Alys Shah
IMDM327

# Final Project Documentation:
# Flow Field Painter

**GitHub Repo**: https://github.com/alysshah/imdm327-final

## 1. Introduction

*Flow Field Painter* is an interactive generative project that was built around the idea of using simple rules to create complex motion. The system is based on a flow field generated with noise, which defines a direction at each point in space. Thousands of small particles move through this field, following those directions and leaving behind trails that, together and on a bigger scale, build up into larger patterns.

The project is meant to be explored through interaction. Instead of watching a fixed simulation, the user can actively influence and reshape the flow field with their mouse, changing how the particles move and respond. As the field shifts, the motion reorganizes itself in real time, creating behavior that feels fluid and responsive, similar to wind or moving currents. Even small interactions can noticeably change the overall structure of the system.

I was interested in flow fields because they sit at a nice intersection between math, simulation, and art/visual design. I enjoy working with generative systems where simple logic can lead to unexpected and visually rich results. By combining a noise-driven vector field with a particle system and persistent trails, my goal was to create something that feels both technical and artistic/visually appealing, but that also invites experimentation through direct interaction.

## 2. Inspiration & Context

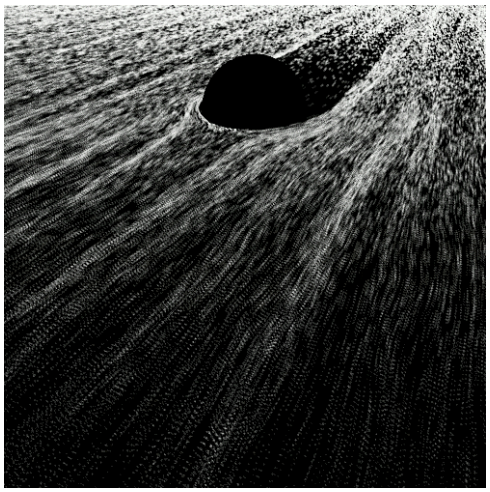
A lot of the inspiration for this project came from looking at natural flow patterns, especially rivers and tributaries. Some of my main references included tributaries and aerial images of rivers splitting and reconnecting, or water moving around rocks in a river. I personally feel that there is something really satisfying about how those shapes form. They feel organic and complex, but they're still being driven by very basic forces.

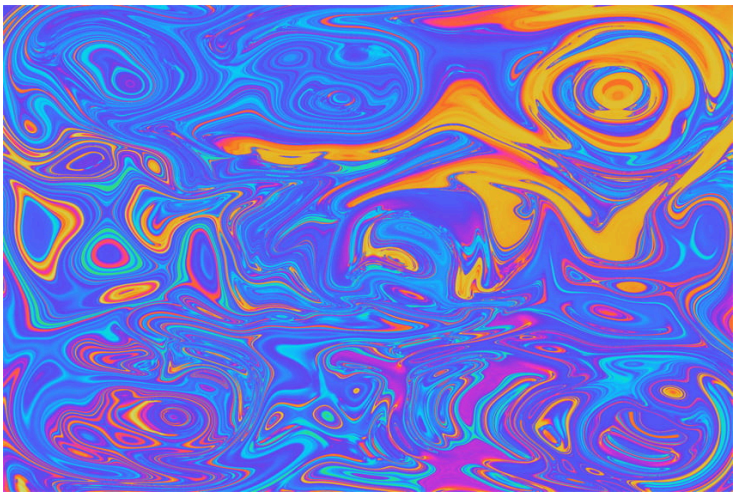
I was also inspired by things like smoke drifting through the air or fluid patterns where different materials swirl together. While there are no actual individual particles that are visible, this overall motion and structure that emerges over time is still very similar to what I was trying to create. I like how those movements leave behind traces and patterns.

From a creative coding perspective, this project fits into my interest in systems where simple rules can create behavior that feels rich and expressive. I'm drawn to projects where I can set up a structure and then explore it, rather than fully controlling the outcome. Flow fields felt like a

good way to do that. They offer a clear underlying logic, but still leave a lot of room for variation and discovery, especially when paired with interaction.

