A Linear Algebra Approach to Procedural Terrain Generation And Texturing

Benedict Presley - 13523067
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13523067@std.stei.itb.ac.id, presleybenedict04@gmail.com

***Abstract*—Procedural terrain generation is a pivotal technique in computer graphics and interactive applications, enabling the creation of expansive, natural-looking environments without the overhead of manual modeling. This paper presents a linear algebra perspective on procedural terrain generation, focusing particularly on Perlin Noise and its extension to fractal Brownian motion. To yield multi-scale detail, multiple octaves of noise are summed according to user-defined lacunarity and persistence, capturing both large landmasses and fine-grained surface features. The noise values are subsequently normalized and mapped to terrain types—such as oceans, beaches, and mountains—through threshold-based color assignments. An implementation is provided that demonstrates dynamic map expansion and user-controlled scrolling, illustrating how these concepts integrate into a scalable, interactive system. Ultimately, this paper aims to unify key mathematical and computational concepts for generating and visualizing procedural terrains that are both efficient to compute and plausible in appearance.**

***Keywords*—Fractal Brownian Motion, Linear Algebra, Perlin Noise, Procedural Terrain Generation**

I. Introduction

Procedural terrain generation has become an essential aspect of various applications, ranging from video game design to geographic simulations and virtual reality environments. This technique allows developers to create complex, realistic, and diverse landscapes algorithmically, while avoiding the time-intensive process of manual design. Traditional procedural terrain generation often relies on heuristic methods such as Perlin noise, this paper introduces a modified approach that integrates linear algebra to enhance the traditional noise-based generation methods.

Perlin noise is a gradient-based noise function commonly used in procedural generation to produce natural-looking textures and terrain. It generates smooth, continuous patterns by interpolating random gradient vectors across a grid. In terrain generation, Perlin noise is often used to define elevation values at each point on a 2D grid, creating hills, valleys, and other natural features. This paper explores a method for calculating noise using linear algebraic concepts, including the Hadamard product, to introduce greater flexibility and control over the generated terrain.

This paper presents an algorithm for terrain generation that reimagines the Perlin noise calculation through linear algebraic techniques. Unlike traditional methods that require fixed-size grid chunks, this algorithm allows for the calculation of terrain chunks of various sizes, offering enhanced flexibility in managing terrain scalability and resolution. To texture the generated terrain, the algorithm utilizes a color distribution map, where each noise value is mapped to a specific color based on predefined thresholds. The integration of a flexible color distribution map also allows for dynamic customization of terrain aesthetics. This adaptability makes the algorithm suitable for a wide range of artistic and functional requirements.

Additionally, the algorithm supports seamless transitions between terrain chunks, ensuring that generated landscapes appear continuous and coherent. This feature is particularly valuable for applications requiring large, interconnected worlds, such as open-world games or virtual simulations.

II. Literature Review

A. Vectors

A vector in the context of linear algebra is a quantity that has both magnitude and direction that can be represented as an ordered list of numbers (often called components). Vectors can exist in various dimensions.

for example, in a 2D plane, a vector v can be written as $\begin{bmatrix} v_x \\ v_y \end{bmatrix}$. In a 3D plane, a vector v and be written as $\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$.

Key properties of vectors include:

1) Addition: Two vectors of the same dimension can be added component-wise. For example, given vectors u and v

$$u \pm v = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \pm \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Note that this operation is commutative and associative.

2) Scalar Multiplication: A vector can be scaled by a scalar (a real number), multiplying each component by that scalar. For example, given a scalar c and a vector v

$$cv = \begin{bmatrix} cv_1 \\ cv_2 \\ \vdots \\ cv_n \end{bmatrix}$$

For two scalars c and d and two vectors u and v, the following applies

$$c(u+v) = cu + cv$$
$$(c+d)u = cu + du$$
$$(cd)u = c(du)$$

3) Magnitude (Norm): The length or magnitude of a vector v is given by

$$||v|| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}$$

4) Direction: The direction of a vector is the orientation in space and is often used in geometry and physics to indicate an object's orientation of movement or force application.

