Perlin Noise Generator

Erich Erstu, Janar Sell, Suido Valli

Introduction

One of the most fascinating fields in computer science is graphics. Especially computer generated graphics. The visual world we see every day is strongly influenced by the fractal aspect of nature. In other words, there are visible patterns everywhere [1].

In figures 1.1 and 1.2 two patterns are shown. However, the latter one is computer generated. This demonstrates perfectly the use of computer generated textures. These textures can then be used to paint the visual world in a computer game or to simply provide interesting graphics for any user interface.

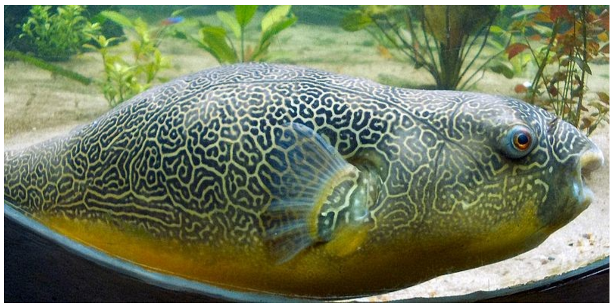
Fig 1.1. Elaborate skin pattern of Giant Freshwater Puffer fish [1]

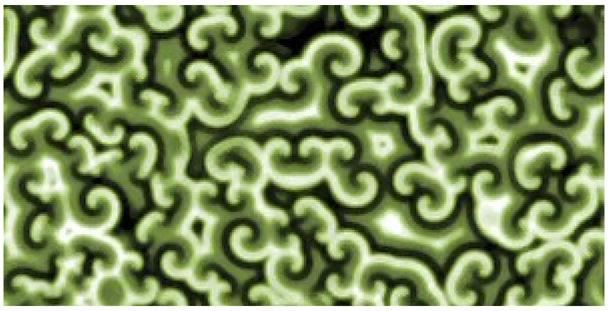
Fig 1.2. Snapshot from computer simulation of Belousov-Zhabotinsky reaction showing dynamic pattern as waves of reaction spread through the reagent mixture [1]

What is more, computer generated textures can not only be used to visualize the natural aspects of the world but to actually describe it as something physical. Something like the solid form of a planet. In terms of computer graphics this refers to a heightmap (see figure 1.3) [2].

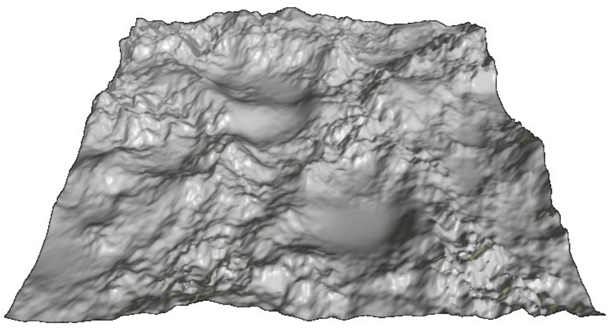
Fig 1.3. A visual image of a typical heightmap, rendered with Anim8or and no textures [2]

Finally, as raster data consumes a lot of memory it is not always optimal to store the data that only represents randomly generated textures. Instead these textures could be generated on demand which exposes the concept of procedural generation [3].

Generation of such textures is rather slow but turns out that it could be optimized by the means of distributed computing. In the next paragraph Perlin noise (one of the most common methods of computational texture generation) is explained. Based on that a working solution for this is given.

1. Perlin noise

The concept of Perlin noise is rather simple but the uses have a really wide range. It all begins with having a texture that contains random noise. This can be achieved with common pseudo random number generators (see figure 2.1a).

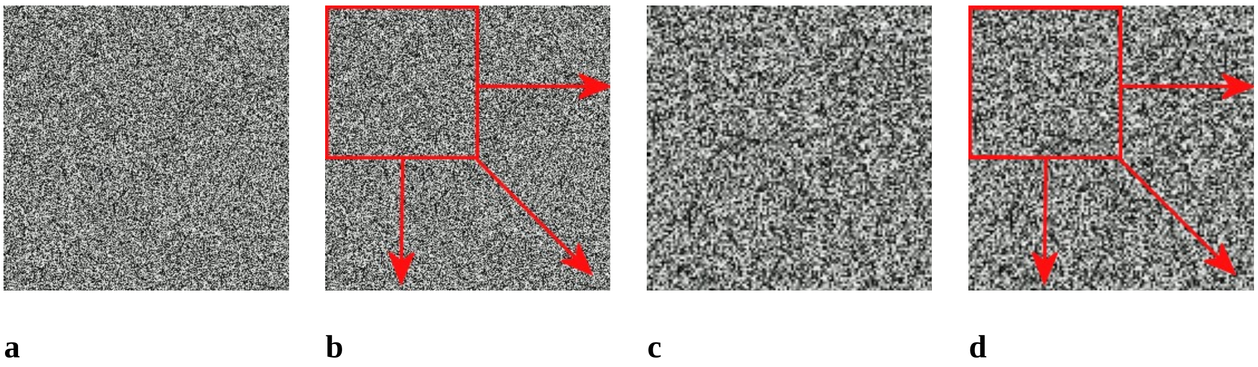

Fig 2.1a. Random noise texture. (**b**) Area to select and enlarge for the next iteration. (**c**) First enlarged texture. (**d**) Area to enlarge to get the second enlarged texture

Each of the next iterations would take an even smaller area (does not necessarily have to be divided by 2) [4]. Therefore, the number of enlarged images increases. All these images are stored for later use.