Below are some examples of these natural flow patterns, from rivers to smoke, even oil spills:



# 3. Project Overview & User Experience

*Flow Field Painter* is built around a simple core loop: a vector field defines direction at every point in space, and particles move through it, leaving behind trails that accumulate into larger visual patterns. The field itself is generated using noise: either Perlin noise or Curl noise, which the user can toggle between. Perlin noise produces smooth, flowing bands, while Curl noise creates more swirling and turbulent motion.

When the project starts, particles are randomly distributed across the screen and immediately begin following the field's flow. Their trails persist for several seconds and then fade gradually, which allows the motion to build up into dense, river-like structures over time. The visual style is intentionally minimal: a black background with white/gray particles and soft trails, creating a high-contrast and very clean look.

The main form of interaction is through brush-based "painting". By clicking and dragging on the canvas, the user can reshape the flow field in real time. There are four brush modes available: *Flow*, which redirects vectors in the direction of the stroke; *Swirl*, which creates circular motion around the brush; *Attract*, which pulls vectors inward toward the cursor; and *Repel*, which pushes them outward. These tools allow the user to sculpt the field like water flowing around obstacles or wind being redirected by barriers.

There is also a collapsible UI panel on the left side of the screen to provide access to all controls. Users can adjust parameters like brush size, flow speed, and turbulence using sliders, and can reset the field to regenerate a fresh noise pattern. A toggle enables sound reactivity, where microphone input influences the turbulence of the particles (louder sounds create more chaotic motion). This adds a layer of responsiveness that makes the system feel alive and reactive to its environment.
For visual output, the project also includes a screenshot feature. Pressing a key activates a frame overlay (a black border that frames the canvas) and exports the current frame as a PNG image. This helps present the work as a more finished, composed piece of art rather than just a screenshot of a simulation. Users can also toggle the visibility of the flow field gizmos, which display the underlying vector directions as small arrows. This is useful both for understanding how the system works and as a visual element in its own right.

Overall, the experience is meant to feel exploratory; the user is free to experiment, observe, and shape the motion however they want. The system responds pretty fluidly to input, and even small gestures can ripple outward and change the entire structure of the flow. It sits somewhere between a tool and art, but is also a toy in the sense that it invites play/interaction.

# 4. Methodology

## 4.1. Flow Field Generation

The flow field is a 2D grid where each cell stores a direction vector. I set the resolution to 128x128, which gave me enough detail to create smooth motion without being too

computationally heavy (also I didn’t really try to push or explore what the limits would be). The field covers the entire visible screen, and each cell's direction is calculated using noise.

I started with Perlin noise because it's straightforward and produces smooth, continuous values. For each cell, I sample the noise at that position, multiply it by 2*pi to get an angle, and convert that into a unit vector. The result is a field that feels more organic as there are no harsh transitions, just gradual changes in direction.

Later, I added curl noise as an option. Curl noise is derived from Perlin noise, but instead of using the noise value directly, you take its gradient and rotate it 90 degrees. This creates a divergence-free field, meaning the vectors naturally form swirls and loops instead of all converging or diverging. It took some time to understand the math, but once I got it working, it produced much more interesting motion that I was happy with.

### 4.2. Particle System

The particle system is not very complex in concept; each particle has a position and a velocity, and every frame it samples the flow field at its current location to figure out which way to go. The field direction gets added to the velocity, and then the particle moves.

I settled on around 1200 particles after some testing (again, didn’t fully push the system to see if adding a little more would cause performance issues). Spawning fewer than that made the trails feel very sparse. Each particle has a Trail Renderer attached, which draws a line behind it as it moves. The trails fade over time, which lets the motion build up into denser, more layered patterns.

When a particle leaves the screen, it wraps around or respawns at a random position. Getting this to work without visual glitches took a lot of iteration, especially since, if you just teleport the particle, the trail draws a line across the entire screen. I ended up pausing each particle briefly, letting its trail fade out, and then moving it. It's a small detail, but it made a big difference in how clean the final output looks.

### 4.3. Motion Variation

Early on, I ran into a problem: all the particles were following the exact same paths and converging into thin lines. It looked more like a circuit diagram than a river. I needed to break up that uniformity without losing the overall flow.