B. Dot Product

The dot product (also known as the scalar product) of two vectors a and b is defined as:

$$a \cdot b = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

This operation results in a single scalar value. The dot product between two vectors measures how much they align or project onto each other. It is related to the cosine of the angle θ:

$$a \cdot b = ||a||||b|| \cos(\theta)$$

The dot product is essential in gradient-based noise functions such as Perlin noise, where the contribution from gradient vectors is computed via dot products with displacement vectors.

C. Matrices

A matrix is a rectangular array of values arranged in rows and columns. An $m \times n$ matrix has m rows and n columns. Matrices are powerful tools in linear transformations and can represent operations such as rotation, scaling, or shear in multiple dimensions.

Key properties of matrices include:

1) Addition: Two matrices of the same dimensions can be added component-wise. For example, given two matrices A and B

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1n} \\ B_{21} & B_{22} & \dots & B_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{n1} & B_{n2} & \dots & B_{nn} \end{bmatrix}$$
$$= \begin{bmatrix} A_{11} + B_{11} & A_{12} + B_{12} & \dots & A_{1n} + B_{1n} \\ A_{21} + B_{21} & A_{22} + B_{22} & \dots & A_{2n} + B_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} + B_{n1} & A_{n2} + B_{n2} & \dots & A_{nn} + B_{nn} \end{bmatrix}$$

2) Scalar Multiplication: Multiplying a matrix by a scalar multiplies each entry in the matrix by that scalar. For example, given a scalar c and a matrix A

$$c\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix} = \begin{bmatrix} cA_{11} & cA_{12} & \dots & cA_{1n} \\ cA_{21} & cA_{22} & \dots & cA_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ cA_{n1} & cA_{n2} & \dots & cA_{nn} \end{bmatrix}$$

3) Matrix Multiplication: If A if an $m \times p$ matrix and B is a $p \times n$ matrix, then their resulting multiplication C is an $m \times n$ matrix. C is given by

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj}$$

4) Transpose: The transpose of a matrix A is formed by flipping it over its diagonal, turning rows into columns and vice versa. The transpose of a matrix A is written as A^T.

D. Hadamard Product

The Hadamard product (also known as the element-wise product) of two matrices A and B of the same dimension is given by multiplying corresponding entries of A and B together.

$$\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix} \circ \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1n} \\ B_{21} & B_{22} & \dots & B_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{n1} & B_{n2} & \dots & B_{nn} \end{bmatrix}$$
$$= \begin{bmatrix} A_{11}B_{11} & A_{12}B_{12} & \dots & A_{1n}B_{1n} \\ A_{21}B_{21} & A_{22}B_{22} & \dots & A_{2n}B_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1}B_{n1} & A_{n2}B_{n2} & \dots & A_{nn}B_{nn} \end{bmatrix}$$

The Hadamard product is used to apply element-wise scaling. In certain noise algorithms, particularly when working with masks or blending different terrain layers, element-wise operations can be applied to combine features pixel-by-pixel or sample-by-sample.

Key properties of Hadamard product include:

1) Commutative: $A \circ B = B \circ A$
2) Associative: $(A \circ B) \circ C = A \circ (B \circ C)$
3) Distributive over addition: $A \circ (B + C) = A \circ B + A \circ C$

E. Perlin Noise

Perlin noise is a gradient-based noise function introduced by Ken Perlin. It is widely used in computer graphics and procedural generation for creating natural-looking textures and terrains. The core idea is to define a pseudo-random gradient at each lattice (grid) point and blend these gradients smoothly to generate coherent noise values across the space.

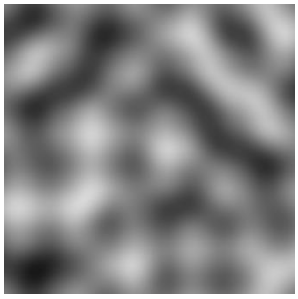

Figure 1. Example of perlin noise map [Source: https://rtouti.github.io/graphics/perlin-noise-algorithm]

Perlin noise is generated in the following steps:

1. Gradient Vector Generation

   The generation of 2D Perlin noise begins by establishing a grid or lattice structure over the 2D plane, with each intersection of grid lines corresponding to a lattice point. For any given position (x, y) within this plane, the surrounding cell, defined by the integer

coordinates of its four corner lattice points, is identified. These lattice points serve as reference anchors for calculating the noise value at the given position.

