Perlin Noise, Procedural Terrain

Procedural generation is a valuable tool for game development and visual effects: instead of manually building every asset (meshes, textures, world maps, etc.), we specify algorithmic rules to create numerous variations of the asset. We have already seen Lindenmayer systems, which used simple rules to draw random instances of plants. This time, we will implement Perlin Noise and use it to generate textures and 3D terrains.

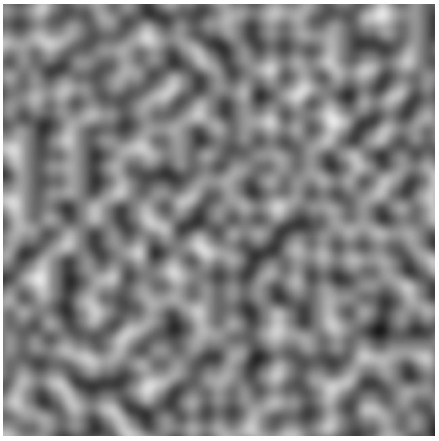


Figure 1: Perlin noise sample

1 Noise

Let us define n-dimensional noise as a function $f : \mathbb{R}^n \to \mathbb{R}$, n-dimensional vector and returns a scalar. For example, 2-dimensional noise can be represented as a 2D texture, where the brightness at point p of the image indicates $f(p)$ [Figure 1]. This function should have the desirable properties mentioned in lecture: no obvious repetitiveness, rotation invariance, smoothness (band-limited), computational efficiency, and reproducibility.

There are many different types of noise functions used in graphics. Two common categories are **value noise** and **gradient noise**. Both approaches generate pseudorandom values on a grid (whose points are typically placed at the integer coordinate locations of $\mathbb{R}^n$) and then interpolate them throughout $\mathbb{R}^n$. Value noise methods generate random *values* at the grid points and interpolate them with standard polynomial interpolation, convolution, or some other technique. Gradient noise methods instead generate random *gradients* at the grid points and apply Hermite interpolation. Interpolating gradients instead of values allows us to get higher degree (smoother) noise functions with more local (efficient) calculations.

In this assignment, we will implement Perlin noise, a gradient noise technique so popular that some use the names "Perlin noise" and "gradient noise" interchangeably. There are many online resources on Perlin noise, for example "Understanding Perlin Noise" (the only difference in their implementation is the hash function they use).

2 Perlin Noise in 1D

Let us investigate the 1D case of Perlin noise step by step. We will be completing the function `perlin_noise_1d` in file `src/shaders/noise.frag`. Run the script `scripts/run_perlin_1d.sh` to see your results.

2.1 Steps

2.1.1 Find the surrounding cell corners

Grid points appear at integral coordinates. In 1D, a cell is just an interval $[c_0, c_1]$. We can find the left endpoint ("cell corner") of this interval by applying the `floor` function to p,

$$c_0 = \lfloor p \rfloor,$$

and right endpoint will be $c_1 = c_0 + 1$.

An example is shown in Figure 2, where $p = 1.7$ is marked with a red star. The neighboring grid points are 1 and 2, marked with vertical gray lines.

2.1.2 Determine gradients g at cell corners

The noise should be deterministic, meaning we get the same result every time we run it—regardless of what computer we run it on—but still "look" random. Therefore, we need a repeatable way to assign gradient vectors to cell corners. In the original Perlin noise method from 1985, a large lookup table of random gradients was pre-computed, and a hash function was used to map nD grid point indices in $\mathbb{Z}^n$ to 1D indices into this lookup table. In the improved version from 2002, we use a table of only 12 specially chosen gradients, listed in Perlin's article. Perlin's justification for this simplified table is that human vision is relatively insensitive to the granularity of orientations, so randomly choosing from 12 different directions is enough to fool your eye into thinking the pattern is rotationally invariant.

So, with this improved approach, we use a hash function $h : \mathbb{Z}^n \to \mathbb{N}$ to deterministically assign an integer to every grid point $c_i \in \mathbb{Z}^n$ and then use this integer to choose from the table of 12 gradients:

$$g_i = \texttt{gradients}[h(c_i) \mod 12].$$

Provided our hash function assigns integers to grid points in a somewhat random-looking way, the gradient assignment will also appear random. Note that, in GLSL, the operation $x \mod y$ is written as $x\%y$.

The hash functions used for Perlin noise are based on a pseudorandom permutation P of the integers $0..N$. For example, for 2D grids, you calculate the hashed value of an integer point (i, j) by evaluating $P(i + P(j))$. Here, we assumed the inputs to P are first reduced modulo N.