The first thing I added was turbulence: a small amount of Perlin noise added to each particle's velocity, unique to that particle. This gave them slight individual wobbles. Then I added spread, which pushes particles perpendicular to the flow direction, helping them fan out instead of all funneling into the same channels.

I also gave each particle its own speed multiplier and noise offset, so they don't all accelerate or jitter in sync. And I added a tiny bit of positional jitter, just a subtle random nudge each frame, to

keep things from feeling too locked in. Together, these forces create motion that follows the field but still feels organic and varied.

### 4.4. Rendering & Visual Output

The visual style came together through a lot of trial and error. I wanted something minimal with a clean black background, light trails, and no textures or complex shading. The trails do most of the visual work, so I focused on getting their length, width, and fade to feel right.

I used Unity's Trail Renderer with a simple unlit material. The trails start slightly thicker and taper off, and they fade in opacity over their lifetime. I kept adjusting the default trail length until the patterns felt dense enough without becoming muddy.

For the final output, I added a frame overlay (just a black border made of a few rectangle images I grouped together under on parent game object) that appears when you take a screenshot. It's a small touch, but I think it helps the piece feel more intentional and more artistic. The gizmos that show the flow field arrows also ended up being part of the aesthetic. I originally added them for debugging, but I liked them and people also responded well to seeing the structure behind the motion, so I kept them as a toggle.

## 5. Code Overview & Breakdown

The project consists of five scripts that work together. Each handles a specific part of the system.

### 5.1. FlowField.cs

This script manages the 2D grid of direction vectors. The main data is stored as a `Vector2[,]` array, along with bounds information so the field covers the camera view.

On `Start()`, `InitializeField()` creates the array, picks random noise offsets, calculates bounds from the camera, then fills the grid with either Perlin or Curl noise.

`GeneratePerlinField()` samples Unity's `Mathf.PerlinNoise` at each cell, converts the 0–1 value to an angle, then to a direction vector:

```
float noiseValue = Mathf.PerlinNoise(
    x * noiseScale + noiseOffsetX,
    y * noiseScale + noiseOffsetY
);
float angle = noiseValue * Mathf.PI * 2f;
field[x, y] = new Vector2(Mathf.Cos(angle), Mathf.Sin(angle));
```

`GenerateCurlField()` approximates the noise gradient using finite differences, then rotates it 90 degrees to get a divergence-free vector:

```
float n1 = Mathf.PerlinNoise(nx + eps, ny);
float n2 = Mathf.PerlinNoise(nx - eps, ny);
float n3 = Mathf.PerlinNoise(nx, ny + eps);
float n4 = Mathf.PerlinNoise(nx, ny - eps);

float dx = (n1 - n2) / (2f * eps);
float dy = (n3 - n4) / (2f * eps);

Vector2 curl = new Vector2(dy, -dx).normalized;
field[x, y] = curl;
```

`Sample()` converts a world position to grid coordinates and uses bilinear interpolation between the four nearest cells to get a smooth direction.

`ApplyBrush()` modifies cells within a radius, blending current directions toward a new direction. `ApplySwirlBrush()` sets vectors perpendicular to the offset from center (tangent to a circle). `ApplyAttractBrush()` points vectors toward or away from the center.

`OnDrawGizmos()` renders the field as cyan arrows in the Scene view (skipping every few cells for readability).

### 5.2. ParticleManager.cs

This script spawns and updates all particles.

Each particle is a `ParticleData` object with transform, trail, position, velocity, and per-particle variation values (speedMultiplier, noiseOffsetX, noiseOffsetY).

`SpawnParticles()` instantiates the prefab at random positions within bounds. Each particle gets random variation values so they don't behave identically.

The main update logic samples the flow field, adds forces, and moves particles:

```
Vector2 flowDirection = flowField.Sample(p.position);

if (flowNoise > 0f) {
    float fnx = Mathf.PerlinNoise(p.noiseOffsetX + time * 0.5f,
p.noiseOffsetY) - 0.5f;
    float fny = Mathf.PerlinNoise(p.noiseOffsetX, p.noiseOffsetY + time *
0.5f) - 0.5f;
    Vector2 flowJitter = new Vector2(fnx, fny) * flowNoise;
    flowDirection = (flowDirection + flowJitter).normalized;
}
```

```
float soundLevel = (soundReactor != null) ? soundReactor.GetVolume() : 0f;
float effectiveTurbulence = turbulence + (soundLevel * soundToTurbulence);
```

Turbulence and spread forces are calculated using per-particle Perlin noise, then all forces are applied to velocity. Damping and speed clamping keep things controlled.