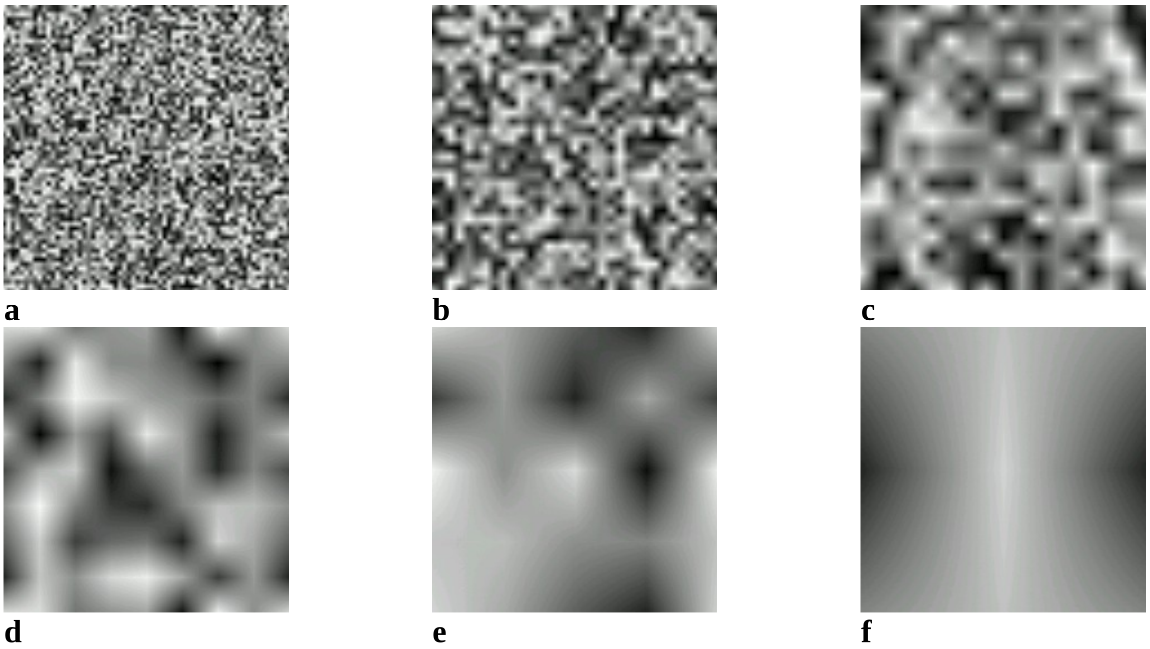

Fig 2.2a. Second enlarged image (4x scaled). (**b**) Third image (8x scaled). (**c**) Fourth image (16x scaled). (**d**) Fifth image (32x scaled). (**e**) Sixth image (64x scaled). (**f**) Seventh image (128x scaled)

By now, eight images are stored and ready to be used in the next step where the principle of turbulence is applied. This means that each of the images received a different value of proportion. The image that has been enlarged the most gets the proportional multiplier of 0.5. Then, as the enlargement factor decreases, the proportion multiplier halves accordingly: 1/4, 1/8, 1/16, 1/32, 1/64, 1/128, 1/256. The proportion does not have to halve for each layer. It can use some other function while still producing a natural output. However, halving the proportion multiplier is the most common approach here [4].

So, starting from the most enlarged image as base each of the enlarged image is added to the previous one according to the proportion multiplier (see figure 2.3).

