UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science

Jaan Janno

Java library for parallel computing of simplex noise

Bachelor's Thesis (9 ECTS)

Supervisor:
Professor Eero Vainikko

Tartu 2015

Java library for parallel computing of simplex noise

Abstract:

The thesis gives an overview of the process of writing the Java library Libjsimplex for parallel calculation of simplex noise. The library can execute its calculations on a GPU if it is available. If not, it falls back to using the CPU. The library allows the user to set various parameters, which alter the properties of the generated noise.

The library can be used to transform generated noise into images or arrays of RGB data. Several parameters can be adjusted to dictate the look of visualized noise.

An overview is given about the concept of noise and its possible use cases. The implementation's performance is compared to Stefan Gustavson's original code [1].

Keywords:

Simplex noise, gradient noise, noise, parallel computing, Java

Java teek simplex müra paralleelseks arvutamiseks

Lühikokkuvõte:

Käesolev bakalaureusetöö kirjeldab protsessi, mille käigus loodi Java teek Libjsimplex, mis suudab arvutada simplex müra. Teek suudab arvutusi teha paralleelselt graafikaprotsessoril või selle puudumisel tavalisel protsessoril. Teek pakub ka mitmeid võimalusi määrata parameetreid, mis müra omadusi muudavad. Toetatud on erinevad võimalused müra graafiliseks töötluseks.

Teeki saab kasutada müra piltideks ja RGB massiivideks muutmiseks. On võimalik muuta mitmeid parameetreid, mis määravad müra väljanägemise.

Selgitatakse ka müra mõistet ning antakse ülevaade müra kasutusvõimalustest. Implementatsiooni jõudlust võrreldakse Stefan Gustavsoni algse koodiga [1].

Võtmesõnad:

Simplex müra, gradient müra, müra, paralleelarvutus, Java

Table of Contents

1 Introduction

The goal of the thesis is to give an overview of the process of creating a Java library for parallel computation of the simplex noise function and to describe its functionality. The concept of simplex noise is explained in section 3.4.

The library does the calculation of noise on the GPU, if the user's system has a GPU that supports OpenCL. GPUs manufactured by Nvidia, AMD and Intel are supported on the Windows platform. On the Linux platform, AMD and Nvidia products are supported. If no GPU is present, the CPU is used instead automatically.

The library allows the user to edit multiple parameters that affect the properties of the noise. These parameters are as follows.

1. Number of dimensions (2 to 4 dimensions supported)
2. The size of the returned array of noise and its location in the generated noise space.
3. Frequency of the noise.
4. Number of octaves. (the number of sub-frequencies in the noise)
5. Persistence. (the weighted extent by which sub-frequencies affect the noise)

The library aims to make handling of the calculated noise arrays user friendly. This consists of several systems.

First, the library supports transformation of noise arrays to bitmap format or an RGB array. It is possible to create custom color transitions and assign contrast.

Secondly, the library can calculate automated transitions from 3 and 4-dimensional noise to 2-dimensional noise, which allows projecting 2-dimensional surfaces with certain properties from higher-dimensional noise space.

Third, the library can handle combining several octaves of noise to create noise with several sub-frequencies.

This library was created, because although open source libraries for noise generating already exist for Java, none of them is parallelized with the support for taking advantage of GPU computation in a way that is hidden from the user of the library and uses simplex noise.

The library doesn't require the users to have an understanding of how the binding with the GPU works and what specifics the library uses for generating and transforming the noise. Especially when it comes to transformations from higher-dimensional noise to 2-dimensional noise, which allow animations and seamless edge transitions for the derived 2-dimensional noise – the user does not need to have an understanding of 4-dimensional geometry.

The first step towards the goals was modifying Stefan Gustavson's Java implementation of simplex noise [1]. It needed several changes to be compatible with the Aparapi library [5], which was used for the parallelization. This is discussed further in the implementation chapter.

The next step was to create a system that takes the users parameters and calls the Aparapi's methods appropriately. That system creates an array containing points in the noise space that need to be calculated and passes them forward.

After performing given steps, it was possible to write methods accessible to the user. This includes the features mentioned earlier.

These steps are further discussed in the implementation chapter.

Another goal is to analyze how well this Java implementation stands against the original implementation. The effectiveness of parallelization is concluded from measuring how much faster it is compared to the original single threaded implementation by Stefan Gustavson, whose work is the basis of the thesis in hand. That gives unbiased information about the parallelization's effectiveness, because it is in essence the same algorithm. It is also be measured how well the parallel calculation scales, when its workload is increased by adding additional octaves.

The following chapter gives a short overview of parallel computing on the GPU and a library called Aparapi, which is used for parallelization in Java.

Chapter 2 explains the concept of parallel computing. Chapter 3 gives an overview about the concept of noise and its uses. Also, the process of writing the library is covered in detail in chapter 4. Last, in chapter 5, the results of the performance tests the library went through are shown.

2 Parallel computing on GPUs

2.1 Overview

Due to the clock rates of processors remaining relatively unchanged and uniprocessor performance growth slowing down, a movement towards adopting parallel computing is taking place. The most accessible parallel architecture available to the masses is the GPU. Modern GPUs can be utilized by adopting parallel programming languages such as CUDA and OpenCL. [4]

The thesis in hand relies on OpenCL to fulfill its goal to parallelize simplex noise computation. It is doing so using a library called Aparapi, which is introduced in the next section.

2.2 Aparapi

Aparapi is a Java library that lets developers utilize GPUs for executing data parallel code fragments. It is able to translate Java bytecode to OpenCL in runtime and execute it on the GPU. While for some reason unable to execute the code on a GPU, it is executed on the CPU using the Java thread pool. [5]

Due to Java and OpenCL having different architecture, Java bytecode cannot be translated in its full extent, which creates some limitations to which Java features can be used along with Aparapi. Those include the inability to use almost all object-oriented approaches and non-primitive data types. Only one-dimensional arrays are allowed. Exceptions cannot be used. [6, 7]

Suppose the code example on Figure 1 needs to be refactored to use parallel computation.

```
final float inA[] = { 1, 2, 3 }; // get a float array of data from somewhere

final float inB[] = { 1, 2, 3 }; // get a float array of data from somewhere

// (inA.length==inB.length)
final float result[] = new float[inA.length];

for (int i = 0; i < result.length; i++) {
      result[i] = inA[i] + inB[i];
}
```

Figure 1: A loop that could be refactored to use parallel computation. [5]

The needed refactor can be fulfilled by creating a new Kernel object and overriding its run() method. When the `execute()` method is called on the Kernel object, the overridden `run()` method is translated to OpenCL if it is the first call. Then, the OpenCL code is executed on the GPU, if a GPU is available. [5]

This refactor can be implemented in a Java project, for example, as illustrated by the code example in Figure 2.

```
final float inA[] = { 1, 2, 3 };

final float inB[] = { 1, 2, 3 };

final float result[] = new float[inA.length];

Kernel kernel = new Kernel() {

	@Override
	public void run() {
		int i = getGlobalId();
		result[i] = inA[i] + inB[i];
	}
};

Range range = Range.create(result.length);
kernel.execute(range);
```

Figure 2: Loop refactored using Aparapi to use parallel computation. [5]

3 Noise

3.1 Overview

The following sections discuss the possible use cases for noise and different methods for generating it. Integer noise and two different types of gradient noise are introduced.

Integer noise is a function, that returns a pseudo-random value given an integer input. It is introduced in the next section. [9]

Gradient noise is a pseudo-random mapping from R^n to R. This means, that a gradient noise function needs n floating point arguments and returns one floating point value. It is ideally isotropic and continuous. Perlin noise and simplex noise, which are also introduced some chapters later, are both forms of gradient noise. The most common use for gradient noise is generating animations and textures. [2]

Another possible use for gradient noise is terrain generation. It can be used for simulating different patterns that exist in nature. [3]

These mentioned possible uses for gradient noise are covered in an upcoming section after integer noise and the two forms of gradient noise have been introduced.

3.2 Integer noise

Integer noise is a function, that outputs a pseudorandom value given an integer input, as Figure 3 illustrates. An implementation of it shown on Figure 4. [9]

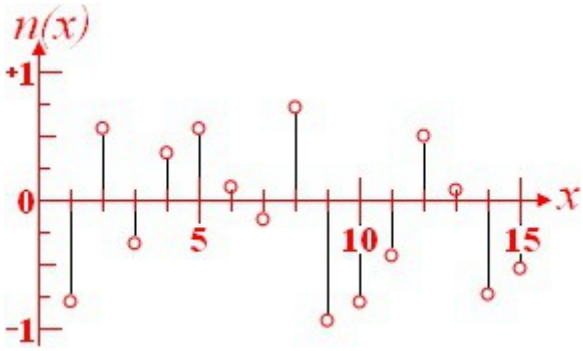


Figure 3: Integer noise function's visualization. [9]

```
double IntegerNoise (int n)
{
  n = (n >> 13) ^ n;
  int nn = (n * (n * n * 60493 + 19990303) + 1376312589) & 0x7fffffff;
  return 1.0 - ((double)nn / 1073741824.0);
}
```

Figure 4: Integer noise code from the Libnoise library. [9]

Integer noise is relevant in this thesis, because a modified version of it was used to optimize the generation of simplex noise. How it was used is covered more specifically in the implementation

chapter. Unlike Perlin and simplex noise, integer noise is not gradient noise as its inputs and output are not continuous.

3.3 Classic Perlin noise

Perlin noise is a predecessor to Simplex noise. Their principle is similar, but Perlin noise has several disadvantages compared to Simplex noise, which are mentioned in the next section. The following section is based upon [1].