At each lattice point, a gradient vector is assigned. These gradient vectors may either be randomly generated and normalized to a fixed length or selected from a predefined set of constant directions, such as vectors uniformly distributed at 45° intervals in 2D. These gradients represent directional influences that modulate the contribution of each lattice point to the noise value at a given position.

The role of gradient vectors is to facilitate a localized and continuous influence on the noise field. For each corner of the cell, a displacement vector is computed, representing the difference between the position (x, y) and the coordinates of the lattice point. The dot product of the gradient vector and the displacement vector is then calculated, yielding a scalar value. This scalar value quantifies the degree of alignment between the displacement and the gradient direction, determining the extent of the lattice point's contribution to the noise value at the position.

2. Permutation Array

Permutation array in Perlin noise is a data structure that facilitates the mapping of integer lattice coordinates to pseudo-random gradients. The permutation array is a shuffled list of integers, typically ranging from 0 to 255 in the classic implementation. (Note that the size of permutation array doesn't have to be 256). To ensure consistency and wrap-around behavior, this array is duplicated, resulting in a structure of length 512, where the first 256 integers are repeated.

The primary purpose of the permutation array is to provide a deterministic yet pseudo-random mapping from grid points to gradient vectors. When computing Perlin noise at a given coordinate (x, y), the integer parts of x and y are converted into lattice indices. This lookup determines which gradient vector is associated with each lattice point in the cell surrounding the input coordinate. Importantly, the permutation array ensures that the same lattice point always maps to the same gradient vector, guaranteeing consistency across the noise field.

The generation of the permutation array begins with a list of sequential integers (e.g., 0 to 255). This list is shuffled using a pseudorandom number generator to introduce randomness. After shuffling, the list is appended to itself, creating a structure that inherently supports wrap-around behavior.

3. Contribution Calculation

In the Perlin noise algorithm, constant vectors are used at lattice points to ensure smooth transitions across the noise field. These constant vectors are typically chosen from a finite set of predefined directions. This approach avoids the need to generate fully random gradient vectors at runtime, which would require computationally expensive normalization. Instead, the predefined set ensures a consistent and uniform distribution of gradients, contributing to the smoothness and continuity of the noise.

The interaction between the constant gradients and the input point is realized through the dot product operation. At each lattice point, the gradient vector represents the local orientation or slope. For a given input position (x, y) within a cell, a displacement vector is computed from each lattice point to the input point, capturing the relative position and direction. The dot product between the gradient vector and the displacement vector yields a scalar value that quantifies how strongly the displacement aligns with the gradient. This scalar value represents the contribution of that lattice point to the overall noise value at (x, y).

The dot product mechanism ensures that the influence of each lattice point is determined not only by its proximity to the input point but also by the alignment of its gradient with the displacement. As a result, the contributions from all four corners of the cell vary smoothly, depending on the position within the cell.

4. Interpolation

To compute the final Perlin noise value at a given point (x, y), the algorithm first determines the cell in which the point lies and identifies the four lattice corners that bound it. For each corner, the displacement vector from the corner to the point is calculated, and the dot product of this displacement vector with the gradient vector at the corner is computed. These dot products, denoted as n_0, n_1, n_2, n_3 in 2D, represent the contribution of each corner to the noise value at the input point.

The next step involves blending these contributions using interpolation. Linear interpolation (lerp) is used to blend two values, a and b, based on a parameter t, as defined by the formula:

$$lerp(a, b, t) = a + t(b - a)$$

In 2D, this process is extended to bilinear interpolation. First, horizontal interpolation is performed between the contributions of the bottom two corners and the top two corners. Then, vertical interpolation is applied between these results to compute the final noise value.

To ensure smooth transitions between cells, the algorithm modifies the interpolation parameter using a fade function. The fade function, typically

$$f(t) = 6t^5 - 15t^4 + 10t^3$$

transforms the fractional component of the input coordinate into smoothed values. This function satisfies the conditions $f(0) = 0$, $f(1) = 1$, and has zero derivatives at both 0 and 1. By ensuring that the slope of the function smoothly transitions to zero at the boundaries, the fade function eliminates discontinuities and sharp transitions.

