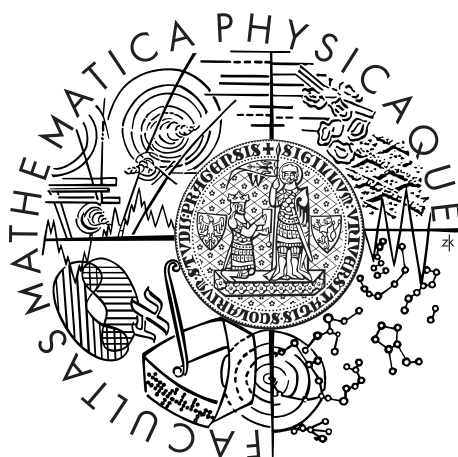


Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Dalibor Frivaldský

Rychlá implementace procedurálních 3D textur

Department of Software and Computer Science Education

Supervisor of the master thesis: doc. Ing. Jaroslav Křivánek, Ph.D.

Study programme: Informatics

Specialization: Software Systems

Prague 2013

I would like to thank my supervisor for his guidance and especially patience during the writing of this thesis. I would also like to thank all the people that supported me and helped me during this endeavour.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In date

Bc. Dalibor Frivaldský

Název práce: Rychlá implementace procedurálních 3D textur

Autor: Bc. Dalibor Frivaldský

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: doc. Ing. Jaroslav Křivánek, Ph.D., Kabinet software a výuky informatiky

Abstrakt: Procedurální šumové textury hrají klíčovou roli ve světě počítačové grafiky. Jejich výpočet je však časově náročný. Generování šumu pro renderování realistických scén může vyžadovat velké množství hardwarových prostředků. Efektivní využití těchto zdrojů je žádoucí vzhledem k trendu neustále se zvyšujících detailů v renderovaných scénách.

V této práci vyvineme sadu optimalizovaných algoritmů pro známý Vylepšený Perlinův šum a nový Gaborův šum. S využitím SIMD prostředků moderní generace procesorů dosáhneme efektivního využití hardwarových zdrojů. Tyto algoritmy následně zintegrujeme do renderovacího systému MentalRay a budeme demonstrovat jejich efektivitu v různých scénách.

Klíčová slova: počítačová grafika, 3D, rendering, procedurální textura

Title: Fast implementation of 3D textures

Author: Bc. Dalibor Frivaldský

Department: Department of Software and Computer Science Education

Supervisor: doc. Ing. Jaroslav Křivánek, Ph.D., Department of Software and Computer Science Education

Abstract: Procedurally generated noise textures play a key part in the world of computer graphics. However, their computation is very time consuming. Generating noise for rendering of photo-realistic scenes may require large amount of hardware resources. Efficient use of these resources is beneficial in the trend of ever-increasing amount of details in the rendered scenes.

In this thesis we develop a series of optimized algorithms for the popular Improved Perlin noise and the new Gabor noise. We will employ SIMD features of the modern generation of general purpose processors to achieve efficient use of hardware resources. These algorithms will be integrated into the MentalRay ray-tracing rendering engine and their effectiveness demonstrated in various scenes.

Keywords: computer graphics, 3D, rendering, procedural texture

Contents

Introduction	3
1 Improved Perlin noise	4
1.1 Introduction to Improved Perlin noise	4
1.1.1 Base algorithm	5
1.2 Algorithm vectorization	8
1.2.1 Initial performance analysis	8
1.2.2 Algorithm vectorization - Iteration 1	10
1.2.3 Algorithm vectorization - Iteration 2, final version	15
1.2.4 Algorithm vectorization - Four values at once	18
1.3 Perlin-derived noises - Billow and RidgedMulti	21
1.4 Integration to Mental-Ray, Maya	25
1.4.1 Exposing the noise	25
1.4.2 Comparing 2D visual output	26
1.4.3 3D volumetric rendering and performance	29
2 Gabor noise	33
2.1 Introduction to Gabor noise	33
2.1.1 Base algorithm	34
2.2 Algorithm vectorization	38
2.2.1 Initial performance analysis	38
2.2.2 Algorithm vectorization - Iteration 1	41
2.2.3 Algorithm vectorization - Fixed-sized local arrays	44
2.2.4 Algorithm vectorization - No arrays, merging	45
2.2.5 Algorithm vectorization - 4 values at once	50
2.3 Gabor surface noise	51
2.3.1 Implementation and vectorization	52
2.4 Integration to Mental-Ray, Maya	52
2.4.1 Exposing the noise	53
2.4.2 Rendering performance	54
3 Implementation details	56
3.1 Design and implementation goals	56
3.2 Implementation	57
3.2.1 Public API layer	57
3.2.2 Specific <i>Module</i> implementation layer	61
3.2.3 Vector Math layer	64
4 Measurements	66
4.1 Improved Perlin and Perlin-based noises	66
4.1.1 Improved Perlin noise, "float" precision	67
4.1.2 Improved Perlin noise, "double" precision	68
4.1.3 Billow noise, "float" precision	69
4.1.4 Billow noise, "double" precision	70
4.1.5 RidgedMulti noise, "float" precision	71

4.1.6	RidgedMulti noise, "double" precision	72
4.2	Gabor noise and Gabor surface noise	73
4.2.1	Gabor noise	73
4.2.2	Gabor surface noise	74
Conclusion		75
Bibliography		77
List of Figures		79
List of Tables		81
List of Abbreviations		82
Attachments		83

Introduction

In the world of computer graphics, procedurally generated noise primitives provide the necessary details to natural looking surfaces (such as wood, marble, cloth) and phenomena (clouds). Dating back to as early as 1984, when Ken Perlin introduced his Perlin noise [9], this field of computer graphics is still evolving and new types of algorithms are being developed.

The algorithms for procedurally generated noise primitives are, however, very time consuming. Real-time rendering of objects with such procedural materials has only been possible with the latest hardware and generating noise patterns for natural phenomena in "off-line" rendering of photo-realistic scenes can take up considerable amount of time. It is only natural that the hardware should be used to its full potential during these computations. Modern, general-purpose processors provide extensive set of features that can be employed to provide optimized versions of many types of algorithms. To some extent, these resources can be tapped automatically by modern compilers through features such as auto-vectorization. However, effective optimization of complex algorithms, such as procedurally generated noises, is currently beyond the capabilities of even the best compilers. Deeper understanding of such algorithms and carefully handwritten optimizations are necessary to get the most out of the available hardware resources.

The choice of programming language and platform has significant influence on the amount of options accessible to the developer when implementing algorithm optimizations. The language should not hide the specifics of the hardware behind a thick layer of abstractions and at the same time offer features of a modern programming language to provide the developer with the ability to create usable, maintainable code. Such language is C++. In terms of available hardware features, most of the currently used general purpose processors offer various sets of vector instructions. The most common of these are the SSE instruction sets, which can provide up 4-times theoretical speed-up over a base, non-vectorized algorithm.

Some work has already been done to provide the most common types of procedurally generated noises in a form of publicly available C++ libraries. One of the most popular of such libraries is the *libnoise*¹ project. However, none of them focuses on providing optimized implementations for modern hardware. Implementations of new types of procedurally generated noises are also scarcely available, if at all.

The goal of this thesis is thus to provide modern, feature-rich C++ library of optimized implementations of common and perspective new types of procedurally generated noises, such as the Improved Perlin noise and Gabor noise. Inseparable part of the work is also exposing the various noise types from the library to the environment of the MentalRay ray-tracing rendering engine through MentalRay shaders.

¹<http://libnoise.sourceforge.net>

1. Improved Perlin noise

1.1 Introduction to Improved Perlin noise

Since its introduction by Ken Perlin 29 years ago [9, 10, 11], Perlin noise has become one the most widely used fundamental algorithms for procedural noise generation. Version of its improved form [12] is the focus of the first part of this thesis.

The Improved Perlin noise is a type of coherent noise, which has 3 main properties, described in terms of the relationship between the input and output values:

1. The same input value always generates the same output value.
2. Small change in the input values will generate small, coherent change in the output values.
3. Large change in the input values will generate pseudo-random change in the output values.

We will describe the fundamentals of the 3D version of the algorithm given a point $P = (x, y, z)$. In an integer cubic lattice, this point is surrounded by eight vertices $V_P = \{(i, j, k)\}$, where $i : \{\lfloor x \rfloor, \lfloor x \rfloor + 1\}$, $j : \{\lfloor y \rfloor, \lfloor y \rfloor + 1\}$ and $k : \{\lfloor z \rfloor, \lfloor z \rfloor + 1\}$ (see figure 1.1).