`WrapPosition()` detects when particles leave bounds. To avoid trail streaks, `FadeAndMoveCoroutine()` pauses the particle, stops emission, waits for the trail to fade, then moves and resumes:

```
p.isPaused = true;
if (p.trail != null) {
    p.trail.emitting = false;
}

float waitTime = Mathf.Min(trailLength, fadeWaitCap);
yield return new WaitForSeconds(waitTime);

p.position = newPos;
p.transform.position = new Vector3(newPos.x, newPos.y, 0);

if (p.trail != null) {
    p.trail.Clear();
    p.trail.emitting = true;
}
p.isPaused = false;
```

### 5.3. BrushController.cs

This script handles mouse input for painting on the flow field.

`Update()` reads the mouse position using the new Input System, converts to world coordinates, and checks if the pointer is over UI:

```
Mouse mouse = Mouse.current;
if (mouse == null)
    return;

Vector3 mouseScreenPos = mouse.position.ReadValue();
mouseScreenPos.z = Mathf.Abs(mainCamera.transform.position.z);
Vector3 worldPos = mainCamera.ScreenToWorldPoint(mouseScreenPos);
currentMouseWorldPos = new Vector2(worldPos.x, worldPos.y);

bool isOverUI = EventSystem.current != null &&
EventSystem.current.IsPointerOverGameObject();
```

When dragging (and not over UI), `ApplyBrush()` calls the appropriate flow field method based on the current `BrushMode` enum (Flow, Swirl, Attract, or Repel).
`OnDrawGizmos()` draws an orange circle at the cursor to show brush radius.

### 5.4. UIController.cs

This script connects UI elements to game systems.

`Start()` wires up button listeners, slider listeners, and the dropdown. Each brush button calls `SetBrushMode()`, which updates the brush controller and highlights the button.

`Update()` checks for key presses:

```
if (keyboard[captureKey].wasPressedThisFrame) {
    StartCoroutine(CaptureScreenshotRoutine());
}
if (keyboard[overlayToggleKey].wasPressedThisFrame && frameOverlay != null) {
    frameOverlay.SetActive(!frameOverlay.activeSelf);
}
if (keyboard[gizmoToggleKey].wasPressedThisFrame && flowField != null) {
    flowField.showGizmos = !flowField.showGizmos;
    if (brushController != null) {
        brushController.showBrushGizmo = flowField.showGizmos;
    }
}
```

`CaptureScreenshotRoutine()` shows the frame overlay, hides debug UI, waits a frame, captures with `ScreenCapture.CaptureScreenshot()`, then restores the original state.
`TogglePanel()` shows/hides the UI panel and syncs the sound reactor's debug bar visibility.

### 5.5. SoundReactor.cs

This script captures microphone input and provides volume for other systems.

`InitializeMicrophone()` checks for available devices and starts recording with `Microphone.Start()`.

`Update()` reads samples from the mic clip and calculates volume:

```
micClip.GetData(sampleBuffer, readPosition);

float sum = 0f;
for (int i = 0; i < sampleBuffer.Length; i++) {
```

```
    sum += sampleBuffer[i] * sampleBuffer[i];
}
float rms = Mathf.Sqrt(sum / sampleBuffer.Length);

targetVolume = Mathf.Clamp01(rms * sensitivity);
currentVolume = Mathf.Lerp(currentVolume, targetVolume, smoothing);
```

`GetVolume()` returns the “smoothed” volume (0–1). `SetEnabled()` toggles the mic on/off at runtime.
`OnGUI()` draws a debug volume bar in the Game view showing current level and mic status.