Perlin noise can, however, be easier to understand and explain. As it is a good algorithm for demonstrating the basic principles of gradient noise, this section covers how Perlin noise works.

The basic principle of the Perlin noise function is that it can take an arbitrary number of real number arguments, depending on how many dimensions are needed, and returns one real number in the interval between -1 and 1.

One-dimensional variant is the easiest to comprehend. That would need just one real number as an argument. The only axis in this 1-dimensional space is divided into unit length parts, as can be seen in Figure 5.

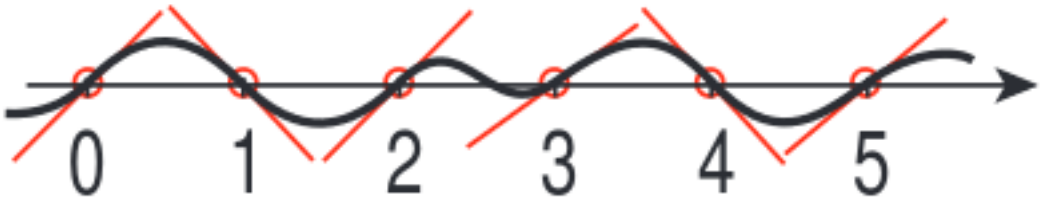


Figure 5: One-dimensional Perlin noise with gradients marked as red lines. [1]

When calculating the value of the noise function at certain coordinates, the first thing to notice is that at integer arguments the value of the noise is 0. As is evident on Figure 5.

Each integer coordinate has a specific gradient parameter associated with it, which is best described as a one-dimensional vector . If the argument to the noise function is not an integer, the value is calculated by interpolation depending on the interval the argument is in. The gradient from the integers on both edges of the interval contribute to the value of the noise function for a specific point. On Figure 5, the gradients are visualized as red lines.

The contribution from a gradient is calculated by a dot product from the gradient vector and the vector that marks the arguments position relative to the integer point. Since both vectors consist of just one dimension, the dot product is just the multiple of those single coordinates.

Those contributions are blended using a polynomial function shown on Figure 6.

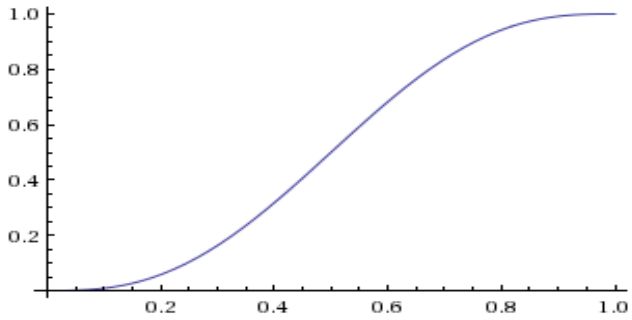


Figure 6: Blending function $f(t) = 6t^5-15t^4+10t^3$ used in Perlin noise, visualized in WolframAlpha.

Figure 7: Visual representation of Perlin noise. The value of -1 is marked as black and 1 as white.

The noise function can have more dimensions. For example 2 dimesnions, as Figure 7 demonstrates, or any arbitrary number, given there's enough computational power. Similarly to the 1-dimensional variant, 2-dimensional Perlin noise sets it's value to 0 in all of its integer coordinates.

3.4 Simplex noise

Simplex noise is also a form of gradient noise. Therefore its properties are quite similar to Perlin noise, as can be seen on Figure 8. As its name would suggest, simplex noise makes use of geometric entities called simplices. [1]

Formally, an n-dimensional simplex is a convex n-dimensional polytope. A simplex is called a regular simplex if it is a regular polytope, which means it that it has vertices of equal length. Simply put, a point, a line, a triangle and tetrahedron are respectively 0, 1, 2 and 3-dimensional simplices. [8]

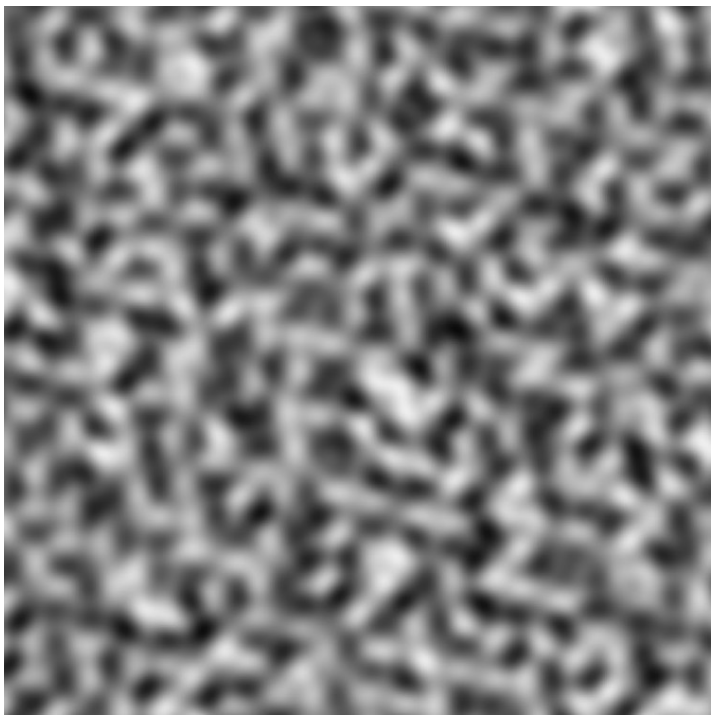


Figure 8: Visual representation of simplex noise. The value of -1 is marked as black and 1 as white.

When used in simplex noise, a grid is constructed using regular simplices to fill an n-dimensional space. That might be the most apparent difference between Perlin noise and simplex noise. For example, similarly to how Perlin noise would divide 2d space into squares in its 2-dimensional variant, simplex noise divides it into triangles. [1]

This means that simplex noise, in its 2-dimensional variant, has noise contributions from 3 points instead of 4, which is the case with Perlin noise. This scales with additional added dimensions. 3 and 4-dimensional simplex noise respectively needs to calculate contributions from 4 and 5 points, while perlin noise would respectively need 8 and 16. This is one of the properties that makes simplex noise less demanding to calculate. [1]

Another difference is how the noise contributions from several points are mixed together. While Perlin noise uses sequential interpolation along each dimension, simplex noise uses direct summation. This again has a speed advantage thanks to using summation instead of multiplication. It should be noted, however, that this also leads to the properties of the noise changing as the number of dimensions is changed. A 2-dimensional projection from 3d simplex noise does not look identical to plain 2-dimensional noise. [1]

3.5 Gradient noise use cases

A common use for gradient noise is the generation of textures and animations. Gradient noise can be manipulated by various functions to generate a variety of interesting visuals. Such as is displayed in Figure 9. [2]

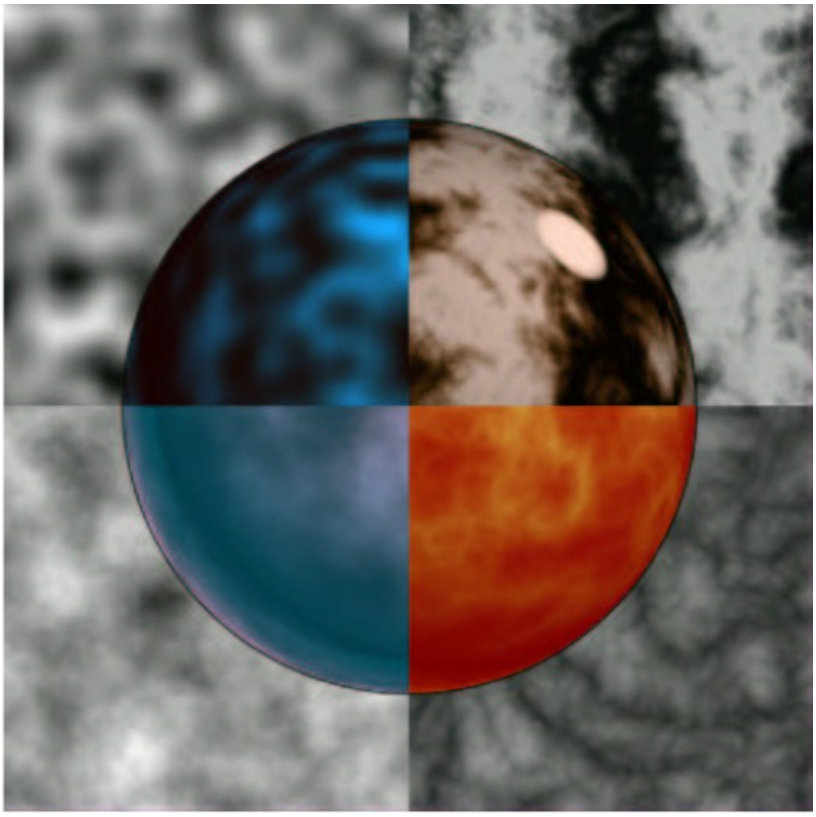


Figure 9: Different textures generated using noise. [2]

On the example in Figure 9, the textures are calculated using the following formulas.

- Upper left: raw noise
- Upper right: `sin(x + sum 1 / f(||noise))`
- Lower left: `sum(1 / f(noise))`
- Lower right: `sum 1 / f(|noise|)`

The scope of this thesis also contains texture generation with several methods of automation. How it was implemented in the library is covered in the implementation chapter.