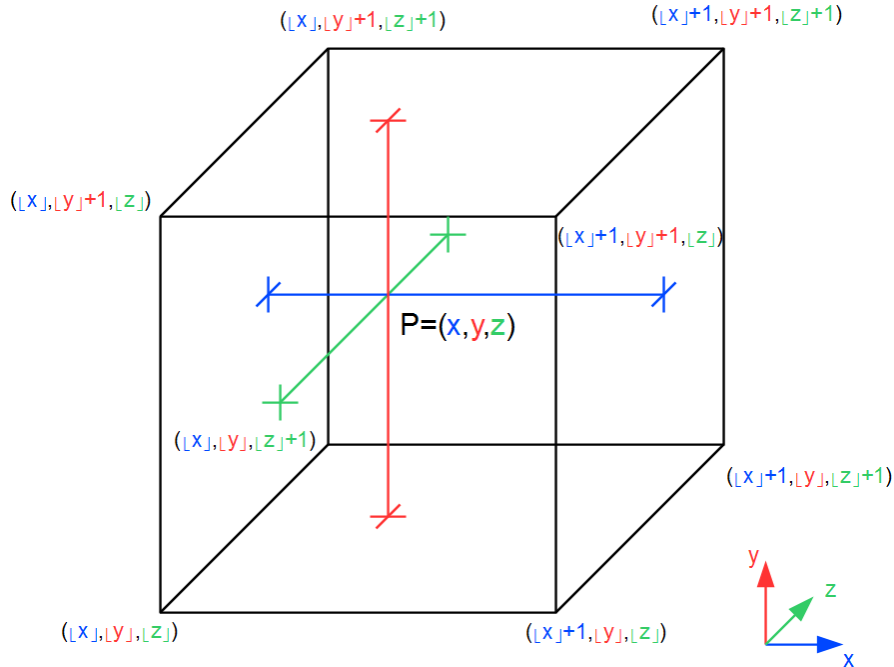


Figure 1.1: Perlin sample point surrounded by integer cubic lattice vertices

Each such unique vertex in the lattice defined by (i, j, k) is assigned a 3D gradient vector $G_{i,j,k} = G[H_s(i, j, k)]$, where G is a precomputed array and H_s a hashing function. G contains a set of pseudo-random unit-length gradient vectors

uniformly distributed over a sphere [12]. The function $H_s(i, j, k)$ computes a hash from the values i, j, k and a seed s , which is then used to index into the array G . The hashing provides a consistent way for associating the gradient vectors to the lattice vertices. Formally,

$$H_s(i, j, k) = Hmad_s(i, j, k) \wedge (Hmad_s(i, j, k) >> 8) \pmod{256} \quad (1.1)$$

where

$$Hmad_s(i, j, k) = s * S + i * I + j * J + k * K \quad (1.2)$$

where S, I, J, K are being prime numbers. Given the gradient vectors of the surrounding lattice cube, we can create eight linear functions

$$N_{i,j,k}(x, y, z) = G_{i,j,k} \cdot (x, y, z) \quad (1.3)$$

which are then tri-linearly interpolated by $sp(x - \lfloor x \rfloor), sp(y - \lfloor y \rfloor), sp(z - \lfloor z \rfloor)$, where

$$sp(t) = 6t^5 - 15t^4 + 10t^3 \quad (1.4)$$

The final noise value is obtained by summing the result of several Improved Perlin noise evaluations with varying amplitude and frequency, called *octaves*. This gives the noise fractal properties.

1.1.1 Base algorithm

The base version of the Improved Perlin noise algorithm, which we obtained from the *libnoise* project, can be divided into 3 parts. The first one serves as an entry point of the Improved Perlin noise (see Algorithm 1). It captures the fractal properties of the noise by calculating noise values for each octave, each time adjusting the input parameters to the *gradientCoherentNoise* function and modulating the output noise value. Sum of the noise values for each octave is the final noise value for the point $P = (x, y, z)$.

The second part, represented by the *gradientCoherentNoise* function (see Algorithm 2), implements the core of the noise algorithm. It begins by finding the surrounding cube of the input point $P = (x, y, z)$ in the integer cubic lattice. Three interpolation values are then prepared - for the three axes x, y, z . They are obtained by evaluating a spline function. Which spline function is evaluated is decided based on a chosen quality. Three levels are provided - level *BEST* evaluates spline described in the Improved Perlin noise ([12], section "MODIFICATIONS", 1st paragraph). Level *STD* evaluates spline from the original Perlin noise and level *FAST* just returns the input value. With all values prepared, the algorithm then begins to evaluate the linear functions $N_{i,j,k}(x, y, z)$ (represented by call to the *gradientNoise* function) and interpolates the resulting gradient dot products to obtain the final noise value.

The third part (see Algorithm 3) implements evaluation of the linear $N_{i,j,k}(x, y, z)$ function. Initially, it calculates, by the use of hashing, an index into the gradient vector table G . Dot product of the obtained gradient vector and the input point

$P = (x, y, z)$ (in the surrounding lattice cube space) yields the resulting value which is scaled into a range of $(-1, 1)$.

Algorithm 1 Improved Perlin entry point

```

1: struct PerlinProps
2:   frequency : float
3:   octaveCount : uint
4:   lacunarity : float // per octave frequency modulation
5:   persistence : float // per octave amplitude modulation
6:   noiseQuality : enum {FAST, STD, BEST}
7:   seed : int
8: end struct

9: function IMPROVEDPERLIN(x, y, z: float; props: PerlinProps)
10:   noise : float
11:   octaveSeed : int
12:   octavePersistence : float
13:
14:   noise  $\leftarrow$  0.0
15:   x  $\leftarrow$  x  $\times$  props.frequency
16:   y  $\leftarrow$  y  $\times$  props.frequency
17:   z  $\leftarrow$  z  $\times$  props.frequency
18:
19:   for i = 0  $\rightarrow$  (props.octaveCount - 1) do
20:     nx  $\leftarrow$  makeInt32Range(x)
21:     ny  $\leftarrow$  makeInt32Range(y)
22:     nz  $\leftarrow$  makeInt32Range(z)
23:     octaveSeed  $\leftarrow$  props.seed + i
24:
25:     signal  $\leftarrow$  gradientCoherentNoise(x, y, z, octaveSeed, props.noiseQuality)
26:     noise  $\leftarrow$  noise + (signal  $\times$  octavePersistence)
27:
28:     x  $\leftarrow$  x  $\times$  props.lacunarity
29:     y  $\leftarrow$  y  $\times$  props.lacunarity
30:     z  $\leftarrow$  z  $\times$  props.lacunarity
31:     octavePersistence  $\leftarrow$  octavePersistence  $\times$  props.persistence
32:   end for
33:   return noise
34: end function

```

Algorithm 2 Improved Perlin gradientCoherentNoise

```
1: function GRADIENTCOHERENTNOISE(x, y, z: float; seed: int; quality:
   enum)
2:    $cellX0 \leftarrow floor(x)$ 
3:    $cellX1 \leftarrow cellX0 + 1$ 
4:    $cellY0 \leftarrow floor(y)$ 
5:    $cellY1 \leftarrow cellY0 + 1$ 
6:    $cellZ0 \leftarrow floor(z)$ 
7:    $cellZ1 \leftarrow cellZ0 + 1$ 
8:
9:    $xs \leftarrow spline(x - cellX0, quality)$ 
10:   $ys \leftarrow spline(y - cellY0, quality)$ 
11:   $zs \leftarrow spline(z - cellZ0, quality)$ 
12:
13:   $n0 \leftarrow gradientNoise(x, y, z, cellX0, cellY0, cellZ0, seed)$ 
14:   $n1 \leftarrow gradientNoise(x, y, z, cellX1, cellY0, cellZ0, seed)$ 
15:   $noiseX0 \leftarrow linearInterp(n0, n1, xs)$ 
16:   $n0 \leftarrow gradientNoise(x, y, z, cellX0, cellY1, cellZ0, seed)$ 
17:   $n1 \leftarrow gradientNoise(x, y, z, cellX1, cellY1, cellZ0, seed)$ 
18:   $noiseX1 \leftarrow linearInterp(n0, n1, xs)$ 
19:   $noiseY0 \leftarrow linearInterp(noiseX0, noiseX1, ys)$ 
20:   $n0 \leftarrow gradientNoise(x, y, z, cellX0, cellY0, cellZ1, seed)$ 
21:   $n1 \leftarrow gradientNoise(x, y, z, cellX1, cellY0, cellZ1, seed)$ 
22:   $noiseX0 \leftarrow linearInterp(n0, n1, xs)$ 
23:   $n0 \leftarrow gradientNoise(x, y, z, cellX0, cellY1, cellZ1, seed)$ 
24:   $n1 \leftarrow gradientNoise(x, y, z, cellX1, cellY1, cellZ1, seed)$ 
25:   $noiseX1 \leftarrow linearInterp(n0, n1, xs)$ 
26:   $noiseY1 \leftarrow linearInterp(noiseX0, noiseX1, ys)$ 
27:
28:   $noiseZ \leftarrow linearInterp(noiseY0, noiseY1, zs)$  return  $noiseZ$ 
29: end function
```

Algorithm 3 Improved Perlin gradientNoise

```
1: NoiseGenX  $\leftarrow$  1619
2: NoiseGenY  $\leftarrow$  31337
3: NoiseGenZ  $\leftarrow$  6971
4: NoiseGenSeed  $\leftarrow$  1013

5: function GRADIENTNOISE(x, y, z: float; cellX, cellY, cellZ: int; seed: int)
6:   tableIndex  $\leftarrow$  cellX * NoiseGenX
7:   tableIndex  $\leftarrow$  tableIndex + cellY * NoiseGenY
8:   tableIndex  $\leftarrow$  tableIndex + cellZ * NoiseGenZ
9:   tableIndex  $\leftarrow$  tableIndex + seed * NoiseGenSeed
10:  tableIndex  $\leftarrow$  tableIndex ^ (tableIndex >> 8)
11:  tableIndex  $\leftarrow$  tableIndex (mod 256)
12:  tableIndex  $\leftarrow$  tableIndex * 4
13:
14:  xGradient  $\leftarrow$  G[tableIndex]
15:  yGradient  $\leftarrow$  G[tableIndex + 1]
16:  zGradient  $\leftarrow$  G[tableIndex + 2]
17:  xPoint  $\leftarrow$  x - cellX
18:  xPoint  $\leftarrow$  y - cellY
19:  xPoint  $\leftarrow$  z - cellZ
20:
21:  noise  $\leftarrow$  xPoint * xGradient
22:  noise  $\leftarrow$  noise + yPoint * yGradient
23:  noise  $\leftarrow$  noise + zPoint * zGradient
24:  noise  $\leftarrow$  noise * 2.12 // get noise value into bounds
25: end function
```

1.2 Algorithm vectorization

We first begin by analyzing the scalar version of the Improved Perlin noise algorithm. We will try to divide it into smaller parts with well-defined interfaces and identify portions suitable for vectorization. The transformation will be described incrementally. Charts representing speed gains will be provided throughout the process of vectorization. This approach should capture the evolution of the vectorized algorithm to its final form.

