The Eigen Valley Project

An Exploration of Linear Algebra in Computer Graphics

Comprehensive Technical Report

Sayali Kadam, 25M1110
Jaideep Biswas, 25M1114

November 27, 2025

Abstract

This report serves as a documentation for "Eigen Valley," a 3D landscape generated through mathematical functions and Linear Algebra concepts. Moving beyond traditional artistic modeling, this project utilizes Vector Fields, Gradient Descent, Cross Products, and Null Spaces to sculpt a living, breathing ecosystem.

The project demonstrates that a digital landscape is not merely a collection of pixels, but a visualization of a vector space $\mathbb{R}^3$ acted upon by linear operators. Through the implementation of a custom engine using WebGL (Three.js), we explore how the rigid axioms of mathematics can be manipulated to produce the organic chaos of nature.

This document provides a rigorous breakdown of the project, spanning from the theoretical underpinnings of gradient noise to the practical implementation of optimization algorithms for particle physics. It includes full code analysis, mathematical proofs for the lighting and terrain generation models, and a philosophical discussion on the intersection of mathematics and generative art.

Contents

1 The Artistic Vision: The Legend of Eigen Valley

1.1 The Origin of the Vector Space

In the beginning, there was only the Null Vector—a void of zero magnitude and undefined direction. The universe was an empty set $\emptyset$. Then, a Basis was chosen.

Eigen Valley is not a place discovered by explorers; it is a place calculated by axioms. It exists within a vector space $\mathbb{R}^3$, governed not by tectonic plates or erosion, but by the rigid laws of matrix transformations and scalar fields. In this valley, the rain does not fall because of gravity; it flows because it seeks to minimize the error of the terrain function, sliding down the gradient vectors toward stability (the local minimum).

1.2 The Philosophy of Math as Art

The central thesis of this artwork is that **Mathematics is the DNA of Nature**. When we look at a mountain range, we are seeing a fractal sum of noise functions. When we see a river, we are seeing a path of least resistance, a gradient descent optimization solving itself in real-time.

The river that cuts through the land is technically the "Null Space"—a region where the elevation vector is forced to zero, erasing all mountains that dare to cross its path. The trees are not biological entities; they are instanced geometries, rotated by affine transformations to face the sun vector.

Eigen Valley is a testament to the hidden order of the universe. It shows that within the chaos of random numbers, if one applies the correct Linear Operators, civilization and structure inevitably emerge.

1.3 Project Goals

The primary objectives of this project were:

1. To create a visually compelling 3D landscape using code only (no external assets).
2. To explicitly utilize concepts from the Linear Algebra curriculum (Vectors, Matrices, Dot/Cross Products, Null Spaces).
3. To build an interactive simulation where users can manipulate the mathematical constants defining the world.

2 Technical Foundation & Origins

This project stands on the shoulders of giants. While the specific ecosystem logic and the "Linear Algebra Visualization" features are our unique contributions, **the foundational rendering and noise algorithms are derived from established computer graphics research.**

2.1 The Base Concept: Gradient Noise (Perlin)

The core mathematical engine driving the terrain generation is Perlin Noise (specifically Gradient Noise), an algorithm developed by Ken Perlin in 1983.

2.1.1 The Gradient Solution

Perlin's solution was to define a grid of control points (lattice). At each grid intersection, we assign a random Gradient Vector ($\vec{g}$).

- For any point $P(x, y)$ inside a grid square, we calculate distance vectors to the four corners.
- We calculate the Dot Product between the distance vector and the gradient vector at each corner.
- We interpolate these four dot products to get the final height.

This ensures that $f(x, y)$ is smooth, resulting in rolling hills.

2.2 References & Frameworks

The project was built upon the Three.js graphics library, which handles the low-level rendering pipeline (rasterization, shader compilation, and the scene graph).

- **Algorithmic Source:** *Perlin, K. (1985). "An Image Synthesizer." ACM SIGGRAPH Computer Graphics, 19(3), 287-296.*
- **Implementation Guide (Sebastian Lague):** The "Procedural Landmass Generation" video series provided the conceptual framework for using "Octaves" (layering noise) to create jagged ridges vs. smooth hills.
- **Mathematical Visualization (The Coding Train):** Daniel Shiffman's tutorials on "Mesh Topology" helped us understand how to map a 2D array index (i, j) to a 3D vertex coordinate (x, y, z) effectively.

3 Development Journey & Challenges

The path to the final Eigen Valley was iterative, filled with mathematical errors and rendering artifacts.

