introduction**:
  - Surface normals
  - Diffuse lighting: dot product with light direction
  - Seeing 3D structure through shading
- **Motivation for raymarching**:
  - Analytical methods don't scale
  - Complex surfaces need flexible approach
- **Signed Distance Functions (SDFs)**:
  - Definition and properties
  - SDFs for primitives: sphere, box, plane, torus
  - Distance as bound for safe marching
- **Raymarching algorithm**:
  - Sphere tracing: march by SDF value
  - Stopping conditions
  - Scene composition (minimum distance)
- **Normal estimation**:
  - Gradient via finite differences
  - Estimating partial derivatives
  - Same lighting applied to raymarched objects
- **Scene progression**:
  - Single sphere
  - Two spheres
  - Sphere and torus

Homework

Required: Algebraic variety rendering - Choose polynomial implicit surface (degree 3 or 4) - Implement root-finding (bisection or Newton's method) - Compute gradient for normals - Optional: bounding sphere optimization

Optional: - Specular lighting (Phong model) - Rotation matrices for object transformation - Complex multi-object scenes

---

Day 5: Choose Your Adventure

The final day will be determined based on pacing, student interest, and energy levels. Two complete lectures are prepared:

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 (Menger sponge)
- Build sophisticated scenes from simple primitives

In-Class Content

- **Domain operations**:
  - Repetition: `mod()` for infinite object grids
  - Symmetry: `abs()` for mirror planes
  - Polar repetition for radial patterns
  - Zero computational 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
- **Menger sponge**:
  - Box folding in 3D
  - Axis-aligned operations
  - Iterated subdivision
  - Connection to 2D fractals
- **Scene building**:
  - Combining techniques
  - Architectural structures
  - Infinite repeated patterns

Homework

Required: Creative scene building - Build complex scene using domain ops and booleans - Experiment with combinations - Focus on mathematical or aesthetic interest

Optional: Sierpinski tetrahedron - Implement via 3D folding (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
  - Multi-pass rendering
  - Simple example: conditional coloring based on buffer
- **Edge detection and the Laplacian**:
  - Discrete Laplacian stencil (5-point or 9-point)
  - Sampling neighboring pixels
  - Spatial derivatives on grids
- **The heat equation**:
  - Mathematical formulation: $u_t = \alpha \nabla^2 u$
  - Applying Laplacian for diffusion
  - Time-stepping: `u_new = u_old + dt * ? * laplacian(u_old)`
  - Initial conditions (e.g., heat in a fractal region)
  - Watching patterns blur and diffuse
- **Boundary conditions**:
  - Zero boundaries (edges set to 0)
  - Avoiding wrap-around
- **Timestep stability**:
  - CFL condition (briefly mentioned)
  - Providing stable `dt` value

Homework

Required: Interactive heat equation or reaction-diffusion - Option 1: Heat source at mouse position, watch it diffuse - Option 2: Gray-Scott reaction-diffusion (pattern formation)

Optional: Wave equation - Requires two buffers (current and previous state) - Implement $u_{tt} = c^2 \nabla^2 u$ - Watch waves propagate and reflect

---

Resources and Further Exploration

Shadertoy

- Main site: https://www.shadertoy.com
- Community examples and tutorials
- GLSL documentation

References

- Complex dynamics and fractals
- Hyperbolic geometry and tilings
- Signed distance functions (Inigo Quilez: https://iquilezles.org/articles/distfunctions/)
- GPU computing for scientific visualization

Advanced Topics

- Path tracing and global illumination
- Non-Euclidean ray tracing
- Real-time denoising
- More complex PDEs and simulations

Assessment Philosophy

This is a workshop-style course focused on skill development. Success means: - Completing required homework to keep pace - Experimenting with optional problems based on interest - Developing intuition for when shader programming is appropriate - Leaving with working code templates for future projects

Philosophy: Getting something working and understanding it is more valuable than perfect, polished results. The goal is to build practical skills and mathematical intuition, not to create production-quality graphics.

Schedule Summary

Day	Topic	Key Concepts
1	Shader Basics	Coordinates, distance fields, implicit curves
2	Complex Dynamics	Mandelbrot, circle inversion, Apollonian gasket
3	Geometric Tilings	Euclidean and hyperbolic tilings, models
4	3D Rendering	Raymarching, SDFs, lighting
5	Advanced (flexible)	Domain ops + fractals OR buffers + PDEs

Each day: 1 hour lecture + ~1.5 hours homework Total: 5 lectures, 10-12 programming assignments (5 required, 5-7 optional)

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.

> **Roadmap for Today**
>
> We'll build up shader programming in layers:
>
> 1. **Core concept**: Shaders as parallel functions (What is a Shader?)
> 2. **Setup**: Coordinate systems and GLSL syntax (First Shader, Coordinate Systems)
> 3. **Basic techniques**: Conditional coloring and distance fields (Half-Planes, Distance Fields)
>
> 4. **Repetition**: Grids via modular arithmetic (Grids and Repetition)
> 5. **Application**: Implicit curves (Implicit Curves)
>
> Each section builds on the previous, so if something feels unclear, it's worth revisiting earlier material before moving forward.

1.2. What is a Shader?

1.2.1. Mathematical Perspective

A shader is fundamentally a function

$$f : \mathbb{R}^2 \times \mathbb{R} \times \cdots \to [0,1]^4$$

that maps pixel coordinates (x, y), time t, and potentially other parameters to RGBA color values. For today, we'll focus on the spatial dependence—thinking of the shader as a function $f : \mathbb{R}^2 \to [0,1]^4$ that assigns a color to each point in the plane. The domain $[0,1]^4$ represents the red, green, blue, and alpha (transparency) channels, each normalized to the unit interval.

Here's the magic: 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. We're not looping over pixels one at a time—we're computing them all at once!

This is completely different from how you might write mathematical visualization code in, say, Python or MATLAB. There you'd have nested loops:

```
for x in range(width):
    for y in range(height):
        color[x,y] = f(x, y)
```

With shaders, there are no loops. You write the function *f*, and the GPU just *does it* everywhere at once. This parallelism is what makes shader-based visualization absurdly fast—fast enough to render complex mathematical objects in real-time, responding to your mouse, animating smoothly, all at 60fps.

The computational model is fundamentally different: in traditional CPU programming you have sequential control flow with explicit loops, while in shader programming you express computation as a pure mathematical function that gets evaluated independently at every pixel. The GPU architecture is specifically designed for this kind of massively parallel workload—it has thousands of small processors that can each evaluate your function simultaneously. This is why a relatively modest GPU can outperform even a powerful CPU on graphics tasks by orders of magnitude.

> **i** Why is this called a "shader"?
>
> Historically, these programs were used for *shading* 3D objects—computing how light interacts with surfaces to create realistic images. The name stuck even though nowadays we use them for all sorts of parallel computation, far beyond just lighting calculations. We're going to use shaders to render implicit curves, fractals, hyperbolic tilings, and solve PDEs—none of which have anything to do with "shading" in the traditional sense!

1.2.2. Why Shadertoy?

Shadertoy is a web-based platform that handles all the annoying GPU setup for you. Normally, working with shaders requires writing a bunch of boilerplate code: setting up OpenGL contexts, compiling shader programs, managing buffers, handling the render loop—it's a pain. Shadertoy abstracts all of that away. You write a single function, hit compile, and instantly see your results.

Shadertoy launched in 2013, created by Pol Jeremias-Vila and Íñigo Quílez (we'll see more of Íñigo's work throughout this week—he's pioneered many shader techniques). Before platforms like Shadertoy, shader programming required managing the entire OpenGL or DirectX pipeline yourself—compiling shaders, linking programs, setting up vertex buffers, managing textures. It was the domain of graphics programmers, not mathematicians.

The genius of Shadertoy was recognizing that for many visualizations, you don't need that complexity. Just give people a function to fill in, handle the boilerplate invisibly, and suddenly shaders become accessible to anyone. It's democratized GPU programming in much the same way that Python notebooks democratized scientific computing—lower the barrier to entry, and a whole new community emerges.

The platform provides several built-in **uniforms** (read-only global variables that are the same for all pixels):

- `iResolution`: screen resolution as a `vec3` (width, height, pixel aspect ratio)
- `iTime`: elapsed time in seconds since the shader started
- `iMouse`: mouse position and click state as a `vec4`

We'll use these throughout the week to create animated, interactive mathematical visualizations.

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` giving the (x, y) position
- `fragColor`: the output color we need to set, as a `vec4` giving the (r, g, b, a) color

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. Not very exciting—but it's a start!

1.3.3. GLSL Syntax Basics

Before we go further, let's talk about some essential GLSL conventions. If you're coming from Python or MATLAB, a few things will feel different:

! GLSL Syntax Rules

Semicolons are required. Every statement must end with a semicolon. This is not Python! Forget one and your shader won't compile.
Floating point literals: Write `1.0` not `1` for floating point values. GLSL is very picky about types—if you write `1`, it's an integer, and mixing types causes errors. Get in the habit of always writing the `.0`.
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 equivalently `v.r`, `v.g` (same thing, different naming convention—use whichever makes sense for your context). Even better, you can rearrange components: `v.yx` swaps the coordinates, `v.xxx` repeats the x-component three times. This is incredibly useful!

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 this to $[0, 1]$ with the affine transformation $t \mapsto \frac{1}{2}(1 + t)$. The screen now pulses between black and red!

This pattern—`0.5 + 0.5 * sin(...)`—comes up constantly when animating. It's the standard way to turn a sinusoid into something that stays in the range $[0, 1]$. You'll use this so often it becomes second nature.

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`

This is fine if you're thinking about pixels, but for mathematical work we want something more natural: coordinates centered at the origin, normalized (not in pixels), and with aspect ratio handled correctly so that squares actually look square!

1.4.2. Centered, Normalized Coordinates

Here's the standard transformation we'll use in every shader:

```
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;

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

Let's understand this transformation rigorously. We're composing four maps. Let's write $w =$ `iResolution.x` and $h =$ `iResolution.y` for the width and height in pixels.

Step 1: Normalization

$$T_1 : [0, w] \times [0, h] \to [0, 1]^2, \quad T_1(x, y) = \left(\frac{x}{w}, \frac{y}{h}\right)$$

This makes our coordinates resolution-independent—the same shader code works whether your screen is 1920×1080 or 800×600. A point that's halfway across the screen is $(0.5, v)$ regardless of how many pixels wide the screen actually is.

Step 2: Centering

$$T_2 : [0, 1]^2 \to [-\tfrac{1}{2}, \tfrac{1}{2}]^2, \quad T_2(u, v) = (u - \tfrac{1}{2}, v - \tfrac{1}{2})$$

Now the origin is at the center of the screen, which is much more natural for mathematical work. We can think about positive and negative coordinates, circles centered at the origin, and so on.

Step 3: Aspect correction

$$T_3(u, v) = \left(\frac{w}{h} \cdot u, v\right)$$

This is crucial! Without it, circles would appear as ellipses on non-square screens. The aspect ratio w/h stretches the x-coordinate so that one unit in x corresponds to the same screen distance as one unit in y. On a typical 16:9 display ($w/h \approx 1.78$), this means the x-axis spans a wider range than the

y-axis—as it should to maintain equal scaling. A circle of radius r will actually appear circular on screen, not squashed.

Step 4: Scaling to viewing window

$$T_4(u, v) = s \cdot (u, v)$$

Finally, we scale by whatever factor gives us the mathematical viewing window we want. If we choose $s = 4$, then on a 16:9 screen our coordinates range roughly from $[-3.56, 3.56]$ in x and $[-2, 2]$ in y—notice the x-range is wider to match the screen aspect ratio.

The composition $T_4 \circ T_3 \circ T_2 \circ T_1$ is our complete coordinate transformation, taking us from raw pixel coordinates to a centered, aspect-corrected mathematical coordinate system.

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."

> **The coordinate transformation boilerplate**
>
> You'll do these first few lines in almost every shader you write. It becomes muscle memory quickly! Some people like to wrap it in a function, but for these lectures we'll just write it out each time so the transformation is explicit and you can modify it when needed.

1.4.3. Visualizing Coordinates

Let's verify our coordinate system is working by coloring pixels according to their position:

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

    // 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. If you don't see this, something went wrong in your coordinate setup! This is a good debugging technique—whenever you're unsure about your coordinates, visualize them directly as colors.

1.5. Conditional Coloring: Half-Planes

1.5.1. The Concept

Given a linear function $L(x, y) = ax + by + c$, 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—the regions where the function is negative versus positive.

The line itself is the zero set: $\{(x, y) : L(x, y) = 0\}$. This is the boundary between the two regions.

1.5.2. Implementation

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

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

    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. The syntax `(condition) ? value_if_true : value_if_false` should be familiar if you've programmed in C, Java, or JavaScript.

1.5.3. The Step Function

GLSL also provides `step(edge, x)` which returns 0 if $x <$ edge and 1 otherwise. The name comes from its graph—a step function in the calculus sense, jumping discontinuously from 0 to 1 at the edge value. This is useful for writing cleaner code without explicit conditionals:

```
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: $(1 - t)a + tb$. So when `s = 0` we get pure red, when `s = 1` we get pure blue. The `mix` function is one of GLSL's most useful tools—you'll use it constantly for blending colors, smoothly transitioning between values, and implementing linear interpolation in all sorts of contexts.

> **Why use `step` instead of the ternary operator?**
>
> Both work fine! The ternary operator `?:` is more explicit and familiar if you know C-like languages. But `step` and `mix` are more idiomatic in shader code, and they compose nicely

with other functions. As you write more shaders, you'll develop a feel for which style is clearer in each situation. For now, use whichever makes sense to you.

1.5.4. Arbitrary Half-Planes

For a general line $ax + by + c = 0$, we just evaluate the corresponding linear function:

```
float a = 1.0, b = 1.0, c = 0.0;
float L = a * p.x + b * p.y + c;
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, b, and c to see different line orientations and positions. The line itself is where $L = 0$, and we're coloring the two sides differently. Notice that scaling (a, b, c) by a positive constant doesn't change the geometry—it's the zero set that matters, not the specific values of the function away from zero.

1.6. Distance Fields and Circles

1.6.1. Distance to Center

The distance from a point $p = (x, y)$ to the origin is just the usual Euclidean distance:

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

In GLSL this is built-in:

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

The `length()` function computes the Euclidean norm of a vector. It works for `vec2`, `vec3`, `vec4`—whatever you need. Under the hood it's computing the square root of the dot product of the vector with itself, but there's no need to write that out explicitly.

1.6.2. Filled Circle

A circle of radius r centered at the origin is the set $\{p : \|p\| < r\}$—just points whose distance from the origin is less than r. So to color the inside versus outside of a circle, we just compare distances:

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

    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);
}
```

That's it! This renders a yellow disk on a dark blue background. Every pixel computes its distance to the origin and decides whether it's inside or outside the circle. Simple, elegant, and fast—millions of distance calculations per frame, all happening in parallel.

1.6.3. Distance-Based Coloring

But we don't have to just make binary inside/outside decisions—we can use the distance value itself to create gradients and other effects. For example, we can make things fade out with distance:

```
float d = length(p);
float intensity = 1.0 - d / 2.0;  // Fades from 1 at center to 0 at distance 2
intensity = clamp(intensity, 0.0, 1.0);  // Keep it in [0, 1]
vec3 color = vec3(intensity);
fragColor = vec4(color, 1.0);
```

This creates a radial gradient—bright at the center, dark at the edges. The `clamp` function ensures we stay within $[0, 1]$ even if our formula would produce values outside that range. Distance fields like this are incredibly versatile: you can use them for smooth transitions, glowing effects, or (as we'll see on Day 4) as the foundation for 3D rendering!

Distance fields will become increasingly important as the week progresses. On Day 4, we'll use them as the foundation for **raymarching**—a technique for rendering 3D geometry without any triangles or polygons, purely by iteratively evaluating distance functions. The `length(p)` function we used for circles today generalizes to arbitrary implicit surfaces: $d(p) = $ "distance to the surface defined by $F(p) = 0$." It's a beautiful connection between analysis and computer graphics.

1.6.4. Circle Outline

What if we want to draw just the *boundary* of a circle—not the filled disk, but the thin curve itself? We need to check if the distance is *approximately equal* to the radius. Mathematically, we're coloring the set $\{p : |d(p) - r| < \epsilon\}$ where ϵ is a small thickness parameter:

```
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);
```

This draws a thin white annulus around the circle. Play with the `thickness` parameter to see how it affects the line width!

For a smoother, anti-aliased edge, GLSL provides `smoothstep`:

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

The `smoothstep(a, b, x)` function performs smooth Hermite interpolation. For $x \in [a, b]$, it returns

$$s(t) = 3t^2 - 2t^3 \quad \text{where } t = \frac{x-a}{b-a}$$

This is a cubic polynomial with $s(0) = 0$, $s(1) = 1$, and crucially $s'(0) = s'(1) = 0$—the zero derivatives at the endpoints mean it transitions smoothly without visible "kinks." For $x < a$ it returns 0, for $x > b$ it returns 1.

The result is anti-aliasing: instead of a hard transition at a single pixel, the edge is blurred over the interval $[a, b]$. For circle outlines, using `smoothstep(r - thickness, r + thickness, d)` creates a smooth transition zone of width $2 \cdot \text{thickness}$ around the target radius. This eliminates jagged edges and makes the circle look much nicer—especially important when you're creating publication-quality mathematical illustrations!

> **Anti-aliasing in shaders**
>
> The harsh cutoffs from using `<` or the ternary operator create jagged, pixelated edges—what computer graphics people call "aliasing" (the signal is being undersampled relative to its frequency content, creating artifacts). Functions like `smoothstep` give you smooth transitions over a few pixels, which is exactly what you want for anti-aliasing. We'll use this technique constantly: anywhere you have a sharp boolean decision, consider replacing it with `smoothstep` for smoother results.

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—it "folds" the entire real line into a finite interval over and over again.

More precisely, recall that $x \bmod T$ is the unique value in $[0, T)$ satisfying $x \equiv r \pmod{T}$—that is, $x = nT + r$ for some integer n. Geometrically, this takes the real line and wraps it into the interval $[0, T)$. Subtracting $T/2$ recenters this to $[-T/2, T/2]$.

In GLSL, `mod(x, T)` computes $x \bmod T$. This is one of the most powerful tools in shader programming!

1.7.2. Creating a Grid

To create a grid of repeated cells, we apply `mod` to our coordinates:

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

    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! The coordinate transformation `cell_p = mod(p + spacing/2.0, spacing) - spacing/2.0` ensures that `cell_p` is always in the range $[-\text{spacing}/2, \text{spacing}/2]$, and this range repeats forever. So every cell of the grid has identical coordinates, and therefore draws identical content.

Think about what just happened: we created infinitely many circles with exactly the same amount of computation as drawing a single circle! There's no loop over grid cells, no array of circle positions—the repetition comes purely from the coordinate transformation.

Compare this to how you might approach this in Python or MATLAB: you'd probably set up a nested loop over grid cells, compute the center of each cell, then draw a circle there. That's $O(n^2)$ work for an $n \times n$ grid. With shaders, it's $O(1)$ in the grid size—the cost is entirely in the number of *pixels*, not the number of *circles*. This is why shaders can render infinitely complex patterns at the same framerate as simple ones.

> **The power of `mod`**
>
> This computational efficiency through coordinate transformations is a recurring theme in shader programming. You'll see it again when we talk about domain repetition for fractals (Day 2), symmetry groups for hyperbolic tilings (Day 3), and space folding for raymarched scenes (Day 4). The key insight is always the same: instead of explicitly iterating over

instances, transform the coordinate system so that all instances share the same local coordinates.

1.7.3. Alternating Pattern

We can create checkerboard-like patterns by using the *cell index* to vary colors. To get the cell index, we divide by the spacing and floor:

```
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 (i, j), and we alternate colors based on the parity of $i + j$. When $i + j$ is even, `checker = 0` (giving us `color_a`), when odd, `checker = 1` (giving us `color_b`).

Notice the elegant separation: `cell_id` tells us *which* cell we're in, while `cell_p` tells us *where within* that cell. This separation of global position and local coordinates is fundamental to working with repeating patterns.

1.7.4. Combining with Circles

Let's put it all together—a grid of circles on an alternating background:

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

    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);
}
```

Try varying the `spacing` and `r` parameters. What happens if you make the circles larger than the cells? (They overlap across cell boundaries!) What if you use different spacing values for `x` and `y`? (You get a rectangular rather than square lattice.) This simple framework is incredibly flexible.

1.8. Implicit Curves

1.8.1. General Principle

An implicit curve is defined by an equation $F(x, y) = 0$. Points on the curve satisfy the equation exactly, while points off the curve have $F(x, y) \neq 0$. To render the curve, 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);
```

This is a remarkably general technique! It works for any curve you can write as an implicit equation—circles, ellipses, hyperbolas, higher-degree algebraic curves, transcendental curves, whatever you want. If you can write down a formula $F(x, y)$, you can visualize its zero set.

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);
```

You should see a yellow parabola on a dark blue background. The curve appears wherever $|F(x, y)| < 0.1$—a thin band around the zero set of F.

One thing to notice: the visual thickness of the curve varies! Near the vertex where the parabola is flat, the curve looks thicker, while in the steep regions it appears thinner. Why does this happen?

We're thresholding on the *value* of F, not the *geometric distance* to the curve. Near the vertex at $(0, 0)$, the parabola is nearly horizontal—small changes in y correspond to small changes in x, so the set $\{p : |y - x^2| < \epsilon\}$ is a thick vertical band. But on the steep parts where $|x|$ is large, the parabola is nearly vertical—now the same change in y corresponds to a large change in x, so the band is thin.

To see this more precisely, consider the gradient: $\nabla F = (-2x, 1)$. Near the vertex this has magnitude close to 1, but for large $|x|$ it has magnitude approximately $2|x|$. The visual thickness is roughly inversely proportional to $|\nabla F|$—where the gradient is small, the level sets are far apart, and where it's large, they're close together.

To get uniform thickness, we'd need the *signed distance function* to the curve:

$$d(p) = \inf\{\|p - q\| : F(q) = 0\}$$

Then thresholding on $|d(p)| < \epsilon$ gives exactly thickness ϵ everywhere. Computing exact signed distance functions is nontrivial (we'll see techniques for this on Day 4 when we discuss raymarching), but for many applications the naive thresholding on $|F|$ works fine—especially if you tune the thickness parameter appropriately or use different thickness values in different regions.

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. The `dot(p, p)` computes $x^2 + y^2$ as a single GPU operation—more efficient than `p.x * p.x + p.y * p.y` and certainly cleaner than writing it out! For circles, the signed distance function and the implicit function are particularly closely related: $d(p) = |\|p\| - r|$, so the naive implicit approach actually works quite well.

1.8.4. More Examples

Let's look at a few more interesting curves:

Hyperbola: $xy = 1$

```
float F = p.x * p.y - 1.0;
```

Ellipse: $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$

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

Lemniscate of Bernoulli: $(x^2 + y^2)^2 = a^2(x^2 - y^2)$

```
float a = 1.0;
float r2 = dot(p, p);
float F = r2 * r2 - a * a * (p.x * p.x - p.y * p.y);
```

Each of these creates beautiful curves! Try implementing them and experimenting with different parameters.

Implicit curves in your homework

When you're implementing the parabola graphing calculator for homework, you'll use this exact implicit curve technique. The key is setting up the equation $F(x, y) = y - (ax^2 + bx + c)$ and thresholding on $|F| < \epsilon$. Make sure to test with various values of a, b, c to ensure your grapher is robust!

1.9. Summary

Today we've learned the fundamental tools of shader programming:

1. **Shaders as parallel functions**: Every pixel evaluates $f(x, y, t, ...) \to$ color simultaneously—no loops required! The computational model is fundamentally different from sequential CPU programming.

2. **GLSL basics**: Syntax rules (semicolons, `.0` for floats), vector types (`vec2`, `vec3`, `vec4`), and essential built-in functions like `length()`, `dot()`, `step()`, and `smoothstep()`

3. **Coordinate systems**: The four-step transformation (normalize, center, aspect-correct, scale) that takes us from pixel coordinates to a mathematical coordinate system suitable for visualization

4. **Conditional coloring**: Using boolean expressions, the ternary operator, and `step()` combined with `mix()` to create discrete color regions based on mathematical predicates

5. **Distance fields**: Using `length()` to create circles and radial patterns—the foundation for much more complex techniques we'll explore on Day 4 with raymarching

6. **Modular arithmetic**: Creating grids and repeating patterns with `mod()`—achieving infinite complexity with finite computation through coordinate transformations rather than explicit iteration

7. **Implicit curves**: Rendering curves defined by $F(x, y) = 0$ by thresholding on $|F|$—a general technique that works for any curve we can express as an equation, though we must be aware of the non-uniform thickness issue

With these tools, you can already create a wide variety of mathematical visualizations! Tomorrow we'll use these same techniques to explore complex dynamics (Mandelbrot and Julia sets) and geometric transformations (circle inversions and the Apollonian gasket). But everything builds on the foundation we've established today.

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—we're not building a GUI yet)
- Draw the x-axis and y-axis as thin lines using the implicit line technique: $|y| < \epsilon$ for the x-axis, $|x| < \epsilon$ for the y-axis
- Plot the parabola $y = ax^2 + bx + c$ as a thicker curve
- Use distinct colors for axes (suggest a neutral gray) and parabola (suggest something bright)
- The visualization should work for any reasonable values of a, b, c—make sure to test edge cases!

What it should look like: A coordinate plane with thin gray axes, and a colored curve tracing out your parabola. The entire parabola should be visible in your viewing window (you may need to adjust your scaling factor depending on your parameters).