Another possible use for gradient noise is terrain generation. It is possible to replicate many natural patterns using it. Terrain generated in this fashion can, for example, be used in games. Figure 10 displays an example of terrain generated using Perlin noise in a terrain generation program called RWG by Erich Erstu. [3]

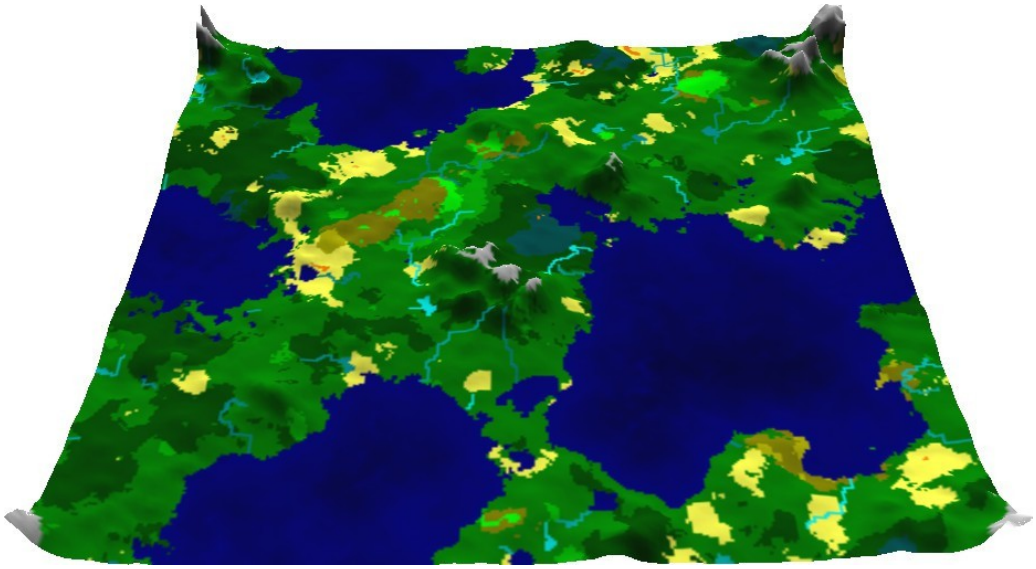


Figure 10: Terrain generated using terrain generation program RWG that makes use of Perlin noise, visualized by Molehill Heightmap Viewer. [3]

The next chapter explains my implementation and parallelization of the simplex noise algorithm. Changes to the original code are shown. An overview is given of the features the library has and how they were implemented.

4 Implementation

4.1 Overview

The implementation of the library is written in the Java programming language. Java was chosen for two main reasons. First, the implementation by Stefan Gustavson, which is described in his paper, is written in Java [1]. His code is used as the basis for noise generation in the library. Second, there is a very suitable library for parallelization for Java, which allows Java code to be executed on a GPU or fall back to a CPU, if a GPU is not available. That library is called Aparapi. The next section gives more detail about how it was used.

The following sections describe the process of implementing the library in more detail. The parallelization technique, optimization and graphical solutions are covered.

4.2 Parallelization

This section describes the steps taken for parallelizing the simplex noise calculation. Several changes had to be done to the code in order to make the calculation parallel and an external parallelization library was used.

A library called Aparapi was used in implementing the library. It is a library that is capable of translating Java bytecode to OpenCL at runtime [1]. Therefore, none of the Java code had to be translated to OpenCL manually.

However, Aparapi does have some drawbacks that have to be considered when using it. A portion of the noise generation code had to be modified to overcome these shortcomings.

One of the notable issues is the lack of most object-oriented approaches to programming. Unlike Java, OpenCL does not support classes. OpenCL does support structs, but those do not have a compatible memory layout with Java classes. This causes Aparapi to only support using arrays of very simple objects. Those objects may have primitive fields and setters and getters. Using those also comes with a performance penalty as Aparapi has to translate them into an appropriate form on each call. [7]

There was one aspect in the original simplex noise code that was object oriented. The gradients used for noise generation were stored in special objects that held data for a gradient vector. Those objects were changed into arrays of floats. It was decided that using Aparapis support for simple objects was not necessary in this case. There were 2 reasons for that. First, it was decided that the performance loss was not worth the effort to make the code more readable. Secondly, the end user of the library does not get access to the classes that directly call the Aparapi library. Therefore, the less easily readable primitive arrays do not have to be manipulated by the users.

Another feature Aparapi does not have is multi-dimensional arrays [7]. Although the simplex noise algorithm itself in its original form did not make of multi-dimensional arrays, that drawback was a hindrance in later use of the calculated noise.

For example, it might be more convenient for the user of the library if the calculated noise came in

an array that has as many dimensions as the desired noise. A 2-dimensional surface cut out from 2-dimensional noise could be most convenient to have in an array with 2 dimensions, for instance.

The only solution was to calculate the noise on the GPU using Aparapi, which stores the data in a 1-dimensional array, and later construct a multi-dimensional array on the CPU. However, this proved to be a serious performance bottleneck. In some cases, where noise calculation was less computationally expensive than average, constructing the appropriate array could take even longer than the noise calculation process itself. For this reason, the library offer the user an ability to either get the noise in a multi-dimensional array or get the raw 1-dimensional array faster.

The parallelized version of the code was shown to be significantly faster when executed on the GPU compared to the original CPU implementation. The exact extent is shown later in the performance tests chapter.

It was possible to make the code even faster in some cases by further optimization. This is covered in the next section.

4.3 Optimization

Although more thorough analysis would have to be done to the code to state it conclusively, the performance of simplex noise generation seems to be bottlenecked by memory bandwidth. Some information about this is given later in the benchmarking chapter.

With that assumption, the code was modified to use less lookup tables. The original code had a table of shuffled integers between 0 and 255 which was used for randomly picking gradients when calculating the noise, and a second array of integers between 0 and 11.

The use of the first table was replaced in most cases with a function that calculates a type of noise called integer noise. Whereas previously the noise code could, for example, take a value from the array from index 77, after the modification it would give this 77 as an argument to the integer noise function. This would also return a value between 0 and 255.

Given the integer noise function was modified significantly, it might not be accurate to call it integer noise any longer, but for the sake of simplicity, from now on the code shown in Figure 11 is referred to as integer noise.

```
private int intNoise(int n) {
	n = ((n + 463856334) >> 13) ^ (n + 575656768);
	return ((n * (n * n * 60493 + 19990303) + 1376312589) & 0x7fffffff & 255);
}
```

Figure 11: Code of the modified version of integer noise.

Replacing the arrays does produce a difference between the optimized and original noise, however. It is visually notable, that the optimized noise can produce more repetitive patterns than the original.

The cause of this could to be, that the original arrays consist of an equal amount of each number that never repeats in succession. The first array consists of 512 numbers, 2 of each number between

0 and 255. Integer noise can, however, produce repeating numbers in succession. It could be argued that in that sense integer noise is *more random,* as it does not prohibit any combination of numbers showing up in succession. However, visual perception might state the opposite.

More analysis would have to be done to state the reasons in more detail, which is beyond the scope of this paper.

For this reason the second array was not removed in favor of using integer noise. Replacing that array proved to produce a lot of those repeating visual features. Replacing just the first array was a good balance between performance and visual properties, as the second array is rarely used in the noise code anyway. It is subjective, but this middle-ground variant seems to produce noise that is almost indistinguishable from the original, as can be observed from the following images on Figure 12.

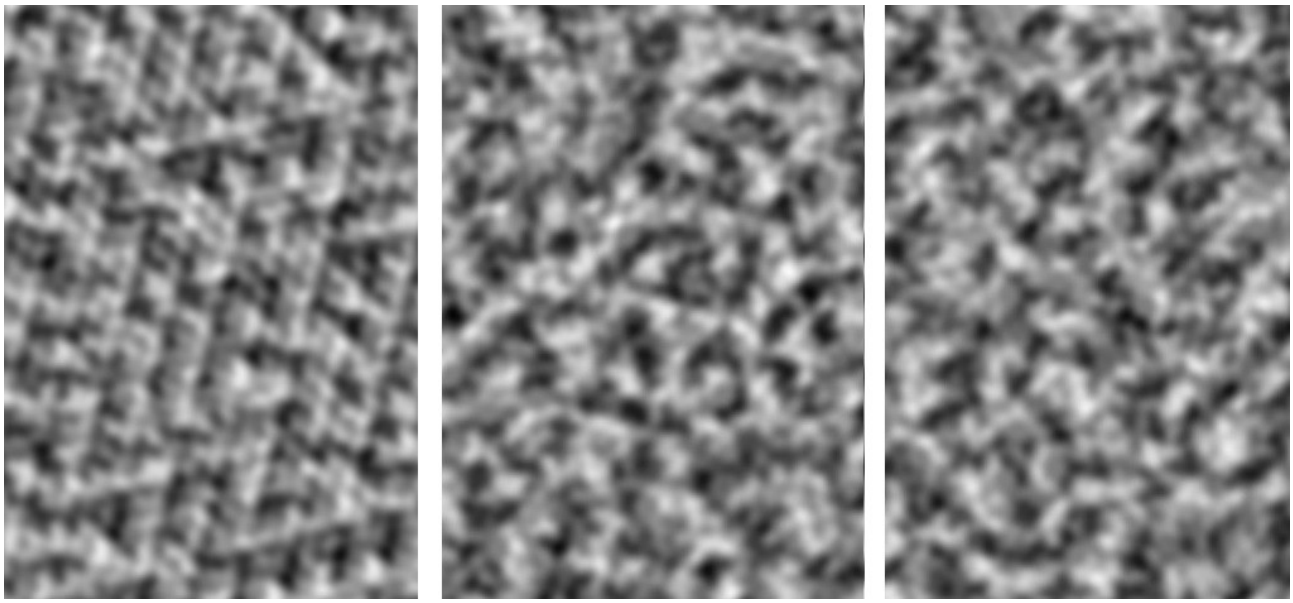


Figure 12: Visualization of full integer noise use, first array replaced with integer noise and original respectively.