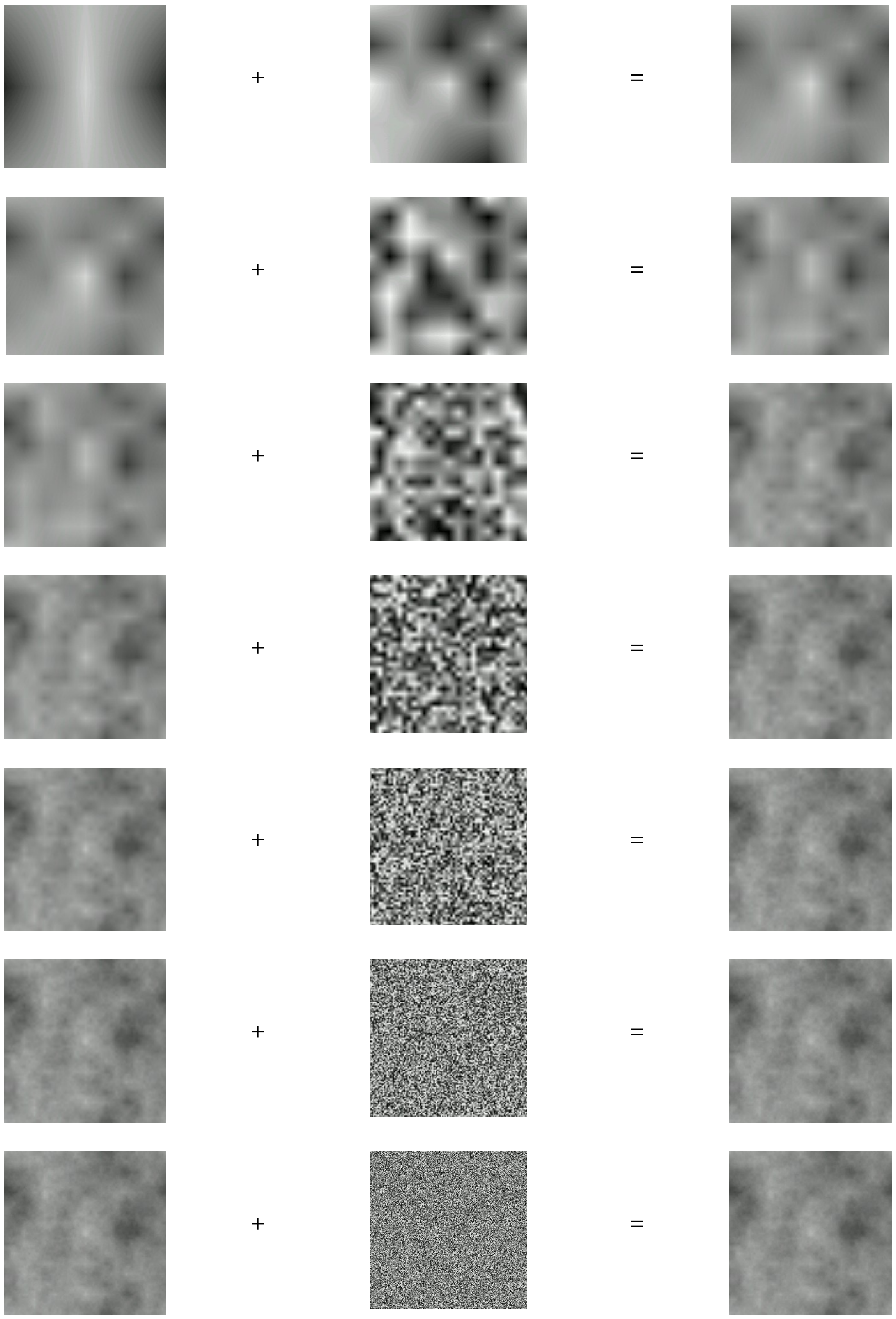

Fig 2.3. Adding textures together to get a Perlin noise.

As an additional step the bilinear interpolation algorithm that is used when enlarging noise could be enhanced to allow generation of seamless textures. So, the previously generated Perlin noise texture when displayed as multiple tiles near to each other produces a seamless texture (see figure 2.4). In figure 2.5 this same heightmap has been rendered in Molehill Heightmap Viewer [6].

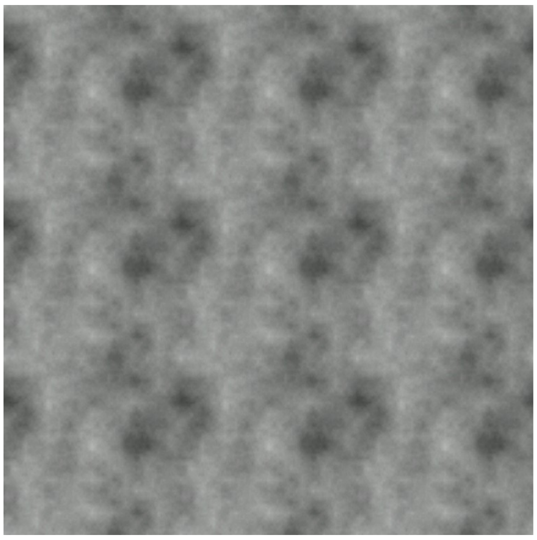
Fig 2.4. Seamless Perlin noise

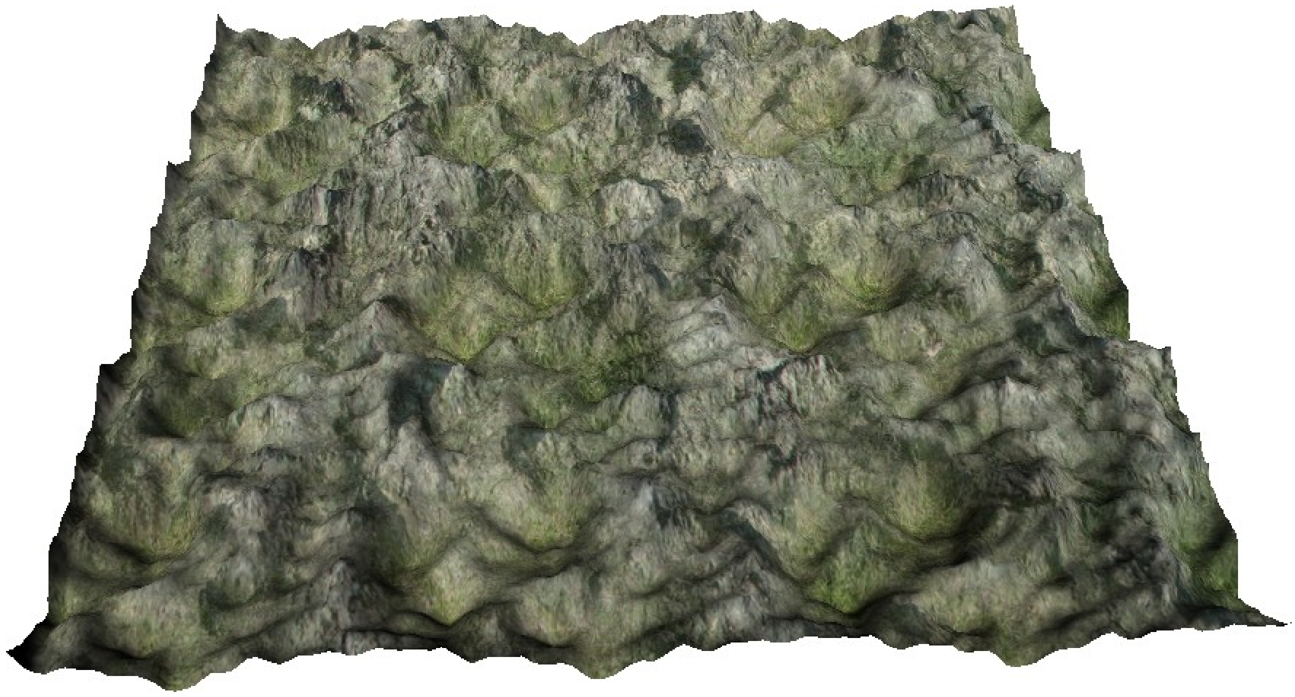
Fig 2.5. Terrain rendered with the heightmap in figure 2.4