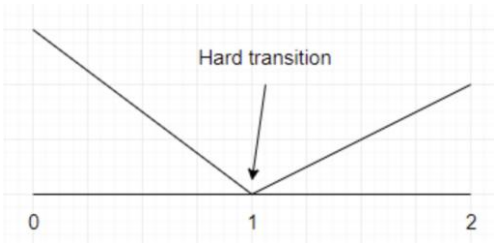

Figure 2. Abrupt transition due to linear interpolation [Source: https://rtouti.github.io/graphics/perlin-noise-algorithm]

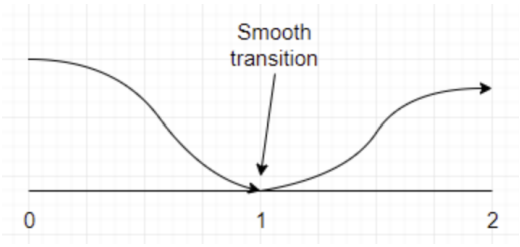

Figure 3. Smoother transition by using fade function [Source: https://rtouti.github.io/graphics/perlin-noise-algorithm]

The final noise value is computed by combining the gradient-based contributions of the four corners through this two-stage interpolation process, with the fade function applied as the smoothing factor. This combination of gradient-based contributions, fade smoothing, and interpolation ensures that the noise transitions smoothly across cells, producing visually pleasing and continuous patterns.

F. *Fractal Brownian Motion (fBm)*

Fractal Brownian Motion (fBm) is a technique used to combine multiple layers (or octaves) of noise, such as Perlin noise, to create complex and natural-looking textures or terrains. Each octave introduces finer details by applying noise at progressively higher frequencies and lower amplitudes. This method enhances the realism of the generated output.

The fBm function is defined as the sum of multiple scaled octaves of noise. Mathematically, it can be expressed as:

$$fBm(x,y) = \sum_{i=0}^{N-1} a_k PerlinNoise(\lambda^k x, \lambda^k y)$$

where

- N is the number of octaves,
- $a_k = p^k$ is the amplitude of the k-th octave, determined by the persistence p, where $0 < p < 1$,
- λ is the lacunarity, which controls the frequency scaling between octaves,
- $PerlinNoise(a, b)$ is the Perlin noise function evaluated at (a, b).

The parameter λ (lacunarity) dictates how the frequency increases between octaves. Each octave's frequency is scaled by a factor of λ. A higher lacunarity ($\lambda > 1$) results in more rapid oscillations at higher octaves, introducing finer details to the noise. Conversely, a lower lacunarity results in slower transitions and less intricate details.

The parameter p (persistence) controls how the amplitude decreases between octaves. Each subsequent octave has an amplitude scaled by p. A higher persistence retains more strength in higher frequency octaves, creating rough or turbulent terrains. A lower persistence diminishes the higher frequency contributions, resulting in smoother terrain.

G. *Min-Max Normalization*

Min-max normalization scales the values of a dataset to a predefined range, often $[0, 1]$ or $[-1, 1]$. This ensures that the minimum value maps to the lower bound of the range and the maximum value maps to the upper bound. All intermediate values are proportionally scaled.

For a value x, the normalized value x' (scales to $[0, 1]$) is computed as:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

To scale to a different range $[a, b]$, the formula becomes:

$$x' = a + \frac{x - x_{min}(b - a)}{x_{max} - x_{min}}$$

By applying the same normalization parameters across chunks (using a global x_{min} and x_{max}), adjacent chunks will have aligned value ranges, resulting in smooth transitions.

H. *Tanh Normalization*

Tanh normalization uses the hyperbolic tangent ($tanh$) function to scale data. It maps values to a range of $[-1, 1]$ with a non-linear curve, compressing extreme values more than intermediate ones. This ensures that the transitions are smooth while reducing the impact of outliers.

For a value x, the normalized value x' is computed as:

$$x' = \tanh(kx)$$

To scale to a different range $[a, b]$, the formula becomes:

$$x' = a + \frac{b - a}{2}(\tanh(kx) + 1)$$

Here, k is a scaling factor that controls the sharpness or sensitivity of the $tanh$ function.

III. IMPLEMENTATION

A. *Programming Language and Supporting Tools*

The programming language used to implement this project is Python. Python is chosen for its simplicity, readability, and the availability of powerful libraries that make it easy to handle complex mathematical calculations and create visualizations.

```
import numpy as np
import matplotlib.pyplot as plt
```

The libraries NumPy and Matplotlib are used in this project. NumPy is chosen for its ability to efficiently perform numerical operations and manage large arrays, which are important for generating and manipulating noise data. Matplotlib is used to create clear and detailed visualizations of the generated patterns, helping to analyze and refine the results.