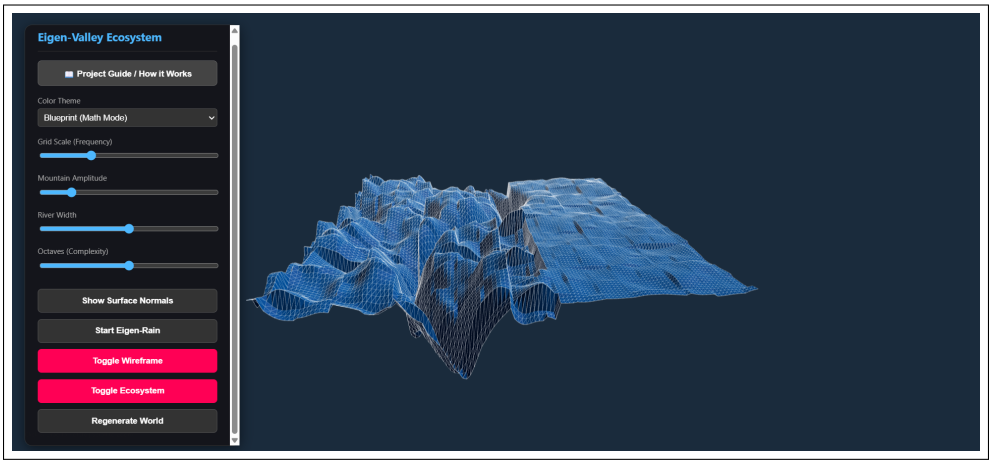

Figure 1: A view of the terrain in 'Blueprint' mode. This wireframe view reveals the discrete 120×120 grid points that form the basis of our vector space. Notice how the grid deforms vertically based on the noise function.

3.1 Phase 1: The Foggy Void

Our initial attempts resulted in a flat, grey mesh obscured by heavy fog.

- **The Issue:** The coordinate mapping was linear, but the noise frequency was too high. This resulted in "Aliasing"—the terrain looked like white noise because the sampling rate was lower than the Nyquist frequency of the noise function.
- **The Fix:** We implemented a 'scale' parameter to divide the input coordinates $(x/scale)$, effectively zooming in on the noise function to reveal the smooth gradients.

3.2 Phase 2: The Failed City

We attempted to force a city onto the terrain by "flattening" the noise where a road should be.

- **The Approach:** We used a parametric equation for a road: $\vec{r}(t) = [\sin(t), 0, t]$. We then checked every terrain point; if distance to $\vec{r}(t)$ was small, we set height to 0.
- **The Failure:** This created unnatural, vertical cliffs bordering the road. It looked like a glitch rather than a construction.

- **The Pivot:** We realized that instead of forcing the terrain to fit the city, we should place the city where the terrain allows. This led to the Gradient Kernel system (Section 5).

3.3 Phase 3: The Ecosystem & Null Space

The final challenge was placing trees. A random scatter algorithm placed trees underwater and on 80-degree slopes.

- **The Solution:** We treated the object placement as a Conditional Probability Distribution based on two variables: Height (h) and Slope ($||\nabla h||$).
- We also introduced the River Null Space to guarantee a water feature, regardless of the random seed.

4 Code Architecture: How It Works

The application is built as a single-file HTML/JS simulation using ES6 modules.

4.1 The "Two Planes" Architecture

The visualization consists of two distinct geometric planes occupying the same vector space. This separation allows us to handle reflection and refraction differently.

1. **The Terrain Plane:** A highly overlapped mesh (120×120 segments, totaling 14,400 vertices). The vertices of this plane are displaced vertically ($\vec{y}$) by the noise function $f(x, z)$.
2. **The Water Plane:** A simple, flat geometric primitive positioned at $y = \text{waterLevel}$.

Visual Intersection: The "River" is not an object; it is the visual intersection of these two subspaces. When the Terrain function outputs a height $h < \text{waterLevel}$, the Terrain mesh physically clips *underneath* the Water mesh. This signifies that the domain of the terrain function has dipped into the "negative" space relative to the water threshold.

4.2 Generating the Mesh (Code Analysis)

The following snippet demonstrates how we iterate through the vertices and apply our Linear Algebra functions.

```
for (let i = 0; i < positions.length; i += 3) {
    const x = positions[i];
    const z = positions[i + 1]; // Plane Y is World Z

    // 1. Calculate Noise (Scalar Field)
```

```
    // H = Sum( Amplitude * Noise( Frequency * Position ) )
    const noiseVal = calculateHeight(x, z, scale, amp, riverWidth,
   octaves);

    // 2. Apply to Y-coordinate (Vertex Displacement)
    positions[i + 2] = noiseVal;

    // 3. Calculate Color based on Height
    // Linearly interpolate (Lerp) between Ground Color and Snow Color
    const c = getColorForHeight(noiseVal, themeName, slope);
    colors.push(c.r, c.g, c.b);
}
```

Listing 1: Terrain Generation Loop

4.3 The Bent Null Space (The River Function)

In standard Linear Algebra, a Null Space (Kernel) is a linear subspace (a flat line or plane through the origin). However, in our creative application, we define the "Null Space" as the set of vectors $\vec{v}$ that satisfy a non-linear equation.