However, the seamless aspect of generating Perlin noise raises many compilcations when attempting to use parallel computing. Therefore, this is specially discussed in the upcoming paragraphs.

2. Workflow

In this work Perlin Noise Generator is built as a modular set of programs: Noise, Zoom, Merge, Turbulence and Render.

In figure 3.1 a general workflow of Perlin Noise Generator is visualized. 3 layers of differently enlarged areas from the base noise are generated. Each of these layers can be produced in parallel. In turn, these layers spawn 3 individual instances of Zoom (9 Zoom instances ran in parallel). Finally, Turbulence adds the layers together making a file that contains the raw Perlin noise. This raw Perlin noise can then be rendered with a single Render instance producing a PNG format picture.

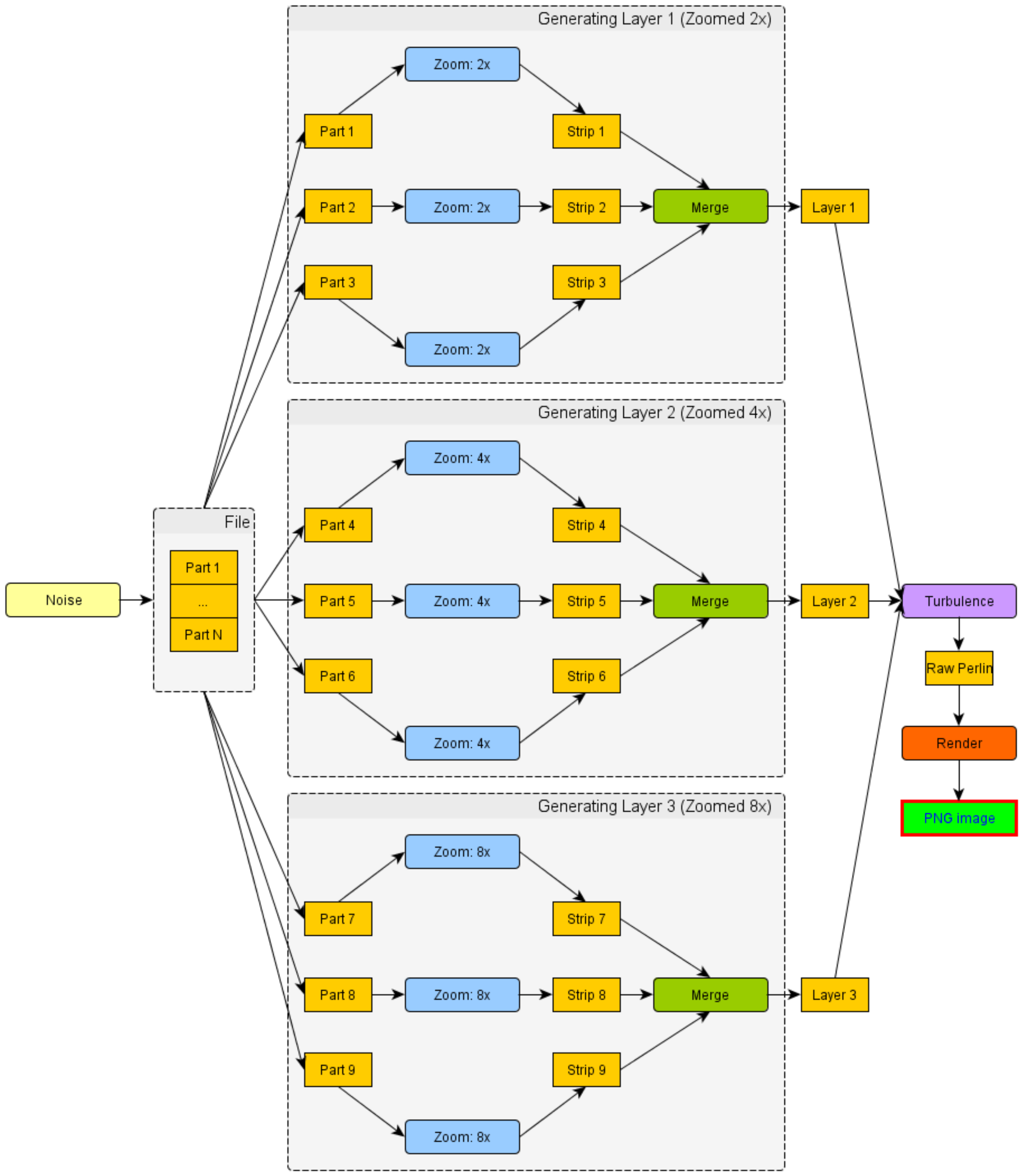

Fig 3.1. Workflow of Perlin Noise Generator

3. Noise

The initial base for all products of Perlin noise is the sequence of random numbers. In the context of this work these random numbers are generated using the pseudo-random number generator that is provided by the C standard library [7].