However, the library still gives the user an option to use the original implementation. The original could either be used because of visual preference or to produce identical results to the original unparallelized code, which could be needed in applications that already use Stefan Gustavson's implementation.

The performance gains from this seem to vary between different systems and noise with different parameters. The gains were higher on integrated graphics cards. This was expected, because those have to use system memory instead of higher bandwidth graphics memory. The gains were also more prominent when calculating noise with multiple octaves. More exact information about this is given in the benchmarking chapter.

4.4 Color transformations

One possible use case for simplex noise is graphics generation. For this purpose, the library offers a system for transforming an array of noise values to an RGB array or a bitmap.

The default is to visualize noise as a grayscale image as was seen in Figure 12 in the previous section or on Figure 13.

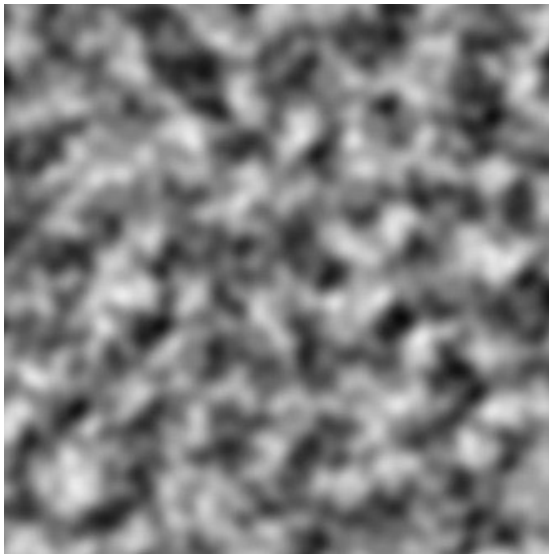


Figure 13: Visual representation of simplex noise in grayscale.

The same color scheme seen on Figure 13 can be achieved manually, by calling the following method on a ColorMapper instance that Figure 14 shows.

```
colormapper.addRange(-1, 1, Color.BLACK, Color.WHITE, 1);
```

Figure 14: Example of a color range declaration.

In this code example the -1 represents the lower limit of the noise and 1 the upper limit of the noise that gets affected by this color range. The third argument's color is assigned to the lower limit and the fourth to the upper limit. The last argument represents the contrast modifier.

The contrast argument can be manipulated to increase or decrease contrast of the output image as is shown on Figure 15.

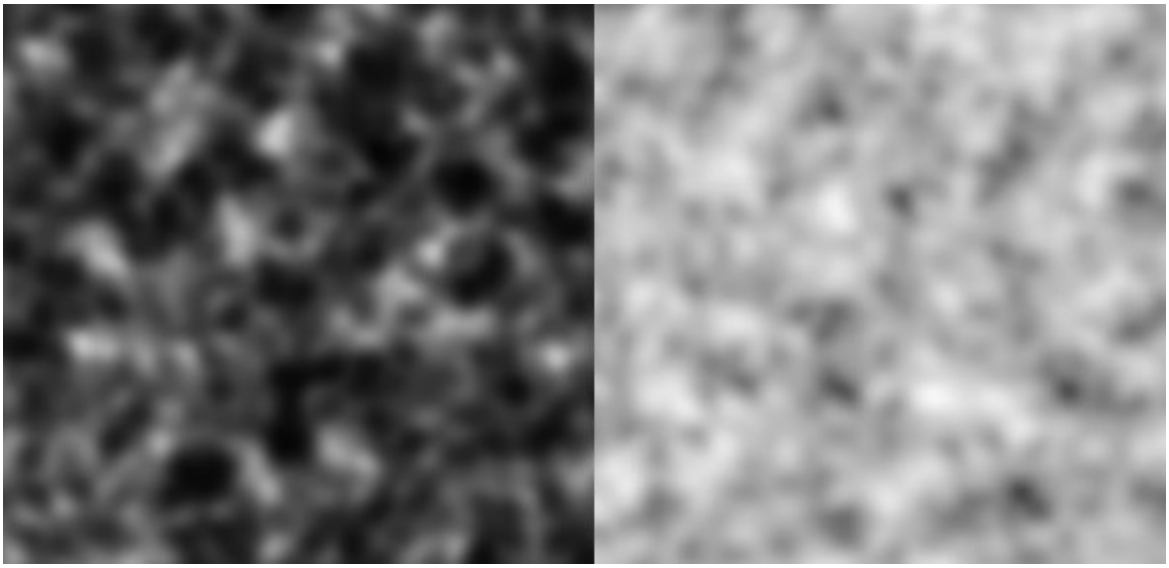


Figure 15: Images of noise with high and low contrast modifiers respectively.

To override the default, the library offers a method for declaring a specific value interval of the noise to be assigned to a specific color range. It takes 3 floating point numbers and 2 color objects to describe a color interval. 2 floating point numbers to declare the lower and upper limit of the colored noise value and another floating point number for its contrast. The color objects declare colors assigned to the lowest and highest value of noise.

These color ranges can be used to generate various visual effects. For example, as shown on Figure 16, one could define 5 color ranges as follows to create a terrain-like image. Define noise values from -1 to 0 to go from dark blue to light blue. 0 to 0.15 from light blue to yellow and so on with colors seen on the image.

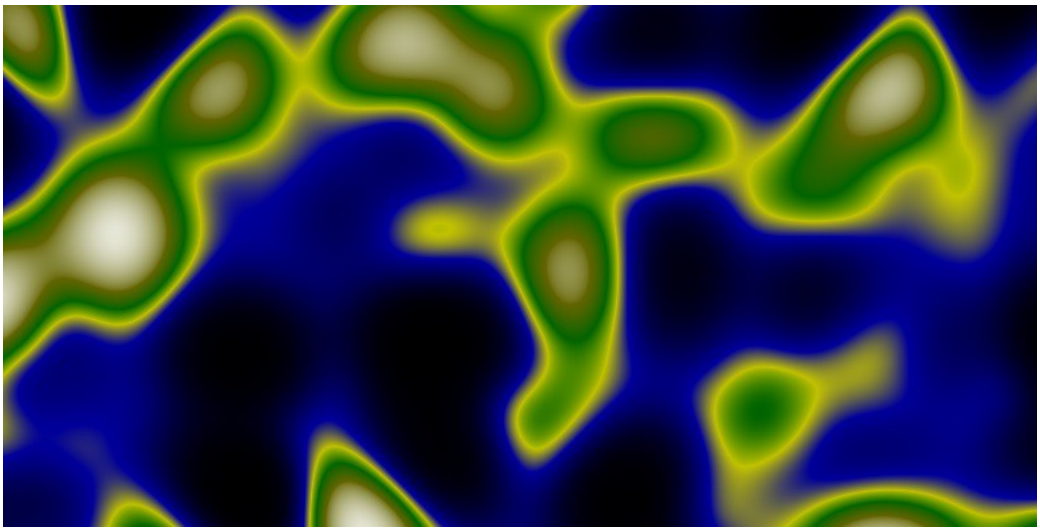


Figure 16: Color ranges defined to generate a terrain-like visualization.

It could be subjectively said, that the previous terrain looks bland. It only has uniform features all around with nothing unique. A way to make it more varied is described on the next section.

4.5 Noise with multiple octaves

Noise with varying frequencies can be summed to create a multi-octave noise. Those can in some situations have desired properties. The library offers an easy way to achieve it. The user can specify the base frequency and the number of upper octaves. It can also be specified, how much stronger or weaker the contribution from the next octave is. The sum of the added octaves is normalized down to the range between -1 and 1, so it can be used for image generation identically to one octave noise.

As mentioned in the end of the previous section, using multiple octaves could, for example be used to generate terrain with more interesting features.

It can be seen from Figure 17, that the lower base frequencies of the noise create larger features of the terrain, such as the larger islands and the sea. The upper octaves with higher frequency generate smaller features such as small islands and lakes. Adding octaves also causes the noise to look more detailed when zoomed in close.

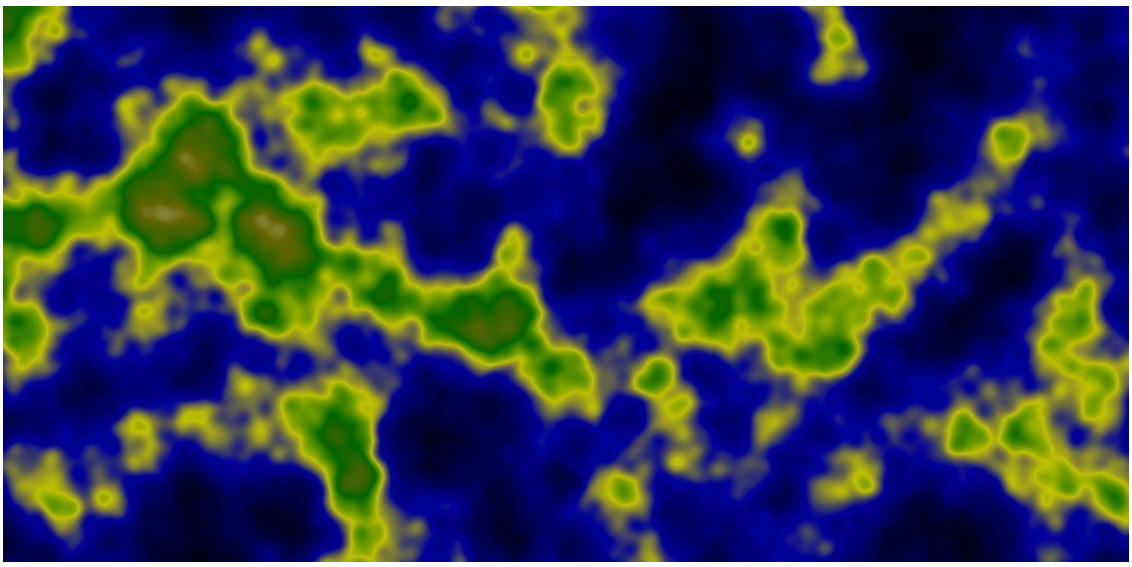


Figure 17: Visualization of terrain generated from 4 octave noise.