Test cases to verify: - $a = 1, b = 0, c = 0$ (standard parabola opening upward) - $a = -1, b = 0, c = 1$ (downward-opening parabola shifted up) - $a = 0.5, b = 1, c = -0.5$ (general case with all parameters nonzero) - $a = 0, b = 1, c = 0$ (degenerate case—just a line! Your code should handle this gracefully)

Suggested approach:

```
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 p = uv * 4.0;

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

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

    // Axes
    float axis_thickness = 0.02;
    float x_axis_mask = abs(p.y) < axis_thickness ? 1.0 : 0.0;
    float y_axis_mask = abs(p.x) < axis_thickness ? 1.0 : 0.0;
    vec3 axis_color = vec3(0.3, 0.3, 0.3);

    // Parabola: F(x,y) = y - (ax² + bx + c) = 0
    float F = p.y - (a * p.x * p.x + b * p.x + c);
    float curve_thickness = 0.08;
```

```
    float parabola_mask = abs(F) < curve_thickness ? 1.0 : 0.0;
    vec3 parabola_color = vec3(1.0, 0.8, 0.0);

    // Combine (axes behind parabola)
    color = mix(color, axis_color, max(x_axis_mask, y_axis_mask));
    color = mix(color, parabola_color, parabola_mask);

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

Try different values of a, b, c and verify your grapher works correctly! What happens with negative a? What about $b \neq 0$? Make sure the axes and parabola remain visible for all parameter values you try. If the parabola goes off-screen, you may need to adjust your coordinate scaling in the setup.

1.10.2. Optional #1: Animated Curve Family

Create a shader that animates through a family of curves—watching how a curve morphs continuously as parameters change is a beautiful way to build geometric intuition!

Easier options:

- **Circle family**: Draw circles of varying radii: $x^2 + y^2 = r^2$ where $r = 1 + 0.5\sin(\text{iTime})$. Simple but mesmerizing!
- **Rotating ellipse**: $(x\cos\theta + y\sin\theta)^2/a^2 + (-x\sin\theta + y\cos\theta)^2/b^2 = 1$ with $\theta = \text{iTime}$. Watch an ellipse rotate continuously.

More challenging options:

- **Lissajous curves**: Use parametric equations $x = A\sin(at + \delta)$, $y = B\sin(bt)$ and animate δ with `iTime`. To render a parametric curve implicitly, you'll need to be clever—one approach is to sample many points along the curve and draw circles at each point (we'll learn better techniques for this later).
- **Cassini ovals**: $(x^2 + y^2)^2 - 2c^2(x^2 - y^2) = a^4 - c^4$. Fix $c = 1$ and vary a with `iTime`. Watch the curve transition from two separate loops to a single figure-eight-like shape as a passes through the critical value $a = c$!
- **Cubic curves**: Take $y^2 = x^3 + ax + b$ and vary one parameter with `iTime`. The topology of the curve changes dramatically as you pass through singular values—this is the beginning of the theory of elliptic curves!

Use `iTime` creatively to create a compelling animation. The goal is to explore how continuous parameter variation produces interesting mathematical families. Bonus points if you can identify special parameter values where the curve topology changes (these are the singularities of the family)!

1.10.3. Optional #2: Beautiful Tiling Pattern

Design an aesthetically pleasing tiling pattern using the `mod()` technique. This is your chance to be creative and make something visually striking!

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 must match up so there are no visible discontinuities at tile boundaries

Ideas to get you started:

- **Geometric patterns**: Nested circles, polygons approximated by implicit curves, star shapes using angular coordinates
- **Color gradients**: Use `cell_id` to vary colors smoothly across tiles, creating large-scale gradient effects superimposed on the local pattern
- **Multiple implicit curves**: Combine several curves within each tile using boolean operations (intersection, union, etc.)
- **Symmetry**: Use `abs()` to create reflections within tiles—this is a simple way to get complex patterns with built-in symmetry
- **Distance-based effects**: Make features pulse or fade based on `iTime` and their position in the grid—create waves propagating across the tiling

Advanced challenge: Can you create a pattern that has different symmetries in different tiles? For example, alternate between rotational and reflectional symmetry using the checkerboard `cell_id` technique. Or create a pattern where the colors vary smoothly across the entire infinite tiling, creating a large-scale gradient effect that's independent of the tile boundaries?

Think about Islamic geometric patterns, Escher tilings, or quasiperiodic tilings (though true quasiperiodicity requires techniques beyond simple `mod`—we'll see that on Day 3!). The goal is to create something mathematically interesting and visually beautiful.

1.11. Looking Ahead

Tomorrow we'll use these techniques to explore **complex dynamics** and **geometric transformations**:

- **Mandelbrot and Julia sets**: Using the implicit curve technique to visualize the boundary of escape sets for complex iteration
- **Circle inversions**: A geometric transformation that takes lines and circles to lines and circles, creating beautiful fractal-like patterns
- **Apollonian gasket**: An infinite packing of circles constructed via repeated inversions—a stunning example of how simple geometric rules create intricate structures

The coordinate systems, distance fields, 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 (the standard four-step transformation from `fragCoord` to centered, aspect-corrected `p`)

- Computing distances with `length()` and dot products
- Using `mod()` for repetition and understanding the separation of global `cell_id` and local `cell_p`
- Conditionally coloring based on mathematical expressions, using both explicit conditionals and smooth interpolation with `smoothstep`

If any of these feel shaky, now is the time to practice! Work through the homework problems, experiment with variations, and make sure you understand not just *how* the code works but *why* the mathematics gives the visual results you see. Everything this week builds on this foundation.

2. Day 2: Complex Dynamics and Iterated Inversions

2.1. Overview

Today we explore the power of iteration to generate fractals. We'll implement complex arithmetic in GLSL and use it to render the iconic Mandelbrot set, then understand its companion, the Julia set. After that, we shift gears to geometric iteration: circle inversion, a beautiful conformal transformation that creates intricate nested patterns when applied repeatedly. We'll see how the Apollonian gasket emerges from iterated inversions of three mutually tangent circles.

By the end of today, you'll understand how simple iterative processes—whether in the complex plane or through geometric transformations—can generate infinitely detailed fractal structures from just a few lines of code.

> **Roadmap for Today**
>
> We'll explore iteration in two different mathematical settings:
>
> 1. **Complex dynamics**: Iterating holomorphic maps (Mandelbrot and Julia sets)
> 2. **Geometric dynamics**: Iterating circle inversions (Apollonian gasket)
>
> Both produce fractals through the same fundamental mechanism: simple rules applied repeatedly reveal infinite complexity. The common thread is **conformality**—both complex multiplication and circle inversion preserve angles, and this angle preservation is key to the beautiful structures we'll see.
> Along the way, we'll learn shader programming techniques for organizing data (structs), implementing mathematical operations efficiently, and creating sophisticated coloring schemes.

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. This is the natural representation—complex numbers are the 2D plane with a particular multiplication structure! 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. You already know complex numbers geometrically as rotations and scalings in the plane—here we're just implementing that algebra in shader code.

2.2.2. Complex Arithmetic

Let $z = a + bi$ and $w = c + di$. We need to implement the basic operations. Some of these are trivial, others require a bit more work:

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

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

Addition of complex numbers is just vector addition—componentwise! You might not even need this function since you can just write `z + w` directly, but it's here for completeness.

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
    );
}
```

This implements the familiar FOIL pattern with $i^2 = -1$, giving us that minus sign in the real part.

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

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

The squared magnitude is just the dot product with itself. This is computationally cheaper than taking the square root, so when we just need to check if $|z| > 2$, we'll check if $|z|^2 > 4$ instead—millions of avoided square roots per frame!

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

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

The magnitude is the Euclidean distance from the origin—exactly what `length()` computes!

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

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

The conjugate flips the sign of the imaginary part—reflection across the real axis.

Division: $\frac{a+bi}{c+di} = \frac{(a+bi)(c-di)}{c^2+d^2}$

```
vec2 cdiv(vec2 z, vec2 w) {
    float denom = dot(w, w);  // c² + d²
    return vec2(
        (z.x * w.x + z.y * w.y) / denom,  // Real part
        (z.y * w.x - z.x * w.y) / denom   // Imaginary part
    );
}
```

Division multiplies numerator and denominator by the conjugate of w to rationalize. The denominator becomes real ($c^2 + d^2$), and the numerator becomes a new complex number we can compute directly.

> **Computational Efficiency in Complex Arithmetic**
>
> Notice we use `dot(z, z)` for magnitude squared—this is a single GPU operation rather than component-wise multiplication and addition. Similarly, `dot(w, w)` in the division routine. For operations you'll compute millions of times per frame, these micro-optimizations add up! We're also using helper functions rather than inlining the formulas everywhere. This makes the code much more readable (`cmul(z, z)` vs `vec2(z.x*z.x - z.y*z.y, 2.0*z.x*z.y)`) and easier to debug. If you make a sign error once in `cmul`, you fix it once. If you inline the formula fifty times, you'll hunt for bugs forever!

These are the building blocks we need for complex dynamics. Let's put them to work!

2.3. The Mandelbrot Set

2.3.1. Definition

The Mandelbrot set $\mathcal{M}$ is one of the most famous objects in mathematics—and for good reason! It's 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 $n \to \infty$.

That's it! Just iterate this simple quadratic map starting from $z_0 = 0$, and see if the orbit escapes to infinity or stays bounded. Points that stay bounded are in the set (traditionally colored black), while points that escape are colored based on how quickly they escape.

2.3.2. The Escape Radius Theorem

In practice, we can't iterate to infinity, so we need a criterion to detect escape. Fortunately, there's a beautiful theorem that tells us exactly when to stop:

Theorem (Escape Radius). If $|z_n| > 2$ for any n, then $|z_n| \to \infty$ as $n \to \infty$.

This means: once the orbit leaves the disk of radius 2, it's definitely escaping to infinity. Points that escape are not in the Mandelbrot set, while points that remain bounded after many iterations are (likely) in the set or very close to its boundary.

Proof. Suppose $|z_n| > 2$ and write $|z_n| = 2 + \epsilon$ for some $\epsilon > 0$. Then

$$\begin{aligned}
|z_{n+1}| &= |z_n^2 + c| \\
&\geq |z_n^2| - |c| \\
&= |z_n|^2 - |c| \\
&> |z_n|^2 - 2 \quad \text{(since } c \text{ is in or near } \mathscr{M}\text{, which fits in } |z| \leq 2) \\
&= (2+\epsilon)^2 - 2 \\
&= 4 + 4\epsilon + \epsilon^2 - 2 \\
&= 2 + 4\epsilon + \epsilon^2 \\
&> 2 + 2\epsilon = |z_n| + \epsilon
\end{aligned}$$

So once $|z_n| > 2$, we have $|z_{n+1}| > |z_n| + \epsilon$, meaning the magnitude grows by at least ϵ each iteration. This linear growth accelerates: if $|z_{n+1}| > 2 + \epsilon$, then $|z_{n+2}| > |z_{n+1}| + \epsilon' > 2 + 2\epsilon$, and so on. More carefully, the orbit actually grows exponentially (roughly like $|z_n| \sim 2^{2^n}$ for large n), but the key point is: it definitely escapes to infinity.

Computational Implication: We only need to check if $|z_n| > 2$. The moment this happens, we can stop iterating—this point will never be in the Mandelbrot set. This single theorem makes efficient rendering possible!

2.3.3. Historical Context

The Mandelbrot set was discovered remarkably recently—1980! Benoit Mandelbrot, working at IBM, was among the first to have access to computers powerful enough to visualize iterative processes in the complex plane. Before computers, studying these sets was nearly impossible—you'd need to manually iterate complex arithmetic hundreds of times for millions of points.

Interestingly, the mathematical theory predates visualization by over 60 years. Gaston Julia and Pierre Fatou studied iterative complex dynamics extensively in 1918, but without computers, they could only reason about these sets abstractly. They knew Julia sets existed and had deep properties, but had never seen one! When Mandelbrot generated the first images in 1980, it revolutionized the field—suddenly the intricate structure of these sets was visible, creating an explosion of interest in fractal geometry and complex dynamics.

The Mandelbrot set became iconic partly because it's so accessible: anyone can understand the definition (iterate $z \mapsto z^2 + c$), yet it produces infinitely intricate beauty. It also sparked broader interest in fractals, chaos theory, and the idea that simple rules can generate complex behavior—themes that would influence everything from physics to economics to art.

2.3.4. Basic Implementation

Let's code it up:

```
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);
}
```

That's the entire Mandelbrot set renderer! The coordinate scaling is tuned to show the "interesting part"—the main cardioid body and its surrounding bulbs. The set extends roughly from -2.5 to 0.5 on the real axis, so we shift our coordinate system accordingly.

> **Why This Is Perfect for GPUs**
>
> Notice what's happening computationally: every pixel performs its own independent calculation. There's no communication between pixels, no shared data structures, no synchronization needed. Each pixel just iterates its own complex number and decides when to stop.
> This is **embarrassingly parallel**—the ideal workload for GPU architecture. A modern GPU has thousands of small processors, and they can all work on different pixels simultaneously. No pixel needs to wait for another pixel's result. The entire screen (potentially millions of pixels) is computed in parallel, which is why we can render the Mandelbrot set at 60fps even with 100+ iterations per pixel.
> Memory-wise, this is also very efficient: each pixel only needs to store its current z value

(two floats) and an iteration counter (one integer). No arrays, no history, no complex data structures. The computation is stateless—we only care about the current iterate, not the full orbit.
This contrasts sharply with sequential CPU code, where you'd iterate over pixels one at a time. Even with clever optimizations and SIMD vectorization, you'd be orders of magnitude slower than a GPU shader doing the same work.

2.3.5. Smooth Coloring

The grayscale rendering shows the structure of the set, but it has harsh banding—sudden transitions between integer iteration counts create visible stripes. We can do much better by interpolating between iteration steps!

The key insight is that near escape, the orbit grows exponentially. Specifically, once $|z_n|$ is large, we have approximately $|z_{n+1}| \approx |z_n|^2$, which means $\log|z_{n+1}| \approx 2\log|z_n|$. Taking logs repeatedly, we get

$$\log\log|z_{n+1}| \approx \log(2\log|z_n|) = \log 2 + \log\log|z_n|$$

This suggests that $\log\log|z_n|$ grows approximately linearly near escape, increasing by $\log 2$ per iteration. We can use this to compute a fractional iteration count!

Here's the formula:

```
if(iter < max_iter) {
    // Smooth iteration count (accounts for continuous escape)
    float log_zn = log(cabs2(z)) / 2.0;  // = log|z_n|
    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);
    vec3 color = palette(t);
} else {
    // Inside the set: black
    vec3 color = vec3(0.0);
}
```

The variable `nu` represents how far we've progressed toward the next integer iteration. When $|z_n| = 2$ exactly (just hitting the escape threshold), $nu = 0$ and we get the integer iteration count. When $|z_n|$ is large (deep into escape), nu approaches 1. Subtracting `nu` from `iter + 1` gives us a continuous, smooth value that transitions gradually between iteration levels.

This eliminates banding entirely! The result is smooth, continuous color gradients that look much more professional and reveal the fractal structure more clearly.

Smooth Coloring as Anti-Aliasing

Smooth coloring is fundamentally an anti-aliasing technique. Without it, nearby pixels with iteration counts of, say, 45 and 46 get completely different colors—creating harsh edges. With smooth coloring, these pixels get nearly identical colors (say, iteration 45.3 and 45.8), producing a smooth gradient.

This is especially important at high zoom levels, where tiny changes in position lead to different integer iteration counts. The smooth interpolation ensures that small changes in c produce small changes in color, which is exactly what we want for a continuous mathematical function.

2.3.6. Color Palettes

Now we need a good color mapping function. A classic approach uses cosines to create smooth, cyclic color palettes:

```
vec3 palette(float t) {
    // Create a cyclic color palette using cosines
    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—perfect for the Mandelbrot set where we want colors to repeat as we zoom in to the fractal boundary. The parameters control different aspects:

- `a` and `b` control the range and center of the colors (here, mapping to $[0, 1]$)
- `c` controls the frequency of color cycling
- `d` controls the phase offset, shifting the entire palette

Play with these parameters to get different color schemes! Try `d = vec3(0.0, 0.1, 0.2)` for a blue-purple palette, or `d = vec3(0.3, 0.2, 0.2)` for warmer tones. You can also adjust `c` to make the colors cycle faster or slower through the iteration range.

The beauty of this cosine approach is that it's smooth (continuous derivatives), cyclic (no seams), and efficient (just a few trig operations). Other approaches exist—hand-picked color stops with interpolation, HSV color spaces, perceptually uniform LAB spaces—but cosine palettes are a great default for fractal visualization.

2.4. Julia Sets

The Julia set is the natural companion to the Mandelbrot set, and understanding their relationship is key to understanding complex dynamics. Where the Mandelbrot set varies c and fixes $z_0 = 0$, the Julia set does the opposite: it fixes c and varies z_0.

2.4.1. Definition

For a fixed complex number c, the **filled Julia set** K_c is the set of initial conditions z_0 for which the iteration

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

remains bounded. The **Julia set** J_c is the boundary of K_c—the set where the dynamics are chaotic, neither definitely bounded nor definitely escaping.

Think about what this means: every point c in the complex plane has an associated Julia set J_c. The Mandelbrot set is telling us about the topology of these Julia sets! Specifically:

- If $c \in \mathscr{M}$ (inside the Mandelbrot set), then J_c is **connected**—a single, intricate curve
- If $c \notin \mathscr{M}$ (outside the Mandelbrot set), then J_c is **totally disconnected**—a Cantor-like dust of points

This is one of the most beautiful connections in mathematics: the Mandelbrot set is a map of parameter space, showing which values of c produce connected Julia sets. Julia and Fatou proved this in 1918 without ever seeing a picture—they understood these sets purely abstractly!

2.4.2. Implementation as Homework

You'll implement Julia set rendering in the homework. The algorithm is nearly identical to the Mandelbrot set—just swap what's fixed and what varies! Instead of setting `z = vec2(0.0)` and varying `c` across pixels, you'll set `c` to a fixed value and let `z` be the pixel coordinate.

The key changes: 1. Fix `c` to an interesting value (we'll give suggestions) 2. Initialize `z` from the pixel position (that's your z_0) 3. Iterate $z \mapsto z^2 + c$ exactly as before 4. Use the same escape criterion and coloring

This will give you a Julia set! Try different values of c to see how the topology changes. You can make `c` depend on time (`iTime`) for animation, or on mouse position (`iMouse`) for interactive exploration.

Some interesting values to try: - $c = -0.7 + 0.27015i$ — classic Julia set, intricate tendrils - $c = -0.4 + 0.6i$ — dendrite-like fractal trees - $c = 0.285 + 0.01i$ — beautiful spiral patterns - $c = -0.8 + 0.156i$ — highly filamentary - $c = -0.70176 - 0.3842i$ — "San Marco dragon"

Values inside the Mandelbrot set give connected Julia sets (single curves), while values outside give disconnected Julia sets (dust). The most interesting Julia sets often come from values right near the boundary of $\mathscr{M}$—this is where the transition between connected and disconnected happens!

> **Exploring Parameter Space**
>
> When you implement Julia sets, try this: tie c to your mouse position. Move the mouse around and watch the Julia set morph in real time! This is an incredibly powerful way to build intuition for how the parameter c affects the dynamics.
> You'll notice that small changes in c can produce dramatic changes in the Julia set topology—this is the sensitive dependence on parameters that makes complex dynamics so rich. Near the boundary of the Mandelbrot set, tiny movements create entirely different structures.

2.5. Interlude: From Complex to Geometric Dynamics

We've been iterating algebraic functions in the complex plane: $z \mapsto z^2 + c$. Now we shift to iterating geometric transformations of the plane itself: circle inversion. The mathematical frameworks are different—holomorphic dynamics versus conformal geometry—but they share fundamental similarities.

Both complex multiplication and circle inversion are **conformal maps**: they preserve angles between curves. In complex dynamics, this comes from the Cauchy-Riemann equations and the geometric interpretation of multiplication as rotation and scaling. In circle inversion, it's a theorem we'll state shortly. This angle preservation is crucial—it's what makes the fractal structures we generate geometrically coherent and visually beautiful.

There's also a deep connection we'll explore tomorrow: circle inversion is actually an isometry of hyperbolic space! The inversions we're about to do are the same transformations that generate Kleinian groups and tessellate the hyperbolic plane. So in some sense, we're already working in non-Euclidean geometry without realizing it. Tomorrow we'll make this explicit when we explore hyperbolic tilings.

For now, let's learn circle inversion and use it to build the Apollonian gasket—a fractal structure every bit as intricate as the Mandelbrot set, but generated through pure Euclidean geometry.

2.6. Circle Inversion

Circle inversion is a beautiful geometric operation that will be the foundation for everything we do with geometric dynamics. It's a transformation of the plane that turns inside into outside, maps circles to circles (or lines), and preserves angles—making it a powerful tool for creating fractal patterns.

2.6.1. Mathematical Definition

Circle inversion is a transformation with respect to a circle. For a circle of radius R centered at a point $\mathbf{c}$, inversion maps a point $\mathbf{p} \neq \mathbf{c}$ to:

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

The vector $\mathbf{p} - \mathbf{c}$ points from the center to $\mathbf{p}$. We normalize this direction by dividing by its squared length, then scale by R^2, and finally translate back by the center. Geometrically:

- Points inside the circle map to points outside (and vice versa)
- Points on the circle are fixed (they map to themselves)
- The closer you are to the center, the farther away you go
- The center itself maps to infinity, and infinity maps to the center

Here's another way to think about it: draw a ray from the center through $\mathbf{p}$. The inversion of $\mathbf{p}$ is the unique point on this ray such that the product of distances from the center is R^2. If $\mathbf{p}$ is at distance r from the center, its image is at distance R^2/r.

2.6.2. Key Properties

Circle inversion has remarkable geometric properties. These aren't obvious from the formula, but they're all classical theorems:

1. **Lines through the center** remain lines through the center (they're "flipped inside out" along the ray)
2. **Lines not through the center** become circles through the center
3. **Circles through the center** become lines (not through the center)
4. **Circles not through the center** generally remain circles, but with different center and radius
5. **Angles are preserved** (conformal property)

The angle preservation is the deepest property. It's not at all obvious from the formula, but it can be proved using the chain rule and careful calculation. The key insight is that inversion is locally similar to a complex conjugation-like operation, which preserves angles.

Why These Properties Matter for Iteration: When we iterate inversions through multiple circles, these properties ensure that the geometry remains coherent. Circles stay circles (or become lines), and the angles between curves are preserved. This means repeated inversion creates intricate but geometrically regular patterns—the hallmark of fractals generated by conformal maps.

Another key fact: inversion is **involutive**—applying it twice returns to the original point (assuming the point isn't the center). Mathematically, $\text{inv}(\text{inv}(\mathbf{p})) = \mathbf{p}$. This makes inversion a geometric reflection of sorts, which will be important when we think about symmetry groups tomorrow.

> **Historical Context: Circle Inversion**
>
> Circle inversion has ancient roots—Apollonius of Perga studied related problems involving tangent circles around 200 BCE. But the modern theory of inversion as a geometric transformation developed in the 19th century as part of projective and non-Euclidean geometry.
> A key insight was recognizing that circle inversion is related to stereographic projection: if you place a sphere on the plane, inversion in a circle corresponds to reflection through the sphere! This connection links circle inversion to spherical geometry and ultimately to hyperbolic geometry.
> In the late 19th and early 20th centuries, mathematicians realized that groups of circle inversions (Kleinian groups) could tessellate hyperbolic space and create fractal limit sets. This anticipates the fractal geometry revolution of the 1970s-80s, though the connection wasn't fully appreciated until computers made visualization possible.

2.6.3. Implementation

The formula translates directly to GLSL:

```
vec2 invertCircle(vec2 p, vec2 center, float radius) {
    vec2 diff = p - center;
    float r2 = dot(diff, diff);  // squared distance from center

    // Handle center (would be division by zero)
```

```
    if(r2 < 0.0001) return vec2(1000.0);  // Map to "infinity"

    return center + (radius * radius) * diff / r2;
}
```

The only tricky part is handling the center point, which mathematically maps to infinity. We approximate this by mapping to a very large value—far enough away that it's effectively off-screen. The threshold `0.0001` is small enough to catch points numerically close to the center but large enough to avoid precision issues.

Computationally, circle inversion is very cheap: just one dot product, a division, a multiplication, and some vector operations. This efficiency is important because we'll be doing many inversions per pixel when generating fractals!

2.6.4. Visualizing Circle Inversion

Let's see what happens when we apply inversion to a grid. This is one of the best ways to understand the transformation visually:

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

    // Inversion circle
    vec2 circleCenter = vec2(0.0, 0.0);
    float circleRadius = 1.0;

    // Apply inversion
    vec2 p_inverted = invertCircle(p, circleCenter, circleRadius);

    // Draw a grid in the inverted space
    vec2 grid = fract(p_inverted * 2.0);  // Create repeating cells
    float gridLine = step(0.95, max(grid.x, grid.y));  // Draw grid lines

    vec3 color = vec3(gridLine);

    // Draw the inversion circle itself (for reference)
    float circDist = abs(length(p) - circleRadius);
    if(circDist < 0.05) color = vec3(1.0, 0.0, 0.0);

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

You'll see straight grid lines transform into beautiful circular arcs! Lines farther from the inversion circle get bent more dramatically, while lines near the circle stay relatively straight. Horizontal and

vertical lines through the center remain horizontal and vertical (but swap inside/outside), while other lines become circles.

This visualization really helps build intuition. You can see: - The circle itself (in red) is unchanged—points on it are fixed - The grid inside the circle maps to a grid outside (and vice versa) - Lines become curves, but the angles where they intersect are preserved - The pattern has a pole singularity at the center (infinite distortion)

> Exploring Circle Inversion
>
> Here are some experiments to try:
>
> 1. **Multiple circles**: Create a grid of circles using `mod()` (like we did on Day 1) and invert through each cell's circle independently. You'll see beautiful overlapping patterns!
> 2. **Animated radius**: Make `circleRadius = 1.0 + 0.5 * sin(iTime)` to watch the grid breathe in and out. This helps you see how the inversion depends on the circle's size.
> 3. **Different patterns**: Instead of a grid, try drawing circles or other shapes in the inverted space. Circles become circles (or lines), creating intricate nested patterns.
> 4. **Off-center inversion**: Move the inversion circle away from the origin. Watch how the asymmetry creates even more complex distortions.
>
> Each of these will give you geometric intuition for how inversion behaves, which will be crucial when we iterate multiple inversions!

2.7. Structs in GLSL

Before we build the Apollonian gasket, we need to talk about organizing our data. We're about to deal with multiple circles, and passing around `center1`, `radius1`, `center2`, `radius2`, etc. gets unwieldy fast. GLSL provides **structs** (just like in C) for grouping related data together.

2.7.1. Defining and Using Structs

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

Now `Circle` is a type we can use just like `vec2` or `float`. Creating and using structs is straightforward:

```
// Declare and initialize
Circle c1 = Circle(vec2(0.0, 0.0), 1.0);

// Or declare first, set later
Circle c2;
c2.center = vec2(1.0, 0.5);
c2.radius = 0.75;

// Pass to functions
vec2 invertThroughCircle(vec2 p, Circle circ) {
    return invertCircle(p, circ.center, circ.radius);
}
```

This makes the code dramatically more readable. Compare:

```
// Without structs - messy!
vec2 iterate(vec2 p, vec2 c1_cen, float c1_rad, vec2 c2_cen, float c2_rad, vec2 c3_cen, float c3_rad)

// With structs - clean!
vec2 iterate(vec2 p, Circle c1, Circle c2, Circle c3);
```

For the Apollonian gasket, we'll have three circles we need to track and pass around. Structs make this much more manageable and semantically clear—we're working with circles as geometric objects, not just pairs of vectors and floats.

2.8. The Apollonian Gasket

Now for the main event! The Apollonian gasket is a fractal generated by iterating circle inversions through three mutually tangent circles. It's named after Apollonius of Perga, who studied the problem of finding circles tangent to three given circles around 200 BCE, though the fractal interpretation is much more modern.

2.8.1. Descartes Circle Theorem

To understand the Apollonian gasket, we need to know about a beautiful theorem: **Descartes Circle Theorem** (1643).

Given four mutually tangent circles (each tangent to the other three), let their curvatures be k_1, k_2, k_3, k_4 where curvature $k = 1/r$ (positive for external tangency, negative for internal). Then:

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

This can be rearranged to solve for the fourth curvature given three:

$$k_4 = k_1 + k_2 + k_3 \pm 2\sqrt{k_1k_2 + k_2k_3 + k_3k_1}$$

The ± gives two solutions—the two circles tangent to the original three (one inside the curvilinear triangle they form, one outside).

What makes this magical for fractals: if you start with three mutually tangent circles with integer curvatures, then **all** circles in the Apollonian gasket have integer curvatures! This is the Apollonian gasket's connection to number theory—it's a fractal made entirely of circles with rational radii.

We won't use this formula directly in our shader (we'll set up circles geometrically), but it explains why certain configurations are special and why the patterns are so regular despite infinite nesting.

2.8.2. Setup: Three Mutually Tangent Circles

We start with three circles that are all tangent to each other—meaning each pair touches at exactly one point. For this to work, the distance between any two circle centers must be exactly twice the radius (so they touch edge-to-edge). The three centers form an equilateral triangle, and we'll position this triangle symmetrically at the origin:

```
void setupApollonianCircles(out Circle c1, out Circle c2, out Circle c3, out Circle outer) {
    float r = 0.5;  // Radius of each inner circle
    // For three circles to be mutually tangent: distance between centers = 2r
    // Centers form equilateral triangle with circumradius = 2r/sqrt(3)
    float d = 2.0 * r / sqrt(3.0);  // ≈ 0.577 for r = 0.5

    // Three inner circles
    c1 = Circle(vec2(0.0, d), r);
    c2 = Circle(vec2(-d * 0.866, -d * 0.5), r);  // 0.866 ≈ sqrt(3)/2
    c3 = Circle(vec2(d * 0.866, -d * 0.5), r);

    // Outer circle tangent to all three, centered at origin
    float R = d + r;  // ≈ 1.077 for r = 0.5
    outer = Circle(vec2(0.0, 0.0), R);
}
```

With $r = 0.5$, the circumradius $d \approx 0.577$, and the outer circle has radius $R = d + r \approx 1.077$. All four circles are mutually tangent—each inner circle touches the other two inner circles and the outer circle.

This is just one possible configuration! You could use different radii (related by Descartes' theorem), different arrangements, or even animated circles. The key is that they start mutually tangent—this ensures the iteration creates a proper Apollonian packing.

2.8.3. The Iteration Algorithm

The algorithm is beautifully simple:

1. Start with a point p
2. Check which circles contain p
3. If p is inside a circle, invert through that circle
4. Repeat until p is outside all circles or we hit max iterations
5. Color based on iteration behavior

Here's the implementation:

```
vec2 iterateApollonian(vec2 p, Circle c1, Circle c2, Circle c3, Circle outer,
                       int maxIter, out int finalIter, out int lastCircle) {
    for(int i = 0; i < maxIter; i++) {
        bool moved = false;

        // Check the three inner circles
        if(length(p - c1.center) < c1.radius) {
            p = invertCircle(p, c1.center, c1.radius);
            lastCircle = 0;
            moved = true;
        }
        else if(length(p - c2.center) < c2.radius) {
            p = invertCircle(p, c2.center, c2.radius);
            lastCircle = 1;
            moved = true;
        }
        else if(length(p - c3.center) < c3.radius) {
            p = invertCircle(p, c3.center, c3.radius);
            lastCircle = 2;
            moved = true;
        }
        // Check if outside the outer circle
        else if(length(p - outer.center) > outer.radius) {
            p = invertCircle(p, outer.center, outer.radius);
            lastCircle = 3;
            moved = true;
        }

        // If we didn't move, we're in the gaps - done!
        if(!moved) {
            finalIter = i;
            return p;
        }
    }

    finalIter = maxIter;
    return p;
}
```

We also track which circle we last inverted through (`lastCircle`) and how many iterations we performed (`finalIter`). These will be useful for coloring!

Why This Creates a Fractal: Each time we invert through a circle, we "push" the point away from that circle's center. But because the four circles are mutually tangent, pushing away from one circle might push us into another circle, triggering another inversion. The interplay between these four inversions (three inner circles plus the outer circle) creates a complex orbit.

Points in the "gaps" between the inner circles (the curvilinear triangles) escape quickly—they're inside the outer circle but not inside any inner circle, so no inversion happens. But points near the tangency points get bounced back and forth between circles many times before escaping. And at

the actual tangency points (where circles touch), the orbit never escapes—these are fixed points or periodic orbits of the iterated inversions.

The fractal structure emerges because these dynamics are self-similar: zooming in near any tangency point reveals the same pattern of nested circles and gaps. This self-similarity is a direct consequence of the conformal nature of circle inversion—the transformation preserves angles, so local geometry looks the same at all scales.

2.8.4. Coloring Strategies

There are several interesting ways to color the Apollonian gasket:

By iteration count (like the Mandelbrot set):

```
float t = float(finalIter) / float(maxIter);
vec3 color = palette(t);
```

This shows the "depth" of the orbit—points that escape quickly are colored differently from points that bounce around many times.

By last circle hit:

```
vec3 colors[4] = vec3[4](
    vec3(1.0, 0.0, 0.0),  // Circle 1: red
    vec3(0.0, 1.0, 0.0),  // Circle 2: green
    vec3(0.0, 0.0, 1.0),  // Circle 3: blue
    vec3(1.0, 1.0, 0.0)   // Outer circle: yellow
);
vec3 color = colors[lastCircle];
```

This shows the basin of attraction—which circle's "influence" each point ultimately fell into. Yellow regions show points that escaped through the outer circle.

By final distance from circles:

```
float d1 = abs(length(p - c1.center) - c1.radius);
float d2 = abs(length(p - c2.center) - c2.radius);
float d3 = abs(length(p - c3.center) - c3.radius);
float d = min(d1, min(d2, d3));
vec3 color = vec3(smoothstep(0.0, 0.05, d));
```

This highlights the circle boundaries themselves, making the geometric structure more apparent.

Each coloring reveals different aspects of the fractal. Try combining them—for example, color by iteration count but modulate brightness by distance to circles.

2.8.5. Putting It All Together

Here's a complete Apollonian gasket renderer:

```
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 p = uv * 4.0;

    // Setup the four circles
    Circle c1, c2, c3, outer;
    setupApollonianCircles(c1, c2, c3, outer);

    // Iterate
    int maxIter = 50;
    int finalIter, lastCircle;
    vec2 final_p = iterateApollonian(p, c1, c2, c3, outer, maxIter, finalIter, lastCircle);

    // Color by iteration count with palette
    float t = float(finalIter) / float(maxIter);
    vec3 color = palette(t);

    // Draw all four circles for reference
    float d1 = abs(length(p - c1.center) - c1.radius);
    float d2 = abs(length(p - c2.center) - c2.radius);
    float d3 = abs(length(p - c3.center) - c3.radius);
    float d_outer = abs(length(p - outer.center) - outer.radius);
    float d = min(min(d1, min(d2, d3)), d_outer);

    if(d < 0.02) color = vec3(1.0);  // White circle outlines

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

You should see a beautiful nested pattern of circles! The fractal structure is immediately apparent—circles within circles within circles, filling every gap with smaller circles.

Zoom in (by scaling p differently in the coordinate setup) to see the self-similarity. No matter how far you zoom, you'll keep finding the same pattern repeated at smaller scales. This is true fractality—infinite detail that never runs out.

Computational Efficiency

Despite the complexity of the output, this algorithm is remarkably efficient. Each iteration just checks three distances (cheap) and potentially does one inversion (also cheap—just a few arithmetic operations). With 50 iterations, we're doing maybe 150 distance checks and 50 inversions per pixel.
Compare this to raymarching (Day 4), where we might do hundreds of distance evaluations per pixel! The Apollonian gasket is very GPU-friendly.
The key is that we terminate early—most pixels escape in just a few iterations. Only points near the fractal boundary require many iterations. This is similar to the Mandelbrot set: most

> of the computational work focuses on the interesting regions (the boundary), while simple regions (deep inside or far outside) are handled quickly.

2.8.6. Historical Context and Connections

The Apollonian gasket connects several mathematical threads across millennia:

Ancient roots: Apollonius of Perga (~200 BCE) studied the problem of constructing circles tangent to three given circles (the "Problem of Apollonius"). He found geometric constructions but couldn't have imagined the infinite fractal structure we're visualizing.

Renaissance mathematics: René Descartes (1643) discovered the circle theorem bearing his name, giving an algebraic formula for tangent circles. This turned Apollonius's geometric problem into arithmetic.

19th century: The connection to projective geometry and circle inversions was developed. Mathematicians realized that packing problems could be studied through group theory—the inversions form a discrete subgroup of the group of Möbius transformations.

20th century: With computers, the fractal nature became visible. The Apollonian gasket was recognized as a limit set of a Kleinian group—a group of isometries of hyperbolic space. This connects to tomorrow's material: the inversions we're doing are actually hyperbolic isometries! In the Poincaré disk model (which we'll see tomorrow), these inversions are reflections through hyperbolic geodesics.

The gasket also has connections to number theory: if the initial circles have integer curvatures, all circles in the packing do too (by Descartes' theorem). This has led to deep questions about the distribution of integers in these packings, which remain active research areas today.

2.9. Summary

Today we've explored two powerful iterative processes that generate fractals:

1. **Complex dynamics**: The Mandelbrot and Julia sets emerge from iterating $z \mapsto z^2 + c$
   - Implemented complex arithmetic in GLSL efficiently
   - Learned escape-time algorithms and the crucial escape radius theorem
   - Developed smooth coloring techniques for anti-aliased rendering
   - Understood the parameter space (Mandelbrot) vs dynamical space (Julia) distinction
   - Saw how 1980s computational power revolutionized a 1918 theory
2. **Geometric dynamics**: The Apollonian gasket emerges from iterating circle inversions
   - Circle inversion as a conformal transformation preserving angles
   - Used structs to organize geometric data cleanly
   - Applied Descartes Circle Theorem to understand tangent circle configurations
   - Iterated inversions through three circles to create fractal patterns
   - Connected ancient Greek geometry to modern fractal theory

Both processes show how incredibly simple rules—a quadratic map, a geometric transformation—generate infinite complexity through iteration. The key in both cases is **conformality**: angle-preserving maps create geometrically coherent fractals. This is one of the core insights of fractal geometry and dynamical systems.

The computational perspective is also crucial: both algorithms are embarrassingly parallel, making them perfect for GPU rendering. Each pixel's calculation is independent, and we can terminate early when orbits escape. The result is real-time rendering of infinitely detailed mathematical objects.

Tomorrow we'll push geometric iteration further, moving from Euclidean to hyperbolic geometry. The circle inversions we've learned today are actually hyperbolic isometries—transformations that preserve distances in hyperbolic space. We'll explore the upper half-plane and Poincaré disk models, create $(2, 3, 7)$ and $(2, 3, \infty)$ triangle tilings, and see how the same algorithmic ideas (iterated geometric transformations) work in non-Euclidean geometry. The results will be even more intricate because hyperbolic space has "more room" than Euclidean space—triangles can have angle sums less than π, allowing denser tilings and more complex fractal structures.

2.10. Homework

2.10.1. Required: Julia Set Explorer

Implement a Julia set renderer starting from the Mandelbrot code. The algorithm is nearly identical—you just need to swap what's fixed and what varies!

Requirements:

1. Start from the Mandelbrot implementation
2. Fix c to a constant value (see suggestions below)
3. Initialize z from the pixel position instead of zero
4. Iterate $z_{n+1} = z_n^2 + c$ exactly as before
5. Use the same escape criterion and coloring

Suggested structure:

```
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 p = uv * 3.0;  // Scale for Julia set viewing

    // Fix c to an interesting value
    vec2 c = vec2(-0.7, 0.27015);  // Classic Julia set

    // Initialize z from pixel position (this is the key change!)
    vec2 z = p;  // z_0 = pixel position
```

```
    // [Rest of iteration exactly like Mandelbrot]
}
```

Interesting parameters to try: - `vec2(-0.7, 0.27015)` — classic Julia set, intricate tendrils - `vec2(-0.4, 0.6)` — dendrite-like fractal trees
- `vec2(0.285, 0.01)` — beautiful spiral patterns - `vec2(-0.8, 0.156)` — highly filamentary structure - `vec2(-0.70176, -0.3842)` — "San Marco dragon" - `vec2(-0.835, -0.2321)` — another classic - `vec2(-0.7269, 0.1889)` — "Douady's rabbit" (famous example)

Extension Options (pick one or both):

Option A: Animated Parameter Space

Make c depend on time to watch the Julia set morph:

```
float angle = iTime * 0.3;
float radius = 0.7885;  // Distance from origin in parameter space
vec2 c = vec2(radius * cos(angle), radius * sin(angle));
```

Watch how the Julia set changes topology as you trace a circle in the complex plane! You'll see it transition from connected to disconnected, develop tendrils, and create organic shapes.

Option B: Mouse-Controlled Exploration

Tie c to mouse position for interactive exploration:

```
vec2 mouse_uv = (iMouse.xy / iResolution.xy) - 0.5;
mouse_uv.x *= iResolution.x / iResolution.y;
vec2 c = mouse_uv * 3.0;  // Scale to cover interesting parameter range
```

Now you can explore parameter space by moving the mouse! This really helps build intuition for how c affects the Julia set. Try finding the boundary of the Mandelbrot set—parameters right at the edge produce the most intricate Julia sets.

> **Understanding Connected vs Disconnected**
>
> As you explore parameter space, pay attention to whether the Julia set appears as a single connected structure or as disconnected dust. Values of c inside the Mandelbrot set give connected Julia sets, while values outside give disconnected ones. The most beautiful Julia sets often come from values right near the boundary of $\mathscr{M}$!

2.10.2. Optional #1: Grid of Julia Sets

Create a grid where each cell shows the Julia set for that value of c, revealing the Mandelbrot set as an emergent pattern!

The idea: The Mandelbrot set is a map of Julia set topology. If we draw a grid of Julia sets for different values of c, we should see the Mandelbrot set emerge in the overall pattern—cells with

connected Julia sets (solid regions) correspond to points in $\mathcal{M}$, while cells with disconnected Julia sets (dust) are outside.

Implementation strategy:

```
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 p = uv * 4.0;

    // Divide screen into grid cells
    float grid_size = 8.0;  // 8×8 grid of Julia sets
    vec2 cell_id = floor(p * grid_size / 4.0);
    vec2 cell_p = fract(p * grid_size / 4.0) - 0.5;
    cell_p *= 4.0;  // Local coordinates within cell

    // Map cell_id to parameter c
    vec2 c = (cell_id / grid_size) * 4.0 - vec2(2.5, 2.0);
    c.x -= 0.5;  // Center on interesting region of Mandelbrot set

    // Run Julia set iteration with z = cell_p, fixed c
    vec2 z = cell_p;
    int max_iter = 50;
    int iter;

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

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

    // Optional: draw grid lines to separate cells
    vec2 grid_edge = abs(fract(p * grid_size / 4.0) - 0.5);
    if(max(grid_edge.x, grid_edge.y) > 0.48) color = vec3(0.0);

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

You should see a grid of tiny Julia sets! If you look carefully at the overall pattern, you'll notice it resembles the Mandelbrot set—cells with connected Julia sets (solid colored regions) correspond to points inside $\mathcal{M}$, while cells with disconnected Julia sets (fine dust patterns) are outside.

This is one of the most beautiful visualizations in complex dynamics—the Mandelbrot set literally encodes the topology of all Julia sets!

Variations to try: - Adjust `grid_size` (larger for more detail, smaller for clearer overview) - Change the parameter space region being sampled - Color cells based on whether the Julia set

appears connected (black) or disconnected (white)—you'll get a pixelated approximation of the Mandelbrot set!

2.10.3. Optional #2: Other Iterated Inversions

Explore variations on the Apollonian gasket theme! The key is setting up circles in interesting configurations and iterating inversions.

Different circle arrangements:

Four circles in a square:

```
void setupSquareCircles(out Circle c1, out Circle c2, out Circle c3, out Circle c4) {
    float r = 0.5;
    float d = 1.5;  // Distance from center
    c1 = Circle(vec2(d, 0.0), r);
    c2 = Circle(vec2(0.0, d), r);
    c3 = Circle(vec2(-d, 0.0), r);
    c4 = Circle(vec2(0.0, -d), r);
}
```

Modify the iteration loop to check four circles instead of three!

Nested circles:

```
// One large circle containing several smaller ones
c1 = Circle(vec2(0.0, 0.0), 2.0);  // Large outer circle
c2 = Circle(vec2(-0.5, 0.0), 0.4);  // Small inner circles
c3 = Circle(vec2(0.5, 0.0), 0.4);
```

Animated radii:

Make the circles pulse:

```
c1.radius = 0.5 + 0.2 * sin(iTime);
c2.radius = 0.5 + 0.2 * sin(iTime + 2.0 * 3.14159 / 3.0);
c3.radius = 0.5 + 0.2 * sin(iTime + 4.0 * 3.14159 / 3.0);
```

Watch the fractal breathe!

Alternating inversion patterns:

Instead of inverting through whichever circle contains the point, try a fixed cycling pattern:

```
// Cycle through circles in order
int circle_index = i % 3;
if(circle_index == 0) {
    p = invertCircle(p, c1.center, c1.radius);
} else if(circle_index == 1) {
    p = invertCircle(p, c2.center, c2.radius);
} else {
    p = invertCircle(p, c3.center, c3.radius);
}
```

This creates very different patterns—more regular and less space-filling than the gasket, but with interesting self-similarity.

Challenge problems: - Can you create a configuration that tiles the plane with circular patterns? - What about spiraling structures? - Can you make a fractal that's asymmetric (not rotationally symmetric)? - Try combining inversion with other transformations (rotation, scaling)

The key is experimentation—try different setups and see what emerges!

2.11. Looking Ahead

Tomorrow we continue with geometric transformations, but move from Euclidean to **hyperbolic geometry**. The circle inversions you've learned today are actually hyperbolic isometries—transformations that preserve distances in hyperbolic space!

We'll explore: - **Multiple models** of the hyperbolic plane (upper half-plane, Poincaré disk) - **Geodesics** (straight lines in hyperbolic geometry—they look like circles in our Euclidean view!) - **Triangle tilings** with $(2, 3, 7)$ and $(2, 3, \infty)$ symmetry groups - **Why hyperbolic space is different**: triangles with angle sum less than π, exponential growth of area

The same algorithmic ideas we've used today (iterated geometric transformations, escape-time coloring, distance-based rendering) will work in hyperbolic space. But the results will be even more intricate because hyperbolic space has "more room" than Euclidean space—allowing denser tilings and more complex fractal structures.

Make sure you're comfortable with: - **Iteration and escape-time algorithms** (we'll use similar ideas for tiling) - **Circle inversion** (this becomes reflection through hyperbolic geodesics!) - **Structs** for organizing geometric data - **Coloring strategies** based on orbit behavior

See you tomorrow!

3. Day 3: Geometric Tilings in Euclidean and Hyperbolic Space

3.1. Overview

Today we explore geometric tilings through reflection operations. We'll start by building a general framework for reflections in Euclidean space, then venture into hyperbolic geometry where the same algorithmic approach produces dramatically different patterns.

The key insight: **the algorithm stays the same across geometries—only the reflection operations change.** This mirrors what we saw on Day 2 with the Apollonian gasket: iteratively apply a geometric transformation until we reach a desired region. But today we'll understand *why* this works through the lens of group theory.

By the end of today, you'll understand: - How to construct reflections using linear algebra - **Why the folding algorithm works** (reflection groups and fundamental domains) - The structure of hyperbolic geometry in the upper half-plane model - How to implement hyperbolic triangle tilings using the same algorithmic pattern - How to convert between different models of hyperbolic space - The connection between circle inversion (Day 2) and hyperbolic isometries

> **i** Roadmap for Today
>
> **Part 1: Euclidean Geometry** - Simple tilings (strip, square) - Half-space abstraction - Triangle tilings - **Why this works**: Reflection groups
> **Part 2: Hyperbolic Geometry** - The upper half-plane model and metric - Geodesics and reflections (connection to Day 2!) - Triangle tilings in $\mathbb{H}^2$ - Multiple models (Poincaré disk, Klein) - Historical context and applications
> The unifying theme is **geometric transformations and their groups**—the same mathematical structure underlies fractals, tilings, and symmetry across all geometries.

3.2. Part 1: Reflection and Tilings in Euclidean Geometry

3.2.1. Starting Simple: The Folding Algorithm

Before we dive into general theory, let's build intuition with the simplest possible example: creating a repeating strip pattern.

3.2.1.1. Tiling a Strip

Imagine we want to tile the plane horizontally. We'll define a fundamental domain—the strip $0 < x < 1$—and reflect any point outside this strip back inside.

The algorithm is remarkably simple: - If $x < 0$, reflect across $x = 0$ - If $x > 1$, reflect across $x = 1$ - Repeat until the point stops moving

For a vertical line at $x = c$, reflection just flips the x-coordinate: $(x, y) \mapsto (2c - x, y)$.

Here's a complete shader:

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

    // Fold into the strip [0, 1]
    for(int i = 0; i < 20; i++) {
        if(p.x < 0.0) p.x = -p.x;
        if(p.x > 1.0) p.x = 2.0 - p.x;
    }

    // Draw something in the fundamental domain
    vec3 color = vec3(0.2, 0.2, 0.3);  // Dark background

    // A circle in the strip
    float d = length(p - vec2(0.5, 0.0));
    if(d < 0.3) {
        color = vec3(1.0, 0.8, 0.3);  // Yellow circle
    }

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

You should see the yellow circle repeat infinitely across the screen! We only drew it once, but the folding algorithm tiles it everywhere.

What's happening geometrically? Every point on the screen gets mapped back to the fundamental domain $[0, 1]$. Points that were in reflected copies of the domain get folded back through a sequence of reflections. Since we draw the same pattern in the fundamental domain, all the reflected copies show the same pattern.

3.2.1.2. Square Tiling

Let's extend to two dimensions. Now we have four boundaries: $x = 0$, $x = 1$, $y = 0$, and $y = 1$. Same algorithm, just more boundaries to check:

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

    // Fold into the square [0,1] × [0,1]
    for(int i = 0; i < 20; i++) {
        if(p.x < 0.0) p.x = -p.x;
        if(p.x > 1.0) p.x = 2.0 - p.x;
        if(p.y < 0.0) p.y = -p.y;
        if(p.y > 1.0) p.y = 2.0 - p.y;
    }

    // Draw something in the fundamental domain
    vec3 color = vec3(0.2, 0.2, 0.3);

    // Circle at center
    float d = length(p - vec2(0.5, 0.5));
    if(d < 0.3) {
        color = vec3(1.0, 0.8, 0.3);
    }

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

Perfect! A full 2D tiling.

3.2.1.3. Square Tiling with Fold Count

Let's track how many reflections were needed. This helps us understand the geometry and creates beautiful visualizations:

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

```
    // Fold into the square [0,1] × [0,1]
    int foldCount = 0;
    for(int i = 0; i < 20; i++) {
        vec2 p_old = p;

        if(p.x < 0.0) p.x = -p.x;
        if(p.x > 1.0) p.x = 2.0 - p.x;
        if(p.y < 0.0) p.y = -p.y;
        if(p.y > 1.0) p.y = 2.0 - p.y;

        // If point didn't move, we're done
        if(length(p - p_old) < 0.0001) break;
        foldCount++;
    }

    // Color based on fold count
    float t = float(foldCount) / 8.0;
    vec3 color = 0.5 + 0.5 * cos(6.28318 * (vec3(1.0) * t + vec3(0.0, 0.33, 0.67)));

    // Draw something in the fundamental domain
    float d = length(p - vec2(0.5, 0.5));
    if(d < 0.3) {
        color = mix(color, vec3(1.0, 1.0, 0.3), smoothstep(0.3, 0.25, d));
    }

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

Beautiful! The color gradient shows how many reflections were needed—points near the fundamental domain require few iterations, while points far away need many.

Notice the convergence check: we save the old position and check if the point stopped moving. When no boundary causes a reflection, the point has reached the fundamental domain and we can stop.

> **Computational Efficiency: Iteration Count**
>
> Why does this converge so quickly? Each reflection moves the point strictly closer to the fundamental domain (in the sense of reducing the number of boundary crossings). For a viewport of size 4×4 and fundamental domain of size 1×1, we need at most $\lceil \log_2(4) \rceil = 2$ reflections per axis, so 4 reflections total in the worst case.
> The 20-iteration limit is very conservative—most pixels converge in under 5 iterations. We could dynamically adjust this based on the coordinate scale, but for real-time rendering, a fixed conservative bound works well.

3.2.2. Abstracting: Half-Spaces

Looking at our square tiling code, we see repetition: check a boundary, reflect if outside, repeat. Let's abstract this pattern so we can handle arbitrary shapes.

3.2.2.1. What is a Half-Space?

A **half-space** is one side of a line. Any line $ax + by = c$ divides the plane into two regions: - Points where $ax + by < c$ - Points where $ax + by > c$

We'll encode a half-space by storing the line parameters and which side we want:

```
struct HalfSpace {
    float a, b, c;  // Line parameters: ax + by = c
    float side;     // +1 or -1 for which side
};
```

The `side` parameter determines which inequality we want: - `side = -1.0` means we want $ax+by > c$ (equivalently, $(ax + by - c) \cdot (-1) < 0$) - `side = 1.0` means we want $ax + by < c$ (equivalently, $(ax + by - c) \cdot (1) < 0$)

This might seem redundant—we could always use $ax + by < c$ and just flip the signs of a, b, c to get the other side. But having an explicit `side` parameter makes the code clearer and will be essential in hyperbolic geometry where sign-flipping doesn't work as cleanly.

3.2.2.2. Visualizing Half-Spaces

Before we implement reflections, let's visualize what a half-space is. Here's a shader that colors one side of a line:

```
struct HalfSpace {
    float a, b, c;
    float side;
};

bool inside(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;
    return (value - hs.c) * hs.side < 0.0;
}

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

    // Define a half-space: x < 1 (left side of vertical line at x=1)
    HalfSpace hs = HalfSpace(1.0, 0.0, 1.0, 1.0);
```

```
    // Color based on whether we're inside
    vec3 color = inside(p, hs) ? vec3(0.3, 0.5, 0.7) : vec3(0.1, 0.1, 0.2);

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

You should see the left side of the line colored blue, the right side dark. Try changing the half-space parameters to see how it affects the coloring!

> Exercise: Drawing the Boundary Line
>
> Want to see where the line is? Add this distance function:
>
> ```
> float distToHalfSpace(vec2 p, HalfSpace hs) {
>     return abs(hs.a * p.x + hs.b * p.y - hs.c) / length(vec2(hs.a, hs.b));
> }
> ```
>
> Then draw the line:
>
> ```
> float d = distToHalfSpace(p, hs);
> if(d < 0.02) color = vec3(1.0);  // White boundary
> ```
>
> This computes the perpendicular distance from the point to the line, then colors points near the line white. See Appendix E4b for the complete implementation!

3.2.2.3. Intersecting Half-Spaces: Making a Square

Now let's intersect four half-spaces to create a square region. We'll use **additive coloring**—each half-space we're inside adds to the color, so the interior (inside all four) will be brightest:

```
struct HalfSpace {
    float a, b, c;
    float side;
};

bool inside(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;
    return (value - hs.c) * hs.side < 0.0;
}

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

```
    // Define the four half-spaces for [0,1] × [0,1]
    HalfSpace left   = HalfSpace(1.0, 0.0, 0.0, -1.0);  // x > 0
    HalfSpace right  = HalfSpace(1.0, 0.0, 1.0,  1.0);  // x < 1
    HalfSpace bottom = HalfSpace(0.0, 1.0, 0.0, -1.0);  // y > 0
    HalfSpace top    = HalfSpace(0.0, 1.0, 1.0,  1.0);  // y < 1

    // Additive coloring - each half-space adds brightness
    vec3 color = vec3(0.0);

    if(inside(p, left))   color += vec3(0.1, 0.15, 0.2);
    if(inside(p, right))  color += vec3(0.1, 0.15, 0.2);
    if(inside(p, bottom)) color += vec3(0.1, 0.15, 0.2);
    if(inside(p, top))    color += vec3(0.1, 0.15, 0.2);

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

You should see the square region brightest (inside all four half-spaces), with regions inside fewer half-spaces progressively darker. This additive approach makes it easy to see how the regions overlap!

Exercise: Improved Visualization

For a cleaner look, you might want to:

1. **Binary coloring** (inside domain or not):

```
bool in_square = inside(p, left) && inside(p, right) &&
                 inside(p, bottom) && inside(p, top);
vec3 color = in_square ? vec3(0.4, 0.6, 0.8) : vec3(0.1, 0.1, 0.2);
```

2. **Draw boundaries** using the distance function from earlier
3. **Create an `insideDomain()` function**:

```
bool insideDomain(vec2 p, HalfSpace hs1, HalfSpace hs2,
                  HalfSpace hs3, HalfSpace hs4) {
    return inside(p, hs1) && inside(p, hs2) &&
           inside(p, hs3) && inside(p, hs4);
}
```

See Appendix E5b for complete enhanced versions!

3.2.2.4. Three Half-Spaces Make a Triangle

Let's visualize three half-spaces defining a triangle:

```
struct HalfSpace {
    float a, b, c;
    float side;
};

bool inside(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;
    return (value - hs.c) * hs.side < 0.0;
}

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

    // Define three half-spaces for equilateral triangle
    HalfSpace hs1 = HalfSpace(1.5, -0.866, -0.866, -1.0);
    HalfSpace hs2 = HalfSpace(0.0, 1.732, -0.866, -1.0);
    HalfSpace hs3 = HalfSpace(-1.5, -0.866, -0.866, -1.0);

    // Additive coloring
    vec3 color = vec3(0.0);

    if(inside(p, hs1)) color += vec3(0.15, 0.2, 0.25);
    if(inside(p, hs2)) color += vec3(0.15, 0.2, 0.25);
    if(inside(p, hs3)) color += vec3(0.15, 0.2, 0.25);

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

You should see a triangle region where all three half-spaces overlap! The additive coloring helps visualize the structure.