The purpose of Noise program is to fill a user defined file with randomly generated numbers. The program takes the width and height of the map to fill as arguments. Optionally a seed for the pseudo-random number generator can be included. Same seed produces always the same sequence of random-like numbers.

4. Zoom

The core worker of Perlin Noise Generator is the Zoom program. As shown in figure 4.1 its only purpose is to read in a part from the initial noise file and enlarge it to the user defined size. As a result a Zoom instance writes the enlarged area to a file.

The fact that in the end these enlarged strips need to match exactly to each other complicates the whole process a lot. In figure 4.2 it is demonstrated how actually a Zoom instance reads the initial noise from a file before enlarging it. Notice that each strip actually contains some of the data from the borders of its closest neighbours. Also, the strips that are located near the borders of the sub area to be enlarged must include some of the data from the strip located near the opposing border. This ensures that the enlarged sub area that is produced with Merge is a seamless pattern.

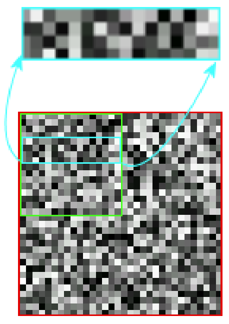
Fig 4.1. Strip from the sub area of noise to be zoomed

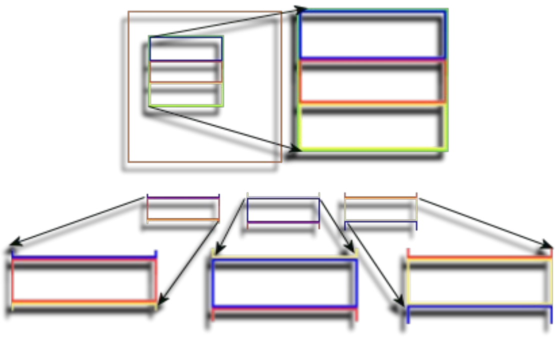
Fig 4.2. Strips to be enlarged must include some data from their neighbours too

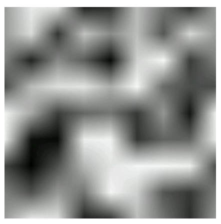
Fig 4.3. Neighbouring strips included

In figure 4.3 there is a sub area of some random noise. This area is enlarged 32 times and the Zoom instances that generated this image took each other into account. In contrast, the figure 4.4 demonstrates the same noise but this time the Zoom instances didn't take each other into account when enlarging their strips. Notice the rough edges in figure 4.4. It is also important to see that there are still major similarities between these two figures. Unfortunately, the figure in right is unacceptable in most cases.

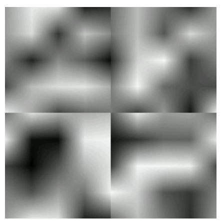
Fig 4.4. Neighbouring strips not included

5. Merge

The Merge program is the most trivial. All it does is writing the input files into a single output file. The content of each file will be appended to the output file.

6. Turbulence

The actual trick of generating Perlin noise is done in Turbulence program. All the other modules serve the purpose of parallel computing and represent the pure fact that modular design helps to keep the project simple.

The Turbulence instance reads in all the layers that hold enlarged areas of the initial noise. Then, by adding these layers together Perlin noise is generated. It is important to understand that each next layer to be added to the base layer has a significantly smaller influence on the final result. The generated Perlin noise is seamless only if all of its component layers are seamless.

To generate other types of Perlin noise this program could be easily enhanced or replaced. As a proof of concept the authors of this work included temporary functionality to demonstrate different types of Perlin noise. See figures 6.1, 6.2 and 6.3.

Fig 6.1. Default Perlin noise

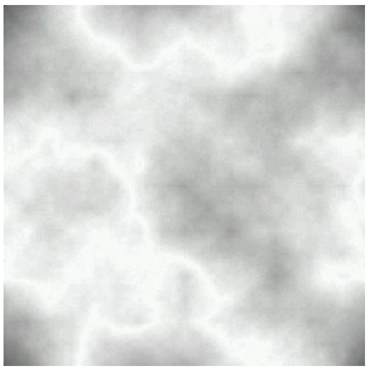
Fig 6.2. Inverted Perlin noise, using absolute function

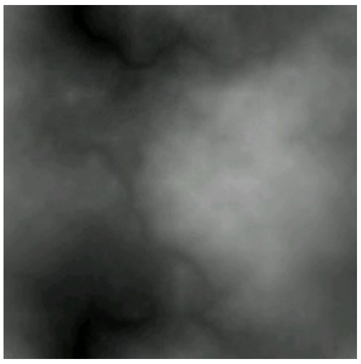
Fig 6.3. Marble texture, using absolute, sine and cosine functions

7. Render

The Render program reads the output file of Turbulence and takes its width and height as parameters. Using the PNG file format library provided by Allegro 5.0.6 [8] the Render instance writes the Perlin noise to a PNG image file to provide a human readable output of the generated Perlin noise.