## 6. Audience Response & Feedback

The audience response to the project was fortunately very positive. Overall, people enjoyed interacting with the system and experimenting with how the flow field responded to their input. A few people mentioned that it was satisfying to slowly shape the motion and watch the trails build up over time, especially as small changes started to affect the entire field.

One thing that stood out during the presentation was how useful the flow field gizmos were. Toggling the gizmos on and off helped people understand how the system worked, since they could clearly see the underlying vector field guiding the particles. At the same time, the gizmos were visually interesting on their own, and some people liked seeing both the structure and the final output layered together. The added frame around the piece was also well received, as it helped the project feel more complete and intentional as a visual experience.

The most helpful feedback I received was about the audio interaction. In its initial state, the system required very loud or sustained sound input to create a noticeable visual response. I even had to have people blow into my computer mic to really observe how sounds affected the particles. A few people pointed out that it might be best if the interaction were more sensitive, responding to normal speaking volume or shorter sound peaks instead of only continuous loud input. I completely agreed with this feedback and continued to adjust the audio settings and variables to try to make the interaction more responsive.

Another suggestion that came up was the idea of using more direct physical input, such as hand tracking with MediaPipe or even VR, so users could shape the flow field with their hands instead of a mouse. This was something I was already interested in, and I agree that it would be an extremely good fit for this project and feels like the natural next step. However, given the scope and time constraints I was dealing with, this was not something I was able to explore fully. It is absolutely something I would want to revisit if I continued developing the project.

## 7. What I Learned / What I Achieved

This project helped me understand flow fields much more deeply, especially how vector math can be used to control motion in a system. I had worked with noise before, but building this from the ground up made things click in a different way. Learning how curl noise works, and how it

can be built using Perlin noise, helped me understand why certain types of motion feel smooth and swirling instead of random.

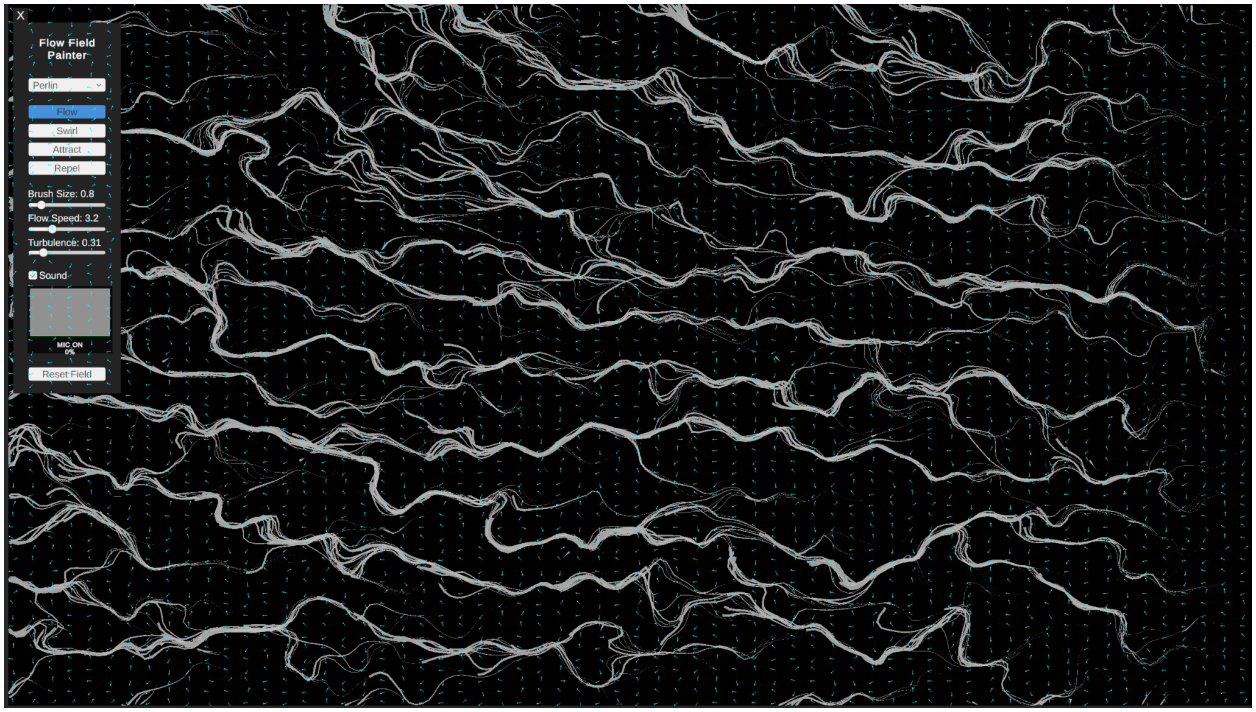
I also learned how valuable good visualization tools are when working with generative systems. Using gizmos in Unity to display the flow field made a huge difference, both for debugging and for understanding what was actually happening under the hood. Being able to see the vectors directly helped me reason through problems instead of guessing. It also ended up becoming a meaningful part of the final piece, not just a development tool.