We define a "River Path" function:

$$Path(z) = A\sin(f \cdot z)$$

We then apply a depression function to the terrain height H:

```
// River Function (Null Space)
const riverPathX = Math.sin(z * 0.05) * 20 - 10;
const distToRiver = Math.abs(x - riverPathX);

// If within the river domain (Kernel)
if (distToRiver < riverWidth) {
    // Force the height towards zero using an inverted Gaussian curve
    const riverDepth = 15 * Math.exp(-(distToRiver * distToRiver) / (
   riverWidth * 2));
    noiseVal -= riverDepth;
}
```

Listing 2: The Null Space Logic

This logic "bends" the null space. We force the terrain height to zero along a sinusoidal curve, guaranteeing a riverbed exists.

5 Linear Algebra Applications

This project is effectively a visual calculator for Linear Algebra. Below are the specific concepts used, their mathematical definitions, and their application in the code.

5.1 Vector Fields (The Terrain)

The terrain is generated by a Vector Field.

- **Definition:** A vector field assigns a vector to every point in a subset of Euclidean space.
- **Application:** The Perlin Noise algorithm assigns a gradient vector $\vec{g}$ to every lattice point on the grid.
- **Operation:** The height at any point is the Dot Product (Inner Product) between the distance vector $\vec{d}$ and the gradient vector $\vec{g}$:

$$h(x, y) = \vec{d} \cdot \vec{g} = ||\vec{d}|| \, ||\vec{g}|| \cos\theta$$

This scalar projection creates the smooth slopes. If the vectors align ($\theta = 0$), the height is maximal. If they are orthogonal ($\theta = 90°$), the height is zero.

5.2 Linear Combinations (Octaves)

Realistic terrain is fractally complex. We achieve this using Linear Combinations.

- **Definition:** A linear combination is an expression constructed from a set of terms by multiplying each term by a constant and adding the results.
- **Application:** We sum multiple noise functions (layers), called octaves.

$$H(\vec{x}) = c_1 N(f_1 \vec{x}) + c_2 N(f_2 \vec{x}) + \cdots + c_n N(f_n \vec{x})$$

  where c_n is the amplitude (weight) and f_n is the frequency.

- **Code Implementation:**

```
    for (let o = 0; o < octaves; o++) {
        noiseVal += noise(x * freq, z * freq) * amp;
        freq *= 2;  // Double the frequency
        amp *= 0.5; // Halve the amplitude
    }

```

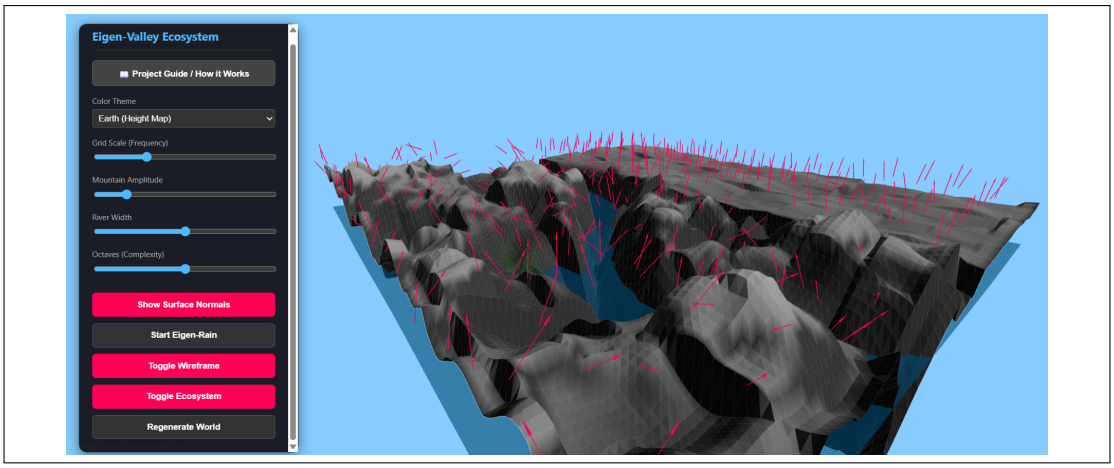

Figure 2: Visualizing the Cross Product. At every grid point, the pink arrow represents the Normal Vector $\vec{n}$. This vector is essential for lighting calculations, as it determines the angle of incidence for light rays.