1.2.1 Initial performance analysis

With the scalar version of the algorithm defined (see section 1.1.1), we will do basic performance analysis to get an initial idea of how it performs. The performance will be measured in a simple scenario of calculating 16 million 3D noise values in a single thread. Three versions of the scenario application will be measured, each compiled with a different compiler:

1. GNU MinGW G++ cross-compiler for 64-bit, version 4.7.2
 - compilation optimization options: -O3

- -O3 - full optimization, function inlining
2. Microsoft MSVC 2012 cl compiler for 64-bit, version 15.00.21022.08
 - compilation optimization options: /Ox /Ob2 /Oi /Ot
 - /Ox - full optimization
 - /Ob2 - inline any suitable function
 - /Oi - enable intrinsic functions
 - /Ot - favor fast code (as opposite to small size)
 3. Intel C++ Composer XE 2013 Icc compiler for 64-bit, version 13.1.1.171
 - compilation optimization options: /O3 /Ob2 /Oi /Ot
 - /O3 - highest optimizations
 - /Ob2 - inline any suitable function
 - /Oi - enable intrinsic functions
 - /Ot - favor fast code (as opposite to small size)

The execution measurements will be done on a system with these properties:

- OS: Windows 8, version 6.2
- CPU: Intel Core i5 3317U, 1.7 GHz, Turbo boost to 2.6 GHz, 2 cores, 4 threads
- Memory: 4 GB DDR3 1600 Mhz, CL11-11-11-28

These compiler parameters and system properties will be used for all subsequent measurements.

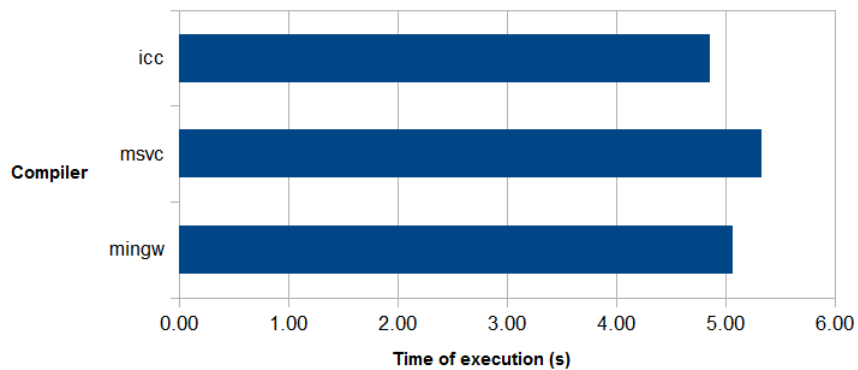


Figure 1.2: Execution time of the base 3D Improved Perlin noise algorithm

To obtain the measurement data, each calculation was performed 7 times for each compiler version. The fastest and slowest results were discarded and the remaining 5 measurements were averaged. Unless stated otherwise, this procedure will be used for all subsequent time measurements.

As can be seen in figure 1.2, calculating Perlin noise values is quite time intensive, producing 3.2 million values per second (for MinGW compiler). Put in a real-time perspective, this would mean only a little more than 1.5 FPS¹ for a full HD² rendering (1920x1080).

Let us take a deeper look at the algorithm execution and hot-spots using Intel's VTune Analyzer. For this purpose, the Intel Icc compiled version was used with function inlining turned off using the `/Ob0` compiler flag. This is necessary to get a better idea in which parts of the code we are spending the most time while calculating.

Function	Total time inside	Self time inside
<code>perlin3</code>	12.07s	0.01s
<code>-GetValue</code>	12.05s	0.44s
<code>-GradientCoherentNoise</code>	11.09s	2.01s
<code>—GradientNoise</code>	6.06s	5.97s
<code>—SCurve5</code>	1.87s	1.87s
<code>—LinearInterp</code>	1.13s	1.13s
<code>-MakeInt32Range</code>	0.43s	0.43s

Table 1.1: Hot-spot analysis of the base Improved Perlin noise algorithm

The table above (1.1) shows how much time was spent in the main functions implementing the algorithm. The second column shows the total time spent somewhere in a function (including inner calls to another functions), the third column shows time spent purely executing the code of that function (without any inner calls). As we can see, turning off function inlining plus the VTune Analyzer's sampling overhead increased the execution time to 12 seconds. The function *perlin3* implements the logic for calculating the 16 million values, *GetValue* is the entry point to the calculation of one Improved Perlin noise value. Function *MakeInt32Range* is, for input values inside half of the range of 32 bit signed integer values, an identity function. The rest of the functions implement the core of the algorithm. This table, along with the algorithms above, tell us which parts of code should our optimization efforts focus on.

1.2.2 Algorithm vectorization - Iteration 1

Initially, a naive, top-to-bottom approach was attempted when transforming the algorithm into a vectorized form. This meant starting at the algorithm's entry point and going deeper and deeper. However, we quickly realized that this is not a way to go. The programming was very cumbersome, almost prohibitive to incremental implementation. Combining the knowledge of the algorithm and the performance analysis table above, vectorization effort was switched to identifying places that are easy and straightforward to vectorize and places where the execution spends the most time in (hot-spots).

¹FPS - Frames Per Second

²HD - High Definition

The best candidate for the first category is the spline evaluation on lines 9-11 of algorithm 2. We have 3 sets of input values for the same logic outputting 3 results - perfectly suited for the SIMD³ programming model of SSE⁴. In practice, all that is needed is to group the input values into a vector and change the scalar mathematical operators for their vectorized counterpart from the SSE instruction set. But as the analysis of the scalar version shows, evaluating the splines takes only about 1/6th of the total execution time. Thus, vectorizing this part will not provide very significant speed-up.

More interesting candidate for speed improvements seems to be *GradientNoise* function. The program spends executing its code half of the total time. Let us take a closer look at the code and see which parts look suitable for vectorization. Calculating an index into the gradient table doesn't look favorable. The only part of it that could be transformed into a SIMD version is the product of the cube cell vertex coordinates and the hashing constants. But these values can be pre-calculated, as every coordinate is always multiplied with the same constant. The rest of the logic, however, looks more promising. Thanks to the way gradient vectors are stored in the gradient table, instead of 3 separate memory accesses, only one vector load from memory can be used to get all the required data. Calculating the distance of the sample point from the cube cell vertex is also straightforward to vectorize. Final dot product of the gradient and distance is, however, a bit tricky. The first part, multiplication, can be done right away. Sum of the results can be obtained by two subsequent *haddps* instructions from SSE3 (see figure 1.3).

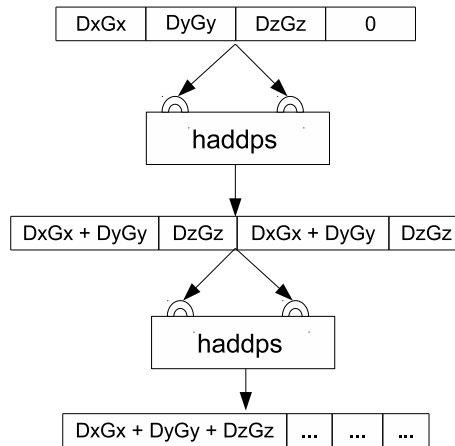


Figure 1.3: Two *haddps* instructions to get a single dot product

Due to the way *haddps* works ([3], section 12.3.5), for one set of input values to *GradientNoise* function, this is a bit wasteful. The first instance of *haddps* uses only a half of the SSE vector for useful calculation, the second only one a quarter. One of the solutions to this problem is to make use of the fact that for 3D improved Perlin noise, *GradientNoise* is called 8 times - once for every vertex of the cube cell. We can thus partially evaluate the *GradientNoise* function for the 8 instances up to the point of the sum and store the results in an array of 8 vectors. We will input these in couples into the horizontal add instruction two

³SIMD - Single Instruction, Multiple Data

⁴SSE - Streaming SIMD Extensions

times. The result will be two vectors containing the 8 required values. This way the number of *haddps* instructions is reduced from 16 (2 for each *GradientNoise*) to 6 (see figure 1.4).