3.2.2.5. The Reflection Formula

Now we're ready to implement reflection. To reflect a point $\mathbf{p} = (x, y)$ across the line $ax + by = c$, we use linear algebra. The normal vector to the line is $\mathbf{n} = (a, b)$. After normalizing to $\hat{\mathbf{n}} = \mathbf{n}/|\mathbf{n}|$, the reflection formula is:

$$\mathbf{p}' = \mathbf{p} - 2d\hat{\mathbf{n}}$$

where d is the signed distance from $\mathbf{p}$ to the line:

$$d = \frac{ax + by - c}{\sqrt{a^2 + b^2}}$$

This is a standard result from linear algebra! The signed distance tells us how far we are from the line (positive on one side, negative on the other), and we move twice that distance in the normal direction to get the reflection.

Our `reflectInto` function checks if we're on the correct side and only reflects if necessary—it **extends** our `inside()` test by conditionally reflecting:

```
vec2 reflectInto(vec2 p, HalfSpace hs) {
    // Compute which side of the line we're on
    float value = hs.a * p.x + hs.b * p.y;

    // Check if we're already on the correct side (this is our inside() test!)
    if((value - hs.c) * hs.side < 0.0) {
        return p;  // Already inside, nothing to do
    }

    // We're on the wrong side - reflect across the boundary line
    vec2 normal = vec2(hs.a, hs.b);
    float norm = length(normal);
    normal = normal / norm;

    float signedDist = (value - hs.c) / norm;
    return p - 2.0 * signedDist * normal;
}
```

This function encapsulates the entire pattern: check if we're on the correct side (the `inside()` test), and only reflect if we're not. So `reflectInto()` extends and renames our visualization function to also perform the reflection!

3.2.3. Square Tiling with Half-Spaces

Let's rewrite our square tiling using this abstraction. For the square $[0, 1] \times [0, 1]$, we need four half-spaces:

- **Left edge** ($x = 0$): We want $x > 0 \longrightarrow$ `HalfSpace(1.0, 0.0, 0.0, -1.0)`
- **Right edge** ($x = 1$): We want $x < 1 \longrightarrow$ `HalfSpace(1.0, 0.0, 1.0, 1.0)`
- **Bottom and top**: Similarly for y

Complete shader:

```
struct HalfSpace {
    float a, b, c;
    float side;
};
```

```
vec2 reflectInto(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;

    if((value - hs.c) * hs.side < 0.0) {
        return p;
    }

    vec2 normal = vec2(hs.a, hs.b);
    float norm = length(normal);
    normal = normal / norm;

    float signedDist = (value - hs.c) / norm;
    return p - 2.0 * signedDist * normal;
}

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

    // Define the four half-spaces for [0,1]×[0,1]
    HalfSpace left   = HalfSpace(1.0, 0.0, 0.0, -1.0);
    HalfSpace right  = HalfSpace(1.0, 0.0, 1.0,  1.0);
    HalfSpace bottom = HalfSpace(0.0, 1.0, 0.0, -1.0);
    HalfSpace top    = HalfSpace(0.0, 1.0, 1.0,  1.0);

    // Fold into the square
    int foldCount = 0;
    for(int i = 0; i < 20; i++) {
        vec2 p_old = p;

        p = reflectInto(p, left);
        p = reflectInto(p, right);
        p = reflectInto(p, bottom);
        p = reflectInto(p, top);

        if(length(p - p_old) < 0.0001) break;
        foldCount++;
    }

    // Color based on fold count
    float t = float(foldCount) / 8.0;
    vec3 color = 0.5 + 0.5 * cos(6.28318 * (vec3(1.0) * t + vec3(0.0, 0.33, 0.67)));

    // Draw something in fundamental domain
    float d = length(p - vec2(0.5, 0.5));
    if(d < 0.3) {
```

```
        color = mix(color, vec3(1.0, 1.0, 0.3), smoothstep(0.3, 0.25, d));
    }

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

This looks identical to our earlier version, but now our code is flexible. The beauty: **changing from a square to a triangle only requires changing the half-space definitions!**

3.2.4. Triangle Tiling

Now we're ready for triangles. We'll use an equilateral triangle with vertices at:

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

This triangle is centered at the origin with one vertex pointing up.

3.2.4.1. Computing Half-Spaces from Edges

For each edge, we need to compute the line parameters (a, b, c) and determine the correct side. The process:

1. Take two vertices defining an edge: $\mathbf{v}_i$ and $\mathbf{v}_j$
2. Compute edge direction: $\mathbf{d} = \mathbf{v}_j - \mathbf{v}_i$
3. Compute perpendicular (rotate 90° counterclockwise): $\mathbf{n} = (-d_y, d_x)$
4. Line equation: $n_x \cdot x + n_y \cdot y = c$ where $c = \mathbf{n} \cdot \mathbf{v}_i$
5. Test origin: if $(n_x \cdot 0 + n_y \cdot 0 - c) < 0$, then `side = -1.0`, else `side = 1.0`

> **i** Derivation of Triangle Half-Space Parameters
>
> **Edge from $v_0 = (0, 1)$ to $v_1 = (-\sqrt{3}/2, -1/2)$:**
>
> - Edge direction: $\mathbf{d} = v_1 - v_0 = (-0.866, -1.5)$
> - Perpendicular (90° CCW): $\mathbf{n} = (1.5, -0.866)$
> - Line: $1.5x - 0.866y = c$ where $c = \mathbf{n} \cdot v_0 = (1.5)(0) + (-0.866)(1) = -0.866$
> - For origin (inside): $1.5(0) - 0.866(0) - (-0.866) = 0.866 > 0$
> - We want inside when $(ax + by - c) < 0$, so we need `side = -1.0`
>
> **Result:** `HalfSpace(1.5, -0.866, -0.866, -1.0)`
> Similar calculations give: - **Edge v_1 to v_2:** `HalfSpace(0.0, 1.732, -0.866, -1.0)` - **Edge v_2 to v_0:** `HalfSpace(-1.5, -0.866, -0.866, -1.0)`

The computed parameters are:

- **Edge v_0 to v_1:** `HalfSpace(1.5, -0.866, -0.866, -1.0)`

- **Edge v_1 to v_2:** `HalfSpace(0.0, 1.732, -0.866, -1.0)`
- **Edge v_2 to v_0:** `HalfSpace(-1.5, -0.866, -0.866, -1.0)`

We already verified these work with our visualization shader above!

3.2.4.2. Triangle Tiling Shader

Now we can implement the tiling by copying our reflection shader and just changing the half-spaces:

```
struct HalfSpace {
    float a, b, c;
    float side;
};

vec2 reflectInto(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;

    if((value - hs.c) * hs.side < 0.0) {
        return p;
    }

    vec2 normal = vec2(hs.a, hs.b);
    float norm = length(normal);
    normal = normal / norm;

    float signedDist = (value - hs.c) / norm;
    return p - 2.0 * signedDist * normal;
}

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

    // Define the three half-spaces for equilateral triangle
    HalfSpace hs1 = HalfSpace(1.5, -0.866, -0.866, -1.0);
    HalfSpace hs2 = HalfSpace(0.0, 1.732, -0.866, -1.0);
    HalfSpace hs3 = HalfSpace(-1.5, -0.866, -0.866, -1.0);

    // Fold into the triangle
    int foldCount = 0;
    for(int i = 0; i < 20; i++) {
        vec2 p_old = p;

        p = reflectInto(p, hs1);
```

```
        p = reflectInto(p, hs2);
        p = reflectInto(p, hs3);

        if(length(p - p_old) < 0.0001) break;
        foldCount++;
    }

    // Color by fold count parity
    float parity = mod(float(foldCount), 2.0);
    vec3 color;
    if(parity < 0.5) {
        color = vec3(0.7, 0.8, 0.9);  // Light blue
    } else {
        color = vec3(0.5, 0.6, 0.8);  // Darker blue
    }

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

Beautiful! You should see an infinite triangle tiling. The alternating colors show which triangles are orientation-preserving vs orientation-reversing reflections of the fundamental domain.

Exercise: Visualizing Triangle Structure

Want to see the edges and vertices of your triangles? This requires computing distances to half-spaces and vertices.
For edges (drawing the boundaries):

```
float distToHalfSpace(vec2 p, HalfSpace hs) {
    return abs(hs.a * p.x + hs.b * p.y - hs.c) / length(vec2(hs.a, hs.b));
}

// In main rendering:
float d1 = distToHalfSpace(p, hs1);
float d2 = distToHalfSpace(p, hs2);
float d3 = distToHalfSpace(p, hs3);
float border = min(d1, min(d2, d3));

if(border < 0.02) color = vec3(1.0);  // White edges
```

For vertices (marking the corners):

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

// Check distance after folding
float dv0 = length(p - v0);
float dv1 = length(p - v1);
float dv2 = length(p - v2);
float vertex_dist = min(dv0, min(dv1, dv2));

if(vertex_dist < 0.05) color = vec3(1.0, 0.0, 0.0);  // Red vertices
```

See Appendix E9 for the complete enhanced version!

3.2.5. Why Does This Algorithm Work? Reflection Groups

We've implemented the folding algorithm, but *why* does it work? Why does iteratively reflecting guarantee we reach the fundamental domain? The answer lies in **group theory**.

3.2.5.1. Reflections Generate a Group

Each reflection r_i across a half-space boundary is an **isometry** of the Euclidean plane—it preserves distances and angles. Composing reflections gives us more isometries. The set of all compositions of our reflections forms a **group** under composition:

- **Identity**: Reflecting twice across the same line returns to the original point ($r_i \circ r_i = \text{id}$)
- **Closure**: Composing reflections gives another isometry (which might be a reflection, rotation, or glide reflection)
- **Inverses**: Every isometry has an inverse (just reflect again)
- **Associativity**: Composition is associative

This group, generated by reflections across the boundaries of our fundamental domain, is called a **reflection group** or **Coxeter group**.

3.2.5.2. The Fundamental Domain and Orbit

Our fundamental domain F (the square $[0, 1]^2$ or triangle) is a **fundamental domain** for the group action. This means:

1. Every point in the plane is equivalent to exactly one point in F (modulo boundary points)
2. The **orbit** of F under the group (all images $g(F)$ for g in the group) tiles the entire plane
3. Different tiles $g(F)$ and $h(F)$ only overlap on their boundaries

When we start with a point p outside F, there exists a sequence of group elements (reflections) that maps p into F. Our algorithm finds this sequence!

3.2.5.3. Why the Algorithm Terminates

Here's the key insight: each reflection across a boundary of F either: - Keeps the point inside F (if it's already on the correct side) - Moves the point strictly closer to F (if it's on the wrong side)

"Closer" here means we reduce some discrete measure—like the number of boundaries we're on the wrong side of. Since this number is finite and decreases with each reflection, the algorithm must terminate.

Formally, we can define a **height function** $h(p)$ that counts how many half-space boundaries p violates. Initially $h(p) \geq 0$. Each reflection that actually moves the point decreases $h(p)$ by at least 1. When $h(p) = 0$, the point is inside F and the algorithm stops.

> **!** The Pattern Across Days
>
> This is the same fundamental principle we've seen throughout:
> **Day 2 (Apollonian gasket):** - Group: Iterated circle inversions - Fundamental domain: The gaps between circles - Algorithm: Invert until inside the domain
> **Day 3 (Euclidean tilings):** - Group: Reflections across boundaries - Fundamental domain: The square/triangle - Algorithm: Reflect until inside the domain
> **Day 3 (Hyperbolic, coming soon):** - Group: Hyperbolic reflections (same structure!) - Fundamental domain: Hyperbolic triangle - Algorithm: Reflect until inside the domain (identical code!)
> The unifying theme is **group actions and fundamental domains**. We're always finding the unique representative of an orbit that lies in the fundamental domain.

3.2.5.4. Computational Implications

Understanding the group theory gives us insight into the computation:

Convergence rate: For a viewport of size V and fundamental domain of size F, we need at most $O(\log(V/F))$ reflections per coordinate axis. This is why small iteration limits (20-30) work well.

Parallelism: Each pixel's orbit is independent—perfect for GPU parallelism. Millions of pixels computing orbits simultaneously with no communication needed.

Threshold choice: The `0.0001` threshold for detecting convergence balances precision and performance. Smaller thresholds catch more subtle movements but risk floating-point noise; larger thresholds might terminate early but rarely matter for visualization.

Why it's efficient: Most tiles are "nearby" in the group—they're reached by short sequences of reflections. Only tiles far from the origin require many reflections, and these appear very small on screen (contributing few pixels).

This completes our Euclidean foundation. We now understand: 1. The folding algorithm in concrete examples 2. The half-space abstraction that makes it general 3. **Why it works**: reflection groups and fundamental domains 4. Computational properties: convergence, parallelism, efficiency

Next, we'll take this exact algorithmic structure into hyperbolic geometry!

3.3. Part 2: Hyperbolic Geometry

3.3.1. Introduction to Hyperbolic Geometry

Hyperbolic geometry is one of the three classical geometries (Euclidean, spherical, and hyperbolic), characterized by constant **negative curvature**. For over two millennia, mathematicians believed Euclidean geometry was the only logically consistent geometry—Euclid's parallel postulate seemed necessary. The discovery of hyperbolic geometry in the early 19th century revolutionized mathematics.

3.3.1.1. Historical Context: The Discovery

The story of hyperbolic geometry is one of the great dramas in mathematical history. For centuries, mathematicians tried to prove Euclid's fifth postulate (the parallel postulate) from the other four axioms. What if you could have multiple parallel lines through a point?

Three mathematicians independently discovered that this "impossible" geometry was actually perfectly consistent:

János Bolyai (1802-1860), a Hungarian mathematician, developed hyperbolic geometry in the 1820s. His father, a mathematician himself, warned him: "For God's sake, I beseech you, give it up. Fear it no less than sensual passions because it too may take all your time and deprive you of your health, peace of mind and happiness in life." But János persisted, publishing his work in 1832 as an appendix to his father's book.

Nikolai Lobachevsky (1792-1856), a Russian mathematician, published the first account of hyperbolic geometry in 1829. He called it "imaginary geometry" and faced considerable resistance from the mathematical establishment. His work was largely ignored during his lifetime.

Carl Friedrich Gauss (1777-1855), the "Prince of Mathematicians," had discovered hyperbolic geometry even earlier but never published it. In his private correspondence, he revealed he'd been working on non-Euclidean geometry since the 1790s but feared the "clamor of the Boeotians" (his term for mathematical philistines). When he read Bolyai's work in 1832, he wrote that he could not praise it "because to praise it would be to praise myself"—he'd discovered the same results years earlier but kept them private.

The discovery had profound implications: geometry was not a single truth about space but a family of possible consistent systems. This philosophical shift influenced everything from Einstein's general relativity (which uses non-Euclidean geometry for curved spacetime) to modern physics and mathematics.

3.3.1.2. Modern Applications

Hyperbolic geometry appears throughout modern mathematics and physics:

- **Complex analysis**: The upper half-plane model is fundamental to the theory of modular forms, elliptic curves, and the Riemann mapping theorem
- **Number theory**: The action of $SL(2, \mathbb{Z})$ on $\mathbb{H}^2$ produces modular forms—functions crucial to the proof of Fermat's Last Theorem
- **Topology**: The study of 3-manifolds and knot theory often requires understanding hyperbolic structures
- **Teichmüller theory**: Moduli spaces of Riemann surfaces have natural hyperbolic metrics

- **Kleinian groups**: Discrete subgroups of hyperbolic isometries produce fractal limit sets (like we saw with the Apollonian gasket!)
- **General relativity**: Anti-de Sitter space has constant negative curvature—hyperbolic geometry in spacetime
- **Machine learning**: Recent work uses hyperbolic embeddings to represent hierarchical data efficiently

3.3.1.3. Key Properties

What makes hyperbolic geometry different from Euclidean geometry?

Parallel lines: Given a line and a point not on it, there are **infinitely many** lines through the point that don't intersect the given line (all parallel to it). This is the defining feature that distinguishes hyperbolic geometry.

Triangle angles: The sum of angles in a triangle is **less than** π. In fact, the **area** of a hyperbolic triangle with angles α, β, γ is exactly:

$$\text{Area} = \pi - (\alpha + \beta + \gamma)$$

This is the **Gauss-Bonnet theorem** for hyperbolic triangles—a beautiful connection between geometry (angles) and topology (area).

Exponential growth: In Euclidean geometry, the circumference of a circle grows linearly with radius ($C = 2\pi r$). In hyperbolic geometry, it grows **exponentially**: $C \sim e^r$ for large r. This means hyperbolic space has "more room" than Euclidean space—there's exponentially more area at distance r from a point.

No similarity: In Euclidean geometry, you can scale any shape—a small triangle and a large triangle with the same angles are similar. In hyperbolic geometry, there's an absolute unit of length built into the curvature. All triangles with angles $(\pi/2, \pi/3, \pi/7)$ are congruent—there's no "scaled version." This makes hyperbolic geometry richer but more rigid.

These properties create the "extra room" that allows much richer tiling structures than Euclidean geometry.

3.3.2. The Upper Half-Plane Model

We'll work in the **upper half-plane model** of hyperbolic geometry, denoted $\mathbb{H}^2$:

$$\mathbb{H}^2 = \{z = x + iy \in \mathbb{C} \,:\, y > 0\}$$

This is just complex numbers with positive imaginary part—the upper half of the complex plane. The **real axis** $\{y = 0\}$ forms the boundary "at infinity"—it's not actually part of $\mathbb{H}^2$, but represents points infinitely far away in hyperbolic distance.

3.3.2.1. The Hyperbolic Metric

The **hyperbolic metric** is what makes $\mathbb{H}^2$ a hyperbolic space:

$$ds^2 = \frac{dx^2 + dy^2}{y^2}$$

This gives $\mathbb{H}^2$ the structure of a complete Riemannian manifold with constant curvature -1.

What does this mean? The factor $1/y^2$ is a **conformal factor** that scales the Euclidean metric. As $y \to 0$ (approaching the boundary), this scaling factor blows up—distances that look small Euclidean-wise are enormous hyperbolically. As $y \to \infty$ (going "up" in the upper half-plane), the scaling factor goes to zero—large Euclidean distances are actually finite hyperbolically.

The metric is **conformal** to the Euclidean metric—it preserves angles but not lengths. If two curves meet at angle θ in the Euclidean sense, they also meet at angle θ in the hyperbolic sense! This is why you can trust your eyes when looking at pictures—angles are what they appear to be.

3.3.2.2. The Distance Formula

Integrating the metric along paths gives the **hyperbolic distance** between two points $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$:

$$d_{\mathbb{H}^2}(z_1, z_2) = \operatorname{arcosh}\left(1 + \frac{|z_1 - z_2|^2}{2y_1 y_2}\right)$$

where $|z_1 - z_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ is the usual Euclidean distance.

Notice the $1/y_1 y_2$ factor—points near the boundary (y small) are very far apart hyperbolically even if they're close Euclidean-wise.

Derivation sketch: For a vertical line from $z_1 = x + iy_1$ to $z_2 = x + iy_2$, the hyperbolic length is:

$$\int_{y_1}^{y_2} \frac{dy}{y} = \log(y_2) - \log(y_1) = \log(y_2/y_1)$$

For a general path, you need to integrate along the geodesic connecting the points (which might not be a straight Euclidean line), giving the arcosh formula above.

3.3.2.3. Visualizing Hyperbolic Distance

Let's make this concrete with an interactive shader. We'll start with Euclidean distance, then switch to hyperbolic to see the difference.

Euclidean distance circles:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 4.0;
    uv.x *= iResolution.x / iResolution.y;
    vec2 p = uv + vec2(0.0, 1.5);  // Shift up so we're in y > 0

    // Mouse position as center (or default)
    vec2 mouse = iMouse.xy / iResolution.xy;
    if(iMouse.z < 0.5) mouse = vec2(0.5, 0.7);  // Default if no click
    mouse = (mouse - 0.5) * 4.0;
    mouse.x *= iResolution.x / iResolution.y;
    vec2 center = mouse + vec2(0.0, 1.5);

    // Euclidean distance
    float dist = length(p - center);

    // Draw a disk of radius 0.5 using two circles
    float radius = 0.5;
    vec3 color = vec3(0.1, 0.1, 0.2);  // Background

    // Outer circle (slightly larger)
    if(dist < radius + 0.02) {
        color = vec3(1.0, 1.0, 0.3);  // Yellow ring
    }

    // Inner circle (slightly smaller) - "cuts out" interior
    if(dist < radius - 0.02) {
        color = vec3(0.4, 0.6, 0.8);  // Blue interior
    }

    // Draw center point
    if(length(p - center) < 0.05) {
        color = vec3(1.0, 0.0, 0.0);
    }

    // Darken outside upper half-plane
    if(p.y < 0.0) {
        color *= 0.3;
    }

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

Click and drag around—the circle stays the same size everywhere. This is Euclidean distance: uniform across the plane. Notice how we draw the boundary: we draw a filled circle at radius $r + \epsilon$ (outer edge), then draw another filled circle at radius $r - \epsilon$ (inner edge) in a different color. The ring between them is our boundary!

Hyperbolic distance circles:

```
float hyperbolicDistance(vec2 z1, vec2 z2) {
    vec2 diff = z1 - z2;
    float diff2 = dot(diff, diff);
    float denom = 2.0 * z1.y * z2.y;
    float arg = 1.0 + diff2 / denom;
    return log(arg + sqrt(arg * arg - 1.0));  // arccosh(arg)
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 4.0;
    uv.x *= iResolution.x / iResolution.y;
    vec2 p = uv + vec2(0.0, 1.5);

    // Mouse position as center
    vec2 mouse = iMouse.xy / iResolution.xy;
    if(iMouse.z < 0.5) mouse = vec2(0.5, 0.7);
    mouse = (mouse - 0.5) * 4.0;
    mouse.x *= iResolution.x / iResolution.y;
    vec2 center = mouse + vec2(0.0, 1.5);

    // Hyperbolic distance
    float dist = hyperbolicDistance(p, center);

    // Draw a hyperbolic disk using two "circles"
    float radius = 0.5;
    vec3 color = vec3(0.1, 0.1, 0.2);  // Background

    // Outer boundary
    if(dist < radius + 0.05) {
        color = vec3(1.0, 1.0, 0.3);  // Yellow ring
    }

    // Inner region
    if(dist < radius - 0.05) {
        color = vec3(0.4, 0.6, 0.8);  // Blue interior
    }

    // Draw center
    if(hyperbolicDistance(p, center) < 0.1) {
        color = vec3(1.0, 0.0, 0.0);
    }

    // Darken outside upper half-plane
    if(p.y < 0.0) {
        color *= 0.3;
    }
```

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

Now drag the center around! Notice how the "circle" changes shape as you move it. Near the bottom ($y \to 0$), the circle appears huge Euclidean-wise—that's because we're near the boundary where hyperbolic distances blow up. Higher up (y large), the circle appears smaller.

This visualization makes the $1/y^2$ conformal factor visceral: **hyperbolic space is compressed near the boundary**.

3.3.3. The Boundary at Infinity

The real axis $\{y = 0\}$ is not part of $\mathbb{H}^2$, but we can think of it as the **boundary at infinity**—points infinitely far away in hyperbolic distance.

Ideal points as equivalence classes: An **ideal point** on the boundary can be defined as an equivalence class of geodesics that asymptotically approach each other. Two geodesics are equivalent if the hyperbolic distance between them goes to zero as you go to infinity along them.

For example, the vertical line $\{x = 0\}$ and the vertical line $\{x = \epsilon\}$ (for small ϵ) both approach the point 0 on the real axis. As you go up ($y \to \infty$), the hyperbolic distance between corresponding points goes to zero—they're asymptotically parallel.

Geometric intuition: In the Poincaré disk model (which we'll see soon), the boundary at infinity is literally the unit circle $|z| = 1$. Points on this circle are infinitely far away from any interior point, but they still have geometric meaning—they represent directions or "points at infinity."

Why this matters: When we define hyperbolic triangles, we can have vertices "at infinity" on the boundary. These are called **ideal vertices** or **ideal triangles**. For instance, our $(2, 3, \infty)$ triangle has one vertex at the point ∞ in the upper half-plane (straight up the imaginary axis). The angle at an ideal vertex is zero—the sides become asymptotically parallel as they approach the boundary.

The Gauss-Bonnet theorem still works: for a triangle with angles α, β, γ, the area is $\pi - (\alpha + \beta + \gamma)$. If one angle is zero (ideal vertex), the area is $\pi - \alpha - \beta > 0$—ideal triangles have finite area!

3.3.4. Geodesics and Reflections

3.3.4.1. Geodesics in the Upper Half-Plane

Geodesics (the "straight lines" of hyperbolic geometry—curves that locally minimize distance) in $\mathbb{H}^2$ have exactly two forms:

1. **Vertical lines**: $\{x = c\}$ for any constant $c \in \mathbb{R}$
2. **Semicircles**: Centered on the real axis, perpendicular to it

These curves meet the boundary at right angles—this is the characterizing property of geodesics in this model.

Why these are geodesics: Reflections across these curves are isometries (they preserve the hyperbolic metric $ds^2 = \frac{dx^2+dy^2}{y^2}$). An isometry's fixed point set is always a geodesic! So we just need to verify that reflection across vertical lines and semicircles preserves the metric.

For **vertical lines**, this is obvious: reflecting across $x = c$ sends $(x, y) \mapsto (2c - x, y)$, which preserves both the Euclidean distance $dx^2 + dy^2$ and the y-coordinate, hence preserves $\frac{dx^2+dy^2}{y^2}$.

For **semicircles**, we use circle inversion from Day 2! A semicircle centered at $(c, 0)$ with radius R is preserved by the inversion:

$$z \mapsto c + R^2 \frac{z - c}{|z - c|^2}$$

Circle inversion is conformal (preserves angles) and scales distances by exactly $1/y^2$ near the boundary—precisely the conformal factor in the hyperbolic metric! So inversions through semicircles are hyperbolic isometries. Their fixed point sets (the semicircles themselves) are therefore geodesics.

> **!** Connection to Day 2: Circle Inversion is a Hyperbolic Isometry!
>
> This is a profound connection: the circle inversions we used on Day 2 for the Apollonian gasket were actually **hyperbolic isometries** all along! The Apollonian gasket lives in hyperbolic space—the gaps between circles are hyperbolic regions, and the inversions are reflections across hyperbolic geodesics (semicircles).
> When we iterated inversions on Day 2, we were doing exactly what we're doing today—finding the fundamental domain of a group action! The Apollonian gasket is a hyperbolic object, just like our triangle tilings. The same group-theoretic principles apply.
> This is why the techniques work across days: we're always working with group actions, whether we realize it or not. The mathematics unifies everything.

3.3.4.2. Implementing Reflections

For **vertical lines** (like $x = c$), reflection is simple—flip the x-coordinate:

```
vec2 reflectIntoVertical(vec2 z, float x_pos, float side) {
    // Check if we're on the correct side
    if((z.x - x_pos) * side < 0.0) {
        return z;  // Already on correct side
    }

    // Reflect: (x,y) → (2c - x, y)
    z.x = 2.0 * x_pos - z.x;
    return z;
}
```

For **semicircles** (geodesics from point p to point q on the real axis), we use circle inversion. The semicircle has center $(c, 0)$ where $c = (p + q)/2$ and radius $R = |p - q|/2$:

```glsl
vec2 reflectIntoCircular(vec2 z, float p, float q, float side) {
    float center = (p + q) / 2.0;
    float radius = abs(p - q) / 2.0;

    vec2 rel = z - vec2(center, 0.0);
    float dist2 = dot(rel, rel);  // Squared distance from center

    // Check if we're on the correct side
    if((dist2 - radius * radius) * side > 0.0) {
        return z;  // Already on correct side
    }

    // Circle inversion (from Day 2!)
    vec2 inverted = vec2(center, 0.0) + (radius * radius) * rel / dist2;
    return inverted;
}
```

Note: We're inverting through a circle in the Euclidean sense (using Euclidean distance `dist2`), but this operation is actually a hyperbolic isometry! The magic is that circle inversion's conformal properties exactly match the hyperbolic metric's requirements.

i Why Two Types of Reflections?

In Euclidean geometry, all reflections across lines look the same—just the orientation changes. Why do we need two different functions in hyperbolic geometry?
The answer is that we're working in a **model** of hyperbolic geometry (the upper half-plane). The vertical lines and semicircles are the images of geodesics in this model. In the intrinsic hyperbolic geometry, all reflections across geodesics are the same—there's only one type of reflection.
But when we represent hyperbolic geometry in the Euclidean upper half-plane, geodesics appear as two different types of curves (vertical lines and semicircles), so we need two different formulas. This is an artifact of the model, not the geometry itself.
In the Poincaré disk model (coming soon), ALL geodesics are circular arcs perpendicular to the boundary circle, so we'd only need one reflection function there!

3.3.5. The (2,3,∞) Triangle

Let's build our first hyperbolic triangle tiling. The notation (p, q, r) means the triangle has angles π/p, π/q, and π/r at its three vertices. So $(2, 3, \infty)$ means angles $\pi/2$, $\pi/3$, and 0 (an ideal vertex at infinity).

Why this triangle exists in hyperbolic geometry: The Gauss-Bonnet theorem tells us the area of a hyperbolic triangle with angles α, β, γ is:

$$\text{Area} = \pi - (\alpha + \beta + \gamma)$$

For a $(2, 3, \infty)$ triangle:

$$\text{Area} = \pi - \left(\frac{\pi}{2} + \frac{\pi}{3} + 0\right) = \pi - \frac{5\pi}{6} = \frac{\pi}{6} > 0$$

So this triangle has finite positive area and can tile the hyperbolic plane. In Euclidean geometry, $\pi/2 + \pi/3 = 5\pi/6 < \pi$, but that's not enough—we'd need the sum to equal *exactly* π to get zero curvature. Since $5\pi/6 < \pi$, there's negative curvature left over, making this a hyperbolic triangle.

Configuration: We'll use a particularly nice setup in the upper half-plane: - **Left boundary**: Vertical line at $x = -1$ - **Right boundary**: Vertical line at $x = 1$
- **Bottom boundary**: Unit semicircle from -1 to 1 (centered at origin, radius 1)

This creates a triangle with: - Two finite vertices at approximately $(-1, 0)$ and $(1, 0)$ (technically infinitesimally above the real axis) - One ideal vertex at ∞ (straight up the imaginary axis) - Angles of $\pi/2$ at the bottom two vertices (vertical line meets semicircle at right angles) - Angle of $\pi/3$ between the two vertical lines when measured hyperbolically - Angle of 0 at the ideal vertex ∞

3.3.5.1. Implementation

Here's the complete shader:

```
vec2 reflectIntoVertical(vec2 z, float x_pos, float side) {
    if((z.x - x_pos) * side < 0.0) return z;
    z.x = 2.0 * x_pos - z.x;
    return z;
}

vec2 reflectIntoCircular(vec2 z, float p, float q, float side) {
    float center = (p + q) / 2.0;
    float radius = abs(p - q) / 2.0;
    vec2 rel = z - vec2(center, 0.0);
    float dist2 = dot(rel, rel);
    if((dist2 - radius * radius) * side > 0.0) return z;

    return vec2(center, 0.0) + (radius * radius) * rel / dist2;
}

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

    // Shift to upper half-plane (need y > 0)
    vec2 z = uv + vec2(0.0, 1.5);

    // Fold into the (2,3,∞) triangle
    int foldCount = 0;
    for(int i = 0; i < 50; i++) {
        vec2 z_old = z;
```

```
        // Reflect across left vertical line (x = -1, want x > -1)
        z = reflectIntoVertical(z, -1.0, -1.0);

        // Reflect across right vertical line (x = 1, want x < 1)
        z = reflectIntoVertical(z, 1.0, 1.0);

        // Reflect across semicircle (from -1 to 1, want outside/above)
        z = reflectIntoCircular(z, -1.0, 1.0, 1.0);

        // If point didn't move, we're inside
        if(length(z - z_old) < 0.0001) break;
        foldCount++;
    }

    // Color by fold count parity
    float parity = mod(float(foldCount), 2.0);
    vec3 color;
    if(parity < 0.5) {
        color = vec3(0.7, 0.8, 0.9);  // Light blue
    } else {
        color = vec3(0.5, 0.6, 0.8);  // Darker blue
    }

    // Darken if below the real axis (outside hyperbolic space)
    if(z.y < 0.0) {
        color *= 0.3;
    }

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

You should see a beautiful hyperbolic tiling! Notice how the triangles appear to get smaller near the bottom of the screen (approaching the real axis $y = 0$)—they're all the same hyperbolic size, but Euclidean distances compress due to the $1/y^2$ metric.