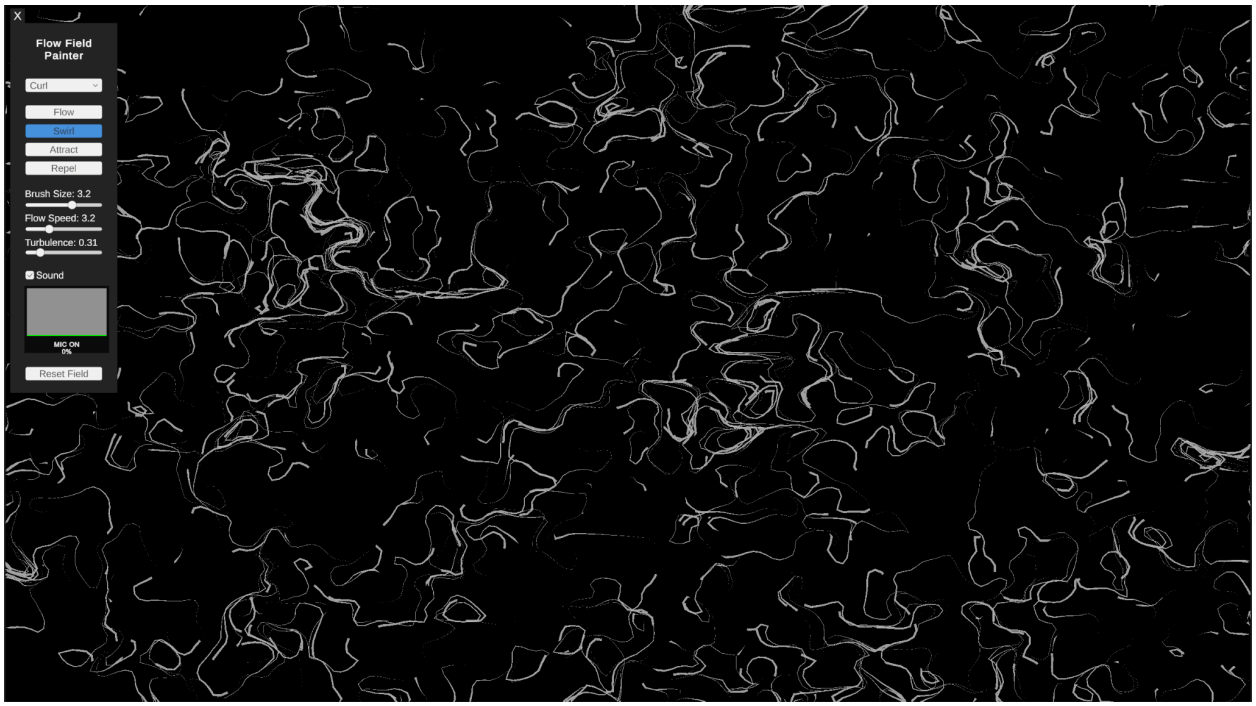
A big part of my workflow involved constant iteration through parameter changes. I made most of my variables public and adjustable in the Inspector, which basically let me build a small control panel for the system. Having sliders, headers, and clear labels made it much easier to experiment and stay organized. Being able to change values like noise scale, speed, damping, turbulence, and spread in real time was essential for finding the final look and behavior I wanted.