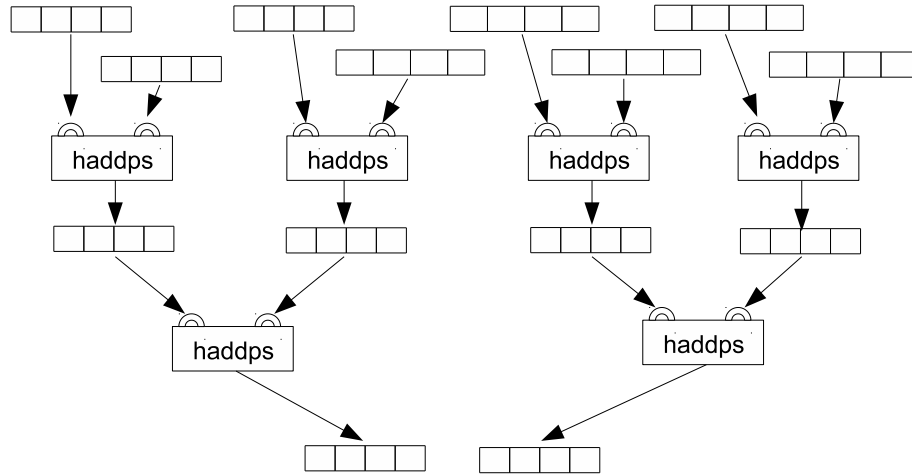


Figure 1.4: Efficient use of *haddps* instruction when calculating gradient dot products

We now have spline evaluation fully vectorized and the gradient linear functions partially vectorized (except for the calculation of the index into the gradient table). With the results from these, all that remains are the linear interpolations. Vectorizing the interpolation algorithm is straightforward, requiring just the substitution of scalar mathematical operators with their SSE counterparts. However, only the first interpolation (along X axis) can fully make use of the 4-element SSE instructions. Interpolations along Y and Z axes will for now remain unchanged.

Let us sum up the first iteration of the algorithm vectorization, now from the top-to-bottom point of view. The first part of the algorithm, dealing with noise octaves (algorithm 1), was not vectorized at all as it is at the very top and requiring only a small fraction of the computation time. Going deeper, calculating the cubic lattice surrounding cell vertices (algorithm 2, lines 1-6) is not vectorized either. Since this represents majority of the actual code of the *GradientCoherentNoise* function (rest of its code are calls to other functions), it uses up around 2 seconds (1/6th) of the CPU time. Its vectorization will be considered later. At this moment, the input point $P = (x, y, z)$ is put into a vector and splines are evaluated (algorithm 2, lines 8-10). From the resulting vector, individual interpolation parameters are extracted. Indices to the gradient table are pre-calculated (in SISD⁵ fashion) and stored into an array. With these, 8 linear functions $N_{i,j,k}(P)$ (*GradientNoise*) are evaluated and the results interpolated - all vectorized. The final Improved Perlin noise value is obtained.

Results

With the core of the algorithm vectorized, we can measure how well the SIMD version performs compared to the original algorithm. As before, the measurement will be done with versions compiled by 3 compilers - MinGW, Intel Icc and

⁵SISD - Single Instruction, Single Data

Microsoft Msvc. For the non-vectorized version, compiler flags are the same as before. The SIMD version uses instructions from the SSE, SSE2, SSE3 instruction sets, so some additional flags are specified:

1. GNU MinGW G++ cross-compiler for 64-bit, version 4.7.2
 - -msse3
2. Microsoft MSVC 2012 cl compiler for 64-bit, version 15.00.21022.08
 - /arch:SSE
3. Intel C++ Composer XE 2013 Icc compiler for 64-bit, version 13.1.1.171
 - /arch:SSE

The program used to evaluate the speed is the same as before - calculating 16 million 3D noise values. Here are the results:

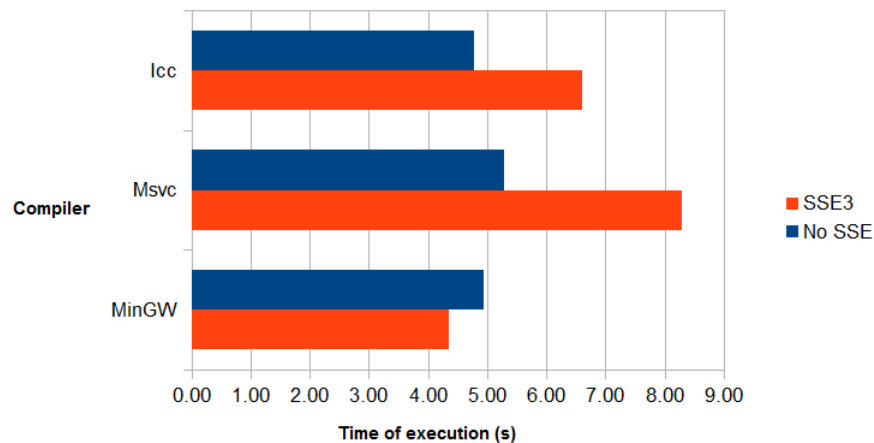


Figure 1.5: Speed measurement of the Improved Perlin noise after the first iteration of vectorization

The results (in figure 1.5) are a bit surprising. Only the MinGW compiled version was able to show some speed gains, albeit very small. The vectorized version is only 14% faster than the original. Icc and Msvc compiled versions run significantly slower, by 38% and 57%, respectively. Let us try to find out what might be causing this behavior. The SSE instructions can theoretically speed-up the execution 4 times. But only for code that has been efficiently vectorized. We will analyze the Intel Icc version again, with function inlining turned off. This should give us an idea of how much time was spent executing non-vectorized and vectorized code.

Function	Total time inside	Self time inside
perlin3	15.08s	0.04s
-GetValue	15.04s	0.46s
-GradientCoherentNoise	14.15s	2.17s
—SCurve5V	2.60s	0.70s
—calculateNoisePartSSELocal0_7	4.88s	2.82s
—initializeTableIndices	1.79s	1.79s
—LinearInterpV	1.17s	0.48s
—LinearInterp	0.67s	0.67s
—vectorize4F	0.50s	0.50s
—vectorSub	0.33s	0.33s
-MakeInt32Range	0.38s	0.38s

Table 1.2: Hot-spot analysis of the Improved Perlin noise algorithm after first iteration of vectoization

The blue cells in the table 1.2 contain times that are spent executing non-vectorized code. In total, that is 5.47 seconds. Let us assume this part of code in the vectorized version is the same as in the original version. The run time of the original algorithm (when analyzing) was 12 seconds. This means we have vectorized around 55% of the code. Theoretically, using SSE instructions should speed-up the vectorized code 4 times (when using single-precision floats). In our case, the program could theoretically finish in $45 + (55/4) = 59\%$ of the time of the original algorithm - a 69% speed-up. Since the results we have gotten are far away from this value (even slower than the base version in some cases), this means our vectorization is very inefficient. We need to find out why and hopefully come up with improvements to the algorithm that will solve these problems.

One of the more obvious reasons is that most of the time, we are using the potential of SSE only for 75%. We are working with a 3D Improved Perlin noise and thus most SIMD instructions are usefully executed only on 3 values instead of the available 4 (the exceptions being horizontal addition when calculating gradient dot products, although only partially, and the first interpolation). Setting data to SSE vectors and retrieving them back also takes some time (see total time of *vectorize4F* in table 1.2). Some vector shuffling was also necessary to get proper data into SSE vectors.

Let us also take a look at how we did the vectorization. Compared to the base algorithm, we are keeping a lot of intermediary results in temporary arrays - pre-calculated table indices and gradient dot products before horizontal addition. In a 64 bit environment, the CPU offers 16 general purpose registers and 16 SSE vector registers XMM0 - XMM15 [3]. This does not leave the compiler much room to operate in (in terms of available registers), especially if it is not able to generate efficiently optimized SSE code and/or lacks efficient register allocation algorithms. Some work has been done comparing these compilers' ability to produce quality vectorized code when using compiler SSE intrinsics [6]. Also, it is known the MinGW compiler (being based on GNU compiler suite) does implement advanced register allocation techniques [7]. Our efforts should thus concentrate on minimising the use of temporary arrays.

1.2.3 Algorithm vectorization - Iteration 2, final version

While analyzing the results obtained after the first iteration of vectorization, we have identified 3 main areas where our work with SSE is not efficient as it could be. The fact that, most of time, we use only 3 out of 4 possible vector elements for useful calculation, is given by the way we chose to vectorize the 3D version of the Improved Perlin noise. The advantage of this approach is that it should be easily extendible to the fourth dimension, which will use the full potential of the 4-element instructions. Thus, in our next efforts, we will focus on increasing the amount of vectorized code and avoiding the use of temporary arrays to improve the work with SSE registers.

As before, let us begin with a bottom-to-top approach and start with calculating the gradient dot products. In this part of the code we use temporary arrays to store both indices to the gradient table as well as the dot products themselves. Additionally, the table indices calculation does not use any SSE features at all. We described in the section 1.2.2 the way gradient dot products calculation was vectorized. The problem is that the tree representing the calculation was evaluated breadth-first - thus the need for the temporary result array (see figure 1.6).

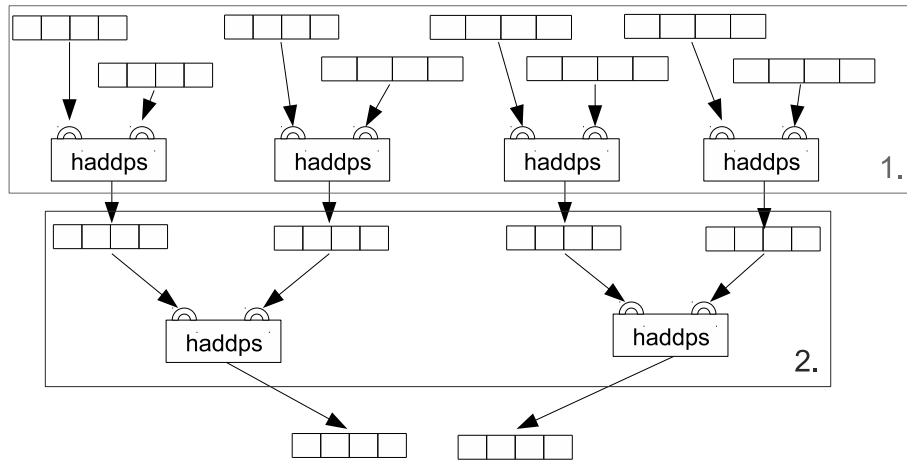


Figure 1.6: Breadth-first calculation of gradient dot products

If we evaluate the tree depth-first, we can significantly reduce the amount of SSE vectors required to store the intermediate results. Following the picture below (1.7) - for the first *haddps* instruction, at most 2 vectors are required; 3 vectors for the second (2 plus the result from the first *haddps*); and just 2 for the third, resulting in one vector containing the gradient dot products. The second half follows the same logic. This means that at the most we require 4 SSE vectors to calculate all the dot products. We were thus able to halve the amount of the necessary computing resources for this calculation.