The alternating colors show which tiles are orientation-preserving vs orientation-reversing reflections of the fundamental domain.

Compare to Euclidean

Look at the structure of this shader compared to the Euclidean triangle tiling:
Euclidean:

```
p = reflectInto(p, hs1);
p = reflectInto(p, hs2);
p = reflectInto(p, hs3);
```

Hyperbolic:

```
z = reflectIntoVertical(z, -1.0, -1.0);
z = reflectIntoVertical(z, 1.0, 1.0);
z = reflectIntoCircular(z, -1.0, 1.0, 1.0);
```

The algorithm is identical! We just have two types of reflection operations instead of one. This is the power of recognizing the pattern.

Computational Analysis

GPU Parallelism: Just like Days 1 and 2, this algorithm is embarrassingly parallel. Each pixel computes independently—no communication, no shared state, perfect for GPU architecture.
Convergence: The folding algorithm works for the same group-theoretic reasons as the Euclidean case. The three reflections generate a discrete group of hyperbolic isometries, and our fundamental triangle is a fundamental domain for this group's action on $\mathbb{H}^2$.
Precision issues: Near $y \to 0$, floating-point precision degrades. The large conformal factor $1/y^2$ amplifies small errors in distance calculations. This is why we darken the region $y < 0$—technically it's not part of the hyperbolic plane, but also our numerics become unreliable there.
Threshold 0.0001: This convergence threshold balances precision and performance. Smaller thresholds catch more subtle movements but risk getting stuck in floating-point noise; larger thresholds might terminate early. For visualization purposes, 0.0001 is a good sweet spot.
Memory efficiency: Each pixel only needs to store its current position z (two floats) and an iteration counter (one integer). No arrays, no history, no complex data structures—just stateless iteration. This is as memory-efficient as you can get!

Exercise: Visualizing Hyperbolic Triangle Structure

Want to see the edges and vertices of your hyperbolic triangles? This requires computing hyperbolic distances.
For hyperbolic distance:

```
float hyperbolicDistance(vec2 z1, vec2 z2) {
    vec2 diff = z1 - z2;
    float diff2 = dot(diff, diff);
    float denom = 2.0 * z1.y * z2.y;
    float arg = 1.0 + diff2 / denom;
    return log(arg + sqrt(arg * arg - 1.0));
}
```

For drawing edges (vertical lines):

```
// Distance to vertical line x = c
float distToVertical = abs(z.x - c);
if(distToVertical < 0.02) color = vec3(1.0);
```

For drawing edges (semicircular geodesics):

```
// Distance to semicircle from p to q
float center = (p + q) / 2.0;
float radius = abs(p - q) / 2.0;
float distToCircle = abs(length(z - vec2(center, 0.0)) - radius);
// Only draw if above real axis
if(z.y > 0.0 && distToCircle < 0.02) color = vec3(1.0);
```

For vertices:

```
// Check hyperbolic distance to vertices after folding
vec2 v1 = vec2(-1.0, 0.01);  // Left vertex (slightly above axis)
vec2 v2 = vec2(1.0, 0.01);   // Right vertex
if(hyperbolicDistance(z, v1) < 0.1 || hyperbolicDistance(z, v2) < 0.1) {
    color = vec3(1.0, 0.0, 0.0);  // Red vertices
}
```

See Appendix H5 for the complete enhanced implementation!

3.3.6. Multiple Models of Hyperbolic Space

The upper half-plane is just one way to represent hyperbolic geometry. There are several other models, each with advantages:

3.3.6.1. The Poincaré Disk Model

The **Poincaré disk model** represents all of $\mathbb{H}^2$ as the interior of the unit disk $\{z \in \mathbb{C} : |z| < 1\}$. The boundary circle $|z| = 1$ represents points at infinity.

Geodesics in this model are: - Diameters of the disk (straight lines through the origin) - Circular arcs perpendicular to the boundary circle

The conformal factor here is $\frac{4}{(1-|z|^2)^2}$, which blows up as $|z| \to 1$ (approaching the boundary).

Advantages: - The entire hyperbolic plane fits in a bounded region (the disk) - All geodesics look similar (circular arcs)—no distinction between vertical and circular - Visually intuitive for understanding the full structure at once

Disadvantages: - Harder to compute distances - More complex reflection formulas

3.3.6.2. Converting Between Models: The Cayley Transform

We can convert between the upper half-plane and Poincaré disk using the **Cayley transform** (also called the Möbius transformation):

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

This maps: - Upper half-plane $\{z : \text{Im}(z) > 0\} \longrightarrow$ Poincaré disk $\{w : |w| < 1\}$ - Real axis $\{z : \text{Im}(z) = 0\} \longrightarrow$ Unit circle $\{w : |w| = 1\}$ - Point i in the upper half-plane $\longrightarrow$ origin 0 in the disk

The inverse transform is:

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

Here's the implementation:

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

vec2 cdiv(vec2 a, vec2 b) {
    float denom = dot(b, b);
    return vec2(a.x * b.x + a.y * b.y, a.y * b.x - a.x * b.y) / denom;
}

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

vec2 diskToUhp(vec2 w) {
    // z = i(1 + w) / (1 - w)
    vec2 i = vec2(0.0, 1.0);
    vec2 one = vec2(1.0, 0.0);
    return cmul(i, cdiv(one + w, one - w));
}
```

To display your tiling in the Poincaré disk:

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup for disk
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 2.5;  // Fit disk in viewport
    uv.x *= iResolution.x / iResolution.y;

    // Convert disk coordinates to upper half-plane
    vec2 z = diskToUhp(uv);

    // Run your tiling algorithm in UHP
    // ... (fold into (2,3,∞) triangle as before)

    // Color and render
    fragColor = vec4(color, 1.0);
}
```

3.3.6.3. The Klein Model

The **Klein model** (also called the **Beltrami-Klein model** or **projective disk model**) is another disk representation. Its defining feature: **geodesics are Euclidean straight lines**! This makes some geometric properties clearer but sacrifices the conformal property—angles are distorted.

Converting from Poincaré disk to Klein disk:

$$\text{Klein}(w) = \frac{2w}{1 + |w|^2}$$

```
vec2 poincareToKlein(vec2 w) {
    float denom = 1.0 + dot(w, w);
    return 2.0 * w / denom;
}
```

The inverse:

$$w = \frac{\text{Klein}(w)}{1 + \sqrt{1 - |\text{Klein}(w)|^2}}$$

```
vec2 kleinToPoincare(vec2 k) {
    float k2 = dot(k, k);
    float denom = 1.0 + sqrt(1.0 - k2);
    return k / denom;
}
```

Advantages of Klein: - Geodesics are straight lines (simplest to compute) - Great for understanding incidence relationships

Disadvantages: - Not conformal—angles are distorted - Harder to see hyperbolic distances

---

3.3.7. Other Triangle Groups

The $(2, 3, \infty)$ triangle is just the beginning. Many other hyperbolic triangles can tile the plane! The most famous is the $(2, 3, 7)$ triangle.

3.3.7.1. The $(2, 3, 7)$ Triangle

This triangle has angles $\pi/2$, $\pi/3$, and $\pi/7$. Its area is:

$$\text{Area} = \pi - \left(\frac{\pi}{2} + \frac{\pi}{3} + \frac{\pi}{7}\right) = \frac{\pi}{42}$$

This is one of the smallest compact hyperbolic triangles! It produces incredibly dense tilings—the $(2, 3, 7)$ tiling has a 7-fold symmetry that creates intricate patterns reminiscent of M.C. Escher's work.

In fact, Escher's famous *Circle Limit* prints (especially Circle Limit III with the fish) are based on tilings by $(2, 3, 7)$ and related triangle groups. The hyperbolic geometry creates the exponential compression toward the boundary that gives these prints their distinctive character.

Classification of hyperbolic triangles: Not every combination (p, q, r) gives a hyperbolic triangle. The Gauss-Bonnet formula tells us we need:

$$\frac{1}{p} + \frac{1}{q} + \frac{1}{r} < 1$$

for a hyperbolic triangle (negative curvature). If the sum equals exactly 1, we get a Euclidean triangle. If the sum is greater than 1, we get a spherical triangle (positive curvature).

Examples: - $(2, 3, 7)$: $\frac{1}{2} + \frac{1}{3} + \frac{1}{7} = \frac{41}{42} < 1$ ☒ Hyperbolic - $(2, 3, \infty)$: $\frac{1}{2} + \frac{1}{3} + 0 = \frac{5}{6} < 1$ ☒ Hyperbolic - $(3, 3, 3)$: $\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1$ ☒ Euclidean (equilateral triangle) - $(2, 3, 5)$: $\frac{1}{2} + \frac{1}{3} + \frac{1}{5} = \frac{31}{30} > 1$ ☒ Spherical (icosahedron face)

Implementing other triangle groups requires computing where the third geodesic should be, given the angle constraints. This is non-trivial and involves hyperbolic trigonometry—we leave this as an advanced homework exercise!

3.4. Summary

Today we learned:

1. **The folding algorithm**: Iteratively reflect across boundaries until reaching the fundamental domain—works in any geometry
2. **Half-space structure**: Boundary + side gives a unified way to encode regions in Euclidean geometry
3. **Why it works**: **Reflection groups** and fundamental domains—the group-theoretic perspective that unifies all our iterative algorithms across Days 2 and 3
4. **Hyperbolic geometry**: Negative curvature space with two types of geodesics (vertical lines and semicircles in the UHP model)
5. **Connection to Day 2**: Circle inversion is a hyperbolic isometry! The Apollonian gasket is actually a hyperbolic object
6. **Two reflection functions**: `reflectIntoVertical` and `reflectIntoCircular` parallel the Euclidean `reflectInto`

7. **Multiple models**: Converting between upper half-plane and Poincaré disk using the Cayley transform

8. **Historical context**: The dramatic discovery of hyperbolic geometry by Gauss, Bolyai, and Lobachevsky in the 19th century

9. **Computational perspective**: GPU parallelism, convergence rates, precision issues, iteration count tuning

Key insight: Clean mathematical abstraction lets us write geometry-independent algorithms. The folding algorithm stays identical; only the reflection operations change. This is the power of recognizing patterns and building flexible abstractions!

The universal pattern: - **Day 1**: Distance fields and implicit curves—visualizing mathematics on the GPU - **Day 2**: Iterating circle inversions to create the Apollonian gasket—first group action - **Day 3 (Euclidean)**: Iterating reflections to create tilings—reflection groups make it work - **Day 3 (Hyperbolic)**: Same algorithm, different geometry—circle inversion reappears as hyperbolic isometry!

The through-line is **geometric transformations and their groups**. When we iterate a group action to reach a fundamental domain, we create tilings, fractals, and beautiful mathematical art.

3.5. Homework

3.5.1. Required #0: Understanding Hyperbolic Distance

Goal: Build geometric intuition for how hyperbolic distance works before implementing tilings.

Tasks:

1. **Euclidean distance visualization**: Implement the shader showing Euclidean distance circles (provided in lecture). Drag the center around and observe that circles stay the same size everywhere.

2. **Hyperbolic distance visualization**: Implement the hyperbolic version (provided in lecture). Drag the center and observe how the "circle" changes shape—growing huge near $y \to 0$ and shrinking as y increases.

3. **Observations**: Write 2-3 sentences describing what you observe about how hyperbolic "circles" behave compared to Euclidean ones. Why does the shape change as you drag the center?

Expected output: Two working shaders demonstrating the difference between Euclidean and hyperbolic distance.

3.5.2. Required #1: Euclidean Triangle Tiling with Edges and Vertices

Goal: Create a beautiful Euclidean triangle tiling that clearly shows the tiling structure.

Tasks:

1. Start with the basic triangle tiling from lecture
2. **Add edge visualization** using distance to half-spaces
3. **Add vertex markers** at the three triangle vertices
4. **Create an attractive color scheme**

Expected output: A clear triangle tiling where you can see individual triangles, their edges, and vertices. Experiment with colors to create an aesthetically pleasing result.

See Appendix E9 for reference implementation.

Bonus: Try different triangles! An isosceles right triangle, a 30-60-90 triangle, etc. Just compute new half-spaces for the edges.

3.5.3. Required #2: Hyperbolic Triangle Tiling with Edges and Vertices

Goal: Create a beautiful hyperbolic triangle tiling with visible structure.

Tasks:

1. Start with the basic $(2, 3, \infty)$ tiling from lecture
2. **Implement hyperbolic distance functions** (see hints in lecture)
3. **Draw geodesic edges** using hyperbolic distance to geodesics
4. **Draw vertices** using hyperbolic distance to vertex points
5. **Create an attractive color scheme**

Expected output: A clear hyperbolic tiling in the upper half-plane showing triangle edges and vertices.

See Appendix H5 for complete reference implementation.

Bonus: Display the same tiling in the Poincaré disk model and compare how edges and vertices appear in the two models.

3.5.4. Required #3: Model Conversions

Goal: See the same tiling in different representations of hyperbolic space.

Tasks:

1. **Poincaré Disk** (already provided in lecture): Verify it works with your tiling code
2. **Klein Model**: Implement the transformation from Poincaré disk to Klein disk (see lecture for formula)

   In the Klein model, geodesics become Euclidean straight lines! Display your tiling and observe this property.
3. **Comparison**: Show your $(2, 3, \infty)$ tiling in both the Poincaré disk and Klein model side-by-side. Write a few sentences about what you observe.

Deliverable: Screenshots of your tiling in at least two models, with brief observations.

See Appendix H7 for Klein model reference.

3.5.5. Required #4: Different Triangle Groups (Challenge!)

Goal: Explore other hyperbolic tilings by implementing different triangle groups.

The Challenge: This is the hardest homework problem! Computing where the third geodesic should be, given angle constraints, requires hyperbolic trigonometry.

Suggested triangles to try: - **(2, 3, 7)**: Creates Escher-like tilings with 7-fold symmetry - **(2, 4, 6)**: Different symmetry pattern - **(3, 3, 3)**: Equilateral hyperbolic triangle

Approach: Use the hyperbolic law of cosines (see lecture for formula) to compute geodesic positions.

Deliverable: Working tiling for at least one triangle group other than $(2, 3, \infty)$. Include screenshots and document your approach.

3.5.6. Optional Exercises

Optional #1: Unified Abstractions - Create structs that work for both Euclidean and hyperbolic geometry

Optional #2: Decorated Tiles - Add Escher-style patterns within fundamental domains

Optional #3: Pentagon Tilings - Right-angled pentagons can tile hyperbolic space!

See lecture for details on these optional exercises.

3.6. Looking Ahead

We've now seen three days of geometric iteration: - **Day 1**: Distance fields and implicit functions - **Day 2**: Circle inversion and the Apollonian gasket - **Day 3**: Reflection groups in Euclidean and hyperbolic geometry

The common thread: **group actions on geometric spaces**. Iteratively applying transformations to reach a desired region, whether it's the fundamental domain of a tiling or the gap structure of a fractal.

The techniques you've learned—GPU parallelism, iterative algorithms, group theory, geometric transformations—apply across a huge landscape of mathematical visualization!

4. Day 4: Introduction to 3D Rendering

4.1. Overview

Today we enter the third dimension! We'll learn how to cast rays from a camera and test for intersections with 3D objects. We'll start with analytical methods (solving equations directly) for spheres and tori, then transition to raymarching with signed distance functions—a more flexible approach that enables complex procedural scenes.

By the end of today, you'll understand: - How to set up a camera and generate rays for each pixel - Analytical ray-object intersection for simple surfaces - Why analytical methods become challenging for complex geometry - Signed distance functions as an alternative representation - The raymarching algorithm (sphere tracing) - How to compose multiple objects and assign materials

---

4.2. Part 1: Analytical Ray Tracing

4.2.1. Camera and Ray Setup

4.2.1.1. The Rendering Pipeline

For each pixel, we need to: 1. Generate a ray from the camera through that pixel 2. Find where (if anywhere) the ray intersects scene geometry 3. Compute color based on surface properties and lighting

4.2.1.2. Coordinate System

We'll use the standard graphics convention: - **Y-axis** points up - **Z-axis** points toward the camera (out of the screen) - **X-axis** points right - Right-handed coordinate system

4.2.1.3. Pinhole Camera Model

Our camera sits at the origin looking down the negative Z-axis. For a pixel at normalized coordinates $(u, v) \in [-1, 1]^2$, we generate a ray:

Ray origin: $\mathbf{o} = (0, 0, 0)$ (camera position)
Ray direction: $\mathbf{d} = \text{normalize}(u, v, -f)$

where f is the focal length, related to field of view by: $f = 1/\tan(\text{FOV}/2)$.

4.2.1.4. Parametric Ray Equation

A ray can be written as:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

where $t \geq 0$ is the parameter. Points along the ray correspond to different values of t.

4.2.1.5. Implementation

Let's start by visualizing our rays without any intersections:

Shader 1: Ray Visualization

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Normalize pixel coordinates to [-1, 1]
    vec2 uv = (fragCoord / iResolution.xy) * 2.0 - 1.0;

    // Correct for aspect ratio
    uv.x *= iResolution.x / iResolution.y;

    // Field of view
    float fov = 45.0;
    float focalLength = 1.0 / tan(radians(fov) / 2.0);

    // Ray direction (camera at origin, looking down -Z)
    vec3 rayDir = normalize(vec3(uv.x, uv.y, -focalLength));

    // Color based on ray direction (visualize the rays)
    vec3 color = rayDir * 0.5 + 0.5;

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

You should see a colorful gradient showing the direction of each ray. This confirms our camera setup is working!

4.2.2. Ray-Sphere Intersection

4.2.2.1. The Sphere Equation

A sphere of radius r centered at position $\mathbf{c}$ is defined by:

$$|\mathbf{p} - \mathbf{c}|^2 = r^2$$

All points $\mathbf{p}$ satisfying this equation lie on the sphere's surface.

4.2.2.2. Finding the Intersection

We want to find where our ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ intersects the sphere. Substituting the ray equation into the sphere equation:

$$|\mathbf{o} + t\mathbf{d} - \mathbf{c}|^2 = r^2$$

Let $\mathbf{oc} = \mathbf{o} - \mathbf{c}$ (vector from sphere center to ray origin). Expanding:

$$|\mathbf{oc} + t\mathbf{d}|^2 = r^2$$

$$|\mathbf{oc}|^2 + 2t(\mathbf{oc} \cdot \mathbf{d}) + t^2|\mathbf{d}|^2 = r^2$$

This is a quadratic equation in t:

$$at^2 + bt + c = 0$$

where: - $a = |\mathbf{d}|^2$ (equals 1 if direction is normalized) - $b = 2(\mathbf{oc} \cdot \mathbf{d})$ - $c = |\mathbf{oc}|^2 - r^2$

The **discriminant** $\Delta = b^2 - 4ac$ tells us: - $\Delta < 0$: no intersection (ray misses sphere) - $\Delta = 0$: one intersection (ray grazes sphere) - $\Delta > 0$: two intersections (ray enters and exits)

We want the smaller positive t (the entry point).

4.2.2.3. Implementation

Shader 2: Basic Sphere

```
float intersectSphere(vec3 rayOrigin, vec3 rayDir, vec3 sphereCenter, float radius) {
    vec3 oc = rayOrigin - sphereCenter;

    float a = dot(rayDir, rayDir);  // Should be 1.0 if rayDir is normalized
    float b = 2.0 * dot(oc, rayDir);
    float c = dot(oc, oc) - radius * radius;

    float discriminant = b * b - 4.0 * a * c;

    if (discriminant < 0.0) {
        return -1.0;  // No intersection
    }

    // Return the closer intersection
    float t1 = (-b - sqrt(discriminant)) / (2.0 * a);
    float t2 = (-b + sqrt(discriminant)) / (2.0 * a);

    // Return closest positive t
    if (t1 > 0.0) return t1;
    if (t2 > 0.0) return t2;
    return -1.0;  // Both behind camera
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Setup ray
```

```
    vec2 uv = (fragCoord / iResolution.xy) * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;

    float fov = 45.0;
    float focalLength = 1.0 / tan(radians(fov) / 2.0);

    vec3 rayOrigin = vec3(0.0, 0.0, 0.0);
    vec3 rayDir = normalize(vec3(uv.x, uv.y, -focalLength));

    // Sphere parameters
    vec3 sphereCenter = vec3(0.0, 0.0, -3.0);
    float sphereRadius = 1.0;

    // Test intersection
    float t = intersectSphere(rayOrigin, rayDir, sphereCenter, sphereRadius);

    vec3 color;
    if (t > 0.0) {
        // Hit the sphere
        color = vec3(1.0, 0.0, 0.0);  // Red
    } else {
        // Background
        color = vec3(0.1, 0.1, 0.2);  // Dark blue
    }

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

You should see a flat red disk! It looks 2D because we don't have lighting yet—we can't see the sphere's curvature.

4.2.3. Adding Lighting

To see the 3D structure, we need to compute lighting based on the **surface normal**.

4.2.3.1. Surface Normal

For a sphere centered at $\mathbf{c}$, the outward normal at surface point $\mathbf{p}$ is:

$$\mathbf{n} = \frac{\mathbf{p} - \mathbf{c}}{r}$$

This is just the vector from center to surface, normalized.

4.2.3.2. Diffuse Lighting

The simplest lighting model: **Lambertian diffuse shading**. Surface brightness depends on the angle between the normal $\mathbf{n}$ and light direction $\mathbf{l}$:

$$\text{brightness} = \max(0, \mathbf{n} \cdot \mathbf{l})$$

The $\max(0, \cdots)$ ensures surfaces facing away from the light remain dark.

Shader 3: Sphere with Lighting

```
float intersectSphere(vec3 rayOrigin, vec3 rayDir, vec3 sphereCenter, float radius) {
    vec3 oc = rayOrigin - sphereCenter;
    float a = dot(rayDir, rayDir);
    float b = 2.0 * dot(oc, rayDir);
    float c = dot(oc, oc) - radius * radius;
    float discriminant = b * b - 4.0 * a * c;

    if (discriminant < 0.0) return -1.0;

    float t1 = (-b - sqrt(discriminant)) / (2.0 * a);
    float t2 = (-b + sqrt(discriminant)) / (2.0 * a);

    if (t1 > 0.0) return t1;
    if (t2 > 0.0) return t2;
    return -1.0;
}

vec3 sphereNormal(vec3 hitPoint, vec3 sphereCenter, float radius) {
    return (hitPoint - sphereCenter) / radius;
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Setup ray
    vec2 uv = (fragCoord / iResolution.xy) * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;

    float fov = 45.0;
    float focalLength = 1.0 / tan(radians(fov) / 2.0);

    vec3 rayOrigin = vec3(0.0, 0.0, 0.0);
    vec3 rayDir = normalize(vec3(uv.x, uv.y, -focalLength));

    // Sphere
    vec3 sphereCenter = vec3(0.0, 0.0, -3.0);
    float sphereRadius = 1.0;

    float t = intersectSphere(rayOrigin, rayDir, sphereCenter, sphereRadius);

    vec3 color;
```

```
    if (t > 0.0) {
        // Hit point
        vec3 hitPoint = rayOrigin + t * rayDir;

        // Surface normal
        vec3 normal = sphereNormal(hitPoint, sphereCenter, sphereRadius);

        // Light direction (from above and to the right)
        vec3 lightDir = normalize(vec3(1.0, 1.0, 1.0));

        // Diffuse lighting
        float diffuse = max(0.0, dot(normal, lightDir));

        // Sphere color
        vec3 sphereColor = vec3(1.0, 0.0, 0.0);  // Red
        color = sphereColor * diffuse;

        // Add ambient light so dark side isn't completely black
        color += sphereColor * 0.1;
    } else {
        color = vec3(0.1, 0.1, 0.2);
    }

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

Now the sphere looks 3D! The lighting reveals its curvature. Beautiful!

4.2.4. Ray-Torus Intersection: Where Analytical Gets Complex

4.2.4.1. The Torus Equation

A torus with major radius R (distance from center to tube center) and minor radius r (tube thickness) has the implicit equation:

$$\left(\sqrt{x^2 + z^2} - R\right)^2 + y^2 = r^2$$

Or in vector form:

$$(|\mathbf{p}_{xz}| - R)^2 + p_y^2 = r^2$$

where $\mathbf{p}_{xz} = (p_x, p_z)$ is the projection onto the XZ-plane.