B. Defining Terrain Types and Colors

```
WEIGHTS = [42, 38, 27, 18, 22, 20, 16, 24, 26]
WEIGHTS = np.array(WEIGHTS) / sum(WEIGHTS)
thresholds = np.cumsum(WEIGHTS)
COLORS = [
      # List of colors in RGB form
   ]

def upper_bound_threshold(height):
       left = 0
       right = len(thresholds) - 1
       pos = -1

       while left <= right:
           middle = (left + right) // 2
           if height < thresholds[middle]:
               pos = middle
               right = middle - 1
           else:
               left = middle + 1

       return pos
```

WEIGHTS is a list of numbers representing the relative proportions of different terrain types (e.g., deep ocean, beach, grassland, or mountains). These weights are normalized to produce probabilities that sum to 1. The cumulative sum of these probabilities, stored in thresholds, is then used to segment the noise values into distinct terrain categories based on their thresholds.

The COLORS list defines specific RGB values corresponding to each terrain category. These colors range from dark blue for deep ocean to browns for higher mountains, ensuring each terrain type is visually distinct and intuitive.

Because thresholds is an increasing array, binary search can be used to optimize color assignment

C. Permutation Array

```
permutation_size = 2**10

def shuffle(array_to_shuffle):
    np.random.shuffle(array_to_shuffle)

def make_permutation():
    P = np.arange(permutation_size, dtype=int)
    shuffle(P)
    return np.concatenate((P, P))

Perm = make_permutation()
```

Another component of Perlin noise is a permutation array. Here, permutation_size is chosen as 2^{10}, though it can be other powers of two or arbitrary sizes. The function make_permutation creates an array P of integers from 0 to 1023, then shuffles them in-place. After shuffling, the code concatenates P to itself, yielding a 2048-element array called Perm. This duplication allows simpler lookups later, because any index beyond 1023 just continues into the second copy of P, thereby simulating a wrap-around without having to explicitly do modular arithmetic each time.

D. Gradient Vectors and the Fade Function

```
def get_constant_vector(v):
    h = v & 3
    if h == 0:
        return np.array([1.0, 1.0])
    elif h == 1:
        return np.array([-1.0, 1.0])
    elif h == 2:
        return np.array([-1.0, -1.0])
    else:
        return np.array([1.0, -1.0])

def fade(t):
    return ((6 * t - 15) * t + 10) * t * t * t
```

To generate noise, the code requires a way to map the shuffled integers in the permutation array into actual gradient directions. The function get_constant_vector takes an integer v, applies a bitwise operation v & 3 (which is equivalent to taking the value v modulo 4) to reduce it to one of four possible values. These vectors act as gradient directions at each lattice point in the noise function. Meanwhile, the function fade defines a polynomial that smooths the interpolation parameter t so that transitions at cell boundaries do not produce visible seams.

E. Single-Scale Noise Calculation

```
   def calculate_map(row1, col1, row2, col2, lacunarity):
       global Perm, permutation_size

       row_length = row2 - row1 + 1
       col_length = col2 - col1 + 1

       TR = np.zeros(row_length * col_length)
       TL = np.zeros(row_length * col_length)
       BR = np.zeros(row_length * col_length)
       BL = np.zeros(row_length * col_length)
       U1 = np.zeros(row_length * col_length)
       V1 = np.zeros(row_length * col_length)

       for i in range(row_length):
           for j in range(col_length):
               curx = (col1 + j) * lacunarity
               cury = (row1 + i) * lacunarity

               x_wrapped    =    int(np.floor(curx))    &
(permutation_size - 1)
               y_wrapped    =    int(np.floor(cury))    &
(permutation_size - 1)

               x_floor = curx - np.floor(curx)
               y_floor = cury - np.floor(cury)

               TRV = np.array([x_floor - 1.0, y_floor -
1.0])
               TLV = np.array([x_floor,        y_floor -
1.0])
               BRV = np.array([x_floor - 1.0, y_floor])
               BLV = np.array([x_floor,       y_floor])

               TRCV                                    =
get_constant_vector(Perm[Perm[x_wrapped + 1] + y_wrapped
+ 1])
               TLCV                                    =
get_constant_vector(Perm[Perm[x_wrapped]     + y_wrapped
+ 1])
               BRCV                                    =
get_constant_vector(Perm[Perm[x_wrapped    +    1]    +
y_wrapped])
               BLCV                                    =
get_constant_vector(Perm[Perm[x_wrapped]               +
y_wrapped])

               TR[i * col_length + j] = np.dot(TRV, TRCV)
               TL[i * col_length + j] = np.dot(TLV, TLCV)
               BR[i * col_length + j] = np.dot(BRV, BRCV)
```

```
            BL[i * col_length + j] = np.dot(BLV, BLCV)

            U1[i * col_length + j] = fade(x_floor)
            V1[i * col_length + j] = fade(y_floor)

    U2 = 1 - U1
    V2 = 1 - V1

    alpha = (U2 * ((V2 * BL) + (V1 * TL))) + (U1 * ((V2
* BR) + (V1 * TR)))

    result = np.zeros((row_length, col_length))

    for i in range(row_length):
        for j in range(col_length):
            result[i, j] = alpha[i * col_length + j]

return result
```

