Generating game terrain using own implementation of the Perlin Noise

Part I:

1. Introduction

For this task I have chosen to implement my own version of the Perlin noise. Perlin noise is a type of gradient noise developed by Ken Perlin in 1983, commonly used in computer graphics to create smooth, natural-looking textures and procedural terrain by generating pseudo-random patterns with gradual transitions. In my case I have decided to create a simple 2D procedurally generated image simulating a simple world filled with lakes, fields and snowy mountains.

2. Explanation of the algorithm and implementation

Perlin noise works by first assigning random directions to unit vectors to the corners of a grid that covers the space being evaluated. Then for any given point, it measures how far away the point is from each grid corner and for each corner calculated the dot product between that distance vector and the gradient vector at the corner. The resulting values from all relevant four corners are then smoothly interpolated using a fade function.

Normally Perlin noise also uses the technique of layering multiple Perlin noise functions with increasing frequency and decreasing amplitude - so called octaves. Which I did not implement due to time constraints.

The actual implementation took place step-by-step without a certain goal in mind, just to see how it goes.

At first I implemented a **Perlin.java** file with functions like:

- **directionArrayGenerate(int numCol, int numRow, int seed)** - function that generates random directions of all the corner unit vectors based on a seeded random function
- **perlinGenerate(float x, float y, int[][] perlinArray)** - function that for a given point returns it's Perlin noise value, with the help of other smaller helper functions

Next was the **CSVwriter.java** file, with which I converted the given Perlin noise value array to a csv file. For which I then wrote a **CSV2PNG.py** file, since I prefer working with .java files, but still the ease of use of working with files in Python made me choose it. In that Python script you can find functions like:

- **read_csv_to_image(csv_file, output_png)** - function which with the help of other functions creates a black and white image, with only ten different shades of gray, based on the .csv file, created in java.
- **png_smooth(input_png, output_png)** - function which from the ten shade greyscale image creates smooth transition using a gaussian filter, which can be directly be controlled by user input, and exports it as another greyscale, in this case smooth, image
- **bw_to_color(input_png, output_png)** - function, which from the smooth greyscale image, creates a colored image, based on the value of gray, of a certain pixel and the **get_color** function which returns a suitable color

3. PCG taxonomy of the algorithm

Perlin noise fits specifically into the family of lattice noise algorithms and ontogenetic algorithms, since it generates content directly from mathematical procedures without simulating a system's evolution over time.

4. Visualization of the algorithm

The algorithm is directly visualized with the three generated ".png" files. The first one to be created is "output.png" **(figure 1.)**, the grayscale pixelated image, then the algorithm generates the smoothed out version of the same image **(figure 2.)**, and the final step of the visualization is the coloring algorithm, that generates the colored "world" image in the "outputFinal.png" file **(figure 3.)**.

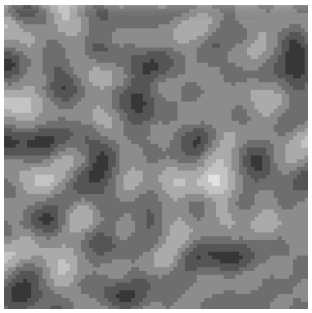

Figure 1.

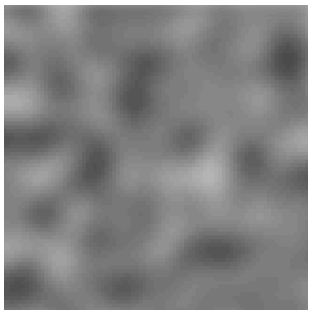

Figure 2.

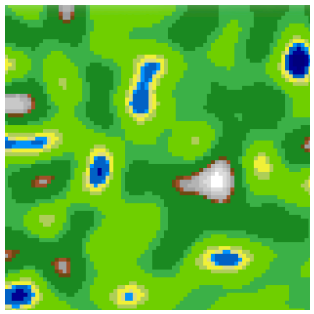

Figure 3.

5. Expressive range of the algorithm

Due to the naive implementation of the said algorithm, the expressive range of the current Perlin noise implementation is limited. Any generated world is always a combination of all basic terrains (lakes, fields, snowy mountains) without distinct spatial separation or thematic

contrast. Variation is present between individual seeds but not within a single generated image. Expanding the expressive range requires additional algorithmic features (octaves, masks, region control), which would allow distinct regions and more complex world building possibilities.

Part II:

1. Critical reflection over the algorithm, content and evaluation

The algorithm successfully implements Perlin noise producing some variation. However, because only single octave is used, the diversity and realism of the environments are limited. The output, despite variation in seed, lacks the expressive range necessary for convincing biomes or large-scale geological features.

2. What would I do different, what worked, what didn't, what would I do next

If I had more time I would definitely try to implement the octave layers, since they help with creating a more realistic and smooth transition between values. I would also try to implement a way to fix the most glaring issue, which is regarding the repetitiveness of the generated content and the lack of diversity.

3. The use of AI in the task

AI was used in this task for proof reading the report and to help generate some minor helper with discrete helper functions when manual effort could be better allocated elsewhere.