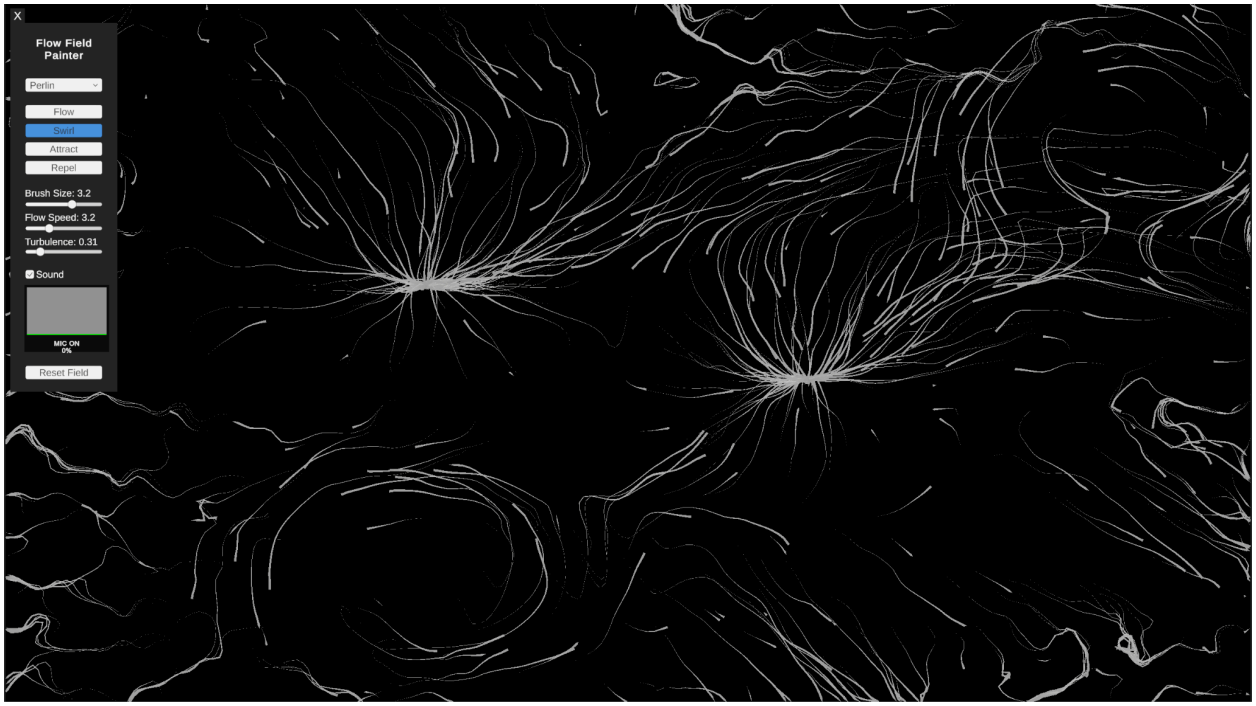
That said, I also learned how easy it is to overcomplicate a system. When the particles were not behaving the way I wanted, I often responded by adding more variables or forces, starting with spread and turbulence and then jitter and keeping adding more on until it made the system messier and harder to understand instead of easier. Clear labeling and real-time controls helped a lot, but I still had to step back and reproach by just playing with variables that actually mattered instead of implementing more patchy modifications and forces.

Overall, I feel like I achieved a system that reflects both the technical and exploratory sides of how I like to work. The project pushed me to think more carefully about structure, iteration, and interaction, and it helped me become more comfortable working with systems that are meant to be explored rather than fully controlled.

## 8. Representation Images


Flow Field Painter
Perlin
Flow
Swirl
Attract
Repel
Brush Size: 0.8
Flow Speed: 3.2
Turbulence: 0.31
Sound
MIC ON
0%
Reset Field


Flow Field Painter
Curl
Flow
Swirl
Attract
Repel
Brush Size: 3.2
Flow Speed: 3.2
Turbulence: 0.31
Sound
MIC ON
0%
Reset Field


X
Flow Field Painter
Perlin
Flow
Swirl
Attract
Repel
Brush Size: 3.2
Flow Speed: 3.2
Turbulence: 0.31
Sound
MIC ON
0%
Reset Field