Most CPU-based Perlin noise implementations use a fixed permutation, stored in an array of integers. However, this table-based approach is not as well suited for the GPU architecture. For this assignment in GLSL, we use the function $P(x) = (34x^2 + x) \mod 289$, which surprisingly gives a random-looking permutation of the integers 0..288, and is efficient to evaluate. A hash function based on this permutation is already implemented for you in `hash_func`.

2.1.3 Calculate contributions

The gradient chosen for each corner c_i defines a linear function over the all of $\mathbb{R}^n$:

$$\phi_i(p) = g_i \cdot (p - c_i). \tag{1}$$

This is the unique linear function taking value 0 and gradient g_i at c_i.

When evaluating the noise function at p, we simply interpolate the values of each corner's linear function at p. The linear functions for each grid point are plotted as dotted lines in Figure 2. The value at $p = 1.7$ will be an interpolation of the linear functions ϕ_1 and ϕ_2, plotted as green and red lines.

2.1.4 Interpolate contributions

The final value of our noise at p is a weighted average of the linear functions ϕ_i for p's surrounding cell corners. The idea is to smoothly interpolate the values of these linear functions based on how close p is to each corner. Let's assume p lies on the interval between corners c_i and c_{i+1} and is at distance $t = p - c_i \in [0, 1]$ along this interval.

We use the mix function, which performs linear interpolation between values x and y with the weight α: when $\alpha = 0$ it yields x and when $\alpha = 1$ it yields y.

$$\text{mix}(x, y, \alpha) = (1 - \alpha)x + \alpha y.$$

However, we do not simply want to linearly interpolate the values of the corner's functions; this would lead to severe discontinuities in the gradient of our noise function. Instead, to ensure our noise function is smooth across the grid points, we calculate the interpolation weights using the polynomial:

$$\alpha(t) = 6t^5 - 15t^4 + 10t^3.$$

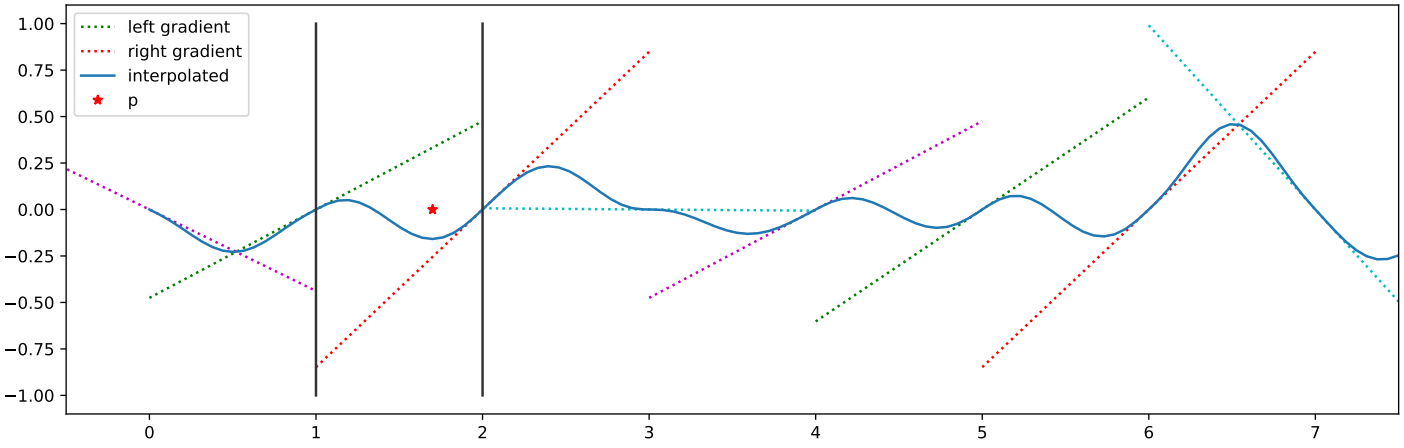


Figure 2: Dotted lines: linear functions at the cell corners that will contribute to the noise values. Blue line: final interpolated result.

The weights of each corner's contributions are plotted in Figure 3. For example, when p is close to $c_0 = 1$, t is small, and therefore the result is dominated by the contribution ϕ_0: near 1, the interpolated blue line follows the green line. When p is close to $c_1 = 2$, t is near 1, and the red line dominates the result.