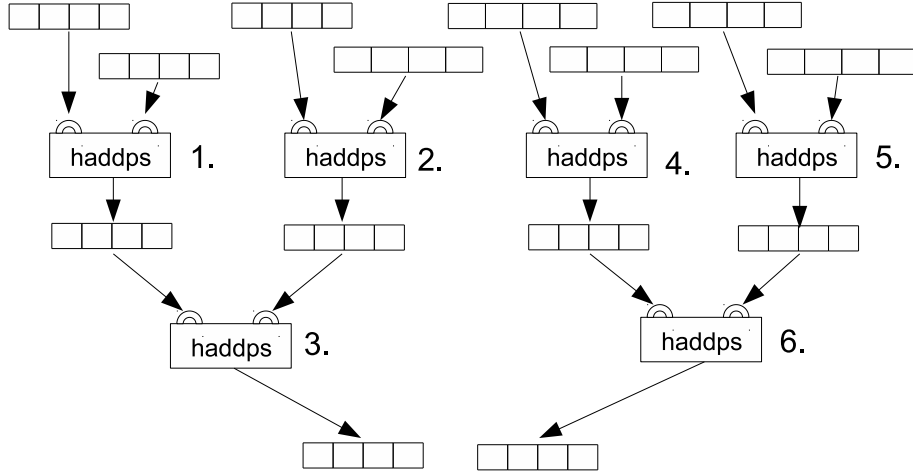


Figure 1.7: Depth-first calculation of gradient dot products

As can be seen above, we are calculating the gradient dot products in groups of 4. Since each of these dot products requires an index to the gradient lookup table, finding these indices can now use the full potential of the SSE SIMD model and will help us remove the second temporary array. The only problem is that efficient multiplication of 4 integer values is available only in the SSE 4.1 instruction set ([3], section 12.10.1). Only relatively new processors support this instruction set. For CPUs supporting only up to SSE2 instruction set, we are using a less efficient way to do integer multiplication.

With the improvements described above we were able to fully vectorize the gradient dot products calculation and get rid of all the temporary arrays. All that is left is a fairly straightforward vectorizing of the rest of the algorithm. So far we have used SSE instructions to calculate only the first set of interpolations after obtaining the gradient dot products. Instead of extracting values from the resulting vector and doing the rest of the interpolation in a scalar manner, we can just shuffle the resulting vector to properly align the values and continue with vectorized interpolation. Two such interpolations are needed to obtain the final noise value. Even though we do not fully use the SSE potential for these calculations (only half of the vector for the first and only one element for the second is used usefully), we can still obtain some speed improvements not just by the vectorization, but also by avoiding the need to extract 4 scalar values from the very first interpolation vector.

Moving further to the top, we still have not done anything with calculation of the surrounding cell of the sample point P in the integer cubic lattice. The core of this piece of code is finding a floor value of a float. SSE does not have direct support for find a floor of a number. However, this operation can be implemented with a few SSE instructions. First, we convert the input value into a 32 bit integer and truncate the decimal part. If the input value is less then zero, we subtract 1 and convert back into a floating point number, which now represents the floor of the input. With the floor calculated, the rest of this part is fairly straightforward to vectorize. With this we have finished our work with calculating noise value for one octave. The last part of the algorithm can be vectorized only partially - the modification of the sample point P for each octave, namely multiplying by *lacunarity* and keeping the modified point inside 32-bit

integer value range boundary. This represents the final version of the Improved Perlin noise vectorization.

Results

Let us see what kind of improvements in the performance were gained with our changes. At this point our implementation supported several levels of SSE features (SSE2, SSE3 and SSE4.1), so we will provide measurements for all of them. These are the compiler flags used for compilation:

- GNU MinGW G++ cross-compiler for 64-bit, version 4.7.2
 - SSE2: DLIBNOISE2_USE_SSE2
 - SSE3: -msse3 -DLIBNOISE2_USE_SSE3
 - SSE4.1: -msse4.1 -DLIBNOISE2_USE_SSE4.1
- Microsoft MSVC 2012 cl compiler for 64-bit, version 15.00.21022.08
 - SSE2: /arch:SSE -DLIBNOISE2_USE_SSE2
 - SSE3: /arch:SSE -DLIBNOISE2_USE_SSE3
 - SSE4.1: /arch:SSE -DLIBNOISE2_USE_SSE4.1
- Intel C++ Composer XE 2013 Icc compiler for 64-bit, version 13.1.1.171
 - SSE2: /arch:SSE -DLIBNOISE2_USE_SSE2
 - SSE3: /arch:SSE -DLIBNOISE2_USE_SSE3
 - SSE4.1: /arch:SSE -DLIBNOISE2_USE_SSE4.1

And here are the results:

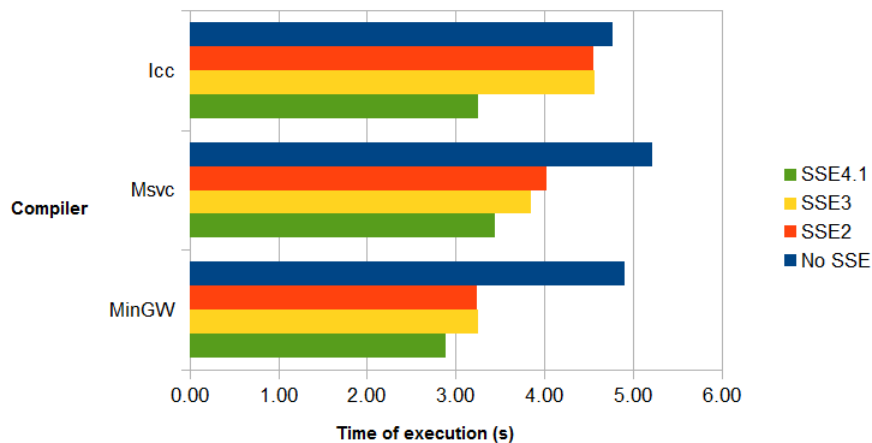


Figure 1.8: Speed measurement of the Improved Perlin noise after the second iteration of vectorization

As can be seen in the figure 1.8, the performance gains we have achieved over the previous version are significant. With every level of SSE support, the

vectorized version runs faster than the base algorithm. The SSE 4.1 version runs, as expected, the fastest, as it uses additional SSE features over the SSE2 and SSE3 versions that help speed up parts of the code (efficient 32-bit integer multiplication). To compare the differences between compilers, the fastest code is produced by MinGW - 2.88s, 70% speed-up over base version (4.90s); then Intel Icc - 3.25s, 47% speed-up (4.77s); and Microsoft Msvc - 3.44s, 52% speed-up (5.22s).

However, even though we were able to vectorize almost the entire algorithm, the speed improvements are far from the theoretical 300% speed-up. This is due to numerous reasons. As already mentioned, the 3D version uses the SSE potential to only 75% most of the time. Also, some addition steps were needed to shuffle the values in the vectors before some useful calculation could be done. The optimization algorithms in the compilers could not be as efficient when working with SSE intrinsics as when optimizing non-vectorized code - indication of this can be the rather significantly different performances across the compilers shown in the results.

Other dimensions - 1D, 2D, 4D

The way we chose to vectorize the computation of one Improved Perlin noise value lends itself to be easily extensible to support other dimensions. The 4D version can gain the most from this approach, as it will remove one of the deficiencies of the 3D version - not making full use of the 4 value vectors in most calculations. However, the 2D version will suffer from this fact even more and thus no significant improvements can be expected. The 1D version becomes in practice exactly the same as the base algorithm and thus for this dimension, no SSE vectorized code was created. Measurements across the various dimensions can be found in chapter 4.

1.2.4 Algorithm vectorization - Four values at once

In sections 1.2.2 and 1.2.3 we were trying to optimize the calculation of a single Improved Perlin noise value. One of the disadvantages of the approach we have chosen was the decreasing utilization of the SSE SIMD potential with lower dimensions - up to a point of 1D version, where no part of the code can be reasonably turned into a vectorized form. One way to overcome this problem is to calculate 4 noise values at once and thus potentially fully utilize the vector capabilities of the SSE instruction sets.

Going through the original algorithm again (algorithms 1, 2, 3), it seems to be fairly straightforward to vectorize the code. Only basic mathematical operations are used for the calculation and for all of them exists an equivalent SSE instruction. Some parts of the code from sections 1.2.2 and 1.2.3 can be reused as well - namely calculating the cell in the integer cubic lattice, interpolations, spline evaluation and modulating a float into a value range of a 32 bit integer. The only problem is an efficient gathering of gradient vectors from the precomputed gradient vector table.