8. Possible enhancements

There are a number of things that can be done to enhance the functionality of Perlin Noise Generator. Some of them are listed below.

Simplex noise

It turns out that generating classic Perlin noise loses efficiency when generating noise in larger dimensions. The original method works well for generating 2D textures but when it comes to 4th and 5th dimentions an alhorithm exists that does the same job with much lower computational cost. This new method was developed by Ken Perlin in 2001 to address the limitations of his classic noise function. Simplex noise requires fewer multiplications, scales to higher dimentions and is easier to implement in hardware [9].

Animated textures

The current solution could be enhanced to support generation of animated textures. Animated textures based on Perlin noise are actually just generated as 3D. Obviously the generated animations should be seamless so that the transition between last and first frame would be smooth. An example of animated Perlin noise is given here: [10].

Variable length bit depth

Currently each pixel in the final Perlin noise is described by 8 bits. This allows only 256 different states per pixel. However, by increasing the bit depth it would become meaningful to generate Perlin noise from more than 8 layers. In turn, this would give a lot more detail to the generated images.

9. Perlin Noise Generator as cloud service

The fact that Perlin noise could be used in many ways makes it reasonable to provide such a service. It could then be enhanced to support procedural texture generation in general. While there exist many other implementations for the same problem they are all dependent on the user's computer. This means that generating really large or detailed textures is not possible as it would consume too much time or memory.

The authors of this work want to put special emphasis on libnoise - a portable, open-source, coherent noise-generating library for C++ [11]. The mentioned library provides a complex toolkit for generating noise based textures. However, even with support for threads generating complex textures can get slow - at least for a typical home user. Hence, there is actually a need for cluster computing within the context of generating Perlin noise.

A typical use case for this service would be comparable to using Google Earth, except in this case it would be up to the user itself to create the world. In a context of a massive multiplayer online game the whole world could be held in cloud so that only the pieces that the player needs would be sent to its game client.

However, an even more interesting idea would be having this service as a global database for computer generated textures. Currently there are many programs that allow the user to generate textures and there are websites that allow downloading the raster images of these textures but it would be more optimal to only store the formulas of these textures and generate them on demand. Finally, providing a user friendly interface for this service targeting non-programmers would make it possible for artists to participate in forging new textures while having the possibility to easily share their work with other users.

MapReduce

Parallel Perlin noise generation’s workflow goes as follows:

1. Noise is generated using Noise program in main method, output is noise file
2. Files with arguments for Zoom program are generated in main method
3. Map method executes Zoom program with arguments in the files generated in previous step. Also it makes a key-value pair, where key is the layer and value is string containing filename of Zoom program generated zoom file.
4. Reduce method merges the stripes of each layer individually.
5. Turbulence program generates the final data needed for getting a pattern, takes merge files from previous steps as input.
6. Render program renders data into a PNG file, so the result could be seen visually

Speed comparison results

The authors used three different settings for setting benchmark of MapReduce approach: Firstly, Perlin noise is generated on single-node, but multithreaded machine, using shell script. Secondly, Perlin noise is generated on single-node, but multithreaded machine, using single-node Hadoop for it. Lastly, Perlin noise is generated on multi-node cluster (4+1 nodes), using Hadoop. Parameters used for this test are as follows:

- Width: 10,000 px
- Height: 10,000 px
- Stripes: 100
- Layers: 8

Results:

Intel Core i5 760 @ 2.8 GHz, 4 threads, shell script: DNF (took more than 30 min.)
SciCloud, 4+1 nodes, Hadoop: 966 seconds.

As results show, using MapReduce on multi node setup is clearly faster than generating Perlin noise locally.

Problems with implementing MapReduce

As Hadoop uses its own distributed file system, called HDFS, then problems arose when trying to pass data between multiple nodes in cluster. During the project authors did not find a solution how to determine on which node the data generated by Zoom and Merge executables in Map and Reduce methods is saved. This determines the fact that our implementation is not finished and it is highly likely that MapReduce is not the best approach to make Perlin noise generation parallel.

Conclusion

While generating natural looking textures is a topic of its own, in the end it still depends on natural randomness. This natural randomness is usually provided by Perlin noise.

The current work set its goal to find a method for largely scalable generation of Perlin noise that could be realized as a cloud service. Nevertheless a well working solution was delivered it did not come easy.

One of the most difficult parts was designing the toolkit so that distributed computing could be easily applicable. This involved mostly getting the Zoom instances to generate strips that would perfectly match with each other.

While the primitive tools were designed to be easily portable several unexpected problems arose when trying to get the project working under SciCloud. As the tools use the natural file system of the operating system to write and read their data Hadoop uses its own file system (HDFS). Therefore, extra work had to be done to get the project working with Hadoop: Original C++ programs had to be rewritten using Hadoop’s own C API for HDFS, called libhdfs. Also, to parallelize noise generation, MapReduce algorithm had to be implemented and altered specifically for this task.