The function calculate_map(row1, col1, row2, col2, lacunarity) generates a slice of noise values for a rectangular region from (row1,col1) to (row2,col2). To handle all points in this rectangular region at once, the code effectively "flattens" or compresses the two-dimensional area into a one-dimensional strip: each position (i, j) in the rectangle is mapped to a single index in arrays such as TR, TL, BR, BL, U1, and V1.

Within this function, TR, TL, BR, and BL store the dot-product contributions from the top-right, top-left, bottom-right, and bottom-left corners of each noise cell. Likewise, U1 and V1 store the fade-interpolation parameters along the horizontal (x-direction) and vertical (y-direction), respectively.

In order to calculate all the noise values at the same time, the following equation is used. The equation applies the interpolation to all values and the same time.

$$\alpha = U' \circ (V' \circ BL + V \circ TL) + U \circ (V' \circ BR + V \circ TR)$$

where

- α is a vector containing the resulting noise value,
- U is a vector containing the fade value of the x-component of each position,
- V is a vector containing the fade value of the y-component of each position,
- BL, TL, BR, and TR each store the dot-product contributions from the top-right, top-left, bottom-right, and bottom-left corners of each noise cell,
- $U' = \begin{bmatrix} 1-U_1 \\ 1-U_2 \\ \vdots \\ 1-U_n \end{bmatrix}, V' = \begin{bmatrix} 1-V_1 \\ 1-V_2 \\ \vdots \\ 1-V_n \end{bmatrix}$

After the calculation, the strip is converted back into matrix form.

F. Fractal Brownian Motion and Normalization

```
def apply_tanh_transform(value_array, k = TANH_K):
    return 0.5 * (np.tanh(TANH_K * (2.0 * value_array -
1.0)) + 1.0)

def calculate_map_with_fbm(row1, col1, row2, col2,
numOctaves):
    global global_minimum, global_maximum,
first_iteration

    persistence = 0.9
    lacunarity  = 0.015
    result = np.zeros((row2 - row1 + 1, col2 - col1 +
1))

    for _ in range(numOctaves):
        result = result + persistence *
calculate_map(row1, col1, row2, col2, lacunarity)

        persistence *= 0.5
        lacunarity  *= 2.0

    result = (result - global_minimum) / (global_maximum
- global_minimum)

    result = apply_tanh_transform(result)

    return result
```

To achieve more varied and natural-looking terrain, the code combines multiple octaves of noise in a function called calculate_map_with_fbm. This function takes parameters defining the rectangular region (row1, col1) to (row2, col2), as well as a number of octaves to generate. Here the equation used is change slightly than the standard fBm formula.

$$fBm(chunk) = \sum_{i=0}^{N-1} \frac{p}{2^i} calculate\_map(chunk, 2^i \lambda)$$

When the sum of these octaves is complete, the code shifts and scales the values according to global minimum and maximum bounds, bringing them into a preliminary [0, 1] range. Afterwards, there is an additional transformation via the function apply_tanh_transform. This tanh transformation helps control extreme values while preserving mid-range variations, often giving a more visually appealing distribution of terrain heights.

