Parallel Computing Lab: Perlin Noise

Silviu Leonard Vatamanelu
silviu.vatamanelu@edu.unifi.it

Abstract

In this paper we implemented Perlin Noise, a coherent noise function, for the generation of procedural terrain. The base noise function had been integrated with a framework that allows the combination of different noise function and the generation of different types of maps. We also implemented the parallel version of this problem using the Intel Thread Building Blocks library leveraging its base functionalities and the flow graph implementation to speed up the generation. We also compared OneTBB with OpenMP to have some baseline results. The code can be found at `https://github.com/ViuTheLumberjack/perlin`.

1. Introduction

Perlin noise is a type of gradient noise developed in 1983 by Ken Perlin. It is mainly used in computer graphics to generate pseudo-random assets, such as procedural terrain, fire effects, water, and clouds.

Figure 1: Example of Perlin noise

Noise function can be expressed in any dimension, a one-dimensional noise might be used for generating handwritten lines, a two-dimensional noise might be used for height maps of surface textures, a tridimensional noise can be useful in terrain generation.
The base noise function produces visually appealing outputs but the results are very homogeneous, as seen in figure 2a, this limit can be overcome by combining different noise functions, at different resolutions with various mathematical functions, such as in figure 2b.

2. Perlin Noise Algorithm

There are 3 main steps in the algorithm:

- Defining a grid of random gradient vectors.
- Computing the dot product between gradient vectors and their offsets.
- Interpolation between these values.

2.1. Grid of random gradient vectors

A random gradient vector is chosen for each intersection point in the 3-D grid. In particular, the grid of random gradients is chosen from 12 possible combinations that can be seen in listing 1.

$$(1,1,0),(-1,1,0),(1,-1,0),(-1,-1,0),$$
$$(1,0,1),(-1,0,1),(1,0,-1),(-1,0,-1),$$
$$(0,1,1),(0,-1,1),(0,1,-1),(0,-1,-1)$$

Listing 1: Random Grandient Vectors

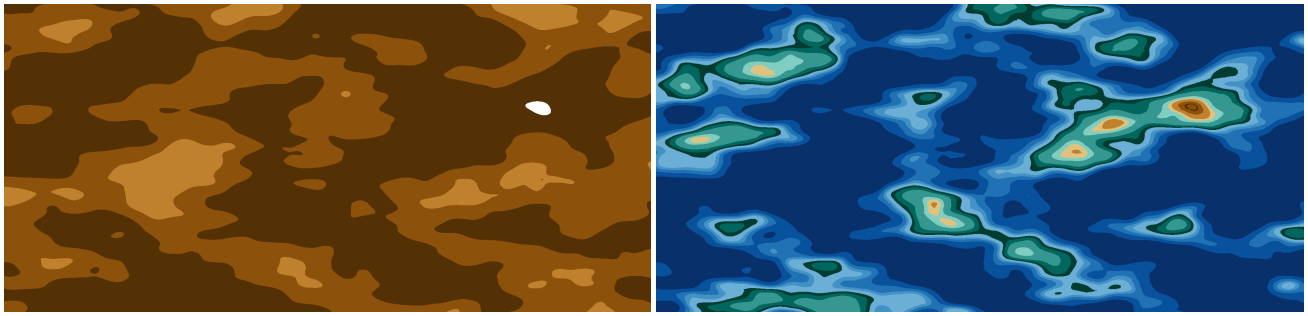
The choice is made using a precomputed permutation table, with uniform values between 1 and 255, that is looked up sequentially for each coordinate to produce a hash, as seen in listing 2, whose last 4 bits are used to determine the pseudo-random gradient.

```
int hash = p[p[p[X] + Y] + Z];
```

Listing 2: Hash Calculation

2.2. Dot Product of gradient vectors

We now need to find the unique grid cell in which the source noise point lies. This yields the knowledge of the random gradients associated with the intersection points of the cell. Then, we can calculate the displacement vectors between the input point and all the corners.
For each random gradient in the corner we take the dot product with the corresponding displacement vector. From this step we can see that the complexity of the algorithm is $O(2^n)$ where n is the number of noise dimensions.