4.6 Multidimensional noise

The library supports calculating 2, 3 and 4-dimensional simplex noise. There are 3 ways the user can request those multiple dimensions to be handled.

One way is to getting an n-dimensional array. For example, 2-dimensional noise can be stored in a 2 dimensional array and so on for higher dimensions. There are separate methods for calculating such an array for each 2, 3 and 4-dimensional noise accordingly. The library also has methods for getting that array in its raw form, as it is calculated on the Aparapi kernel. This means all the data is in a one-dimensional array, although it still represents the same n-dimensional noise.

Another way is getting a flat 2-dimensional surface from 3 or 4-dimensional noise. This means the user can declare the x, y, z and possibly w coordinate and a width and height for the surface. X and y directly affect the dimensions into which the width and height extend. Z and w can be used as additional parameters, that make several useful things possible.

One possible example is animation. When using 3 dimensional noise, the third z dimension could be used to represent the progression of time. For example, the x and y could both be constant and z could be changed over time, as shown on Figure 18. Figure 18 shows gradual change to the surface as the z coordinate is slightly altered.

A possible addition would be to make the z parameter oscillate between certain values. This would create a seamless endless animation. In the example of Figure 18, this could be used to animate waves in the ocean covering different parts of the land on different times. However, it would be noticeable, that the animation keeps reversing periodically.

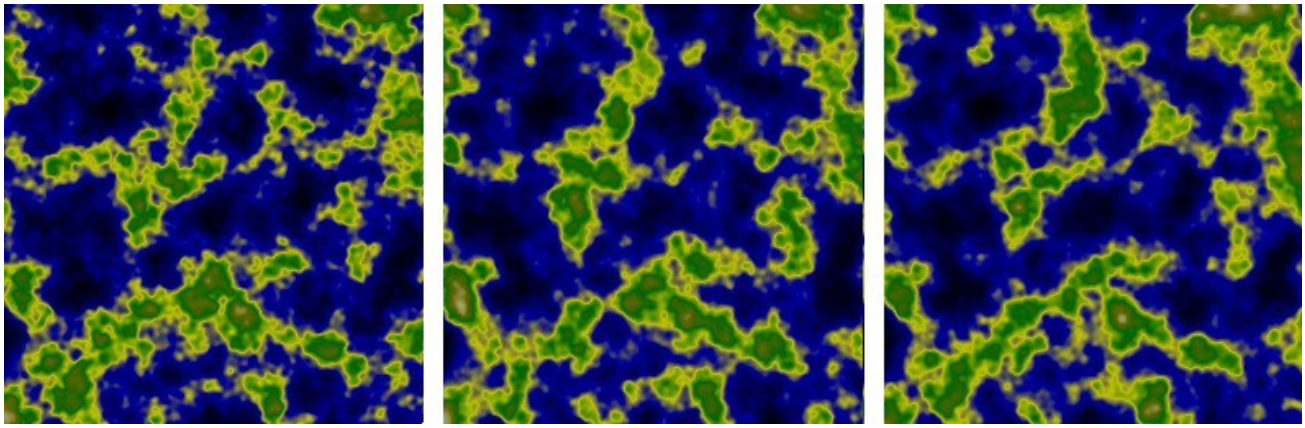


Figure 18: Visualization of changing the z parameter of 3d noise when picking a 2d subsurface at 3 different time points.

There is a way to overcome this periodic reversing while generating seamless repeating animations. Unintuitively, more than one dimension can be used to represent the progression of time. Time in the traditional sense could be represented by a single dimension, but if a second dimension can be used, it would be possible to, in a sense, *make a circle* in this 2-dimensional time, as is shown on Figure 19. This avoids reversing through the same time coordinates when making the oscillation.

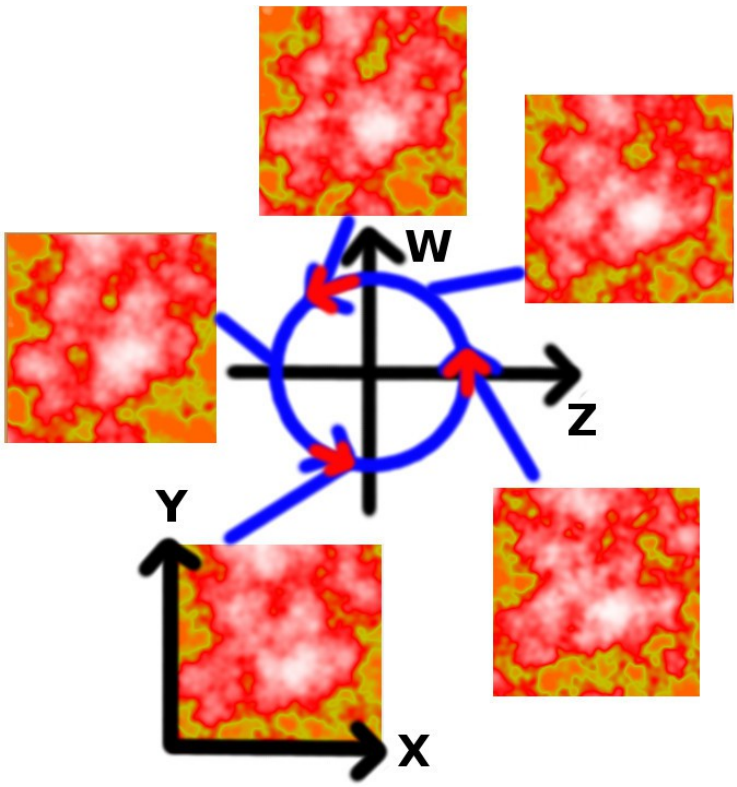


Figure 19: Visualization of 2d surfaces with constant x and y and 2 changing time coordinates z and w.

Another use for higher dimensions is creating seamlessly tiling graphics. This means that if identical copies of a surface are placed next to each other, the edges would be identical and fit together, as Figure 20 illustrates.

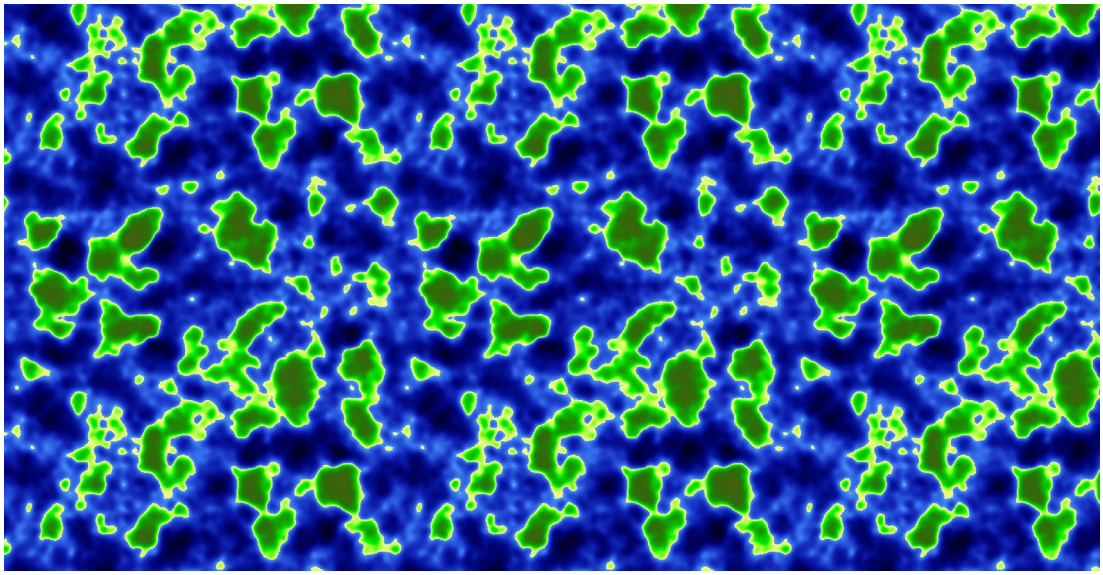


Figure 20: Visualization of seamlessly tiling noise.

The easiest to explain example is seamless 1-dimensional noise. When choosing a random interval from 1-dimensional noise, such as is illustrated on Figure 21, it is highly probable that the end points would have different values and derivatives, therefore they would not create a seamless transition when tiled.

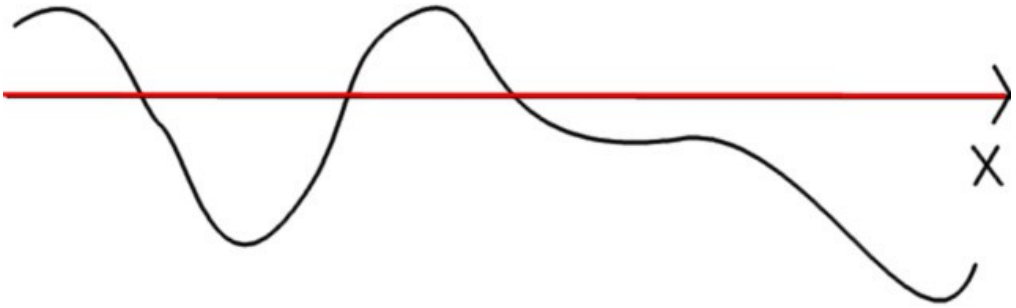


Figure 21: Visualization of 1-dimensional noise.