F. Procedural Generation of New Chunks

```
def calculate_new_map():
    global MAP, SIZE, step

    fbm_map = calculate_map_with_fbm(SIZE, 0, SIZE +
step - 1, SIZE - 1, OCTAVES)

    for i in range(SIZE, SIZE + step):
        MAP.append([])

        for j in range(SIZE):
            k = upper_bound_threshold(fbm_map[i - SIZE,
j])
            MAP[i].append(COLORS[k])

    fbm_map = calculate_map_with_fbm(0, SIZE, SIZE +
step - 1, SIZE + step - 1, OCTAVES)

    for i in range(SIZE + step):
        for j in range(step):
            k = upper_bound_threshold(fbm_map[i, j])
            MAP[i].append(COLORS[k])

    SIZE += step
```

The function calculate_new_map is responsible for extending the existing map boundaries whenever the user scrolls beyond the currently generated terrain. The process begins by generating an additional vertical strip of heightmap data. This is achieved through the call calculate_map_with_fbm(SIZE, 0, SIZE + step - 1, SIZE - 1, OCTAVES), which creates a new region of noise values that

cover rows from SIZE to SIZE + step - 1 and columns from 0 to SIZE - 1. In the subsequent loop from i = SIZE to i < SIZE + step, the function appends new rows to MAP and assigns a color to each cell based on its noise value through upper_bound_threshold. By doing so, the existing map is extended downward by step rows.

After creating the vertical strip, the code then produces a horizontal strip to complete the expansion. The call calculate_map_with_fbm(0, SIZE, SIZE + step - 1, SIZE + step - 1, OCTAVES) computes noise data for rows from 0 to SIZE + step - 1 and columns from SIZE to SIZE + step - 1. A nested loop then assigns colors to these newly added columns of cells. At the end of this process, SIZE is incremented by step, meaning the map is now larger in both dimensions. As a result, the user can continue scrolling seamlessly in any direction without encountering an "edge," because fresh terrain is always generated and appended to the map structure.

IV. Results and Discussion

Below is the result of generating a 150 pixel by 150 pixel map.

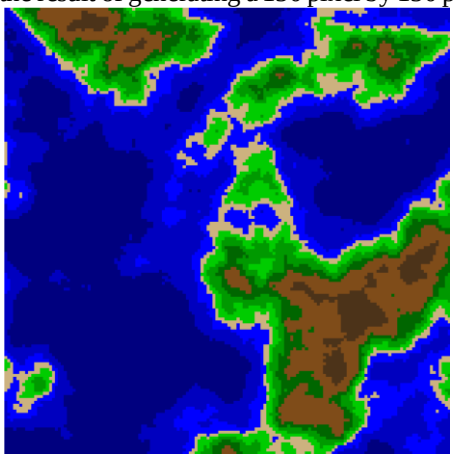

Figure 4. First generation: 150 pixel by 150 pixel map

After moving the map 5 times to the right and 5 times down, we get the resulting figure.

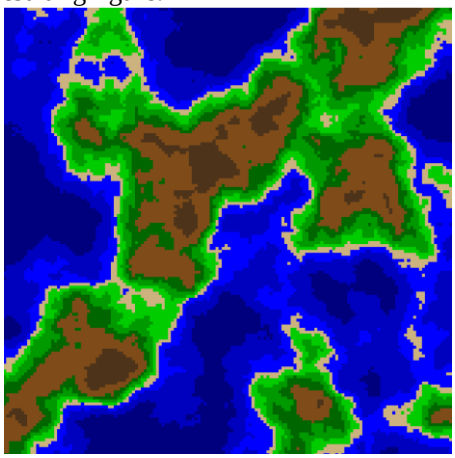

Figure 5. First generation: 150 pixel by 150 pixel map after translation

As can be seen above, the new map is consistent with the old map which demonstrates that procedural generation has worked successfully.