5.3 The Cross Product (Surface Normals)

To shade the mountains, the rendering engine must know the "Normal" vector ($\vec{n}$), which is orthogonal to the surface at every point. We calculate this using the Cross Product.

- **Definition:** The cross product of two vectors $\vec{a}$ and $\vec{b}$ is a vector $\vec{c}$ that is perpendicular to both.
- **Derivation:** We approximate the tangent vectors at a point (x, z) by checking the height of neighbors (Finite Difference Method).

$$\vec{t}_x = \begin{bmatrix} \Delta x \\ \frac{\partial h}{\partial x} \\ 0 \end{bmatrix}, \quad \vec{t}_z = \begin{bmatrix} 0 \\ \frac{\partial h}{\partial z} \\ \Delta z \end{bmatrix}$$

- **Calculation:**

$$\vec{n} = \vec{t}_z \times \vec{t}_x = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 0 & \partial_z h & \Delta z \\ \Delta x & \partial_x h & 0 \end{vmatrix}$$

- **Application:** In the "Show Normals" mode (Figure 2), we visualize $\vec{n}$ as pink arrows. This vector $\vec{n}$ is then used in the lighting equation: Brightness $= \vec{n} \cdot \vec{L}$ (Dot product with light direction).

5.4 Gradient Descent (Eigen-Rain)

The "Eigen-Rain" (Figure 3) is a direct visualization of Gradient Descent, an optimization algorithm.

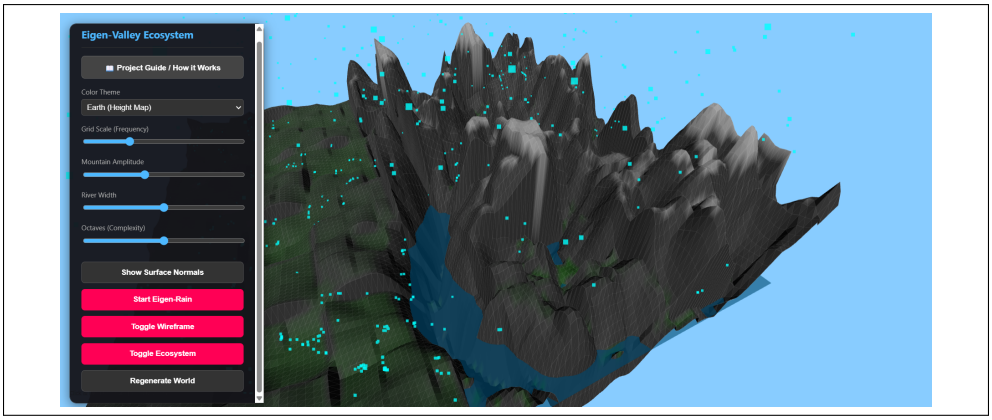

Figure 3: Gradient Descent in action. The glowing blue particles do not fall by gravity alone; they calculate the local gradient ∇h and move in the direction of steepest descent, naturally finding the river (the global minimum).

- **The Problem:** A raindrop wants to find the lowest energy state (the river). It doesn't know where the river is; it only knows the local slope.
- **The Math:** We calculate the Gradient Vector ∇h at the particle's position $\vec{p}$. The gradient points in the direction of steepest ascent.

$$\nabla h = \begin{bmatrix} \frac{\partial h}{\partial x} \\ \frac{\partial h}{\partial z} \end{bmatrix}$$

- **The Update Rule:** To move downhill, we subtract the gradient:

$$\vec{p}_{new} = \vec{p}_{old} - \alpha \nabla h$$

  where α is the "learning rate" (speed).

- **Code Implementation:**

```
    // Calculate Partial Derivatives (Slope)
    const dHdX = (heightRight - heightLeft) / delta;
    const dHdZ = (heightUp - heightDown) / delta;

    // Update Position (Gradient Descent)
    particle.x -= learningRate * dHdX;
    particle.z -= learningRate * dHdZ;

```

5.5 Subspace Projection (Asymmetry)

The terrain is asymmetric (mountains on one side, plains on the other) because we applied a Projection Operator.

- **The Logic:** We defined a subspace condition ($x > 10$).
- **The Operation:** Inside this subspace, we apply scalar multiplication to the height vector.

$$h_{new} = 0.2 \cdot h_{old}$$

- **Result:** This effectively projects the high-magnitude mountain vectors onto a lower subspace (a plain), flattening the right side of the map to allow for civilization.