However, 2-dimensional noise can be used to create a seamlessly tiling 1-dimensional noise interval. This can be done by picking a circular pattern from 2-dimensional noise, as Figure 22 demonstrates.

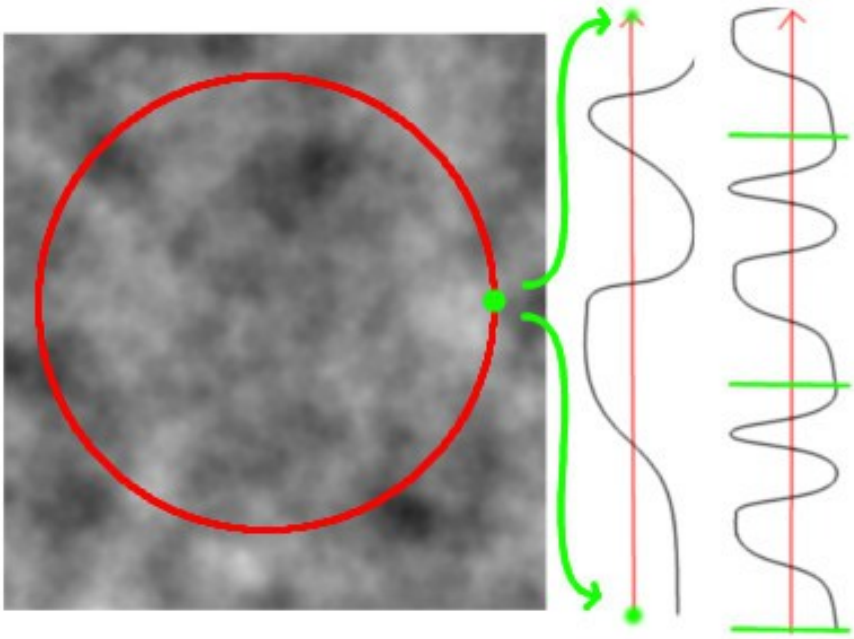


Figure 22: Transformation of a circular shape in 2d noise to a 1-dimensional tiling noise wave.

The same principle can be generalized to higher dimensions. For instance, the analogue for a circle in 3-dimensional space would be a cylinder in this use case. Intuitively, a cylinder could be used to create a 2-dimensional surface, that tiles on one of its edges, as a cylinders surface area is essentially a rectangle.

Making a 2-dimensional surface tile both vertically and horizontally was a bit more complex. This requires the use of 4-dimensional noise, which is hard to visualize, but as a concept it is not much different from 2 or 3-dimensional noise. Its analogue to the 3d cylinder is somewhat similar to a torus shape, or a doughnut, with the main difference that a torus does not have a rectangle shaped surface, but the 4-dimensional variant does. In Figure 20, it can be observed, that the terrain tiles both vertically and horizontally – this was achieved by the same method of using 4-dimensional space.

5 Benchmarking

5.1 Parallelization performance tests

This chapter determines how large the large performance gains were. It compares the original noise function from Stefan Gustavson, which is run on a CPU, to the modified parallel version which can utilize a GPU.

Six different PC's were tested. Four of them desktops and two laptops.

The tested hardware specifications were as follows.

- Desktop 1
  - CPU: Intel i7-4770
  - GPU: Nvidia GTX 770
- Desktop 2
  - CPU: Intel i3-4130
  - GPU: Nvidia GTX 660
- Desktop 3
  - CPU: Intel i5-4570

    GPU: AMD 280X
- Desktop 4
  - CPU: AMD Athlon II X4 640
  - GPU: AMD HD6670
- Laptop 1
  - CPU: Intel i5-3210M
  - GPU: Intel HD4000
- Laptop 2
  - CPU: Intel i7-4710MQ
  - GPU: Nvidia 860M

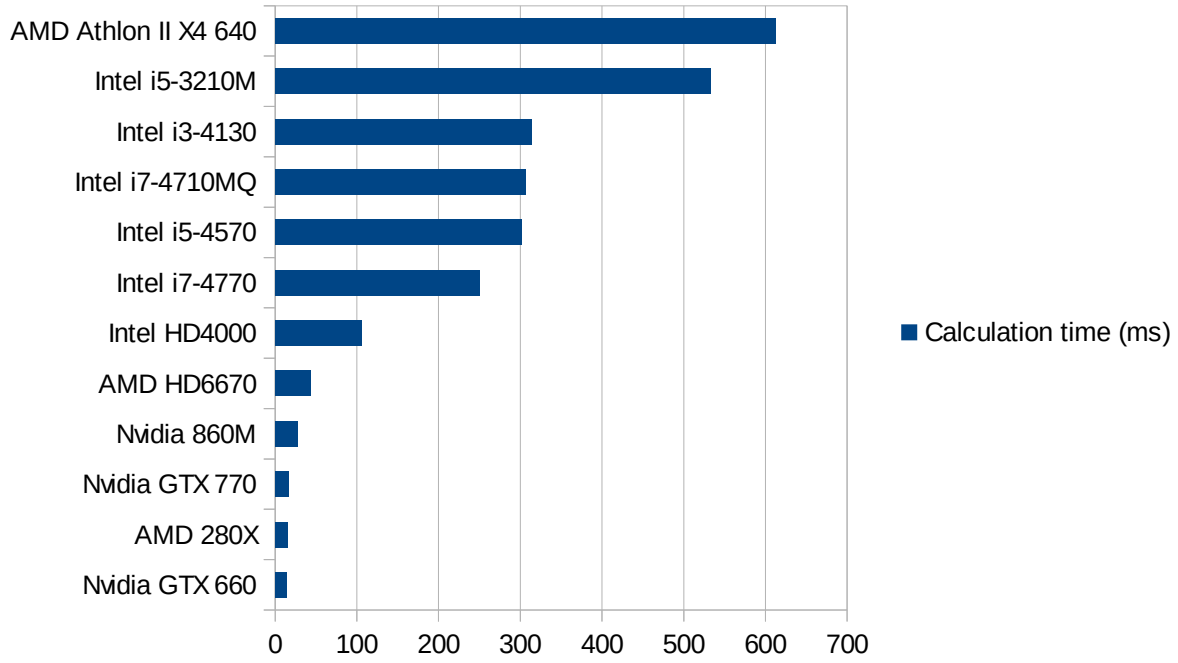


Figure 23: Times that different pieces of hardware took to calculate the test data.

The test consisted of calculating 3,686,400 points of Simplex noise in 4-dimensional space. Same for both the CPU and GPU. The test was run 10 times on each platform and the average of the results was used.

The results, as seen on Figure 23, conclude that the extent of the performance gain can be well over an order of magnitude for the GPU's. Even when comparing the most powerful tested CPU, the Intel i7-4770, and least powerful integrated GPU, the Intel HD4000, the GPU is still over two times faster.

An interesting fact was that the top tier graphics cards had similar performance. In raw gflop/s performance stated by its specifications, the GTX 660 should for example actually fall behind the GTX 770 and 280X.

The deciding factor might be something other than raw performance. Most likely, that would be memory bandwidth. Some evidence in favor of that is shown on the next section.

5.2 Optimization performance tests

This chapter describes the final optimized code's performance against the unmodified, but parallelized version and also the original implementation for the CPU. This was measured by making different pieces of hardware calculate noise that consists of a variable number of octaves.

The test were run on 3 different computers. 2 of them desktops and 1 laptop.

- Desktop 1
  - CPU: Intel i7-4770
  - GPU: Nvidia GTX 770

- Desktop 2
  - CPU: Intel i3-4130
  - GPU: Nvidia GTX 660
- Laptop 1
  - CPU: Intel i5-3210M
  - GPU: Intel HD4000

The test were run 10 times and their average was calculated. A single test consisted of calculating 3,686,400 points of simplex noise for each octave. That number of points can be multiplied by the number of octaves to get the total amount of calculation needed for each iteration, as table 1 shows. Octave numbers of 1, 3, 5, 7, 10, 30 and 100 were calculated.

However, it must be mentioned that this only has benchmarking value. In real use such high octave number are not particularly useful – about 10 would work in a majority of cases in real world use.

Table 1: The number of noise function calls a number of octaves requires.

Number of octaves	Calls to the noise function
1	3,686,400
3	11,059,200
5	18,432,000
7	25,804,800
10	36,864,000
30	110,592,000
100	368,640,000

Figure 24 shows the performance of the 2 higher end GPUs in the test systems and Figure 25 adds an integrated GPU to the comparison. It shows them separately using the unoptimized and optimized version of the noise algorithm.

In can be concluded, that the optimized version can in some cases have up to double the performance. This is notable in the higher and medium numbers of octaves in higher end GPUs and low and medium numbers of octaves for the HD4000.

It is also noticeable, that the computation time increases slower than linearly when additional octaves are added. A probable cause for this is that the computation is bounded by memory bandwidth and the additional amount of computation the octaves require do not have full effect.

In the HD4000's case, it can be noticed that the optimized algorithm starts to take as much time as the original for the highest octave number. It is possible, that at this point the computation required

is large enough to overcome the memory bandwidth bottleneck and therefore the memory use optimization has no effect.