The gradient vector table contains 256 precomputed, 4 element gradient vectors (G_x, G_y, G_z, G_w). Evaluation of the noise at each sample point generates 2, 4, 8 or 16 lookups into the table, for 1D, 2D, 3D and 4D, respectively. These

individual lookups are unrelated between each other and thus each can extract values from a different part of the table. Since we will be calculating 4 values at once, we will be calculating also 4 gradient dot products at once, meaning each such grouped calculation will require 4 lookups into the table (see figure 1.9). To be able to use the gradient values we will need to move the individual elements around the vectors so that gradient elements corresponding to each axis end up together - for example, for the X axis a vector $(G_{x1}, G_{x2}, G_{x3}, G_{x4})$. Instead of extracting the values into temporary arrays and move around one by one, we will employ the features of SSE to make this process more efficient. First, we interleave the low elements (elements 1 and 2) of the gradient vectors in groups of two using the *unpcklps* instruction ([3], section 10.4.2.2). Afterwards, we shuffle the resulting 2 vectors using the *shufps* instruction ([3], section 10.4.2.2) to obtain the required gradient vector (see figure 1.10). This same approach can be applied for all axes x, y, z and w , except that for the z, w axes, interleave of the higher elements (3 and 4) is required using the *unpckhps* instruction ([3], section 10.4.2.2).

As a side note, AVX2 instruction set supports gathering data from different parts of memory into the vector registers using the *vgatherdps* ([3], table 13-20) instruction, which could be used to avoid the need to move the gradient vector values altogether. However this instruction set was at the time of writing this thesis available only in the latest Intel® CPU line codenamed Haswell [2], which was only beginning to appear on the market.

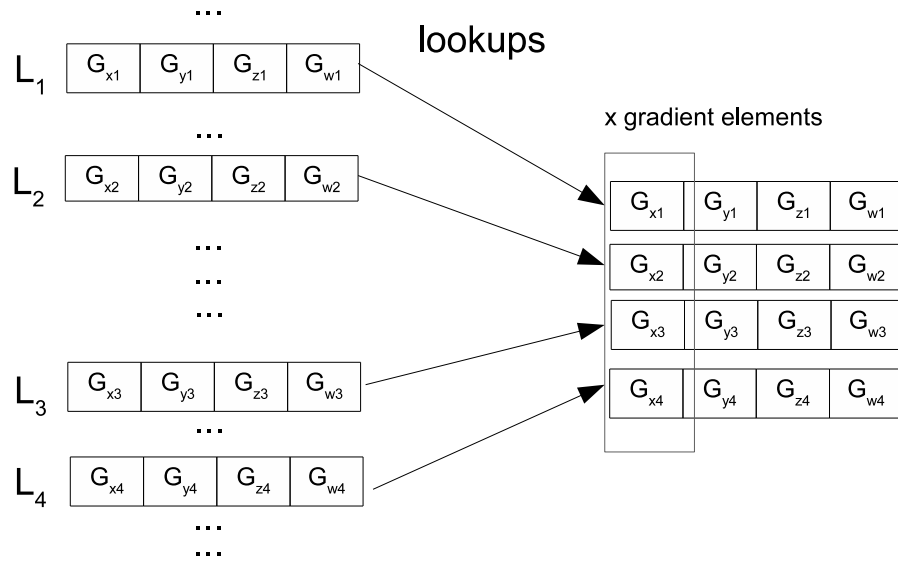


Figure 1.9: Gathering of gradient vectors from the gradient vector table

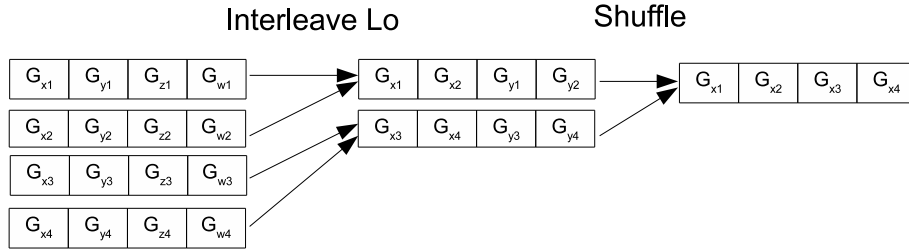


Figure 1.10: Moving gradient vector values to group them according to axes

Results

Let us now take a look at how vectorization of the computation of 4 noise values at once performs. The evaluation program is the same as before - calculating sum of 16 million noise values. We will look at the 3D version to get an idea of the speed-ups achieved. Measurement for all the dimensions can be found in chapter 4. The same compiler flags were used as in all the other measurements.

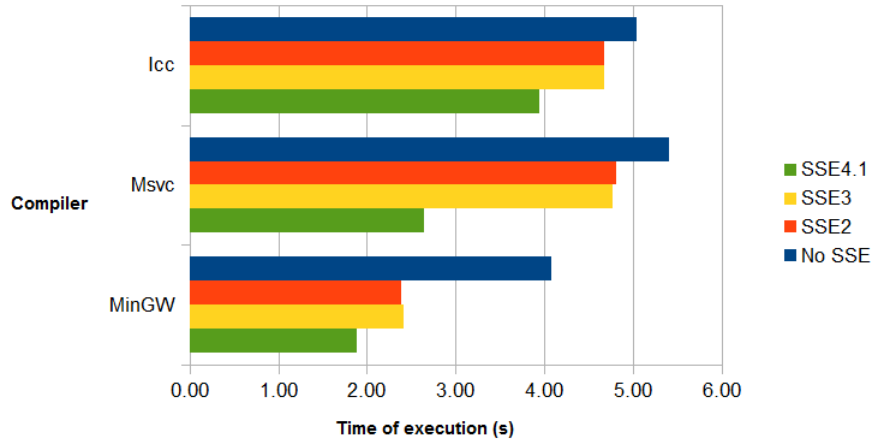


Figure 1.11: Speed measurement of the Improved Perlin noise 4-at-once version

Taking the results achieved with the MinGW compiler, we were able to achieve speed-up of 116% in comparison to the 70% speed-up achieved with calculating only one value. This results were somewhat expected, as almost the whole algorithm could be very efficiently vectorized. The only part of code that required some additional computation on top of the base algorithm was shuffling the gradient vector elements. This required both additional logic and vector registers for temporary results, which contributed to the fact that the observed speed-up does not approach the theoretical limits.

Interesting finding is the not so good performance of the Intel Icc compiled version, achieving speed-up of only 28%. This is significantly slower than what was achieved with the vectorized calculation of a single noise value. It is even slower than the version produced by the Microsoft Msvc, both in terms of relative speed-up (28% for Intel Icc vs 104% for Msvc) and absolute execution time - 3.94s for Intel Icc vs 2.64s for Msvc.

Even though we were able to achieve better efficiency and utilization of the SSE resources when calculating 4 noise values at once, this approach cannot be

effectively used in every use-case. An example can be a ray-tracing engine with support for procedural surface evaluation, such as Mental-Ray. In it, programs describing object materials are evaluated for individual points (surface-ray intersections). Depending on the material, only one noise value may be required for the calculation. We will focus on integrating the vectorized Improved Perlin noise into Mental Ray and Maya in section 1.4.

1.3 Perlin-derived noises - Billow and Ridged-Multi

We will look briefly onto two other types of noise functions that are based on the Perlin noise - Billow noise and RidgedMulti noise. The core of their algorithm is exactly the same as that of the Improved Perlin noise. The difference is the way each one modulates intermediate noise values for individual octaves.

The Billow noise algorithm modulates the octave noise value simply - it finds its absolute value. As can be seen in algorithm 4, when calculating one noise value, this part of code cannot be vectorized at all. Since the amount of code that can be vectorized becomes smaller, we expect the resulting speed-up to be slightly lower than in the case of the Improved Perlin noise.

Algorithm 4 Improved Perlin gradientCoherentNoise

```

1: struct BillowProps
2:     frequency : float
3:     octaveCount : uint
4:     lacunarity : float // per octave frequency modulation
5:     persistence : float // per octave amplitude modulation
6:     noiseQuality : enum {FAST, STD, BEST}
7:     seed : int
8: end struct

9: function BILLOW(x, y, z: float; props: BillowProps)
10:    // ...
11:    for ... do // inside the octave calculation loop
12:        // signal ← gradientCoherentNoise(x, y, z, octaveSeed, props.noiseQuality)
13:        // noise ← noise + (signal × octavePersistence)
14:        signal ← gradientCoherentNoise(x, y, z, octaveSeed, props.noiseQuality)
15:        signal ← 2.0 × abs(signal) − 1.0)
16:        noise ← noise + (signal × octavePersistence)
17:    end for
18:    // ... return noise
19: end function

```

The RidgedMulti algorithm follows a similar pattern, but the modulation is a bit more complex (see algorithm 5). We can thus expect the speed-up to have a similar character as with the Billow noise - it should even be a little lower, since the amount of non-vectorizable code is larger than in the Billow noise. As a side note, the values returned from the SpectralWeight function can be pre-calculated

for a given set of RidgedMulti noise properties, as they do not depend on the sample point. In the presented algorithm, calling the function for every octave and sample point is done only for convenience.