4.2.4.2. The Challenge

Substituting our ray equation into this gives a **quartic polynomial** (degree 4):

$$at^4 + bt^3 + ct^2 + dt + e = 0$$

Unlike quadratics (which have a simple formula), quartic equations require sophisticated algebraic methods. Here's what solving it actually looks like:

Shader 4: Analytical Torus

```
// From Inigo Quilez - https://www.shadertoy.com/view/XdSGWy
// Analytical quartic solver for torus intersection
float intersectTorus(vec3 ro, vec3 rd, vec2 tor)
{
    float po = 1.0;
    float Ra2 = tor.x * tor.x;
    float ra2 = tor.y * tor.y;

    float m = dot(ro, ro);
    float n = dot(ro, rd);

    // Bounding sphere check
    float h = n*n - m + (tor.x + tor.y) * (tor.x + tor.y);
    if(h < 0.0) return -1.0;

    // Find quartic coefficients
    float k = (m - ra2 - Ra2) / 2.0;
    float k3 = n;
    float k2 = n*n + Ra2*rd.z*rd.z + k;
    float k1 = k*n + Ra2*ro.z*rd.z;
    float k0 = k*k + Ra2*ro.z*ro.z - Ra2*ra2;

    // Prevent numerical issues
    if(abs(k3*(k3*k3 - k2) + k1) < 0.01)
    {
        po = -1.0;
        float tmp = k1; k1 = k3; k3 = tmp;
        k0 = 1.0/k0;
        k1 = k1*k0;
        k2 = k2*k0;
        k3 = k3*k0;
    }

    float c2 = 2.0*k2 - 3.0*k3*k3;
    float c1 = k3*(k3*k3 - k2) + k1;
    float c0 = k3*(k3*(-3.0*k3*k3 + 4.0*k2) - 8.0*k1) + 4.0*k0;

    c2 /= 3.0;
    c1 *= 2.0;
    c0 /= 3.0;
```

```
float Q = c2*c2 + c0;
float R = 3.0*c0*c2 - c2*c2*c2 - c1*c1;

float h2 = R*R - Q*Q*Q;
float z = 0.0;

if(h2 < 0.0)
{
    // 4 intersections
    float sQ = sqrt(Q);
    z = 2.0*sQ*cos(acos(R/(sQ*Q)) / 3.0);
}
else
{
    // 2 intersections
    float sQ = pow(sqrt(h2) + abs(R), 1.0/3.0);
    z = sign(R)*abs(sQ + Q/sQ);
}

z = c2 - z;

float d1 = z - 3.0*c2;
float d2 = z*z - 3.0*c0;

if(abs(d1) < 1.0e-4)
{
    if(d2 < 0.0) return -1.0;
    d2 = sqrt(d2);
}
else
{
    if(d1 < 0.0) return -1.0;
    d1 = sqrt(d1/2.0);
    d2 = c1/d1;
}

float result = 1e20;

h2 = d1*d1 - z + d2;
if(h2 > 0.0)
{
    h2 = sqrt(h2);
    float t1 = -d1 - h2 - k3;
    float t2 = -d1 + h2 - k3;
    t1 = (po < 0.0) ? 2.0/t1 : t1;
    t2 = (po < 0.0) ? 2.0/t2 : t2;
    if(t1 > 0.0) result = t1;
    if(t2 > 0.0) result = min(result, t2);
}
```

```
    h2 = d1*d1 - z - d2;
    if(h2 > 0.0)
    {
        h2 = sqrt(h2);
        float t1 = d1 - h2 - k3;
        float t2 = d1 + h2 - k3;
        t1 = (po < 0.0) ? 2.0/t1 : t1;
        t2 = (po < 0.0) ? 2.0/t2 : t2;
        if(t1 > 0.0) result = min(result, t1);
        if(t2 > 0.0) result = min(result, t2);
    }

    return result;
}

vec3 torusNormal(vec3 pos, vec2 tor)
{
    return normalize(pos * (dot(pos,pos) - tor.y*tor.y - tor.x*tor.x*vec3(1.0,1.0,-1.0)));
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Setup ray
    vec2 uv = (fragCoord / iResolution.xy) * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;

    float fov = 45.0;
    float focalLength = 1.0 / tan(radians(fov) / 2.0);

    vec3 rayOrigin = vec3(0.0, 0.0, 0.0);
    vec3 rayDir = normalize(vec3(uv.x, uv.y, -focalLength));

    // Torus parameters
    vec2 torus = vec2(1.0, 0.4);  // (major radius, minor radius)
    vec3 torusCenter = vec3(0.0, 0.0, -3.5);

    // Adjust ray for torus position
    vec3 ro = rayOrigin - torusCenter;

    float t = intersectTorus(ro, rayDir, torus);

    vec3 color;
    if (t > 0.0) {
        vec3 hitPoint = ro + t * rayDir;
        vec3 normal = torusNormal(hitPoint, torus);

        vec3 lightDir = normalize(vec3(1.0, 1.0, 1.0));
        float diffuse = max(0.0, dot(normal, lightDir));

        vec3 torusColor = vec3(0.0, 0.7, 1.0);  // Cyan
```

```
        color = torusColor * diffuse + torusColor * 0.1;
    } else {
        color = vec3(0.1, 0.1, 0.2);
    }

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

Look at that intersection code! Over 80 lines of complex algebra just to render one torus. And this is still a relatively simple surface—imagine arbitrary algebraic varieties, or trying to combine multiple objects with boolean operations.

Analytical methods work beautifully for simple geometry, but we need a more flexible approach for complex scenes.

4.3. Part 2: Signed Distance Functions and Raymarching

4.3.1. Introduction to SDFs

A **signed distance function** (SDF) gives the distance from any point in space to the nearest surface:

$$d(\mathbf{p}) = \begin{cases} > 0 & \text{outside surface} \\ = 0 & \text{on surface} \\ < 0 & \text{inside surface} \end{cases}$$

Crucially, $|d(\mathbf{p})|$ is the actual Euclidean distance to the surface.

4.3.1.1. Why SDFs?

SDFs have a powerful property: if we're at point $\mathbf{p}$ and the surface is distance d away, we can safely move d units in *any* direction without hitting anything. This enables **sphere tracing**—we march along the ray taking steps proportional to the SDF value.

4.3.1.2. SDF Examples

Let's see SDFs for shapes we've already rendered analytically:

Sphere:

```
float sdSphere(vec3 p, vec3 center, float radius) {
    return length(p - center) - radius;
}
```

Compare this to our 30+ line analytical intersection! Much simpler.

Torus:

```
float sdTorus(vec3 p, vec3 center, float majorRadius, float minorRadius) {
    vec3 q = p - center;
    vec2 pxz = vec2(q.x, q.z);
    float d = length(pxz) - majorRadius;
    return length(vec2(d, q.y)) - minorRadius;
}
```

Again, dramatically simpler than the quartic solver!

Other Primitives

Many more SDFs exist: boxes, cylinders, capsules, cones, pyramids, etc. See Inigo Quilez's comprehensive library for a complete reference. Each SDF is typically just a few lines of code.

4.3.2. The Raymarching Algorithm

Sphere tracing works like this:

1. Start at the ray origin
2. Evaluate the SDF at current position
3. March forward along the ray by that distance (safe step!)
4. Repeat until:
    - Very close to surface (SDF ≈ 0) $\longrightarrow$ hit!
    - Too far away $\longrightarrow$ miss
    - Too many steps $\longrightarrow$ give up

Here's the algorithm:

```
float sceneSDF(vec3 p) {
    // Define scene geometry (we'll implement this)
    return 0.0;
}

bool raymarch(vec3 rayOrigin, vec3 rayDir, out float hitDist, out vec3 hitPos) {
    float t = 0.0;

    for(int i = 0; i < 100; i++) {
        vec3 pos = rayOrigin + t * rayDir;
        float d = sceneSDF(pos);

        // Close enough to surface?
        if(abs(d) < 0.001) {
            hitDist = t;
            hitPos = pos;
            return true;
```

```
        }

        // March forward
        t += d;

        // Too far?
        if(t > 100.0) {
            return false;
        }
    }

    return false;
}
```

4.3.2.1. Normal Estimation via Gradient

For an SDF $d(\mathbf{p})$, the gradient ∇d points perpendicular to the surface (it's the normal direction). We estimate it using finite differences:

$$\frac{\partial d}{\partial x} \approx \frac{d(x+\epsilon, y, z) - d(x-\epsilon, y, z)}{2\epsilon}$$

```
vec3 estimateNormal(vec3 p) {
    float eps = 0.001;
    float dx = sceneSDF(p + vec3(eps, 0, 0)) - sceneSDF(p - vec3(eps, 0, 0));
    float dy = sceneSDF(p + vec3(0, eps, 0)) - sceneSDF(p - vec3(0, eps, 0));
    float dz = sceneSDF(p + vec3(0, 0, eps)) - sceneSDF(p - vec3(0, 0, eps));
    return normalize(vec3(dx, dy, dz));
}
```

4.3.2.2. First Raymarch Shader

Shader 5: Single Sphere with Raymarching

```
float sdSphere(vec3 p, vec3 center, float radius) {
    return length(p - center) - radius;
}

float sceneSDF(vec3 p) {
    return sdSphere(p, vec3(0.0, 0.0, -3.0), 1.0);
}

vec3 estimateNormal(vec3 p) {
    float eps = 0.001;
    float dx = sceneSDF(p + vec3(eps, 0, 0)) - sceneSDF(p - vec3(eps, 0, 0));
    float dy = sceneSDF(p + vec3(0, eps, 0)) - sceneSDF(p - vec3(0, eps, 0));
    float dz = sceneSDF(p + vec3(0, 0, eps)) - sceneSDF(p - vec3(0, 0, eps));
    return normalize(vec3(dx, dy, dz));
```

```
}

bool raymarch(vec3 rayOrigin, vec3 rayDir, out vec3 hitPos) {
    float t = 0.0;

    for(int i = 0; i < 100; i++) {
        vec3 pos = rayOrigin + t * rayDir;
        float d = sceneSDF(pos);

        if(abs(d) < 0.001) {
            hitPos = pos;
            return true;
        }

        t += d;

        if(t > 100.0) return false;
    }

    return false;
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Setup ray
    vec2 uv = (fragCoord / iResolution.xy) * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;

    float fov = 45.0;
    float focalLength = 1.0 / tan(radians(fov) / 2.0);

    vec3 rayOrigin = vec3(0.0, 0.0, 0.0);
    vec3 rayDir = normalize(vec3(uv.x, uv.y, -focalLength));

    // Raymarch
    vec3 hitPos;
    bool hit = raymarch(rayOrigin, rayDir, hitPos);

    vec3 color;
    if(hit) {
        vec3 normal = estimateNormal(hitPos);
        vec3 lightDir = normalize(vec3(1.0, 1.0, 1.0));
        float diffuse = max(0.0, dot(normal, lightDir));

        vec3 sphereColor = vec3(1.0, 0.0, 0.0);
        color = sphereColor * diffuse + sphereColor * 0.1;
    } else {
        color = vec3(0.1, 0.1, 0.2);
    }
```

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

Same result as the analytical sphere, but now we have a flexible framework!

4.3.3. The Power of SDFs: Instant Shape Swapping

Here's where SDFs shine: **changing shapes is trivial**. Just swap out one SDF for another!

Shader 6: Shapeshifting

```
float sdSphere(vec3 p, vec3 center, float radius) {
    return length(p - center) - radius;
}

float sdTorus(vec3 p, vec3 center, float majorRadius, float minorRadius) {
    vec3 q = p - center;
    vec2 pxz = vec2(q.x, q.z);
    float d = length(pxz) - majorRadius;
    return length(vec2(d, q.y)) - minorRadius;
}

float sdBox(vec3 p, vec3 center, vec3 halfExtents) {
    vec3 q = abs(p - center) - halfExtents;
    return length(max(q, 0.0)) + min(max(q.x, max(q.y, q.z)), 0.0);
}

float sceneSDF(vec3 p) {
    // Uncomment ONE of these to see different shapes!
    // Everything else stays the same - same raymarch, same lighting, same normal calculation

    return sdSphere(p, vec3(0.0, 0.0, -3.0), 1.0);
    //return sdTorus(p, vec3(0.0, 0.0, -3.0), 1.0, 0.4);
    //return sdBox(p, vec3(0.0, 0.0, -3.0), vec3(0.8, 0.8, 0.8));

    // Try any SDF from https://iquilezles.org/articles/distfunctions/
}

vec3 estimateNormal(vec3 p) {
    float eps = 0.001;
    float dx = sceneSDF(p + vec3(eps, 0, 0)) - sceneSDF(p - vec3(eps, 0, 0));
    float dy = sceneSDF(p + vec3(0, eps, 0)) - sceneSDF(p - vec3(0, eps, 0));
    float dz = sceneSDF(p + vec3(0, 0, eps)) - sceneSDF(p - vec3(0, 0, eps));
    return normalize(vec3(dx, dy, dz));
}

bool raymarch(vec3 rayOrigin, vec3 rayDir, out vec3 hitPos) {
```

```
    float t = 0.0;

    for(int i = 0; i < 100; i++) {
        vec3 pos = rayOrigin + t * rayDir;
        float d = sceneSDF(pos);

        if(abs(d) < 0.001) {
            hitPos = pos;
            return true;
        }

        t += d;
        if(t > 100.0) return false;
    }

    return false;
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 uv = (fragCoord / iResolution.xy) * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;

    float fov = 45.0;
    float focalLength = 1.0 / tan(radians(fov) / 2.0);

    vec3 rayOrigin = vec3(0.0, 0.0, 0.0);
    vec3 rayDir = normalize(vec3(uv.x, uv.y, -focalLength));

    vec3 hitPos;
    bool hit = raymarch(rayOrigin, rayDir, hitPos);

    vec3 color;
    if(hit) {
        vec3 normal = estimateNormal(hitPos);
        vec3 lightDir = normalize(vec3(1.0, 1.0, 1.0));
        float diffuse = max(0.0, dot(normal, lightDir));

        vec3 objectColor = vec3(0.0, 0.7, 1.0);  // Cyan
        color = objectColor * diffuse + objectColor * 0.1;
    } else {
        color = vec3(0.1, 0.1, 0.2);
    }

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

Comment/uncomment different SDFs in `sceneSDF()` to instantly see different shapes! Try adding more from Quilez's library. The raymarching algorithm doesn't care what shape you use—it just follows the distance field.

4.3.4. Composing Multiple Objects

To combine multiple objects, we simply take the **minimum distance** to any object. The closest surface wins!

Shader 7: Multiple Objects with Materials

```
float sdSphere(vec3 p, vec3 center, float radius) {
    return length(p - center) - radius;
}

float sdTorus(vec3 p, vec3 center, float majorRadius, float minorRadius) {
    vec3 q = p - center;
    vec2 pxz = vec2(q.x, q.z);
    float d = length(pxz) - majorRadius;
    return length(vec2(d, q.y)) - minorRadius;
}

float sdPlane(vec3 p, float height) {
    return p.y - height;
}

// Global variable to track which object we hit
float gMaterialID;

float sceneSDF(vec3 p) {
    float d = 1e10;  // Start with very large distance

    // Sphere
    float sphere = sdSphere(p, vec3(-1.2, 0.0, -3.5), 0.8);
    if(sphere < d) {
        d = sphere;
        gMaterialID = 1.0;
    }

    // Torus
    float torus = sdTorus(p, vec3(1.2, 0.0, -3.5), 1.0, 0.3);
    if(torus < d) {
        d = torus;
        gMaterialID = 2.0;
    }

    // Ground plane
    float plane = sdPlane(p, -1.0);
    if(plane < d) {
        d = plane;
        gMaterialID = 3.0;
    }
```

```
    return d;
}

vec3 getMaterialColor(float matID) {
    if(matID < 1.5) return vec3(1.0, 0.0, 0.0);      // Sphere: red
    if(matID < 2.5) return vec3(0.0, 0.7, 1.0);      // Torus: cyan
    return vec3(0.5, 0.5, 0.5);                       // Plane: gray
}

vec3 estimateNormal(vec3 p) {
    float eps = 0.001;
    float dx = sceneSDF(p + vec3(eps, 0, 0)) - sceneSDF(p - vec3(eps, 0, 0));
    float dy = sceneSDF(p + vec3(0, eps, 0)) - sceneSDF(p - vec3(0, eps, 0));
    float dz = sceneSDF(p + vec3(0, 0, eps)) - sceneSDF(p - vec3(0, 0, eps));
    return normalize(vec3(dx, dy, dz));
}

bool raymarch(vec3 rayOrigin, vec3 rayDir, out vec3 hitPos, out float matID) {
    float t = 0.0;

    for(int i = 0; i < 100; i++) {
        vec3 pos = rayOrigin + t * rayDir;
        float d = sceneSDF(pos);

        if(abs(d) < 0.001) {
            hitPos = pos;
            matID = gMaterialID;
            return true;
        }

        t += d;
        if(t > 100.0) return false;
    }

    return false;
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 uv = (fragCoord / iResolution.xy) * 2.0 - 1.0;
    uv.x *= iResolution.x / iResolution.y;

    float fov = 45.0;
    float focalLength = 1.0 / tan(radians(fov) / 2.0);

    vec3 rayOrigin = vec3(0.0, 0.0, 0.0);
    vec3 rayDir = normalize(vec3(uv.x, uv.y, -focalLength));

    vec3 hitPos;
    float matID;
```

```
    bool hit = raymarch(rayOrigin, rayDir, hitPos, matID);

    vec3 color;
    if(hit) {
        vec3 normal = estimateNormal(hitPos);
        vec3 lightDir = normalize(vec3(1.0, 1.0, 1.0));
        float diffuse = max(0.0, dot(normal, lightDir));

        vec3 objectColor = getMaterialColor(matID);
        color = objectColor * diffuse + objectColor * 0.1;
    } else {
        color = vec3(0.1, 0.1, 0.2);
    }

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

Three objects with different colors! Adding more objects is trivial—just add another SDF check in `sceneSDF()`. Compare this to analytical methods where combining objects requires sophisticated CSG (constructive solid geometry) techniques.

4.4. Summary

Today we learned two approaches to 3D rendering:

Analytical Ray Tracing: - Solve equations directly for ray-surface intersection - Exact solutions, very efficient for simple geometry - Sphere: straightforward quadratic equation - Torus: complex quartic equation requiring sophisticated algebra - Becomes increasingly difficult for complex surfaces - Standard in production ray tracers for well-defined geometry

SDF-Based Raymarching: - Represent surfaces as distance fields - March along rays using sphere tracing - Simple, uniform code for any geometry - Easy composition: just take minimum distance - Flexible—works for procedural, implicit, or arbitrary surfaces - Slightly slower than analytical, but much more versatile

Key Concepts: - Camera setup and ray generation - Parametric ray equation: $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ - Surface normals for lighting - Diffuse (Lambertian) shading - Signed distance functions (SDFs) - Sphere tracing algorithm - Normal estimation via gradient (finite differences) - Material tracking for multiple objects

Tomorrow we'll explore advanced raymarching: domain operations for infinite repetition, boolean operations for smooth blending, and 3D fractals!

4.5. Homework

4.5.1. Required: Algebraic Variety Rendering

Implement analytical ray tracing for an interesting polynomial implicit surface.

Goal: Experience the challenges of analytical methods firsthand, then appreciate SDFs even more!

Suggested surfaces: - **Degree 3**: Saddle surfaces, cubic varieties - **Degree 4**: Klein bottle projections, quartic surfaces with interesting topology - **Cassini ovals** in 3D: $(x^2+y^2+z^2)^2-2a^2(x^2-y^2) = b^4 - a^4$

Implementation steps:

1. **Define your implicit function** $F(x, y, z) = 0$

Example—a quartic surface:

```
float implicitFunction(vec3 p) {
    float r2 = dot(p, p);
    return r2 * r2 - (p.x*p.x + p.y*p.y - 2.0*p.z*p.z);
}
```

2. **Implement root finding** (bisection method)

```
float intersectImplicit(vec3 rayOrigin, vec3 rayDir) {
    float tMin = 0.0;
    float tMax = 10.0;

    // Check for sign change
    float fMin = implicitFunction(rayOrigin + tMin * rayDir);
    float fMax = implicitFunction(rayOrigin + tMax * rayDir);

    if(fMin * fMax > 0.0) return -1.0;  // No root

    // Bisection
    for(int i = 0; i < 50; i++) {
        float tMid = (tMin + tMax) / 2.0;
        float fMid = implicitFunction(rayOrigin + tMid * rayDir);

        if(abs(fMid) < 0.001) return tMid;

        if(fMin * fMid < 0.0) {
            tMax = tMid;
            fMax = fMid;
        } else {
            tMin = tMid;
            fMin = fMid;
        }
    }

    return (tMin + tMax) / 2.0;
}
```

3. **Compute normal via gradient**

```
vec3 implicitNormal(vec3 p) {
    float eps = 0.001;
    float dx = implicitFunction(p + vec3(eps, 0, 0)) - implicitFunction(p - vec3(eps, 0, 0));
    float dy = implicitFunction(p + vec3(0, eps, 0)) - implicitFunction(p - vec3(0, eps, 0));
    float dz = implicitFunction(p + vec3(0, 0, eps)) - implicitFunction(p - vec3(0, 0, eps));
    return normalize(vec3(dx, dy, dz));
}
```

4. **Optimization: Bounding volume** (optional but recommended)

Use a bounding sphere to avoid checking the entire ray:

```
// First check if ray intersects bounding sphere
// Only compute implicit function if inside bounds
```

Expected output: A rendered algebraic surface with proper lighting showing its geometric features.

Reflection question: After implementing this, compare the effort to using SDFs. Which approach would you prefer for a complex scene with many objects?

4.5.2. Optional Exercises

4.5.2.1. 1. Specular Lighting (Phong Model)

Add shiny highlights using the Phong reflection model:

$$\text{specular} = (R \cdot V)^n$$

where R is reflected light direction, V is view direction, n is shininess.

```
vec3 R = reflect(-lightDir, normal);  // Reflected light
vec3 V = -rayDir;                      // View direction
float spec = pow(max(0.0, dot(R, V)), 32.0);
color += vec3(1.0) * spec * 0.5;  // White specular highlight
```

Try different shininess values (8, 16, 32, 64, 128) to see the effect!

4.5.2.2. 2. Camera Movement

Implement an orbiting camera using time:

```
float angle = iTime * 0.5;
vec3 rayOrigin = vec3(3.0 * cos(angle), 1.0, 3.0 * sin(angle));

// Look-at matrix
vec3 target = vec3(0.0, 0.0, -3.0);
vec3 forward = normalize(target - rayOrigin);
vec3 right = normalize(cross(vec3(0, 1, 0), forward));
vec3 up = cross(forward, right);

// Transform ray direction
vec3 rd = normalize(uv.x * right + uv.y * up + focalLength * forward);
```

4.5.2.3. 3. Complex SDF Scene

Create a scene with 5+ objects using different SDFs from Quilez's library: - Mix primitives: spheres, boxes, cylinders, tori, cones - Position them creatively - Use different materials - Add interesting lighting

4.5.2.4. 4. Soft Shadows (Preview of Day 5)

Cast rays from the surface toward the light to check for occlusion:

```
float softShadow(vec3 pos, vec3 lightDir) {
    float t = 0.01;  // Start slightly above surface
    float shadow = 1.0;

    for(int i = 0; i < 50; i++) {
        float d = sceneSDF(pos + lightDir * t);
        shadow = min(shadow, 8.0 * d / t);  // Penumbra factor
        t += d;
        if(t > 10.0 || d < 0.001) break;
    }

    return clamp(shadow, 0.0, 1.0);
}
```

Apply this to your diffuse lighting for more realistic shadows!

4.6. Looking Ahead to Day 5

Tomorrow we'll explore advanced raymarching techniques that would be nearly impossible with analytical methods:

- **Domain operations**: Infinite repetition, symmetry, twisting
- **Boolean operations**: Union, intersection, smooth blending

- **3D fractals**: Menger sponge, Mandelbulb via iterated transformations
- **Advanced lighting**: Ambient occlusion, global illumination

Make sure you're comfortable with: - The raymarching algorithm (it's the foundation) - How SDFs compose (taking minimum/maximum) - Normal estimation via gradients - The material tracking pattern we developed

See you tomorrow!

5. Day 5a

6. Day 5bs

A. Appendix: Complete Shader Code for Day 1

This appendix provides complete, standalone Shadertoy code for each shader program presented in Day 1. Each listing includes all necessary setup and can be copied directly into Shadertoy (https://www.shadertoy.com/new) and run immediately.

A.1. A1. Basic Red Screen

The simplest possible shader - every pixel is red.

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

A.2. A2. Animated Color (Pulsing Red)

Using `iTime` to animate the red channel.

```
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);
}
```

A.3. A3. Coordinate Visualization

Visualizing the coordinate system by mapping position to color.

```
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 p = uv * 4.0;

    // 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);
}
```

A.4. A4. Half-Plane Coloring (Ternary Operator)

Dividing the plane into two regions based on a linear function.

```
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 p = uv * 4.0;

    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);
}
```

A.5. A5. Half-Plane with Step Function

Same as above but using `step()` and `mix()`.

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

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

A.6. A6. Arbitrary Half-Plane

Dividing along an arbitrary line $ax + by + c = 0$.

```
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 p = uv * 4.0;

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

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

A.7. A7. Filled Circle

Using distance from origin to create a disk.

```
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 p = uv * 4.0;

    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);
}
```

A.8. A8. Distance-Based Coloring (Radial Gradient)

Using distance value itself to create a gradient.

```
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 p = uv * 4.0;

    float d = length(p);
    float intensity = 1.0 - d / 2.0;  // Fades from 1 at center to 0 at distance 2
    intensity = clamp(intensity, 0.0, 1.0);  // Keep it in [0, 1]

    vec3 color = vec3(intensity);
    fragColor = vec4(color, 1.0);
}
```

A.9. A9. Circle Outline (Hard Edge)

Drawing just the boundary of a circle with hard threshold.

```
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 p = uv * 4.0;

    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);
}
```

A.10. A10. Circle Outline (Smooth with Smoothstep)

Anti-aliased circle outline using `smoothstep()`.

```
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 p = uv * 4.0;

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

    float circle_mask = 1.0 - smoothstep(r - thickness, r + thickness, d);
    vec3 color = vec3(circle_mask);
    fragColor = vec4(color, 1.0);
}
```

A.11. A11. Grid of Circles

Using `mod()` to create repeating circles.

```
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 p = uv * 4.0;

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

    // 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);
}
```

A.12. A12. Grid with Alternating Background

Creating a checkerboard pattern behind the circles.

```
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 p = uv * 4.0;

    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);
}
```

A.13. A13. Complete Grid Pattern

Full example combining grid cells, checkerboard, and circles.

```
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 p = uv * 4.0;

    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);
}
```

A.14. A14. Implicit Curve: Parabola

Drawing a parabola using the implicit equation $y = x^2$.

```
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 p = uv * 4.0;

    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);
}
```

A.15. A15. Implicit Curve: Circle

Drawing a circle using the implicit equation $x^2 + y^2 = r^2$.

```
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 p = uv * 4.0;

    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;

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

A.16. A16. Implicit Curve: Hyperbola

Drawing a hyperbola $xy = 1$.

```
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 p = uv * 4.0;

    float F = p.x * p.y - 1.0;
    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);
}
```

A.17. A17. Implicit Curve: Ellipse

Drawing an ellipse $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$.

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

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

A.18. A18. Parabola Graphing Calculator (Homework Template)

Template for the required homework assignment.