Figures 26 and 27 compare the CPU performance to GPUs. Expectedly it is a lot slower and scales linearly with added octaves.

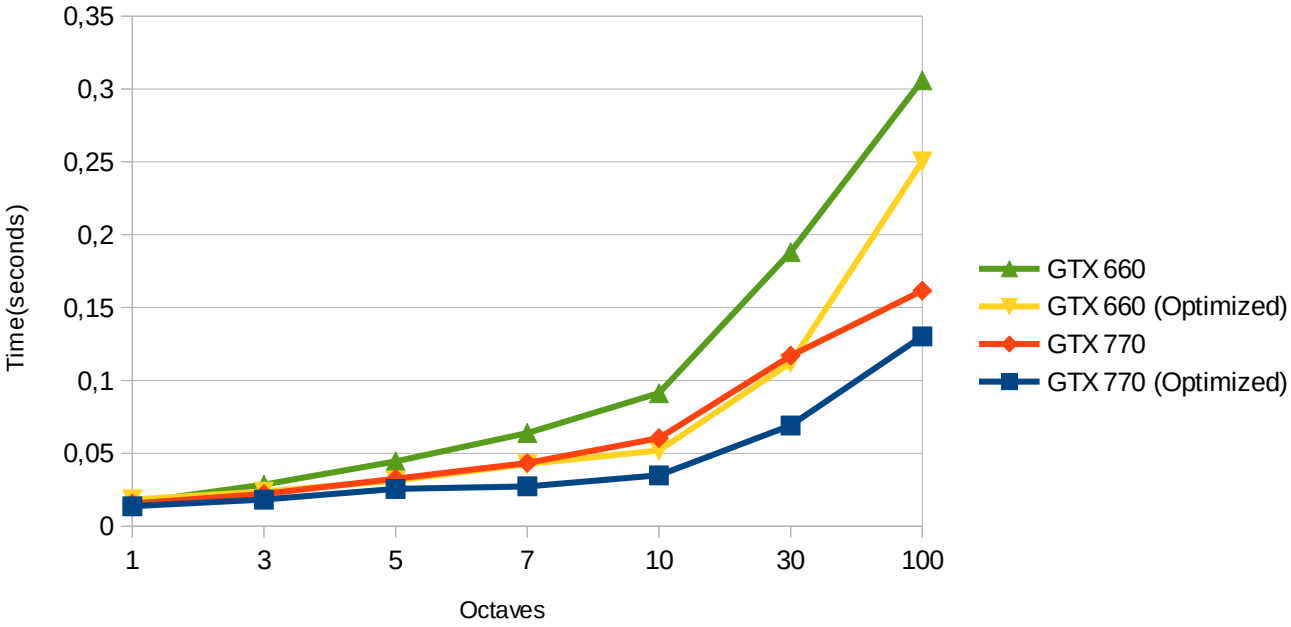


Figure 24: Higher end GPUs measured with the optimized and original algorithm.

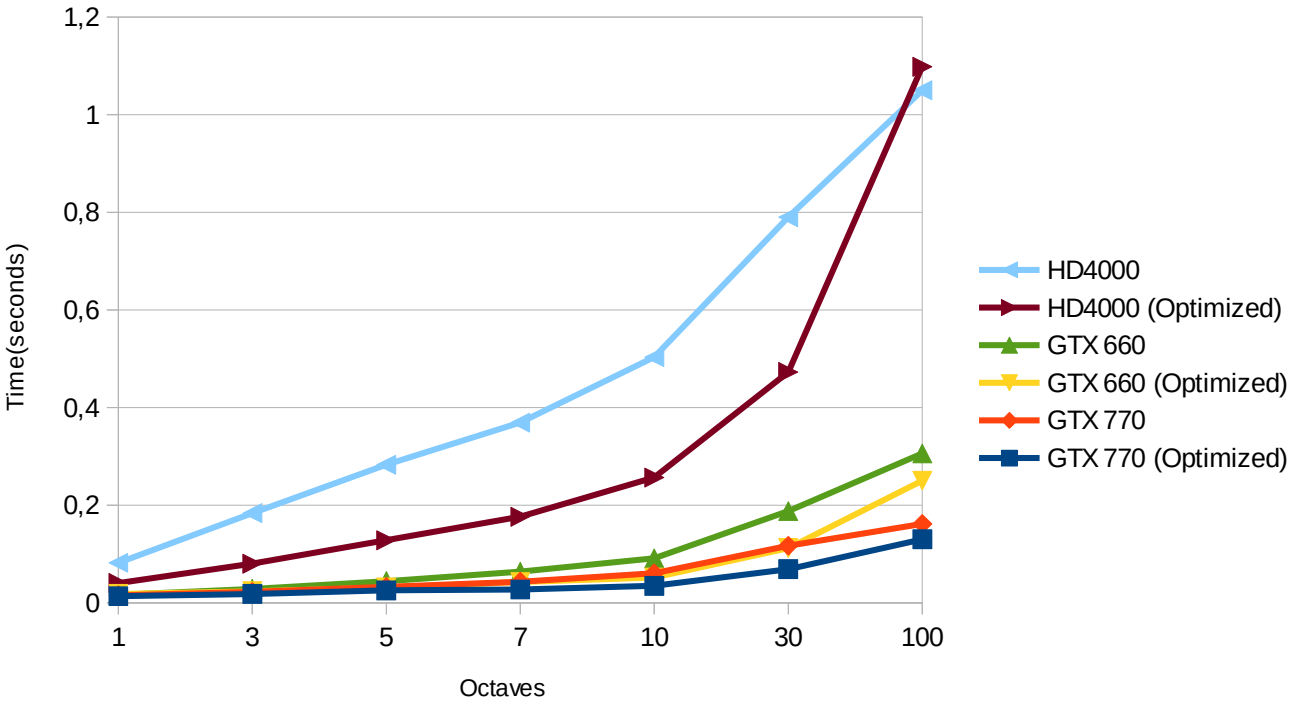


Figure 25: Integrated HD4000 graphics card with the optimized and original algorithm in comparison to higher end graphics cards.

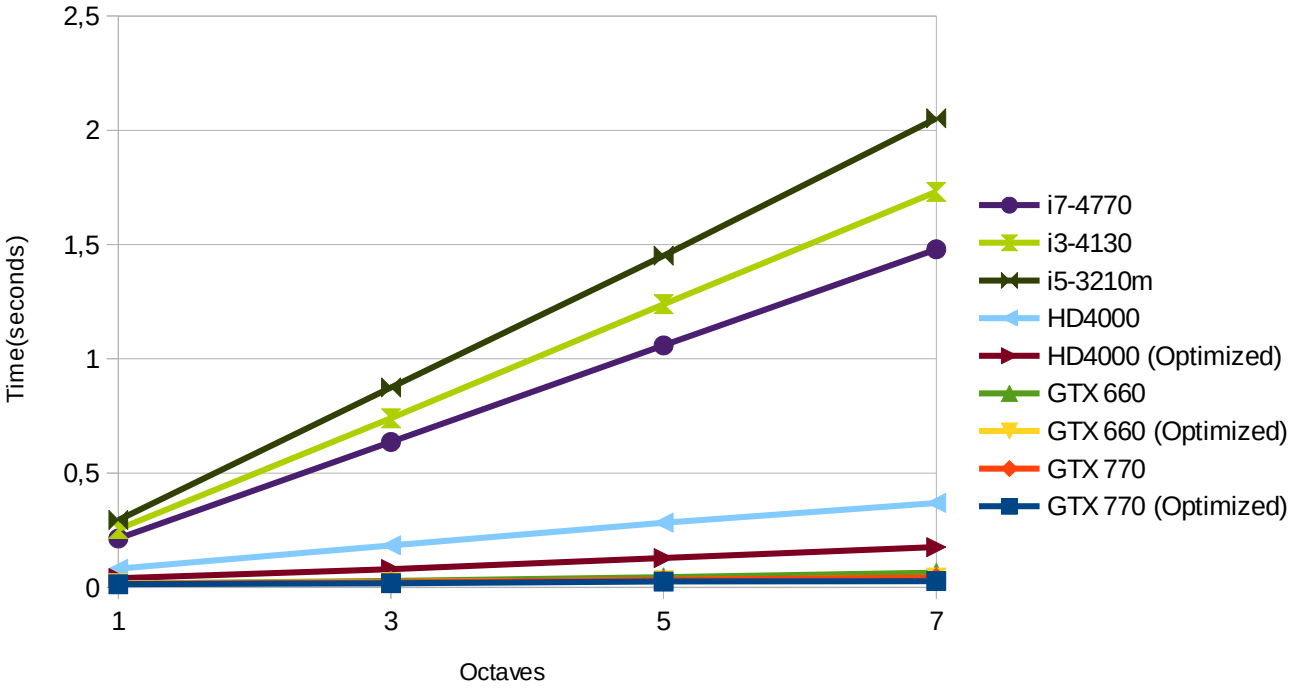


Figure 26: CPUs performance compared to GPUs for lower octave numbers.