(a) Perlin noise (b) Sawtooth of exponential of Perlin noise

Figure 2: Comparison between Perlin noise and a composition of functions over Perlin noise

2.3. Interpolation

In the end we need to interpolate between the 2^n dot products, this can be done with the formula 1.

$$f(x) = a_0 + \text{fade}(x) * (a_1 - a_0), \text{for} x \in (0,1] \quad (1)$$

The fade function used is the one proposed by Perlin, which has the property of having the first 2 derivatives at 0. It produces smoother results than the first order one.

$$\text{fade}(x) = 6x^5 - 15x^4 + 10x^3, \text{for} x \in (0,1] \quad (2)$$

In figure 4a we can see the noise produced only with this steps.

2.4. Frequency and Octaves

To make the noise more appealing and natural, we can think of adding noise at different frequencies and amplitudes together. This means we are adding more octaves to the noise, like in music. The amplitude and frequency of each subsequent octave are calculated as

$$frequency = 2^i$$
$$amplitude = persistence^i$$

where i indicates the octave produced and persistence is measure of how much influence the octave should have.

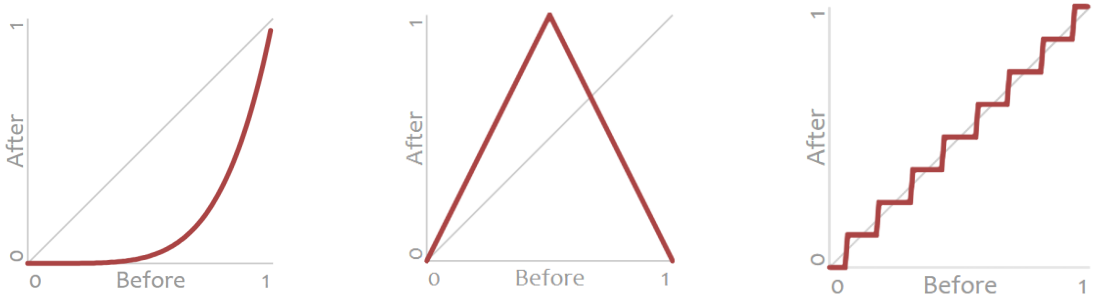
Figure 4b show the influence of octaves.

2.5. Redistribution

The output now looks more varied, but when dealing with random terrain generation, we would like to span between valleys and mountain ranges. This can be accomplished by redistributing the noise values. In figure 3 we can see a few different redistribution techniques:

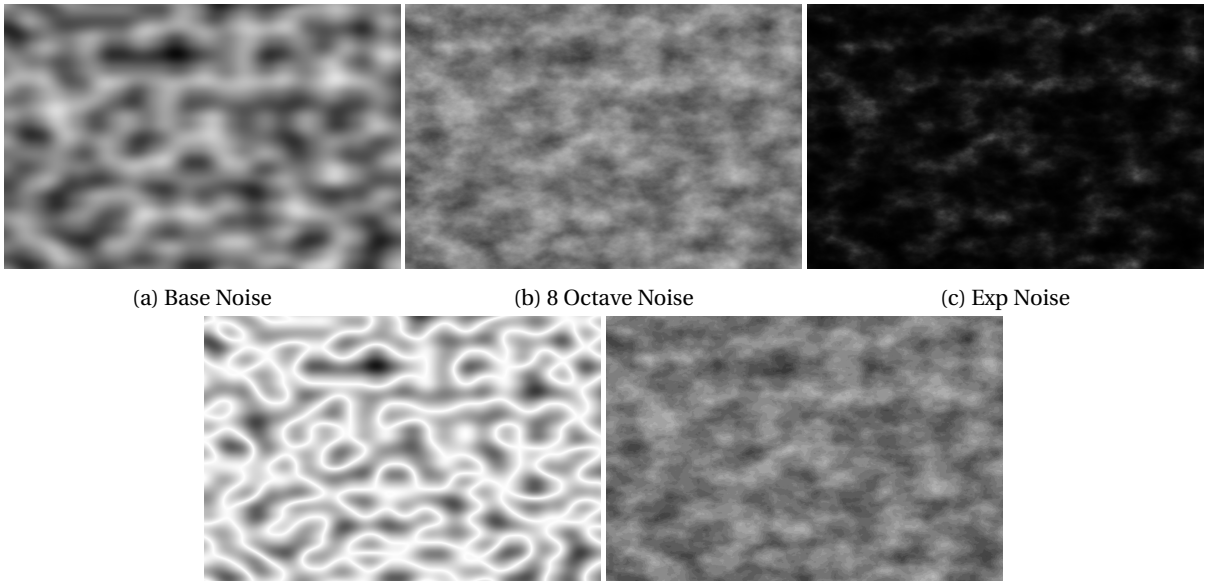
- Exponential, can push down values near zero and raise values near one. In figure 4c.
- Sawtooth, middling values (the most frequent ones) become higher, while border values go to zero. Like in figure 4d.
- Step function, gives a quantization-like output, with a specific set of uniformly spaced outputs. Figure 4e contains an example.