There are several things that can be done to enhance the Perlin Noise Generator. One of the most interesting ones is support for animated textures but to get an out of the box solution a lot more effort has to be put into this project. The authors of this work believe that the whole concept of having a Perlin Noise Generator as a cloud service has a great potential but fully realizing it falls out of the scope of this course.

Perlin noise sure has its place in the world of procedural content generation but it still remains questionable if there could ever be a strong need for vast seamless textures to be generated. After all, procedural generation is usually done on demand. However, one thing is for sure - generating seamless Perlin noise can be done in parallel. This gives the algorithm a tremendous speed boost that many of the already existing solutions lack.

To fully understand this work it is advisable to see the practical solution developed within the context of this work. It is included in the appendix B. The user manual is in the appendix A. For the user manual of Hadoop implementation, please see appendix C.

References

1. Wikipedia, Patterns in nature, viewed May 17 2012, http://en.wikipedia.org/wiki/Fractal#Natural_phenomena_with_fractal_features.

2. Wikipedia, Heightmap, viewed May 17 2012, http://en.wikipedia.org/wiki/Heightmap.

3. Wikipedia, Procedural generation, viewed May 17 2012, http://en.wikipedia.org/wiki/Procedural_generation.

4. Perlin Noise, viewed May 17 2012, http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

5. Making noise, Ken Perlin, viewed May 17 2012, http://www.noisemachine.com/talk1/.

6. Molehill Heightmap Viewer, viewed May 17 2012, http://disturbedcoder.com/?p=38.

7. rand - C++ reference, viewed May 27 2012, http://www.cplusplus.com/reference/clibrary/cstdlib/rand/.

8. About Allegro.cc, viewed May 27 2012, http://www.allegro.cc/about.

9. Simplex noise, viewed May 27 2012, http://en.wikipedia.org/wiki/Simplex_noise.

10. Animated perlin noise, viewed May 27 2012, http://www.youtube.com/watch?v=csYzDia5Ckw

11. libnoise, viewed May 27 2012, http://libnoise.sourceforge.net/

Appendix A

Short user guide for Perlin Noise Generator

This user guide collects all the information concerning the usage of Perlin Noise Generator developed as part of this work.

System requirements to compile this project

The GCC low-level runtime library
The GNU Standard C++ Library v3
Allegro 5.0.6 (Only needed to compile Render)

In Appendix B there are prebuilt Win32 programs included.

Workflow

First, as shown in figure 3.1, noise should be called to generate the initial noise file filled with randomly generated bytes. Then, for each layer of zoomed noise a user defined number of Zoom instances should be called to generate the enlarged strips based on that initial noise. The number of layers to generate is user defined too but for practical reasons it is not meaningful to generate more than 8 layers. For each set of strips a Merge instance should be spawned (it is important to wait for all the Zoom instances to finish). Each of the Merge instance produces a file that holds an enlarged area from the original noise. These files are given as input to the Turbulence instance which in turn generates the raw Perlin noise. Finally, to visualize the Perlin noise the output of Turbulence should be given as an input to Render to produce a PNG image of the generated Perlin noise.

Definition of arguments

Noise

```
Arguments: FNAME WIDTH HEIGHT [SEED]
Example:   Noise.exe noise.dat 256 256

FNAME    - File to fill with random bytes.
WIDTH    - Integer width of the output noise map.
HEIGHT   - Integer height of the output noise map.
SEED     - Seed for the pseudo-random number generator. Uses program's starting
           time as seed when left blank.
```

Zoom

```
Arguments: INPUT WIDTH HEIGHT SX SY SW SH SubW SubH OUTPUT DW DH
Example:   Zoom.exe noise.dat 256 256 0 0 256 128 256 256 zoom1_1.dat 256 128

INPUT    - File name that contains the random noise to be zoomed.
WIDTH    - Integer width of the input noise map.
HEIGHT   - Integer height of the input noise map.
SX       - Integer x-position of the strip in the input file to be zoomed in.
SY       - Integer y-position of the strip in the input file to be zoomed in.
SW       - Integer width of the strip in the input file.
SH       - Integer height of the strip in the input file.
SubW     - Width of the sub area in the input file where the strip belongs to.
SubH     - Height of the sub area in the input file where the strip belongs to.
OUTPUT   - Name of the file to write the zoomed in data to.
DW       - Desired integer width of the output file.
DH       - Desired integer height of the output file.
```

Merge

```
Arguments: FNAME INPUT_1 [INPUT_2 [...]]
Example:   Merge.exe merge1.dat zoom1_1.dat zoom1_2.dat

FNAME    - Name of the output file.
INPUT_1  - Name of the first input file to be added.
...      - ...
INPUT_N  - Name of the last input file to be added.
```

Turbulence

```
Arguments: FNAME INPUT_1 [INPUT_2 [...]]
Example:   Turbulence.exe turbulence.dat merge3.dat merge2.dat merge1.dat

FNAME    - Name of the output file.
INPUT_1  - Name of the first input file. This should be the one most zoomed in.
...      - ...
INPUT_N  - Name of the last input file. This should be the one least zoomed in.
```

Render

```
Arguments: INPUT OUTPUT WIDTH HEIGHT
Example:   Render.exe turbulence.dat perlin.png 256 256

INPUT    - File name of the raw Perlin noise.
OUTPUT   - File name to write PNG format picture to.
WIDTH    - Integer width of the input Perlin noise.
HEIGHT   - Integer height of the input Perlin noise.
```