```
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 p = uv * 4.0;

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

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

    // TODO: Draw x-axis (|y| < thickness)
    // TODO: Draw y-axis (|x| < thickness)
    // TODO: Draw parabola (|y - (ax² + bx + c)| < thickness)

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

A.19. A19. Parabola Graphing Calculator (Complete Solution)

Complete implementation of the required homework.

```
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 p = uv * 4.0;

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

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

    // Axes
    float axis_thickness = 0.02;
    float x_axis_mask = abs(p.y) < axis_thickness ? 1.0 : 0.0;
    float y_axis_mask = abs(p.x) < axis_thickness ? 1.0 : 0.0;
    vec3 axis_color = vec3(0.3, 0.3, 0.3);

    // Parabola: F(x,y) = y - (ax² + bx + c) = 0
    float F = p.y - (a * p.x * p.x + b * p.x + c);
    float curve_thickness = 0.08;
    float parabola_mask = abs(F) < curve_thickness ? 1.0 : 0.0;
    vec3 parabola_color = vec3(1.0, 0.8, 0.0);

    // Combine (axes behind parabola)
    color = mix(color, axis_color, max(x_axis_mask, y_axis_mask));
    color = mix(color, parabola_color, parabola_mask);

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

A.20. A20. Animated Curve Family: Circle

Animating through different circle radii.

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

```
    // Animated radius
    float r = 1.0 + 0.5 * sin(iTime);

    float F = dot(p, p) - r * r;
    float thickness = 0.08;
    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);
}
```

A.21. A21. Animated Curve Family: Rotating Ellipse

Ellipse that rotates over time.

```
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 p = uv * 4.0;

    // Rotation angle from time
    float theta = iTime * 0.5;

    // Rotate coordinates
    vec2 p_rot = vec2(
        p.x * cos(theta) + p.y * sin(theta),
        -p.x * sin(theta) + p.y * cos(theta)
    );

    // Ellipse in rotated coordinates
    float a = 2.0, b = 1.0;
    float F = (p_rot.x * p_rot.x) / (a * a) + (p_rot.y * p_rot.y) / (b * b) - 1.0;
    float thickness = 0.08;
    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);
}
```

A.22. A22. Beautiful Tiling: Geometric Pattern

Example of a custom tiling pattern using circles and symmetry.

```
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 p = uv * 4.0;

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

    // Use symmetry within each cell
    cell_p = abs(cell_p);  // 4-fold symmetry

    // Multiple circles at different positions
    float d1 = length(cell_p - vec2(0.2, 0.2));
    float d2 = length(cell_p - vec2(0.4, 0.0));
    float d3 = length(cell_p - vec2(0.0, 0.4));

    float r = 0.15;
    float mask = (d1 < r || d2 < r || d3 < r) ? 1.0 : 0.0;

    // Vary color by cell position
    float checker = mod(cell_id.x + cell_id.y, 2.0);
    vec3 color1 = vec3(0.2, 0.4, 0.6);
    vec3 color2 = vec3(0.6, 0.2, 0.4);
    vec3 bg = mix(color1, color2, checker);

    vec3 circle_color = vec3(1.0, 0.9, 0.7);
    vec3 color = mix(bg, circle_color, mask);

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

A.23. A23. Beautiful Tiling: Distance-Based Animation

Pattern that pulses based on time and distance.

```
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 p = uv * 4.0;

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

    // Distance from cell center
    float d = length(cell_p);

    // Distance from origin (in cell coordinates)
    float cell_dist = length(cell_id);

    // Animated radius that propagates outward
    float r = 0.3 + 0.1 * sin(iTime * 2.0 - cell_dist * 0.5);

    float mask = smoothstep(r + 0.05, r - 0.05, d);

    // Color based on cell distance
    vec3 color1 = vec3(0.2, 0.3, 0.5);
    vec3 color2 = vec3(0.8, 0.3, 0.4);
    float t = fract(cell_dist * 0.2);
    vec3 circle_color = mix(color1, color2, t);

    vec3 bg = vec3(0.1, 0.1, 0.15);
    vec3 color = mix(bg, circle_color, mask);

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

A.24. Notes on Using These Shaders

A.24.1. Getting Started

1. Go to https://www.shadertoy.com/new
2. Delete the default code
3. Copy and paste any of the above listings
4. Click the play button (▶) or press Alt+Enter

A.24.2. Coordinate System

All shaders (except A1 and A2) use the standard coordinate transformation:

```
vec2 uv = fragCoord / iResolution.xy;  // Normalize to [0,1]
uv = uv - 0.5;                         // Center at origin
uv.x *= iResolution.x / iResolution.y; // Aspect ratio correction
vec2 p = uv * 4.0;                     // Scale viewing window
```

This gives you coordinates centered at the origin with equal scaling in x and y.

A.24.3. Modifying Parameters

Coordinate scaling: - Change `uv * 4.0` to zoom in/out (smaller number = zoom in)

Colors: - Modify `vec3(r, g, b)` values (each in range [0, 1]) - Red: `vec3(1.0, 0.0, 0.0)` - Green: `vec3(0.0, 1.0, 0.0)` - Blue: `vec3(0.0, 0.0, 1.0)` - Yellow: `vec3(1.0, 1.0, 0.0)` - Cyan: `vec3(0.0, 1.0, 1.0)` - Magenta: `vec3(1.0, 0.0, 1.0)`

Distance and thickness: - `thickness` parameters control line width - Larger thickness = thicker lines/curves

Grid patterns: - `spacing` controls grid cell size - Smaller spacing = more cells

Animation: - Use `iTime` for time-based animation - `sin(iTime)` oscillates between -1 and 1 - `0.5 + 0.5 * sin(iTime)` oscillates between 0 and 1

A.24.4. Common Modifications to Try

Make circles pulse:

```
float r = 0.3 + 0.1 * sin(iTime);
```

Make grid spacing animate:

```
float spacing = 1.0 + 0.3 * sin(iTime * 0.5);
```

Add mouse interaction:

```
vec2 mouse = iMouse.xy / iResolution.xy;
mouse = mouse - 0.5;
mouse.x *= iResolution.x / iResolution.y;
// Use mouse position to control parameters
```

Combine techniques: - Put implicit curves on a grid using `mod()` - Add animation to any parameter with `iTime` - Use distance fields to create smooth transitions

A.24.5. Troubleshooting

Shader won't compile: - Check for missing semicolons - Make sure all numbers are floats: `1.0` not `1` - Verify parentheses and braces are balanced

Nothing shows up: - Check your coordinate scaling - might be zoomed too far in/out - Verify colors are in [0, 1] range - Make sure alpha channel is 1.0: `vec4(color, 1.0)`

Circles look like ellipses: - Make sure you include the aspect ratio correction: `uv.x *= iResolution.x / iResolution.y;`

A.24.6. Next Steps

Once you're comfortable with these basics: - Combine multiple techniques in one shader - Create your own implicit curves - Design custom tiling patterns - Add animation and interactivity - Experiment with color palettes and smooth transitions

The goal is to understand how coordinate transformations, distance functions, and conditionals work together to create mathematical visualizations on the GPU!

B. Appendix: Complete Shader Code for Day 2

This appendix provides complete, standalone Shadertoy code for each shader program presented in Day 2. Each listing includes all necessary helper functions and can be copied directly into Shadertoy (https://www.shadertoy.com/new) and run immediately.

B.1. A1. Basic Mandelbrot Set (Grayscale)

This is the simplest Mandelbrot renderer, showing just the escape-time iteration count in grayscale.

```
// Complex number operations
float cabs2(vec2 z) {
    return dot(z, z);
}

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

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;

    // Scale to view the Mandelbrot set
    vec2 c = uv * 3.5;
    c.x -= 0.5;

    // 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;
    }

    // Grayscale coloring
    float t = float(iter) / float(max_iter);
    vec3 color = vec3(t);

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

B.2. A2. Mandelbrot Set with Smooth Coloring

This adds smooth coloring and a cosine-based color palette for much better visual quality.

```
// Complex number operations
float cabs2(vec2 z) {
    return dot(z, z);
}

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

// Cosine-based color 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(0.0, 0.33, 0.67);

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

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;

    // Scale to view the Mandelbrot set
```

```
    vec2 c = uv * 3.5;
    c.x -= 0.5;

    // 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;
    }

    // Smooth coloring
    vec3 color;
    if(iter < max_iter) {
        // Smooth iteration count
        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);
        color = palette(t);
    } else {
        // Inside the set: black
        color = vec3(0.0);
    }

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

B.3. A3. Julia Set Explorer

Template for Julia set implementation. Students fill in the iteration code based on Mandelbrot.

```
// Complex number operations
float cabs2(vec2 z) {
    return dot(z, z);
}

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

```
// Cosine-based color 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(0.0, 0.33, 0.67);

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

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 p = uv * 3.0;

    // Fix c to an interesting value
    vec2 c = vec2(-0.7, 0.27015);

    // HOMEWORK: Initialize z from pixel position
    // HOMEWORK: Iterate z_{n+1} = z_n^2 + c
    // HOMEWORK: Use smooth coloring like Mandelbrot

    // Placeholder color (replace with your implementation)
    vec3 color = vec3(0.5);

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

B.3.1. A3b. Julia Set (Complete Solution)

Here's a complete working Julia set for reference:

```
// Complex number operations
float cabs2(vec2 z) {
    return dot(z, z);
}

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

// Cosine-based color 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(0.0, 0.33, 0.67);

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

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 p = uv * 3.0;

    // Fix c to an interesting value
    vec2 c = vec2(-0.7, 0.27015);

    // Initialize z from pixel position (key difference from Mandelbrot!)
    vec2 z = p;

    // Julia set iteration
    int max_iter = 100;
    int iter;

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

    // Smooth coloring
    vec3 color;
    if(iter < max_iter) {
        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);
        color = palette(t);
    } else {
        color = vec3(0.0);
    }

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

B.3.2. A3c. Julia Set with Mouse Control

For interactive exploration of parameter space:

```
// Complex number operations
float cabs2(vec2 z) {
    return dot(z, z);
}

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

// Cosine-based color 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(0.0, 0.33, 0.67);

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

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 p = uv * 3.0;

    // c controlled by mouse position
    vec2 mouse_uv = (iMouse.xy / iResolution.xy) - 0.5;
    mouse_uv.x *= iResolution.x / iResolution.y;
    vec2 c = mouse_uv * 3.0;

    // Fallback if mouse hasn't been clicked
    if(iMouse.z < 0.5) {
        c = vec2(-0.7, 0.27015);
    }

    // Initialize z from pixel position
    vec2 z = p;

    // Julia set iteration
    int max_iter = 100;
    int iter;
```

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

    // Smooth coloring
    vec3 color;
    if(iter < max_iter) {
        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);
        color = palette(t);
    } else {
        color = vec3(0.0);
    }

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

B.3.3. A3d. Julia Set with Animation

Animating through parameter space:

```
// Complex number operations
float cabs2(vec2 z) {
    return dot(z, z);
}

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

// Cosine-based color 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(0.0, 0.33, 0.67);

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

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 p = uv * 3.0;

    // Animate c around a circle in parameter space
    float angle = iTime * 0.3;
    float radius = 0.7885;
    vec2 c = vec2(radius * cos(angle), radius * sin(angle));

    // Initialize z from pixel position
    vec2 z = p;

    // Julia set iteration
    int max_iter = 100;
    int iter;

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

    // Smooth coloring
    vec3 color;
    if(iter < max_iter) {
        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);
        color = palette(t);
    } else {
        color = vec3(0.0);
    }

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

B.4. A4. Circle Inversion Visualization

Visualizes circle inversion by showing how it deforms a grid.

```
// Circle inversion function
vec2 invertCircle(vec2 p, vec2 center, float radius) {
    vec2 diff = p - center;
```

```
    float r2 = dot(diff, diff);

    // Handle center (would be division by zero)
    if(r2 < 0.0001) return vec2(1000.0);

    return center + (radius * radius) * diff / r2;
}

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 p = uv * 4.0;

    // Inversion circle
    vec2 circleCenter = vec2(0.0, 0.0);
    float circleRadius = 1.0;

    // Apply inversion
    vec2 p_inverted = invertCircle(p, circleCenter, circleRadius);

    // Draw a grid in the inverted space
    vec2 grid = fract(p_inverted * 2.0);
    float gridLine = step(0.95, max(grid.x, grid.y));

    vec3 color = vec3(gridLine);

    // Draw the inversion circle itself (for reference)
    float circDist = abs(length(p) - circleRadius);
    if(circDist < 0.05) color = vec3(1.0, 0.0, 0.0);

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

B.5. A5. Apollonian Gasket

Complete Apollonian gasket implementation with iteration coloring.

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

```
// Circle inversion
vec2 invertCircle(vec2 p, vec2 center, float radius) {
    vec2 diff = p - center;
    float r2 = dot(diff, diff);

    if(r2 < 0.0001) return vec2(1000.0);

    return center + (radius * radius) * diff / r2;
}

// Setup four mutually tangent circles (three inner + one outer)
void setupApollonianCircles(out Circle c1, out Circle c2, out Circle c3, out Circle outer) {
    float r = 0.5;  // Radius of each inner circle
    // For three circles to be mutually tangent: distance between centers = 2r
    // Centers form equilateral triangle with circumradius = 2r/sqrt(3)
    float d = 2.0 * r / sqrt(3.0);  // ≈ 0.577 for r = 0.5

    // Three inner circles
    c1 = Circle(vec2(0.0, d), r);
    c2 = Circle(vec2(-d * 0.866, -d * 0.5), r);  // 0.866 ≈ sqrt(3)/2
    c3 = Circle(vec2(d * 0.866, -d * 0.5), r);

    // Outer circle tangent to all three, centered at origin
    float R = d + r;  // ≈ 1.077 for r = 0.5
    outer = Circle(vec2(0.0, 0.0), R);
}

// Iterate inversions through four circles (three inner + one outer)
vec2 iterateApollonian(vec2 p, Circle c1, Circle c2, Circle c3, Circle outer,
                       int maxIter, out int finalIter, out int lastCircle) {
    for(int i = 0; i < maxIter; i++) {
        bool moved = false;

        // Check the three inner circles
        if(length(p - c1.center) < c1.radius) {
            p = invertCircle(p, c1.center, c1.radius);
            lastCircle = 0;
            moved = true;
        }
        else if(length(p - c2.center) < c2.radius) {
            p = invertCircle(p, c2.center, c2.radius);
            lastCircle = 1;
            moved = true;
        }
        else if(length(p - c3.center) < c3.radius) {
            p = invertCircle(p, c3.center, c3.radius);
            lastCircle = 2;
            moved = true;
        }
        // Check if outside the outer circle
```

```
        else if(length(p - outer.center) > outer.radius) {
            p = invertCircle(p, outer.center, outer.radius);
            lastCircle = 3;
            moved = true;
        }

        if(!moved) {
            finalIter = i;
            return p;
        }
    }

    finalIter = maxIter;
    return p;
}

// Cosine-based color 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(0.0, 0.33, 0.67);

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

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 p = uv * 4.0;

    // Setup circles
    Circle c1, c2, c3, outer;
    setupApollonianCircles(c1, c2, c3, outer);

    // Iterate
    int maxIter = 50;
    int finalIter, lastCircle;
    vec2 final_p = iterateApollonian(p, c1, c2, c3, outer, maxIter, finalIter, lastCircle);

    // Color by iteration count
    float t = float(finalIter) / float(maxIter);
    vec3 color = palette(t);

    // Draw all four circles for reference
    float d1 = abs(length(p - c1.center) - c1.radius);
    float d2 = abs(length(p - c2.center) - c2.radius);
```

```
    float d3 = abs(length(p - c3.center) - c3.radius);
    float d_outer = abs(length(p - outer.center) - outer.radius);
    float d = min(min(d1, min(d2, d3)), d_outer);

    if(d < 0.02) color = vec3(1.0);

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

B.5.1. A5b. Apollonian Gasket (Basin Coloring)

Alternative coloring showing which circle's basin each point falls into:

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

// Circle inversion
vec2 invertCircle(vec2 p, vec2 center, float radius) {
    vec2 diff = p - center;
    float r2 = dot(diff, diff);

    if(r2 < 0.0001) return vec2(1000.0);

    return center + (radius * radius) * diff / r2;
}

// Setup four mutually tangent circles (three inner + one outer)
void setupApollonianCircles(out Circle c1, out Circle c2, out Circle c3, out Circle outer) {
    float r = 0.5;  // Radius of each inner circle
    // For three circles to be mutually tangent: distance between centers = 2r
    // Centers form equilateral triangle with circumradius = 2r/sqrt(3)
    float d = 2.0 * r / sqrt(3.0);  // ≈ 0.577 for r = 0.5

    // Three inner circles
    c1 = Circle(vec2(0.0, d), r);
    c2 = Circle(vec2(-d * 0.866, -d * 0.5), r);  // 0.866 ≈ sqrt(3)/2
    c3 = Circle(vec2(d * 0.866, -d * 0.5), r);

    // Outer circle tangent to all three, centered at origin
    float R = d + r;  // ≈ 1.077 for r = 0.5
    outer = Circle(vec2(0.0, 0.0), R);
}

// Iterate inversions through four circles (three inner + one outer)
vec2 iterateApollonian(vec2 p, Circle c1, Circle c2, Circle c3, Circle outer,
                       int maxIter, out int finalIter, out int lastCircle) {
```

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

        // Check the three inner circles
        if(length(p - c1.center) < c1.radius) {
            p = invertCircle(p, c1.center, c1.radius);
            lastCircle = 0;
            moved = true;
        }
        else if(length(p - c2.center) < c2.radius) {
            p = invertCircle(p, c2.center, c2.radius);
            lastCircle = 1;
            moved = true;
        }
        else if(length(p - c3.center) < c3.radius) {
            p = invertCircle(p, c3.center, c3.radius);
            lastCircle = 2;
            moved = true;
        }
        // Check if outside the outer circle
        else if(length(p - outer.center) > outer.radius) {
            p = invertCircle(p, outer.center, outer.radius);
            lastCircle = 3;
            moved = true;
        }

        if(!moved) {
            finalIter = i;
            return p;
        }
    }

    finalIter = maxIter;
    return p;
}


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 p = uv * 4.0;

    // Setup circles
    Circle c1, c2, c3, outer;
    setupApollonianCircles(c1, c2, c3, outer);

    // Iterate
```

```
    int maxIter = 50;
    int finalIter, lastCircle;
    vec2 final_p = iterateApollonian(p, c1, c2, c3, outer, maxIter, finalIter, lastCircle);

    // Color by which circle we last hit
    vec3 colors[4];
    colors[0] = vec3(1.0, 0.0, 0.0);  // Circle 1: Red
    colors[1] = vec3(0.0, 1.0, 0.0);  // Circle 2: Green
    colors[2] = vec3(0.0, 0.0, 1.0);  // Circle 3: Blue
    colors[3] = vec3(1.0, 1.0, 0.0);  // Outer circle: Yellow

    vec3 color = colors[lastCircle];

    // Draw all four circles for reference
    float d1 = abs(length(p - c1.center) - c1.radius);
    float d2 = abs(length(p - c2.center) - c2.radius);
    float d3 = abs(length(p - c3.center) - c3.radius);
    float d_outer = abs(length(p - outer.center) - outer.radius);
    float d = min(min(d1, min(d2, d3)), d_outer);

    if(d < 0.02) color = vec3(1.0);

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

B.6. A6. Grid of Julia Sets (Optional Homework)

Shows many Julia sets in a grid, revealing the Mandelbrot set structure.

```
// Complex number operations
float cabs2(vec2 z) {
    return dot(z, z);
}

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

// Cosine-based color 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(0.0, 0.33, 0.67);

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

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

    // Divide screen into grid cells
    float grid_size = 50.0;  // 8×8 grid
    vec2 cell_id = floor(p * grid_size / 4.0);
    vec2 cell_p = fract(p * grid_size / 4.0) - 0.5;
    cell_p *= 4.0;  // Local coordinates within cell

    // Map cell_id to parameter c
    vec2 c = (cell_id / grid_size) * 3.0 - vec2(2.5, 1.5);
    c.x -= 0.5;  // Center on Mandelbrot set

    // Run Julia set iteration
    vec2 z = cell_p;
    int max_iter = 50;
    int iter;

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

    // Color
    float t = float(iter) / float(max_iter);
    vec3 color = vec3(t);

    // Draw grid lines
    vec2 grid_edge = abs(fract(p * grid_size / 4.0) - 0.5);
    if(max(grid_edge.x, grid_edge.y) > 0.48) color = vec3(0.0);

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

B.7. Notes on Using These Shaders

B.7.1. Getting Started

1. Go to https://www.shadertoy.com/new
2. Delete the default code
3. Copy and paste any of the above listings
4. Click the play button (▶) or press Alt+Enter

B.7.2. Modifying Parameters

Each shader has parameters you can adjust at the top of `mainImage()`:

Mandelbrot/Julia: - `max_iter` - More iterations reveal finer detail (try 200) - Color scale factor in `uv * 3.5` - Zoom in/out - Offset `c.x -= 0.5` - Pan the view

Julia specific: - `vec2 c = ...` - Change the parameter to see different Julia sets - Try values from the homework section

Circle Inversion: - `circleCenter` - Move the inversion circle - `circleRadius` - Change the size - Grid frequency in `p_inverted * 2.0` - Denser or sparser grid

Apollonian Gasket: - `maxIter` - More iterations show deeper nesting - `R` and `r` in setup function - Change circle sizes - Color palette parameters in `palette()` function

B.7.3. Performance Tips

If a shader runs slowly: - Reduce `max_iter` (try 50 instead of 100) - Lower the resolution (bottom right resolution dropdown in Shadertoy) - Some computers may struggle with smooth coloring - remove it for speed

B.7.4. Exploring Further

All of these shaders are starting points! Try: - Combining techniques (Julia set with Apollonian coloring scheme) - Animating parameters with `iTime` - Adding mouse interaction with `iMouse` - Creating your own color palettes - Experimenting with different circle configurations

The goal is to understand how simple iterative processes create complex fractals, and how to implement them efficiently on the GPU!

C. Appendix: Complete Shader Code for Day 3

This appendix provides complete, standalone Shadertoy code for each shader program presented in Day 3. Each listing includes all necessary helper functions and can be copied directly into Shadertoy (https://www.shadertoy.com/new) and run immediately.

C.1. Part 1: Euclidean Tilings

C.1.1. E1. Strip Tiling (Basic)

Simple horizontal strip tiling showing the folding algorithm in one dimension.

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

    // Fold into the strip [0, 1]
    for(int i = 0; i < 20; i++) {
        if(p.x < 0.0) p.x = -p.x;
        if(p.x > 1.0) p.x = 2.0 - p.x;
    }

    // Draw something in the fundamental domain
    vec3 color = vec3(0.2, 0.2, 0.3);  // Dark background

    // A circle in the strip
    float d = length(p - vec2(0.5, 0.0));
    if(d < 0.3) {
        color = vec3(1.0, 0.8, 0.3);  // Yellow circle
    }

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

C.1.2. E2. Square Tiling (Basic)

2D square tiling extending the folding algorithm to both dimensions.

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

    // Fold into the square [0,1] × [0,1]
    for(int i = 0; i < 20; i++) {
        if(p.x < 0.0) p.x = -p.x;
        if(p.x > 1.0) p.x = 2.0 - p.x;
        if(p.y < 0.0) p.y = -p.y;
        if(p.y > 1.0) p.y = 2.0 - p.y;
    }

    // Draw something in the fundamental domain
    vec3 color = vec3(0.2, 0.2, 0.3);

    // Circle at center
    float d = length(p - vec2(0.5, 0.5));
    if(d < 0.3) {
        color = vec3(1.0, 0.8, 0.3);
    }

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

C.1.3. E3. Square Tiling with Fold Count

Square tiling colored by the number of reflections needed to reach the fundamental domain.

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

    // Fold into the square [0,1] × [0,1]
    int foldCount = 0;
    for(int i = 0; i < 20; i++) {
```

```
        vec2 p_old = p;

        if(p.x < 0.0) p.x = -p.x;
        if(p.x > 1.0) p.x = 2.0 - p.x;
        if(p.y < 0.0) p.y = -p.y;
        if(p.y > 1.0) p.y = 2.0 - p.y;

        // If point didn't move, we're done
        if(length(p - p_old) < 0.0001) break;
        foldCount++;
    }

    // Color based on fold count
    float t = float(foldCount) / 8.0;
    vec3 color = 0.5 + 0.5 * cos(6.28318 * (vec3(1.0) * t + vec3(0.0, 0.33, 0.67)));

    // Draw something in the fundamental domain
    float d = length(p - vec2(0.5, 0.5));
    if(d < 0.3) {
        color = mix(color, vec3(1.0, 1.0, 0.3), smoothstep(0.3, 0.25, d));
    }

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

C.1.4. E4a. Single Half-Space Visualization

Visualizes one side of a line (a half-space).

```
struct HalfSpace {
    float a, b, c;
    float side;
};

bool inside(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;
    return (value - hs.c) * hs.side < 0.0;
}

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

```
    // Define a half-space: x < 1 (left side of vertical line at x=1)
    HalfSpace hs = HalfSpace(1.0, 0.0, 1.0, 1.0);

    // Color based on whether we're inside
    vec3 color = inside(p, hs) ? vec3(0.3, 0.5, 0.7) : vec3(0.1, 0.1, 0.2);

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

C.1.5. E4b. Single Half-Space with Boundary Line

Same as E4a but with the boundary line drawn.

```
struct HalfSpace {
    float a, b, c;
    float side;
};

bool inside(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;
    return (value - hs.c) * hs.side < 0.0;
}

float distToHalfSpace(vec2 p, HalfSpace hs) {
    return abs(hs.a * p.x + hs.b * p.y - hs.c) / length(vec2(hs.a, hs.b));
}

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

    // Define a half-space: x < 1
    HalfSpace hs = HalfSpace(1.0, 0.0, 1.0, 1.0);

    // Color based on whether we're inside
    vec3 color = inside(p, hs) ? vec3(0.3, 0.5, 0.7) : vec3(0.1, 0.1, 0.2);

    // Draw the boundary line
    float d = distToHalfSpace(p, hs);
    if(d < 0.02) color = vec3(1.0);  // White boundary

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

C.1.6. E5a. Four Half-Spaces (Additive Coloring)

Intersecting four half-spaces to create a square using additive coloring.

```
struct HalfSpace {
    float a, b, c;
    float side;
};

bool inside(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;
    return (value - hs.c) * hs.side < 0.0;
}

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

    // Define the four half-spaces for [0,1] × [0,1]
    HalfSpace left   = HalfSpace(1.0, 0.0, 0.0, -1.0);  // x > 0
    HalfSpace right  = HalfSpace(1.0, 0.0, 1.0,  1.0);  // x < 1
    HalfSpace bottom = HalfSpace(0.0, 1.0, 0.0, -1.0);  // y > 0
    HalfSpace top    = HalfSpace(0.0, 1.0, 1.0,  1.0);  // y < 1

    // Additive coloring - each half-space adds brightness
    vec3 color = vec3(0.0);

    if(inside(p, left))   color += vec3(0.1, 0.15, 0.2);
    if(inside(p, right))  color += vec3(0.1, 0.15, 0.2);
    if(inside(p, bottom)) color += vec3(0.1, 0.15, 0.2);
    if(inside(p, top))    color += vec3(0.1, 0.15, 0.2);

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

C.1.7. E5b. Four Half-Spaces with Boundaries

Enhanced version with binary coloring and boundary lines.