The redistribution techniques are not limited to the one presented. But for the sake of brevity only the ones described are treated.



$f(x) = x^y$ $f(x) = 2 \cdot (0.5 - |0.5 - x|)$ $f(x) = \lfloor x \cdot \text{steps} \rfloor \cdot \text{steps}$

Figure 3: On the left, an exponential, on the center a sawtooth, on the right a step function.



(a) Base Noise (b) 8 Octave Noise (c) Exp Noise

(d) Sawtooth Noise (e) Stepped Noise

Figure 4: Example of different noise functions on same noise input

3. Project

The main structure of the code that represents the noise and it's modifiers is shown in the UML in figure 5. We leverage the power of the Decorator pattern to build a composition of the base noise functions, ConstantNoise and PerlinNoise, with the redistribution operators discussed before and a few other operators that allow operations between different noise maps.

This creates a tree structure of the code to calculate the noise function that will be discussed later on.
Each type of terrain applies the appropriate morphology to translate the coordinates of the flat map into the ones used to sample the noise.

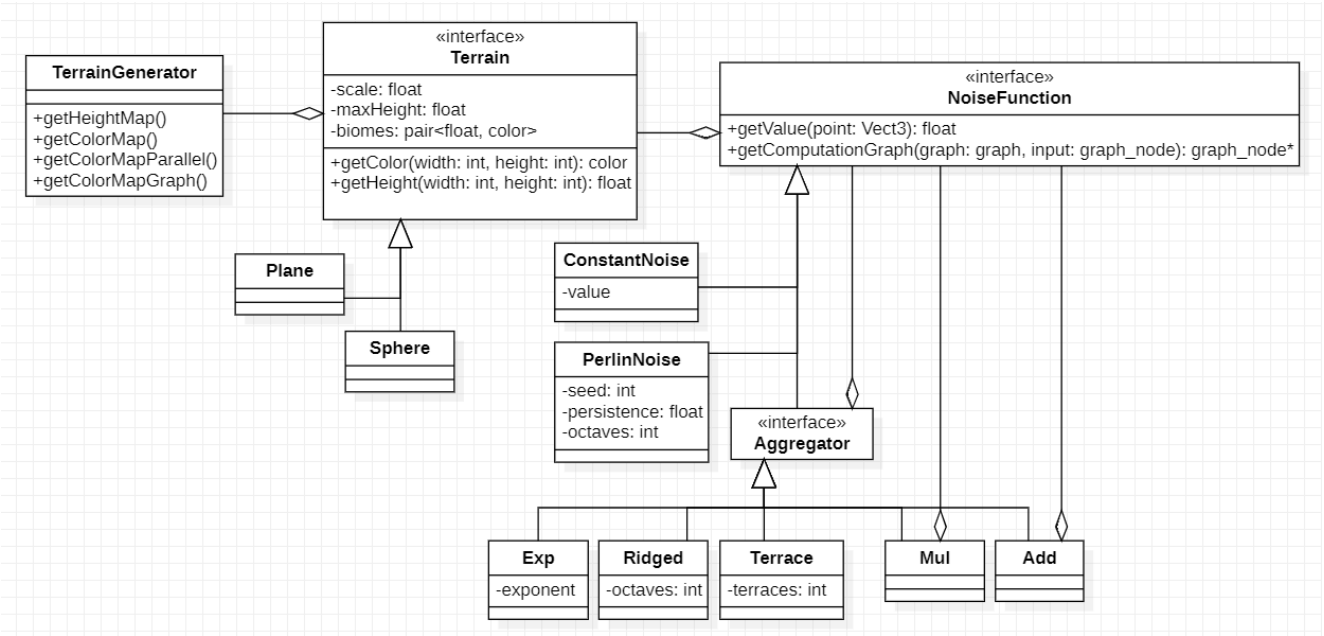


Figure 5: UML of the noise structure

3.1. Parallel Version

The algorithm described is embarrassingly parallel, meaning we can improve its performance with some parallelism.
We decided to use **Intel OneAPI Thread Building Blocks** to implement the parallel version. One might think that this library and OpenMP are alternatives, in reality, they serve different purposes and can blend together nicely. In particular, OpenMP has a coarser control over the code while OneTBB can be used for finer details.
In this use case we only use the *parallel_for* function and the capabilities offered by the *flow* namespace.
The *parallel_for* is used to compute in parallel the value of the noise function at different points, meaning we generate the entire map in parallel. It takes as parameters a *blocked_range*, an iterator over the the pixels that need to be calculated, and a functional, so a class with the *operator=* redefined, or a lambda that represents the code that needs to be run. Each thread is assigned automatically a portion of the entire *blocked_range*, using an internal partitioner. In listing 3 an example of parallel_for block.

```
parallel_for(
    blocked_range<int>(0, width * height),
    [this, &colorMap, width, height](
        blocked_range<int> const &r) -> void {
    }
```

Listing 3: Paraller For Block

The default one is the *auto_partitioner* that equally distributes work among the threads. Other partitioner can be used, like the *affinity_partitioner* that takes into account cache affinity for work distribution.

The other construct used is the *graph*, a class that represents a computation graph with nodes and edges representing data flow dependencies. It is constructed from the decorator pattern and encapsulates all the simple steps functional to produce the end noise value.