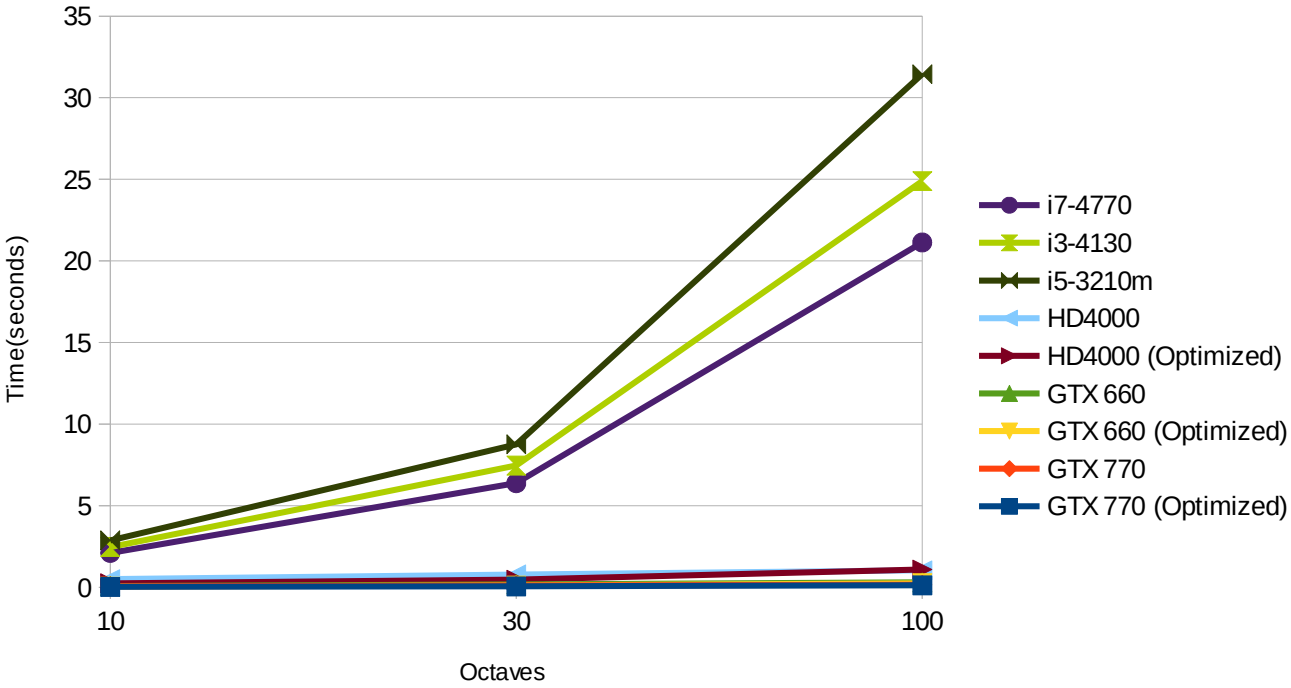


Figure 27: CPUs performance compared to GPUs for higher octave numbers.

6 Conclusions

As a result of the thesis, a Java library was written, which performs parallel computation of simplex noise. The library is able to utilize a GPU if it is present and fall back to the CPU if a GPU is not available. It allows the user to specify various parameters, which enhance the properties of the calculated noise. These include the number of dimensions, coordinates, the calculated area's length along each axis, frequency, the number of higher sub-frequencies/octaves and their weighted contribution to the resulting sum of frequencies. The end result of noise generation is an array of floating point numbers, which represents the noise.

The library also offers several methods for visualizing the generated noise. It is possible to assign color parameters to the visualization. Those determine which noise value intervals are transformed to a specific color interval and the contrast of the color gradient can be manipulated. The end result can either be received as an array of RGB data or as a Java BufferedImage object.

One measured property was how much faster Stefan Gustavson's original simplex noise implementation was on a GPU without modifications to the principle of the algorithm. The performance gains from parallelization were significant.

Another benchmark tested, how well the parallelized code scales, when noise with a high number of octaves is calculated. It was concluded that the number of calculated noise octaves had little impact on the performance, if below 3 digit numbers, which is far more that normally necessary. That would imply computational power is not the bottleneck for that task.

This scaling was also compared to multiple octave calculation on the CPU, which took a linear amount of time to calculate additional octaves. This means, that with more calculated octaves, the performance gain from parallelization increases.

In the future, several possible additions could be done to the library. It could be analyzed, whether generating RGB arrays from noise arrays would also benefit from parallel computing. If that is the case, generating Java's buffered images could be made faster. Another possible addition would be to make the noise depend on parameters that are not always constant in all coordinates. For example the contribution from higher frequencies could be different in different coordinates, which would create more variety in the noise.

7 References

[1] Gustavson, Stefan. "Simplex noise demystified." *Linköping University, Linköping, Sweden, Research Report* (2005).

[2] Perlin, Ken. "Noise hardware." *Real-Time Shading SIGGRAPH Course Notes* (2001).

[3] Erstu, Erich. *Maakaartide juhugeneraator* Diss. Tartu Ülikool, 2012.

[4] Keckler, Stephen W., et al. "GPUs and the future of parallel computing." *IEEE Micro* 31.5 (2011): 7-17.

[5] Aparapi, parallelization library, https://code.google.com/p/aparapi/ *

[6] Aparapi, parallelization library FAQ, https://code.google.com/p/aparapi/wiki/FrequentlyAskedQuestions

[7] Aparapi, parallelization library Kernel guidelines, https://code.google.com/p/aparapi/wiki/JavaKernelGuidelines

[8] Wikipedia, Simplex definition, http://en.wikipedia.org/wiki/Simplex

[9] Libnoise, noise library, http://libnoise.sourceforge.net/noisegen/index.html

* All the mentioned web pages were last visited on 11.05.2015.

8 Appendix

8.1 Repository

The source code can be found in the following repository.

https://github.com/JaanJanno/libjsimplex

8.2 Use example

The flow of using the library for an user in need of a texture can be as follows.

Use case: The user needs to generate a cloudy sky texture for his Java program. He intends to store the image as an instance of BufferedImage.

Step 1: The user downloads the library from the repository.

Step 2: He imports the library in a way specific to his preferred IDE.

Step 3: He creates a new ColorMapper instance as follows, which can be used for creating color ranges.

```
ColorMapper m = new ColorMapper();
```

Step 4: He assigns that noise with value between -1 and 0.5 has to be a blue gradient from the RGB value of (55,55,105) to (195,195,195), or blue to grey. Similarly, for noise value of 0.5 to 1, the gradient is from (195,195,195) to (255,255,255), or grey to white. This can be declared to the ColorMapper instance as follows.

```
m.addRange(-1,0.5f, new Color(55,55,105), new Color(195,195,195), 1);
m.addRange(0.5f,1, new Color(195,195,195), new Color(255,255,255), 1);
```

Step 5: He calls the following method from NoiseSurface class and receives an array of noise.

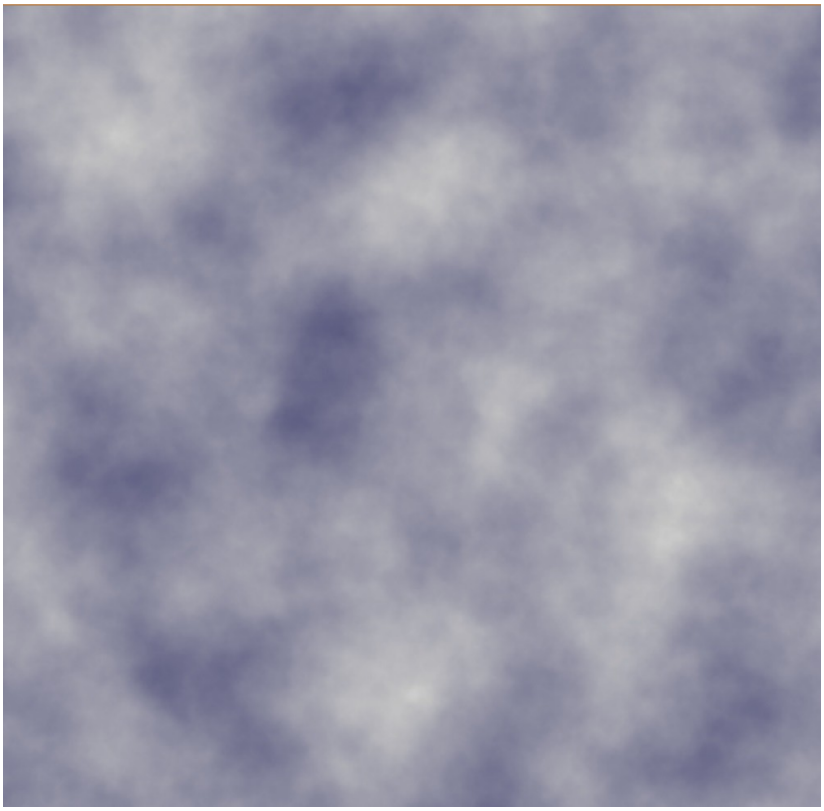
```
// Position in the noise space.
float xPosition = 0, yPosition = 0;
// How large the main patterns of the noise are.
float frequency = 0.003f;
// How large of an effect smaller frequencies have.
float persistence = 0.5f;
// Number of lower frequencies.
int octaves = 50;
// How many points of noise we need along a dimension.
int width = 1024, height = 1024;
// Determines if the library uses the optimized noise functions.
boolean fast = true;

float[] noise = NoiseSurface.generate2dRawOctaved(xPosition, yPosition, width,
      height, frequency, persistence, octaves, fast);
```

Step 6: The ColorMapper instance can be used to transform this array of noise into a BufferedImage as follows.

```
int width = 1024;
int height = 1024;
BufferedImage image = m.getBufferedImage(noise, width, height);
```

Step 6: The user can use the BufferedImage in his program according to his needs. This example produces an image as follows.



8.3 License

Non-exclusive licence to reproduce thesis and make thesis public

I, **Jaan Janno** (date of birth: 30.12.1991),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

    b. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

    c. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

of my thesis

Java library for parallel computing of simplex noise,

supervised by Professor Eero Vainikko,

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **11.5.2015**