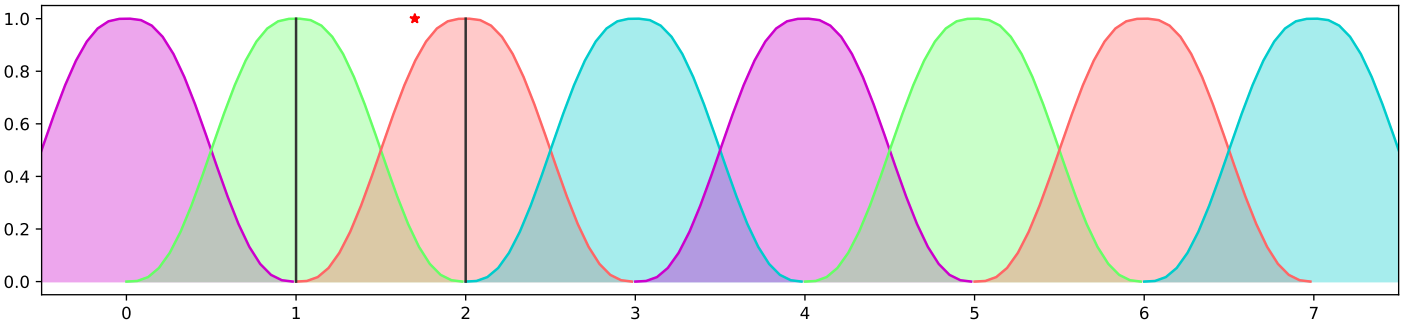


Figure 3: Interpolation weights for the 1D Perlin noise.

Task 2.1 Implement the 1D Perlin noise in function `perlin_noise_1d`. Run the script `scripts/run_perlin_1d.s` to see your results and produce the image (similar to Figure 4). The different plots display your noise function at different frequencies (see section below). The last plot is fBm, which you will implement later.

3 Fractional Brownian Motion

Textures in nature have both low frequency (general shape) and high frequency (detail) components. In order to mimic this effect, we combine noise at different frequencies; the resulting noise function is called fractional Brownian motion (fBm). It is also explained in [tutorial 1] and [tutorial 2].

(Brownian motion is the physical phenomenon where many small particles hit a heavy particle, and each hit contributes momentum in a different direction, resulting in random irregular motion of the big particle.)

To obtain the noise at frequency ω, we multiply the input point p by ω, the noise value is therefore $f(p\,\omega)$. The noise functions at the various scaled frequencies are called *octaves*, and just as with musical octaves, the frequency rises exponentially. So for the i-th octave, we have $\omega_i = \omega_1^i$.

However, we want the contributions of the high octaves to be small details on top of dominant lower frequencies. Therefore, we reduce the amplitude of the noise with each octave, also exponentially. The weight for the i-th octave is $A_i = A_1^i$, where $A_1 \in [0, 1]$.
The final result is therefore:

$$\text{fbm}(p) = \sum_{i=0}^{N-1} A_1^i f(p\ \omega_1^i)$$

In the code, ω_1 is `freq_multiplier`, A_1 is `ampl_multiplier`, and N is `num_octaves`.
Turbulence is very similar to fBm: the only difference is we sum the *absolute values* of octaves:

$$\text{turbulence}(p) = \sum_{i=0}^{N-1} A_1^i |f(p\ \omega_1^i)|$$

Task 3.1 Implement the 1D fractional Brownian motion in function `perlin_fbm_1d`. It will be shown in the bottom plot.

Figure 4: Top 4 plots depict the Perlin noise function at different frequencies. The bottom plot shows the weighted sum of these octaves (fBm). The Y range of the plots is $[-1, 1]$.

4 Perlin Noise 2D

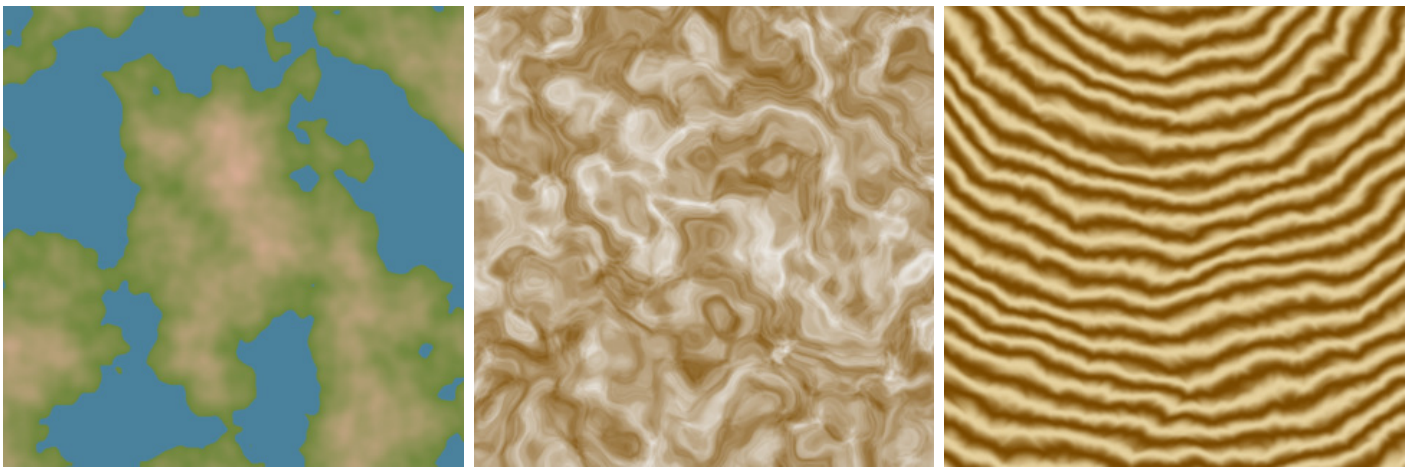
Now we extend the algorithm to 2D; the 2D process is also described in the lecture slide: `2D_Perlin_Diagram.pdf`. There are several differences compared to the 1D case:

- Each cell has 4 corners. We can still get one of the corners using $c_{00} = \texttt{floor}(p)$ and obtain the others by adding appropriate offsets to c_{00}.
- The operations in Equation 1 now operate on 2D vectors, and $\cdot$ is the dot product.
- The interpolation now has two steps, one per dimension: the two pairs of contributions at the top and bottom of the cell are first interpolated along the X axis (st and uv in the diagram), and then this result is interpolated along the Y axis. (The x and y in the slide's formulas mean *p's relative position inside the cell.*)

With this 2D Perlin noise function, we can use exactly the same procedure described in Section 3 to implement 2D variants of fBm and turbulence.

Task 4.1 Implement 2D Perlin noise in the function `perlin_noise`. Run the script `scripts/run_perlin_2d.sh` to see your results and produce the image.
Task 4.2 Implement 2D fBm in the function `perlin_fbm`. Run the script `scripts/run_fbm.sh` to see your results and produce the image.
Task 4.3 Implement 2D turbulence in the function `turbulence`. Run the script `scripts/run_turbulence.sh` to see your results and produce the image.

Shader viewer: The code we provide includes an interactive noise viewer. You can scroll around your texture with the *WASD* or arrow keys, and change the scale with the scroll-wheel. Press *P* or *F12* to make a screenshot.



(a) *Map* (b) *Marble* (c) *Wood*

Figure 5: Samples of the textures produced by the formulas below.

5 Textures

The primitive noise functions you've implemented can be used to procedurally generate many different textures.

World Map We interpret the noise value $s = \text{fbm}(p)$ as a terrain elevation. If s is below the water level, we give it the color of water. Otherwise we interpolate (mix) between between the grass color and the mountain color with weight $\alpha = (s - s_{\text{water}})$.
Wood, described in this article, is an interpolation between dark brown and light brown, with the following weight α:

$$\alpha = \frac{1}{2}(1 + \sin(100\,(\|p\| + 0.15\ \text{turbulence}(p)))).$$

Marble, described in this article, is an interpolation between white and dark brown, with the following weight:

$$\begin{aligned}\alpha &= \frac{1}{2}(1 + \text{fbm}(p + 4q)), \text{ where}\\ q &= (\text{fbm}(p), \text{fbm}(p + (1.7, 4.6)).\end{aligned}$$

Note, this is an example of the domain distortion/warping technique, where the output of one noise function is used to perturb the input to another. The result is the swirly pattern shown in Figure 5 (b).

Task 5.1 Implement those textures in functions `tex_map`, `tex_wood`, `tex_marble`. Use the scripts `scripts/run_map.sh`, `scripts/run_wood.sh`, `scripts/run_marble.sh` to see your result and generate images. As before, you can move around, zoom in/out or take screenshots.

6 Terrain

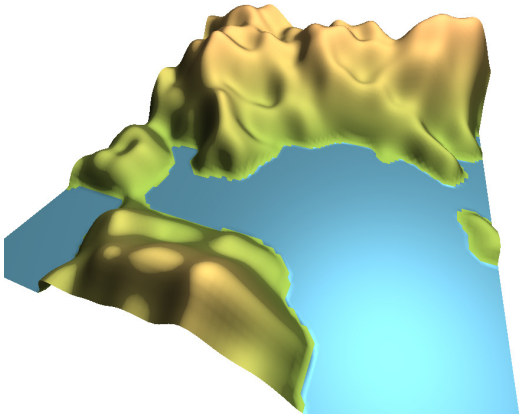


Figure 6: Procedural terrain: the elevation is equal to the value of fBm.