6 Procedural Logic & Aesthetics

Beyond raw terrain generation, the project employs Linear Algebra to define the "look and feel" of the world.

6.1 Color Themes via Vector Interpolation

The colors of the terrain are not textures; they are vectors calculated dynamically.

- We define a "Bottom Vector" $\vec{C}_{bot}$ (e.g., Dark Green) and a "Top Vector" $\vec{C}_{top}$ (e.g., White Snow).
- For any vertex with height h, we perform Linear Interpolation (LERP):

$$\vec{C}_{final} = (1-t)\vec{C}_{bot} + t\vec{C}_{top}$$

  where $t = \frac{h-h_{min}}{h_{max}-h_{min}}$.
- This ensures the color gradient is mathematically smooth across the surface.

6.2 Sun Rotation

The sun is modeled as a Directional Vector $\vec{L}$ located at infinity. To simulate day and night, we rotate this vector using a time parameter t.

$$\vec{L}(t) = \begin{bmatrix} \cos(t) \\ \sin(t) \\ 0 \end{bmatrix}$$

As t increases, the x and y components oscillate, causing shadows to lengthen and rotate across the terrain.

6.3 Flora & Architecture Generation

The houses and trees are not manually placed. They are generated using a Probability Density Function derived from the terrain's properties.

1. **Grid Scan:** We iterate through every possible coordinate (x, z).
2. **Gradient Check:** We calculate the magnitude of the gradient (slope) $||\nabla h||$.
3. **Conditional Instantiation:**
   - **Trees:** Spawn if $||\nabla h|| < 0.8$ (Moderate Slope) AND $h >$ waterLevel.
   - **Houses:** Spawn if $||\nabla h|| < 0.4$ (Flat Ground) AND $h >$ waterLevel.
   - **Rocks:** Spawn if $||\nabla h|| > 0.6$ (Steep Cliffs).

This ensures that our ecosystem adheres to physical constraints without manual intervention.

7 Lighting Physics

7.1 Specular vs. Diffuse Reflection

The visual difference between the water (shiny) and the ground (matte) is defined by how they treat the light vector.

1. **Terrain (Diffuse/Lambertian):** The ground scatters light in all directions. The brightness is purely a function of the angle between the Surface Normal $\vec{N}$ and the Light Vector $\vec{L}$.
$$I_{diffuse} = \max(0, \vec{N} \cdot \vec{L})$$
2. **Water (Specular):** The water reflects light in a specific direction. We calculate the Reflection Vector $\vec{R}$:
$$\vec{R} = 2(\vec{N} \cdot \vec{L})\vec{N} - \vec{L}$$
If the camera vector $\vec{V}$ is aligned with $\vec{R}$ (high dot product), we see a bright specular highlight (the sun's reflection).

8 Conclusion

The Eigen Valley project demonstrates that Linear Algebra is not merely a tool for solving systems of equations, but a language for creation.

8.1 Creativity in Mathematics

By treating the computer screen as a vector space, we were able to sculpt mountains using scalar multiplication, carve rivers using null spaces, and simulate weather using gradient descent. The project proves that the rigid rules of mathematics can produce the organic chaos of nature. It challenges the notion that math is "rigid" and art is "fluid"—here, fluidity is defined by the Gradient vector.

8.2 Future Improvements

To further extend the Linear Algebra concepts, we could implement:

- **Eigenvector Ridge Detection:** We currently use $|noise|$ for ridges. A more advanced method would be calculating the Hessian Matrix (matrix of second derivatives) at each point. The Eigenvectors of the Hessian would point exactly along the principal curvature (the spine of the mountain), allowing us to place trees exactly on ridge lines.
- **Linear Transformation Monoliths:** Floating zones that apply Shear or Rotation matrices to any object that passes beneath them, visualizing the distortion of space.

References

[1] Lague, S. (2016). *Procedural Landmass Generation [Video Series]*. YouTube. Retrieved from https://www.youtube.com/user/Cercopithecan. (Source for Octave/Layering logic).

[2] Shiffman, D. (2016). *Coding Challenge #11: 3D Terrain Generation with Perlin Noise*. The Coding Train. (Source for Mesh Topology and Vertex mapping).

[3] Three.js Authors. (2024). *Three.js Documentation*. https://threejs.org/docs/ (Reference for the WebGL rendering pipeline).

[4] Perlin, K. (1985). *An Image Synthesizer*. ACM SIGGRAPH Computer Graphics, 19(3), 287-296. (The foundational paper on Gradient Noise).