Algorithm 5 Improved Perlin gradientCoherentNoise

```

1: struct RidgedMultiProps
2:   frequency : float
3:   octaveCount : uint
4:   lacunarity : float // per octave frequency modulation
5:   offset : float
6:   gain : float
7:   h : float // spectral weight exponent
8:   noiseQuality : enum {FAST, STD, BEST}
9:   seed : int
10: end struct

11: function RIDGEDMULTI(x, y, z: float; props: RidgedMultiProps)
12:   // ...
13:   weight : float
14:   weight  $\leftarrow$  0.0
15:   for ... do // inside the octave calculation loop
16:     // signal  $\leftarrow$  gradientCoherentNoise(x, y, z, octaveSeed, props.noiseQuality)
17:     // noise  $\leftarrow$  noise + (signal  $\times$  octavePersistence)
18:     signal  $\leftarrow$  abs(signal)
19:     signal  $\leftarrow$  props.offset - signal
20:     signal  $\leftarrow$  signal  $\times$  signal
21:     signal  $\leftarrow$  signal  $\times$  weight
22:     weight  $\leftarrow$  signal  $\times$  props.gain
23:     if weight > 1.0 then
24:       weight  $\leftarrow$  1.0
25:     else if weight < 0.0 then
26:       weight  $\leftarrow$  0.0
27:     end if
28:     noise  $\leftarrow$  noise + (signal  $\times$  SpectralWeight(curOctave))
29:   end for
30:   // ... return noise
31: end function

32: function SPECTRALWEIGHT(octave: int; props: RidgedMultiProps)
33:   freq : float
34:   freq  $\leftarrow$  props.frequency
35:   for i = 0  $\rightarrow$  octave do
36:     freq  $\leftarrow$  freq  $\times$  props.lacunarity
37:   end for
38:   return pow(freq, -props.h)
39: end function

```

On the contrary, calculating 4 values at once should provide better results, as

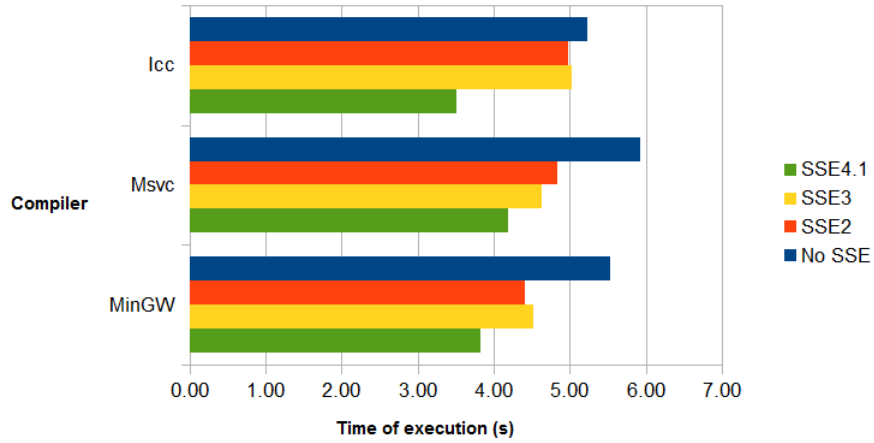


Figure 1.12: Speed measurement of 3D Billow noise

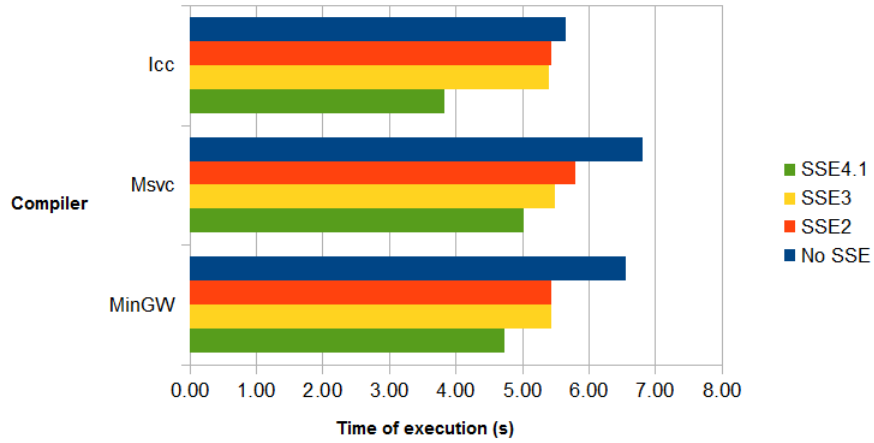


Figure 1.13: Speed measurement of 3D RidgedMulti noise

these parts of code can be straightforwardly vectorized, increasing the portion of code following the SIMD model. We can thus expect the SSE version of these noises, when calculating 4 values at once, will provide better overall speed-up than the Improved Perlin noise.

Results

All settings and measurement were done as before, so let us look directly at the results.

As was expected, due to the increased amount of code that cannot be vectorized when calculating single noise value, the total speed-up provided by the vectorization is lower than the calculation of pure Improved Perlin noise (see figures 1.12, 1.13). For Billow - 41%, 45%, 49% for Msvc, MinGW and Intel Icc, respectively. For RidgedMulti - 35%, 38%, 48% for Msvc, MinGW and Intel Icc, respectively.

Calculating 4 values once also show results that were expected, providing

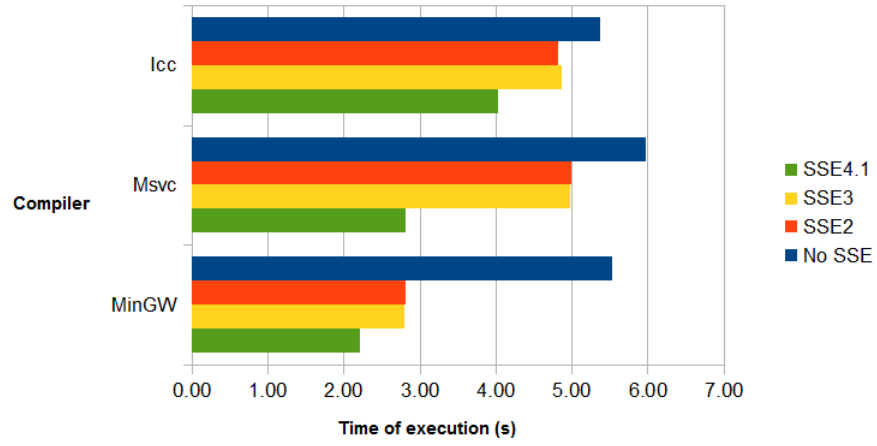


Figure 1.14: Speed measurement of 3D Billow noise, 4 values at once

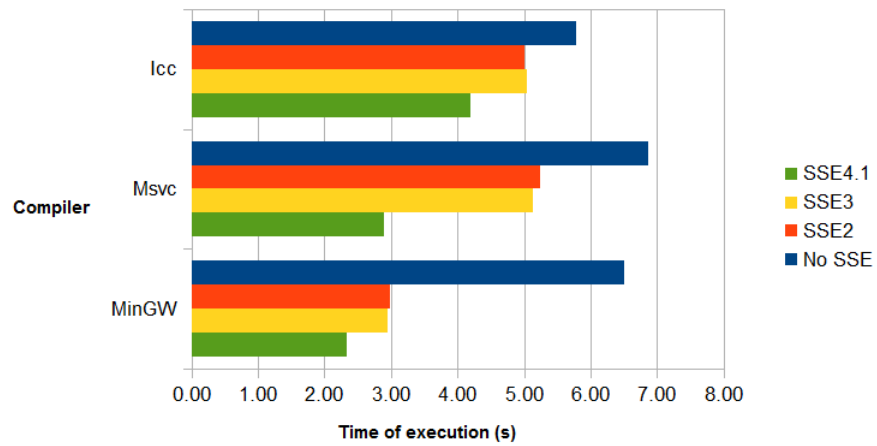


Figure 1.15: Speed measurement of 3D RidgedMulti noise, 4 values at once

higher overall speed-up than when calculating Improved Perlin noise (see figures 1.14, 1.15). However, the Intel Icc compiler continues to struggle during these calculations. For the Billow noise, relative speed-ups are 33%, 112%, 151% for Intel Icc, Msvc and Mingw, respectively; RidgedMulti relative speed-ups are 38%, 138%, 180% for Intel Icc, Msvc and MinGW, respectively.

1.4 Integration to Mental-Ray, Maya

The Mental Images MentalRay⁶ is a ray-tracing based rendering engine integrated into the Autodesk Maya 3D modelling and animation studio⁷. Due to their popularity, we will integrate our vectorized noise algorithms into their environment and show how they perform compared to the built-in noise routines both in terms of visual output and speed of rendering. We will use Maya 2012 x64 with MentalRay version 3.9.1.36.