Example batch file for Windows

```
Noise.exe temp\noise.dat 256 256 1337

Zoom.exe temp\noise.dat 256 256 0   0 256 128 256 256 temp\zoom1_1.dat 256 128
Zoom.exe temp\noise.dat 256 256 0 128 256 128 256 256 temp\zoom1_2.dat 256 128

Zoom.exe temp\noise.dat 256 256 0   0 128  64 128 128 temp\zoom2_1.dat 256 128
Zoom.exe temp\noise.dat 256 256 0  64 128  64 128 128 temp\zoom2_2.dat 256 128

Zoom.exe temp\noise.dat 256 256 0   0  64  32  64  64 temp\zoom3_1.dat 256 128
Zoom.exe temp\noise.dat 256 256 0  32  64  32  64  64 temp\zoom3_2.dat 256 128

Zoom.exe temp\noise.dat 256 256 0   0  32  16  32  32 temp\zoom4_1.dat 256 128
Zoom.exe temp\noise.dat 256 256 0  16  32  16  32  32 temp\zoom4_2.dat 256 128

Zoom.exe temp\noise.dat 256 256 0   0  16   8  16  16 temp\zoom5_1.dat 256 128
Zoom.exe temp\noise.dat 256 256 0   8  16   8  16  16 temp\zoom5_2.dat 256 128

Zoom.exe temp\noise.dat 256 256 0   0   8   4   8   8 temp\zoom6_1.dat 256 128
Zoom.exe temp\noise.dat 256 256 0   4   8   4   8   8 temp\zoom6_2.dat 256 128

Zoom.exe temp\noise.dat 256 256 0   0   4   2   4   4 temp\zoom7_1.dat 256 128
Zoom.exe temp\noise.dat 256 256 0   2   4   2   4   4 temp\zoom7_2.dat 256 128

Zoom.exe temp\noise.dat 256 256 0   0   2   1   2   2 temp\zoom8_1.dat 256 128
Zoom.exe temp\noise.dat 256 256 0   1   2   1   2   2 temp\zoom8_2.dat 256 128

Merge.exe temp\merge1.dat temp\zoom1_1.dat temp\zoom1_2.dat
Merge.exe temp\merge2.dat temp\zoom2_1.dat temp\zoom2_2.dat
Merge.exe temp\merge3.dat temp\zoom3_1.dat temp\zoom3_2.dat
Merge.exe temp\merge4.dat temp\zoom4_1.dat temp\zoom4_2.dat
Merge.exe temp\merge5.dat temp\zoom5_1.dat temp\zoom5_2.dat
Merge.exe temp\merge6.dat temp\zoom6_1.dat temp\zoom6_2.dat
Merge.exe temp\merge7.dat temp\zoom7_1.dat temp\zoom7_2.dat
Merge.exe temp\merge8.dat temp\zoom8_1.dat temp\zoom8_2.dat

Turbulence.exe  temp\turbulence.dat  temp\merge8.dat  temp\merge7.dat  temp\merge6.dat  temp\merge5.dat
temp\merge4.dat temp\merge3.dat temp\merge2.dat temp\merge1.dat

Render.exe temp\turbulence.dat perlin.png 256 256
```

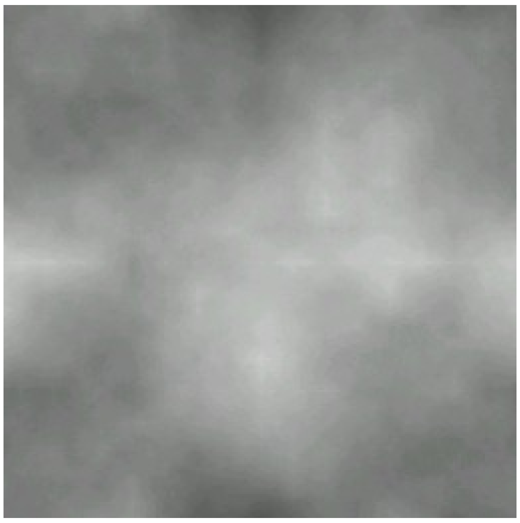

Fig A.1. Output of the batch file.

In figure A.1 there is an output of the previously defined batch file. Notice that Noise.exe is called with the seed 1337. The given figure is generated with this same seed. To generate a different Perlin noise each time the seed parameter should be removed.

Appendix B

Perlin Noise Generator source code

The source code for Perlin Noise Generator is available at its SVN repository:

- svn://ats.cs.ut.ee/u/rwg/fun/Perlin

Appendix C

Hadoop Perlin noise generator user manual

Perlin noise generator is run in Hadoop with following command:

```
hadoop jar MapReduce.jar MapReduce <path> <width> <height> <stripes> <parallel jobs> <layers>
<filename> <temp path>
```

Meanings of arguments:

- <path> determines, where on HDFS is the “home directory” of Perlin generator
- <width> determines the width of resulting picture
- <height> determines the height of resulting picture
- <stripes> determines into how many parts will the picture be divided
- <parallel jobs> will determine how many parallel jobs will be run on Hadoop
- <layers> determines how many layers will be made (needed for optimal result)
- <filename> determines the filenames of some temporary files
- <temp path> determines into which folders are temporary files put

Also, Merge, Zoom, Turbulence and Noise executables should be put into HDFS “home directory”. For this, one can use `hadoop fs -copyFromLocal <source path> <destination path>` command.