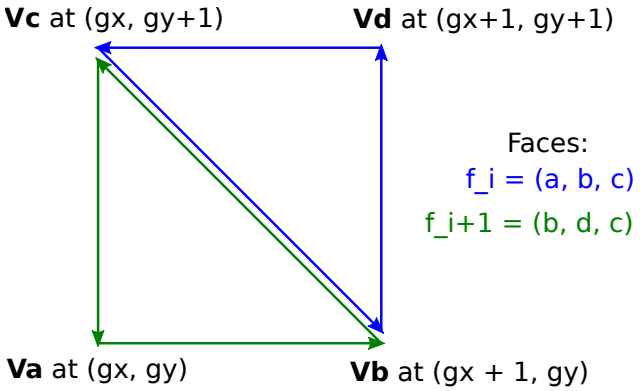


Figure 7: Tiling a grid cell with triangles.

In the previous section, we built a map texture of our procedurally generated world. Now let us build the terrain in 3D, as shown in Figure 6. To that end, we will build a 3D triangle mesh, which consists of:

- **Vertices** $v_0, v_1, \ldots v_{N_v-1} \in \mathbb{R}^3$: points in 3D space.
- **Faces** $f_0, \ldots f_{N_f-1} \in [0, N_v - 1]^3$. Each face is a triangle spanning between three of the mesh's vertices specified by index. For example, the face $f_i = (a, b, c)$ spans between v_a, v_b and v_c.

Please look at the function `build_terrain_mesh` in `src/main_terrain.cpp`.

In the first part, we construct the vertices on a rectangular grid in the XY plane. The **world XY** coordinates of the vertex should be such that the vertices span a uniform grid from -0.5 to 0.5 (this is an arbitrary choice, which we made to have good proportions with the height). Please convert the grid integral coordinates (g_x, g_y) into 2D points in this range.

The **world Z** coordinate of each vertex is determined by the fBm value at that point. We already wrote the code to capture your GLSL fBm output into a 2D array called `height_map`, which you can access with the grid coordinates (g_x, g_y). However if a point falls below the `WATER_LEVEL`, it should be clamped back to `WATER_LEVEL` to produce a flat lake.

Vertices must be fed into OpenGL in a 1D sequence $v_0, v_1, \ldots v_{N_v-1}$ so we map the grid position (g_x, g_y) to index $(g_x + \text{width} * g_y)$; this conversion is done by the function `xy_to_v_index`.

Finally, we must connect these grid vertices with triangles to cover the plane. To that end, in the second loop, we will cover each grid square with two triangles, as shown in Figure 7.

Notice: To complete the rendering pipeline, please:

- Fill in the terrain coloring shader `src/shaders/terrain.frag`, by adapting your solution for the map texture. Apart from setting the color (variable `material`), fill in the `shininess`, which is 8.0 for water and 0.5 for terrain. Additionally, insert the **phong shading** function from Assignment 7. We use the same material color (`material`) for both the specular and diffuse components.
- Insert the vertex normal computation from Assignment 3 into `Mesh::compute_normals` in `src/render/Mesh.cpp`. Insert `n_matrix` from Assignment 7 into `MeshViewer::draw_scene` in `src/render/MeshViewer.cpp` (this is not graded).

Task 6.1 Implement the mesh construction function `build_terrain_mesh` in `src/main_terrain.cpp` to fill in the `vertices` and `faces` arrays. Also, complete the terrain fragment shader `src/shaders/terrain.frag`. Run `scripts/terrain.sh` to see your results.

7 Grading

The scores for this assignment are broken down as follows:

- 15%: 1D Perlin noise (Task 2.1)
- 35%: 2D Perlin noise (Task 4.1)
- 15%: fBm and Turbulence (Tasks 3.1, 4.2, and 4.3; 5% each)
- 15%: Textures (Task 5.1; 5% each)
- 20%: Terrain (Task 6.1; Terrain mesh construction: 15%, shading: 5%)

8 What to hand in

Please submit:

- `src/shaders/noise.frag` - noise and texture functions,
- Noise images in `out`: *perlin_1d_plot, perlin_2d, fbm, turbulence*,
- Texture images in `out`: *map, wood, marble*,
- `src/main_terrain.cpp` - mesh building function,
- `src/shaders/terrain.frag` - terrain coloring shader,
- `readme.txt` containing the percentage of contribution by each team member and any other remarks you have.

Note that the noise and texture `png` output images described above are generated by running the scripts in `scripts/` (or `scripts_win/` on Windows) from the root codebase directory.