The construction of the graph starts from the top of the object tree, applying a post-order tree traversal that creates the appropriate computation node and makes an edge between the new node to be added to the graph and the parent one. The graph is then used in a *parallel_for* work sharing region, where each tread constructs a copy of the computation graph and uses it to calculate the final noise value.
We decide to use the computation graph in order to parallelize the evaluation of noise branches that that are independent from each other until a join node, like the **add** block or the **mult** block that require 2 different noise function to be evaluated.

4. Experiments

We tested out the code we implemented on an *AMD Ryzen 5 4600U* CPU, with a base clock speed of 2.1 GHz, and with 6 physical cores and 12 logical threads. This information is crucial to determine how many tests we have to take.
All the data taken into consideration is produced by repeating 20 time each test, evaluating the sequential, the parallel for implementation and the parallel graph implementation. We varied the threads from 1 to 24 with stride 2 on three different noise functions of increasing difficulty, in listing 4, and at 3 different resolutions: 640x480, 1024x786, 1920x1080. This allows us to see how the algorithms perform based on **thread number**, **noise function** evaluated, **resolution** and parallel **algorithm**.

```
std::make_unique<PerlinNoise>(69, 0.5f, 3));
std::make_unique<RidgedNoise>(
  std::make_unique<Exp>(10.0f, 1.2f,
    std::make_unique<PerlinNoise>(9, 0.5f, 3)),
    8));
std::make_unique<RidgedNoise>(
  std::make_unique<Add>(
    std::make_unique<Exp>(10.0f, 1.2f,
      std::make_unique<PerlinNoise>(6, 0.5f, 3)),
    std::make_unique<Mult>(
      std::make_unique<PerlinNoise>(3, 0.5f, 3),
      std::make_unique<ConstantNoise>(0.25f)),
    2.0f, 1.0f),
 8));
```

Listing 4: Paraller For Block

For each test we measure the elapsed time from the beginning of the terrain generation until the end.
We also measure the speedup, equation 3, to have a better understanding of the influence of parallelization.

$$Speedup = \frac{SequentialTime}{ParallelTime} \tag{3}$$

Before presenting the parallel results, in table 1 we have a comparison between the TBB implementations and the OpenMP one on a fixed resolution over all noise functions in the case defined by the compiler , hence when we are not manually defining the number of threads we want, and we are using the number of logical processors.

	OpenMP	**TBB Par For**	**TBB graph**
Perlin	6.43×10^5	5.20×10^6	6.58×10^6
Ridged	5.47×10^6	45.01×10^6	7.07×10^6
Complex	1.07×10^7	8.74×10^7	1.45×10^7

Table 1: OpenMP vs OneTBB with 12 threads time in microseconds

We can see that always OpenMP performs better that OneTBB, this comes with no surprise because OpenMP works through compiler directives so a higher level of compiler optimization benefits more such directives than compiled library bindings. This test can be repeated by manually compiling OneTBB with the same optimization (`-O3`), but for the sake of brevity this test has not been performed.