By tweaking the persistence, lacunarity, and number of octaves, different terrain styles can be obtained

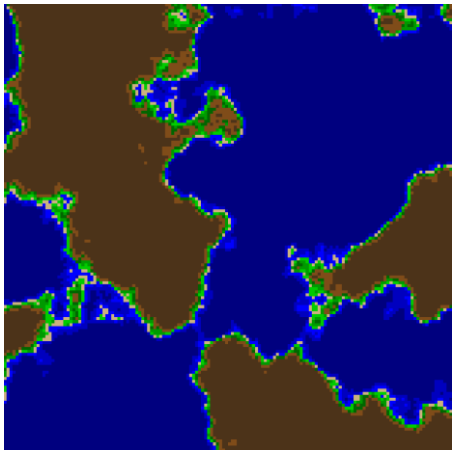

Figure 6. Second generation: 150 pixel by 150 pixel map with larger persistence

Increasing persistence causes the terrain tends to be more homogenous and the colors tends toward the extremes.

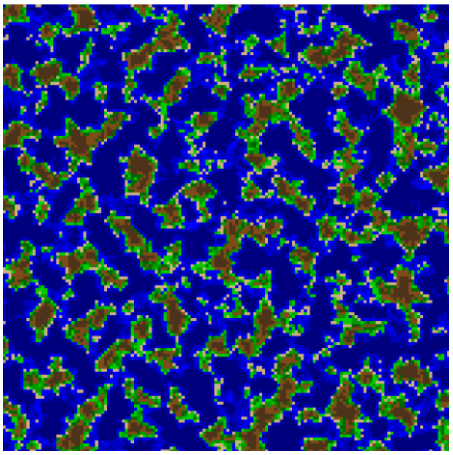

Figure 7. Third generation: 150 pixel by 150 pixel map with higher lacunarity

Increasing lacunarity causes the terrain to be more dispersed and have more small details.

Below is a 500 pixel by 500 pixel map with larger number of octaves.

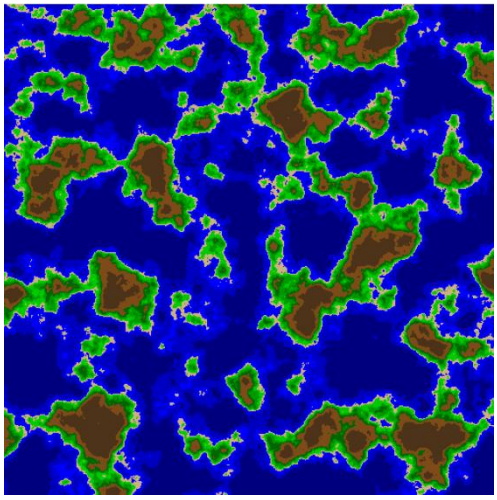

Figure 8. Fourth generation: 500 pixel by 500 pixel map with larger number of octaves

By increasing the number of octaves, the map contains finer details but still retains a degree of homogenicity.

V. APPENDIX

GitHub: https://github.com/BP04/Procedural-Terrain-Generation-Makalah-IF2123

Explanation Video: https://youtu.be/ogeeM9kZqOI

VI. ACKNOWLEDGMENT

The author is deeply thankful to God Almighty for providing strength, determination, and opportunity to bring this paper to completion. The author also expresses deep appreciation to Ir. Rila Mandala, M.Eng., Ph.D., the lecturer of the IF2123 Linear Algebra and Geometry course, for his dedicated guidance and support, which have been a source of inspiration throughout his teaching journey with the students.

REFERENCES

[1] D. H. Million, "A Project on the Mathematics of Voting and Apportionment," 2007. [Online]. Available: http://buzzard.ups.edu/courses/2007spring/projects/million-paper.pdf. [Accessed: Dec. 31, 2024]

[2] R. Touti, "Perlin Noise Algorithm," 2023. [Online]. Available: https://rtouti.github.io/graphics/perlin-noise-algorithm. [Accessed: Dec. 31, 2024].

[3] Munir, Rinaldi, "Review Matriks," 2023. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-01-Review-Matriks-2023.pdf. [Accessed: Dec. 31, 2024].

[4] Munir, Rinaldi, "Vektor di Ruang Euclidean (Bagian 1)," 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2024-2025/Algeo-11-Vektor-di-Ruang-Euclidean-Bag1-2024.pdf. [Accessed: Dec. 31, 2024].

[5] Munir, Rinaldi, "Vektor di Ruang Euclidean (Bagian 1)," 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-12-Vektor-di-Ruang-Euclidean-Bag2-2023.pdf. [Accessed: Dec. 31, 2024].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 31 Desember 2024

Benedict Presley
13523067