```glsl
struct HalfSpace {
    float a, b, c;
    float side;
};

bool inside(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;
    return (value - hs.c) * hs.side < 0.0;
}

float distToHalfSpace(vec2 p, HalfSpace hs) {
    return abs(hs.a * p.x + hs.b * p.y - hs.c) / length(vec2(hs.a, hs.b));
}

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

    // Define the four half-spaces for [0,1] × [0,1]
    HalfSpace left   = HalfSpace(1.0, 0.0, 0.0, -1.0);  // x > 0
    HalfSpace right  = HalfSpace(1.0, 0.0, 1.0,  1.0);  // x < 1
    HalfSpace bottom = HalfSpace(0.0, 1.0, 0.0, -1.0);  // y > 0
    HalfSpace top    = HalfSpace(0.0, 1.0, 1.0,  1.0);  // y < 1

    // Binary coloring: inside domain or not
    bool in_square = inside(p, left) && inside(p, right) &&
                     inside(p, bottom) && inside(p, top);
    vec3 color = in_square ? vec3(0.4, 0.6, 0.8) : vec3(0.1, 0.1, 0.2);

    // Draw boundaries
    float d1 = distToHalfSpace(p, left);
    float d2 = distToHalfSpace(p, right);
    float d3 = distToHalfSpace(p, bottom);
    float d4 = distToHalfSpace(p, top);
    float d = min(min(d1, d2), min(d3, d4));

    if(d < 0.02) color = vec3(1.0);  // White boundaries

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

C.1.8. E6. Three Half-Spaces for Triangle (Additive)

Visualizing three half-spaces defining an equilateral triangle.

```
struct HalfSpace {
    float a, b, c;
    float side;
};

bool inside(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;
    return (value - hs.c) * hs.side < 0.0;
}

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

    // Define three half-spaces for equilateral triangle
    HalfSpace hs1 = HalfSpace(1.5, -0.866, -0.866, -1.0);
    HalfSpace hs2 = HalfSpace(0.0, 1.732, -0.866, -1.0);
    HalfSpace hs3 = HalfSpace(-1.5, -0.866, -0.866, -1.0);

    // Additive coloring
    vec3 color = vec3(0.0);

    if(inside(p, hs1)) color += vec3(0.15, 0.2, 0.25);
    if(inside(p, hs2)) color += vec3(0.15, 0.2, 0.25);
    if(inside(p, hs3)) color += vec3(0.15, 0.2, 0.25);

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

C.1.9. E7. Euclidean Triangle Tiling (Basic)

Full triangle tiling using half-space reflections.

```
struct HalfSpace {
    float a, b, c;
    float side;
};

vec2 reflectInto(vec2 p, HalfSpace hs) {
```

```
    float value = hs.a * p.x + hs.b * p.y;

    if((value - hs.c) * hs.side < 0.0) {
        return p;
    }

    vec2 normal = vec2(hs.a, hs.b);
    float norm = length(normal);
    normal = normal / norm;

    float signedDist = (value - hs.c) / norm;
    return p - 2.0 * signedDist * normal;
}

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

    // Define three half-spaces for equilateral triangle
    HalfSpace hs1 = HalfSpace(1.5, -0.866, -0.866, -1.0);
    HalfSpace hs2 = HalfSpace(0.0, 1.732, -0.866, -1.0);
    HalfSpace hs3 = HalfSpace(-1.5, -0.866, -0.866, -1.0);

    // Fold into triangle
    for(int i = 0; i < 20; i++) {
        p = reflectInto(p, hs1);
        p = reflectInto(p, hs2);
        p = reflectInto(p, hs3);
    }

    // Simple coloring
    vec3 color = vec3(0.3, 0.5, 0.7);

    // Draw a circle in fundamental domain
    float d = length(p - vec2(0.0, -0.3));
    if(d < 0.2) {
        color = vec3(1.0, 0.8, 0.3);
    }

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

C.1.10. E8. Euclidean Triangle Tiling with Fold Count

Triangle tiling colored by reflection count with convergence check.

```
struct HalfSpace {
    float a, b, c;
    float side;
};

vec2 reflectInto(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;

    if((value - hs.c) * hs.side < 0.0) {
        return p;
    }

    vec2 normal = vec2(hs.a, hs.b);
    float norm = length(normal);
    normal = normal / norm;

    float signedDist = (value - hs.c) / norm;
    return p - 2.0 * signedDist * normal;
}

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

    // Define three half-spaces for equilateral triangle
    HalfSpace hs1 = HalfSpace(1.5, -0.866, -0.866, -1.0);
    HalfSpace hs2 = HalfSpace(0.0, 1.732, -0.866, -1.0);
    HalfSpace hs3 = HalfSpace(-1.5, -0.866, -0.866, -1.0);

    // Fold into triangle with iteration count
    int foldCount = 0;
    for(int i = 0; i < 20; i++) {
        vec2 p_old = p;

        p = reflectInto(p, hs1);
        p = reflectInto(p, hs2);
        p = reflectInto(p, hs3);

        if(length(p - p_old) < 0.0001) break;
        foldCount++;
    }

    // Color by fold count
```

```
    float t = float(foldCount) / 10.0;
    vec3 color = 0.5 + 0.5 * cos(6.28318 * (vec3(1.0) * t + vec3(0.0, 0.33, 0.67)));

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

C.1.11. E9. Euclidean Triangle Tiling with Edges and Vertices

Complete triangle tiling with visible structure (reference implementation for homework).

```
struct HalfSpace {
    float a, b, c;
    float side;
};

vec2 reflectInto(vec2 p, HalfSpace hs) {
    float value = hs.a * p.x + hs.b * p.y;

    if((value - hs.c) * hs.side < 0.0) {
        return p;
    }

    vec2 normal = vec2(hs.a, hs.b);
    float norm = length(normal);
    normal = normal / norm;

    float signedDist = (value - hs.c) / norm;
    return p - 2.0 * signedDist * normal;
}

float distToHalfSpace(vec2 p, HalfSpace hs) {
    return abs(hs.a * p.x + hs.b * p.y - hs.c) / length(vec2(hs.a, hs.b));
}

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

    // Define three half-spaces for equilateral triangle
    HalfSpace hs1 = HalfSpace(1.5, -0.866, -0.866, -1.0);
    HalfSpace hs2 = HalfSpace(0.0, 1.732, -0.866, -1.0);
    HalfSpace hs3 = HalfSpace(-1.5, -0.866, -0.866, -1.0);
```

```
    // Fold into triangle with parity tracking
    int foldCount = 0;
    for(int i = 0; i < 20; i++) {
        vec2 p_old = p;

        p = reflectInto(p, hs1);
        p = reflectInto(p, hs2);
        p = reflectInto(p, hs3);

        if(length(p - p_old) < 0.0001) break;
        foldCount++;
    }

    // Color by parity (alternating pattern)
    float parity = mod(float(foldCount), 2.0);
    vec3 color;
    if(parity < 0.5) {
        color = vec3(0.7, 0.8, 0.9);  // Light blue
    } else {
        color = vec3(0.5, 0.6, 0.8);  // Darker blue
    }

    // Draw edges
    float d1 = distToHalfSpace(p, hs1);
    float d2 = distToHalfSpace(p, hs2);
    float d3 = distToHalfSpace(p, hs3);
    float d_edge = min(min(d1, d2), d3);

    if(d_edge < 0.02) {
        color = vec3(1.0, 1.0, 1.0);  // White edges
    }

    // Draw vertices (approximate positions)
    vec2 v1 = vec2(-0.577, -0.333);
    vec2 v2 = vec2(0.577, -0.333);
    vec2 v3 = vec2(0.0, 0.667);

    float d_v1 = length(p - v1);
    float d_v2 = length(p - v2);
    float d_v3 = length(p - v3);
    float d_vert = min(min(d_v1, d_v2), d_v3);

    if(d_vert < 0.08) {
        color = vec3(1.0, 0.0, 0.0);  // Red vertices
    }

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

C.2. Part 2: Hyperbolic Tilings

C.2.1. H1. Euclidean Distance Visualization (For Comparison)

Shows standard Euclidean distance circles for comparison with hyperbolic.

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 4.0;
    uv.x *= iResolution.x / iResolution.y;
    vec2 p = uv + vec2(0.0, 1.5);  // Shift up so we're in y > 0

    // Mouse position as center (or default)
    vec2 mouse = iMouse.xy / iResolution.xy;
    if(iMouse.z < 0.5) mouse = vec2(0.5, 0.7);  // Default if no click
    mouse = (mouse - 0.5) * 4.0;
    mouse.x *= iResolution.x / iResolution.y;
    vec2 center = mouse + vec2(0.0, 1.5);

    // Euclidean distance
    float dist = length(p - center);

    // Draw a disk of radius 0.5 using two circles
    float radius = 0.5;
    vec3 color = vec3(0.1, 0.1, 0.2);  // Background

    // Outer circle (slightly larger)
    if(dist < radius + 0.02) {
        color = vec3(1.0, 1.0, 0.3);  // Yellow ring
    }

    // Inner circle (slightly smaller) - "cuts out" interior
    if(dist < radius - 0.02) {
        color = vec3(0.4, 0.6, 0.8);  // Blue interior
    }

    // Draw center point
    if(length(p - center) < 0.05) {
        color = vec3(1.0, 0.0, 0.0);
    }

    // Darken outside upper half-plane
    if(p.y < 0.0) {
        color *= 0.3;
    }

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

C.2.2. H2. Hyperbolic Distance Visualization

Shows hyperbolic distance "circles" in the upper half-plane model.

```
float hyperbolicDistance(vec2 z1, vec2 z2) {
    vec2 diff = z1 - z2;
    float diff2 = dot(diff, diff);
    float denom = 2.0 * z1.y * z2.y;
    float arg = 1.0 + diff2 / denom;
    return log(arg + sqrt(arg * arg - 1.0));  // arccosh(arg)
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 4.0;
    uv.x *= iResolution.x / iResolution.y;
    vec2 p = uv + vec2(0.0, 1.5);

    // Mouse position as center
    vec2 mouse = iMouse.xy / iResolution.xy;
    if(iMouse.z < 0.5) mouse = vec2(0.5, 0.7);
    mouse = (mouse - 0.5) * 4.0;
    mouse.x *= iResolution.x / iResolution.y;
    vec2 center = mouse + vec2(0.0, 1.5);

    // Hyperbolic distance
    float dist = hyperbolicDistance(p, center);

    // Draw a hyperbolic disk using two "circles"
    float radius = 0.5;
    vec3 color = vec3(0.1, 0.1, 0.2);  // Background

    // Outer boundary
    if(dist < radius + 0.05) {
        color = vec3(1.0, 1.0, 0.3);  // Yellow ring
    }

    // Inner region
    if(dist < radius - 0.05) {
        color = vec3(0.4, 0.6, 0.8);  // Blue interior
    }

    // Draw center
    if(hyperbolicDistance(p, center) < 0.1) {
        color = vec3(1.0, 0.0, 0.0);
    }
```

```
    // Darken outside upper half-plane
    if(p.y < 0.0) {
        color *= 0.3;
    }

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

C.2.3. H3a. Single Vertical Geodesic Half-Space

Visualizes one side of a vertical geodesic (hyperbolic "line").

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 4.0;
    uv.x *= iResolution.x / iResolution.y;
    vec2 z = uv + vec2(0.0, 1.5);

    // Vertical geodesic at x = 0, showing right side (x > 0)
    float x_pos = 0.0;
    float side = -1.0;  // side = -1.0 means x > x_pos

    // Check which side we're on
    bool on_right_side = (z.x - x_pos) * side < 0.0;

    vec3 color = on_right_side ? vec3(0.3, 0.5, 0.7) : vec3(0.1, 0.1, 0.2);

    // Draw the geodesic (vertical line)
    if(abs(z.x - x_pos) < 0.02) {
        color = vec3(1.0, 1.0, 1.0);
    }

    // Darken outside upper half-plane
    if(z.y < 0.0) {
        color *= 0.3;
    }

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

C.2.4. H3b. Single Circular Geodesic Half-Space

Visualizes one side of a semicircular geodesic.

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 4.0;
    uv.x *= iResolution.x / iResolution.y;
    vec2 z = uv + vec2(0.0, 1.5);

    // Semicircular geodesic from p to q on real axis
    float p = -1.0;
    float q = 1.0;
    float center = (p + q) / 2.0;
    float radius = abs(p - q) / 2.0;

    // Distance from center
    vec2 rel = z - vec2(center, 0.0);
    float dist2 = dot(rel, rel);

    // side = 1.0 means outside the circle
    float side = 1.0;
    bool outside_circle = (dist2 - radius * radius) * side > 0.0;

    vec3 color = outside_circle ? vec3(0.3, 0.5, 0.7) : vec3(0.1, 0.1, 0.2);

    // Draw the geodesic (semicircle)
    float dist_to_circle = abs(length(rel) - radius);
    if(z.y > 0.0 && dist_to_circle < 0.02) {
        color = vec3(1.0, 1.0, 1.0);
    }

    // Darken outside upper half-plane
    if(z.y < 0.0) {
        color *= 0.3;
    }

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

C.2.5. H3c. Three Geodesics Additively Colored (Building the Triangle)

Shows the three geodesics of the (2,3,∞) triangle using additive coloring.

```
void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 4.0;
    uv.x *= iResolution.x / iResolution.y;
    vec2 z = uv + vec2(0.0, 1.5);

    // Three geodesics of (2,3,∞) triangle
    // Left vertical line: x = -1, want x > -1
    bool inside_left = (z.x - (-1.0)) * (-1.0) < 0.0;

    // Right vertical line: x = 1, want x < 1
    bool inside_right = (z.x - 1.0) * 1.0 < 0.0;

    // Semicircle from -1 to 1, want outside (above)
    float center = 0.0;
    float radius = 1.0;
    vec2 rel = z - vec2(center, 0.0);
    float dist2 = dot(rel, rel);
    bool inside_circle = (dist2 - radius * radius) * 1.0 > 0.0;

    // Additive coloring
    vec3 color = vec3(0.0);

    if(inside_left)   color += vec3(0.15, 0.2, 0.25);
    if(inside_right)  color += vec3(0.15, 0.2, 0.25);
    if(inside_circle) color += vec3(0.15, 0.2, 0.25);

    // Draw the three geodesics
    if(abs(z.x - (-1.0)) < 0.02) color = vec3(1.0);  // Left line
    if(abs(z.x - 1.0) < 0.02) color = vec3(1.0);     // Right line

    float dist_to_circle = abs(length(rel) - radius);
    if(z.y > 0.0 && dist_to_circle < 0.02) color = vec3(1.0);  // Semicircle

    // Darken outside upper half-plane
    if(z.y < 0.0) {
        color *= 0.3;
    }

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

C.2.6. H4. Basic (2,3,∞) Triangle Tiling

Complete hyperbolic triangle tiling with simple coloring.

```glsl
vec2 reflectIntoVertical(vec2 z, float x_pos, float side) {
    if((z.x - x_pos) * side < 0.0) return z;
    z.x = 2.0 * x_pos - z.x;
    return z;
}

vec2 reflectIntoCircular(vec2 z, float p, float q, float side) {
    float center = (p + q) / 2.0;
    float radius = abs(p - q) / 2.0;
    vec2 rel = z - vec2(center, 0.0);
    float dist2 = dot(rel, rel);
    if((dist2 - radius * radius) * side > 0.0) return z;

    return vec2(center, 0.0) + (radius * radius) * rel / dist2;
}

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

    // Shift to upper half-plane (need y > 0)
    vec2 z = uv + vec2(0.0, 1.5);

    // Fold into the (2,3,∞) triangle
    for(int i = 0; i < 50; i++) {
        vec2 z_old = z;

        // Reflect across left vertical line (x = -1, want x > -1)
        z = reflectIntoVertical(z, -1.0, -1.0);

        // Reflect across right vertical line (x = 1, want x < 1)
        z = reflectIntoVertical(z, 1.0, 1.0);

        // Reflect across semicircle (from -1 to 1, want outside/above)
        z = reflectIntoCircular(z, -1.0, 1.0, 1.0);

        // If point didn't move, we're inside
        if(length(z - z_old) < 0.0001) break;
    }

    // Simple coloring
    vec3 color = vec3(0.6, 0.7, 0.9);

    // Darken if below the real axis (outside hyperbolic space)
    if(z.y < 0.0) {
        color *= 0.3;
    }
```

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

C.2.7. H5. (2,3,∞) Triangle Tiling with Fold Count

Hyperbolic tiling colored by iteration count showing alternating pattern.

```
vec2 reflectIntoVertical(vec2 z, float x_pos, float side) {
    if((z.x - x_pos) * side < 0.0) return z;
    z.x = 2.0 * x_pos - z.x;
    return z;
}

vec2 reflectIntoCircular(vec2 z, float p, float q, float side) {
    float center = (p + q) / 2.0;
    float radius = abs(p - q) / 2.0;
    vec2 rel = z - vec2(center, 0.0);
    float dist2 = dot(rel, rel);
    if((dist2 - radius * radius) * side > 0.0) return z;

    return vec2(center, 0.0) + (radius * radius) * rel / dist2;
}

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

    // Shift to upper half-plane (need y > 0)
    vec2 z = uv + vec2(0.0, 1.5);

    // Fold into the (2,3,∞) triangle
    int foldCount = 0;
    for(int i = 0; i < 50; i++) {
        vec2 z_old = z;

        // Reflect across left vertical line (x = -1, want x > -1)
        z = reflectIntoVertical(z, -1.0, -1.0);

        // Reflect across right vertical line (x = 1, want x < 1)
        z = reflectIntoVertical(z, 1.0, 1.0);

        // Reflect across semicircle (from -1 to 1, want outside/above)
        z = reflectIntoCircular(z, -1.0, 1.0, 1.0);
```

```
        // If point didn't move, we're inside
        if(length(z - z_old) < 0.0001) break;
        foldCount++;
    }

    // Color by fold count parity
    float parity = mod(float(foldCount), 2.0);
    vec3 color;
    if(parity < 0.5) {
        color = vec3(0.7, 0.8, 0.9);  // Light blue
    } else {
        color = vec3(0.5, 0.6, 0.8);  // Darker blue
    }

    // Darken if below the real axis (outside hyperbolic space)
    if(z.y < 0.0) {
        color *= 0.3;
    }

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

C.2.8. H6. (2,3,∞) Triangle Tiling with Edges and Vertices

Complete hyperbolic tiling with visible structure (reference for homework).

```
vec2 reflectIntoVertical(vec2 z, float x_pos, float side) {
    if((z.x - x_pos) * side < 0.0) return z;
    z.x = 2.0 * x_pos - z.x;
    return z;
}

vec2 reflectIntoCircular(vec2 z, float p, float q, float side) {
    float center = (p + q) / 2.0;
    float radius = abs(p - q) / 2.0;
    vec2 rel = z - vec2(center, 0.0);
    float dist2 = dot(rel, rel);
    if((dist2 - radius * radius) * side > 0.0) return z;

    return vec2(center, 0.0) + (radius * radius) * rel / dist2;
}

float hyperbolicDistance(vec2 z1, vec2 z2) {
    vec2 diff = z1 - z2;
    float diff2 = dot(diff, diff);
    float denom = 2.0 * z1.y * z2.y;
    float arg = 1.0 + diff2 / denom;
```

```
    return log(arg + sqrt(arg * arg - 1.0));
}

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

    // Shift to upper half-plane (need y > 0)
    vec2 z = uv + vec2(0.0, 1.5);

    // Fold into the (2,3,∞) triangle
    int foldCount = 0;
    for(int i = 0; i < 50; i++) {
        vec2 z_old = z;

        z = reflectIntoVertical(z, -1.0, -1.0);
        z = reflectIntoVertical(z, 1.0, 1.0);
        z = reflectIntoCircular(z, -1.0, 1.0, 1.0);

        if(length(z - z_old) < 0.0001) break;
        foldCount++;
    }

    // Color by fold count parity
    float parity = mod(float(foldCount), 2.0);
    vec3 color;
    if(parity < 0.5) {
        color = vec3(0.7, 0.8, 0.9);  // Light blue
    } else {
        color = vec3(0.5, 0.6, 0.8);  // Darker blue
    }

    // Draw geodesic edges
    // Left vertical line (x = -1)
    if(abs(z.x - (-1.0)) < 0.02) {
        color = vec3(1.0, 1.0, 1.0);
    }

    // Right vertical line (x = 1)
    if(abs(z.x - 1.0) < 0.02) {
        color = vec3(1.0, 1.0, 1.0);
    }

    // Semicircle from -1 to 1
    vec2 rel = z - vec2(0.0, 0.0);
    float dist_to_circle = abs(length(rel) - 1.0);
    if(z.y > 0.0 && dist_to_circle < 0.02) {
```

```
        color = vec3(1.0, 1.0, 1.0);
    }

    // Draw vertices using hyperbolic distance
    vec2 v1 = vec2(-1.0, 0.01);  // Left vertex (slightly above axis)
    vec2 v2 = vec2(1.0, 0.01);   // Right vertex

    if(hyperbolicDistance(z, v1) < 0.15 || hyperbolicDistance(z, v2) < 0.15) {
        color = vec3(1.0, 0.0, 0.0);  // Red vertices
    }

    // Darken if below the real axis
    if(z.y < 0.0) {
        color *= 0.3;
    }

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

C.2.9. H7. Poincaré Disk Model

Same (2,3,∞) tiling displayed in the Poincaré disk using the Cayley transform.

```
vec2 reflectIntoVertical(vec2 z, float x_pos, float side) {
    if((z.x - x_pos) * side < 0.0) return z;
    z.x = 2.0 * x_pos - z.x;
    return z;
}

vec2 reflectIntoCircular(vec2 z, float p, float q, float side) {
    float center = (p + q) / 2.0;
    float radius = abs(p - q) / 2.0;
    vec2 rel = z - vec2(center, 0.0);
    float dist2 = dot(rel, rel);
    if((dist2 - radius * radius) * side > 0.0) return z;

    return vec2(center, 0.0) + (radius * radius) * rel / dist2;
}

// Cayley transform: Poincaré disk -> Upper half-plane
vec2 cayleyDiskToUHP(vec2 w) {
    // z = i(1-w)/(1+w)
    vec2 numer = vec2(-w.y, 1.0 - w.x);  // i(1-w) = i - iw
    vec2 denom = vec2(1.0 + w.x, w.y);    // 1 + w

    float denom_mag2 = dot(denom, denom);
    return vec2(
```

```
        (numer.x * denom.x + numer.y * denom.y) / denom_mag2,
        (numer.y * denom.x - numer.x * denom.y) / denom_mag2
    );
}

// Inverse Cayley: Upper half-plane -> Poincaré disk
vec2 cayleyUHPToDisk(vec2 z) {
    // w = (z-i)/(z+i)
    vec2 numer = vec2(z.x, z.y - 1.0);    // z - i
    vec2 denom = vec2(z.x, z.y + 1.0);    // z + i

    float denom_mag2 = dot(denom, denom);
    return vec2(
        (numer.x * denom.x + numer.y * denom.y) / denom_mag2,
        (numer.y * denom.x - numer.x * denom.y) / denom_mag2
    );
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup for Poincaré disk
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 2.5;
    uv.x *= iResolution.x / iResolution.y;
    vec2 w = uv;  // Point in Poincaré disk

    // Convert to upper half-plane
    vec2 z = cayleyDiskToUHP(w);

    // Fold into the (2,3,∞) triangle (in UHP)
    int foldCount = 0;
    for(int i = 0; i < 50; i++) {
        vec2 z_old = z;

        z = reflectIntoVertical(z, -1.0, -1.0);
        z = reflectIntoVertical(z, 1.0, 1.0);
        z = reflectIntoCircular(z, -1.0, 1.0, 1.0);

        if(length(z - z_old) < 0.0001) break;
        foldCount++;
    }

    // Color by fold count parity
    float parity = mod(float(foldCount), 2.0);
    vec3 color;
    if(parity < 0.5) {
        color = vec3(0.7, 0.8, 0.9);  // Light blue
    } else {
        color = vec3(0.5, 0.6, 0.8);  // Darker blue
    }
```

```
    // Darken outside unit disk
    if(length(w) > 1.0) {
        color *= 0.3;
    }

    // Draw boundary circle
    if(abs(length(w) - 1.0) < 0.02) {
        color = vec3(1.0, 1.0, 0.0);  // Yellow boundary
    }

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

C.2.10. H8. Klein Disk Model

Same tiling in the Klein model where geodesics appear as straight lines.

```
vec2 reflectIntoVertical(vec2 z, float x_pos, float side) {
    if((z.x - x_pos) * side < 0.0) return z;
    z.x = 2.0 * x_pos - z.x;
    return z;
}

vec2 reflectIntoCircular(vec2 z, float p, float q, float side) {
    float center = (p + q) / 2.0;
    float radius = abs(p - q) / 2.0;
    vec2 rel = z - vec2(center, 0.0);
    float dist2 = dot(rel, rel);
    if((dist2 - radius * radius) * side > 0.0) return z;

    return vec2(center, 0.0) + (radius * radius) * rel / dist2;
}

// Poincaré disk -> Klein disk
vec2 poincareToKlein(vec2 w) {
    float w_mag2 = dot(w, w);
    return 2.0 * w / (1.0 + w_mag2);
}

// Klein disk -> Poincaré disk
vec2 kleinToPoincare(vec2 p) {
    float p_mag2 = dot(p, p);
    float denom = 1.0 + sqrt(1.0 - p_mag2);
    return p / denom;
}

// Cayley transform: Poincaré disk -> Upper half-plane
```

```
vec2 cayleyDiskToUHP(vec2 w) {
    vec2 numer = vec2(-w.y, 1.0 - w.x);
    vec2 denom = vec2(1.0 + w.x, w.y);

    float denom_mag2 = dot(denom, denom);
    return vec2(
        (numer.x * denom.x + numer.y * denom.y) / denom_mag2,
        (numer.y * denom.x - numer.x * denom.y) / denom_mag2
    );
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    // Coordinate setup for Klein disk
    vec2 uv = fragCoord / iResolution.xy;
    uv = (uv - 0.5) * 2.5;
    uv.x *= iResolution.x / iResolution.y;
    vec2 k = uv;  // Point in Klein disk

    // Convert Klein -> Poincaré -> Upper half-plane
    vec2 w = kleinToPoincare(k);
    vec2 z = cayleyDiskToUHP(w);

    // Fold into the (2,3,∞) triangle (in UHP)
    int foldCount = 0;
    for(int i = 0; i < 50; i++) {
        vec2 z_old = z;

        z = reflectIntoVertical(z, -1.0, -1.0);
        z = reflectIntoVertical(z, 1.0, 1.0);
        z = reflectIntoCircular(z, -1.0, 1.0, 1.0);

        if(length(z - z_old) < 0.0001) break;
        foldCount++;
    }

    // Color by fold count parity
    float parity = mod(float(foldCount), 2.0);
    vec3 color;
    if(parity < 0.5) {
        color = vec3(0.7, 0.8, 0.9);  // Light blue
    } else {
        color = vec3(0.5, 0.6, 0.8);  // Darker blue
    }

    // Darken outside unit disk
    if(length(k) > 1.0) {
        color *= 0.3;
    }
```

```
    // Draw boundary circle
    if(abs(length(k) - 1.0) < 0.02) {
        color = vec3(1.0, 1.0, 0.0);  // Yellow boundary
    }

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

C.3. Notes on Using These Shaders

C.3.1. Getting Started

1. Go to https://www.shadertoy.com/new
2. Delete the default code
3. Copy and paste any of the above listings
4. Click the play button (▶) or press Alt+Enter

C.3.2. Modifying Parameters

Euclidean Tilings: - Adjust fold iteration count (20 is conservative, 10 often sufficient) - Change fundamental domain by modifying half-space parameters - Experiment with different shapes (triangles, pentagons, hexagons) - Try different color palettes

Hyperbolic Tilings: - Mouse interaction in H1/H2 (click and drag to move center) - Adjust the shift in `z = uv + vec2(0.0, 1.5)` to change visible region - Increase iteration count (50) for more precision near boundaries - Try different triangle configurations (requires computing new geodesics)

C.3.3. Performance Tips

If a shader runs slowly: - Reduce iteration count - Lower resolution in Shadertoy settings - Simplify edge/vertex drawing code

C.3.4. Exploring Further

Euclidean extensions: - Implement other regular tilings (hexagons, pentagons) - Add animations by making half-spaces time-dependent - Create compound patterns with multiple fundamental domains

Hyperbolic extensions: - Implement (2,3,7) or (2,4,6) triangles - Decorate fundamental domains with patterns (Escher-style) - Explore pentagon tilings - Animate between different models

C.3.5. Key Observations

Euclidean vs Hyperbolic: Compare E7 and H4 - notice how the algorithm structure is identical but the reflection operations differ. This demonstrates the power of mathematical abstraction!

Model Comparisons: Run H4 (upper half-plane), H7 (Poincaré disk), and H8 (Klein disk) side by side. The same mathematical object looks dramatically different depending on the model, but the underlying hyperbolic geometry is identical.

Edge Behavior: In H6, notice how triangle edges near the boundary ($y \to 0$) appear more compressed. This visualizes the $1/y^2$ conformal factor in the hyperbolic metric.

D. GLSL