We now proceed analyzing the mean times and speedup varying the thread numbers. In figure 6 we have the time comparison for each noise type, while figure 7 contains the speedups, once again grouped by the noise type.

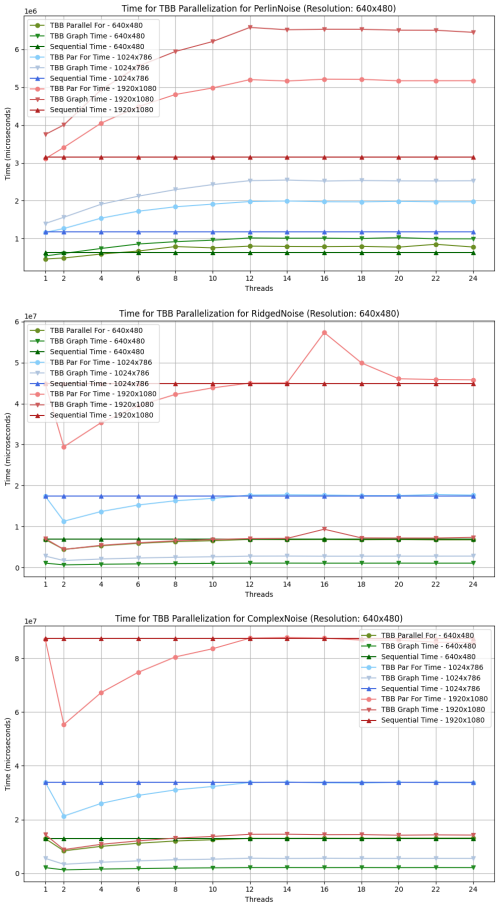


Figure 6: From top to bottom: Perlin noise, RidgedNoise and ComplexNoise noises times at each resolution varying threads

We can notice that for easy problems (PerlinNoise), almost always the parallel versions are much worse than the sequential one. With medium size problems (RidgedNoise) and complex problems (ComplexNoise), parallelization improves our performance a lot, especially the graph version.

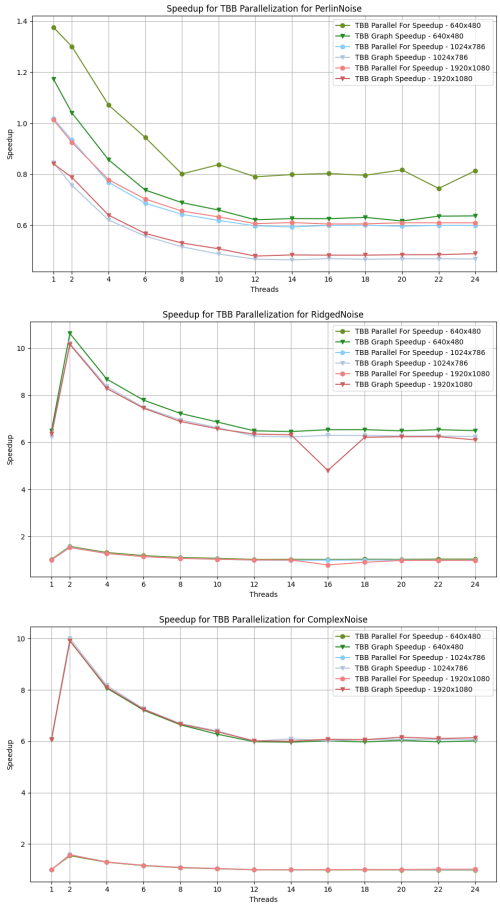


Figure 7: From top to bottom: speedup at each resolution varying threads

We can also notice that in the last two cases, with OneTBB the time converges towards the sequential when increasing the thread number, meaning we have a safe net if we mistakenly change the thread count value with a wrong number. We also notice that the minimum, with the default partitioner is always found at 2 threads, we could further develop this study with other partitioners.

5. Conclusions

We have seen how to tackle the Map Generation problem leveraging Perlin Noise, a continuous gradient based noise. We have seen how to redistribute the noise values to produce varied outputs and have stunning maps that can model many scenarios. Then we have seen how to implement and parallelize the code, comparing OneTBB with OpenMP and taking a look at the difference in time between the sequential and parallel implementation. We have noticed that the library chosen excels at handling complex dependencies with smart internal logic which we tried to leverage.