1.4.1 Exposing the noise

We will provide the noise in form of a MentalRay volume shader (3D) with support for time-based change (1D). We will thus use the 4D noise modules. Due to the nature of ray-tracing engines, the vectorized version calculating 4 values at once can not be used efficiently. The version calculating single noise value will thus form core of the volumetric shader. It will be parametrizable with the following values:

- NoiseType: type of noise encoded in 4-digit integer (more details below)
- Frequency: noise frequency, default 1.0
- Lacunarity: per-octave frequency modulator
- Persistence: per-octave amplitude modulator
- OctaveCount: number of noise octaves
- Seed: seed for the pseudo-random number generator
- Time: 4-th noise dimension, mostly used for time-based noise change

As mentioned in the list above, the noise type is encoded using a 4 digit integer $d_l d_c d_s d_n$, where:

- d_l - encodes which noise implementation library to use
 - 1 - the original libnoise library
 - 2 - to use the implementation developed for this thesis
- d_c - encodes code to use based on the compiler that generated it
 - 1 - Microsoft MSVC

⁶<http://www.nvidia-arc.com/products/nvidia-mental-ray/>

⁷<http://www.autodesk.com/products/autodesk-maya/overview>

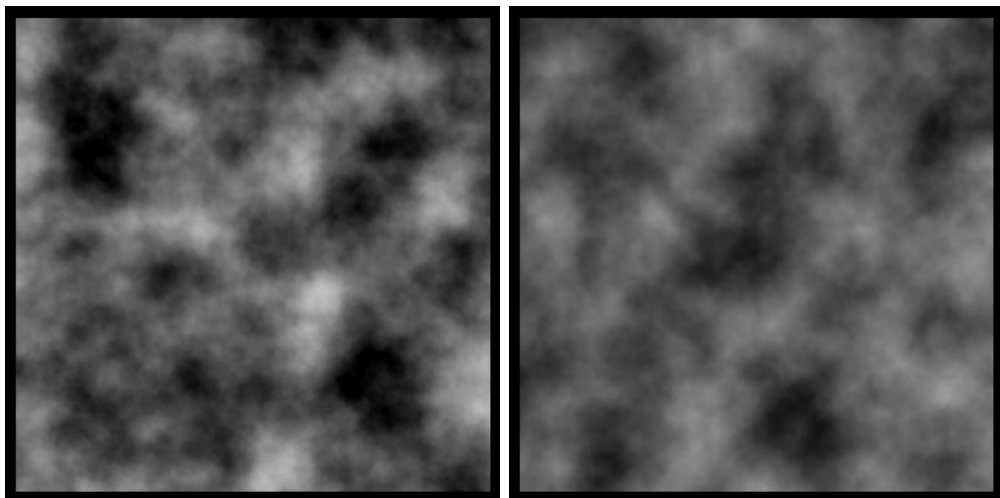
- 2 - Intel Icc
- 3 - GNU MinGW
- d_s - encoded whether to use version with SSE enabled or disabled
 - 0 - SSE disabled
 - 1 - SSE enabled
- d_n - encodes, which noise type to use
 - 1 - Improved Perlin noise
 - 2 - Billow noise
 - 3 - RidgedMulti noise

1.4.2 Comparing 2D visual output

Before we get to volumetric rendering and performance measurements, let us first take a look at renderings of a 2D plane. We will apply to it several materials based on our noise implementation and Maya's built-in noise shaders. This should allow us to compare the visual properties of the noises and find a mapping between the parameters of our noise shader and the built-in noises.

Improved Perlin noise

We will begin with the Improved Perlin noise. We will use the MinGW compiled version as this one performed the best when calculating single noise values. In the pictures below, the one on the left (1.16a) was generated by our noise shader, the one on the right (1.16b) by Maya's "Volumen noise" shader. Specific parameters used are listed in the table below (1.3), where each parameter of our noise on the left is mapped (if possible) to Maya's Volume noise's parameter on right. As can be seen, the generated noises look very similar in terms of noise features.



(a) 2D Perlin, our

(b) 2D Perlin, Maya

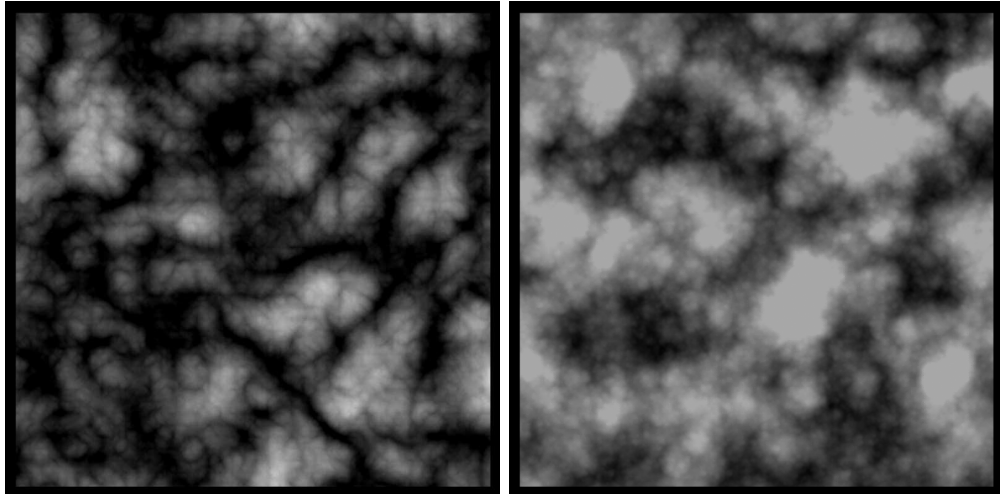
Figure 1.16: 2D Perlin noise renderings

Our noise params		Maya's Volume noise params	
Noise Type	2311	Noise Type	Perlin Noise
N/A		Threshold	0.0
N/A		Amplitude	1.0
Persistence	0.5	Ratio	0.5
Lacunarity	2.0	Frequency Ratio	2.0
Octave Count	6	Depth Max	6
Time	0.0	Time	0.0
Frequency	1.0	Frequency	1.0
N/A		Scale	(1.0, 1.0, 1.0)
N/A		Origin	(0.0, 0.0, 0.0)
N/A		Implode	0.0
N/A		Implode Center	(0.0, 0.0, 0.0)
N/A		Inflection	false
Seed	0	N/A	

Table 1.3: Our Perlin noise to Maya Perlin noise parameter mapping and values

Billow noise

We will now compare the Billow type of noise. As can be seen in the images below, the Billow Noise type of the Maya's Volume shader (figure 1.17b) looks very differently than our Perlin-based Billow noise implementation (figure 1.17a).

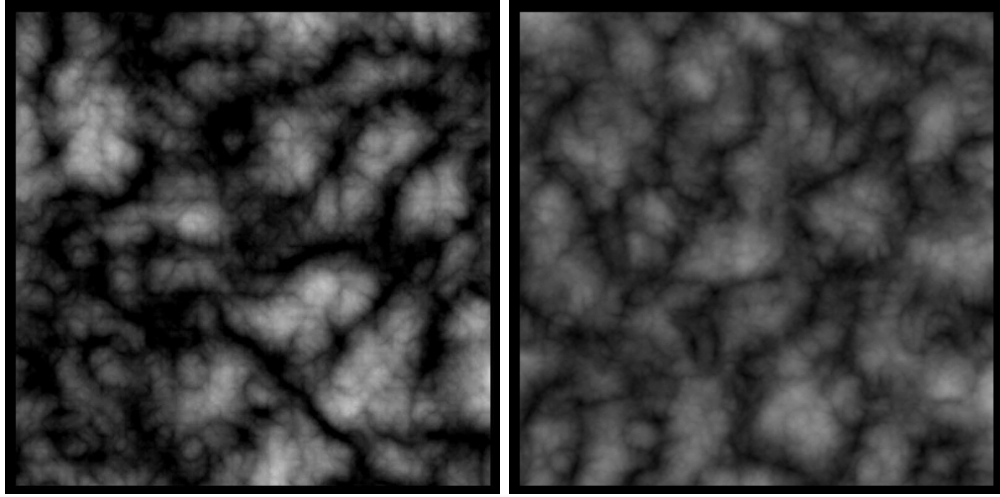


(a) 2D Billow, our

(b) 2D Billow, Maya

Figure 1.17: 2D Billow noise renderings

However, after playing around with different parameters of the Perlin-type Maya's Volume shader, we were able to find out that enabling the option "Inflection" produces very similar results to our Billow noise. These are the results (see figures 1.18a and 1.18b), with table of parameters below (1.4).



(a) 2D Billow, our

(b) 2D Perlin + Inflection, Maya

Figure 1.18: 2D Billow and Perlin with inflection renderings

Our noise params		Maya's Volume noise params	
Noise Type	2312	Noise Type	Perlin Noise
N/A		Threshold	0.0
N/A		Amplitude	1.0
Persistence	0.5	Ratio	0.5
Lacunarity	2.0	Frequency Ratio	2.0
Octave Count	6	Depth Max	6
Time	0.0	Time	0.0
Frequency	1.0	Frequency	1.0
N/A		Scale	(1.0, 1.0, 1.0)
N/A		Origin	(0.0, 0.0, 0.0)
N/A		Implode	0.0
N/A		Implode Center	(0.0, 0.0, 0.0)
N/A		Inflection	true
Seed	0	N/A	

Table 1.4: Our Billow noise to Maya Perlin noise parameter mapping and values

RidgedMulti noise

We were unable to find any built-in Maya material that would produce anything similar to the RidgedMulti noise type in our noise shader. Nevertheless, we present here the same scene as above with applied RidgedMulti noise (see figure 1.19) with table of parameters below for completeness (see table 1.5).

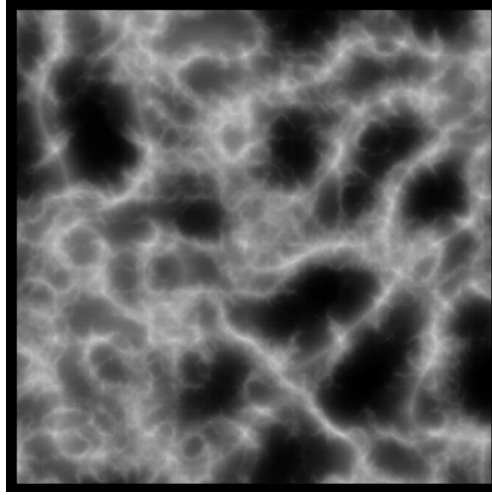


Figure 1.19: 2D RidgedMulti noise rendering

Our noise params	
Noise Type	2313
Persistence	0.5
Lacunarity	2.0
Octave Count	6
Time	0.0
Frequency	1.0
Seed	0

Table 1.5: RidgedMulti noise parameter values

1.4.3 3D volumetric rendering and performance

In section 1.4.2 we have focused on presenting detailed visual features of the various Perlin-based noises implemented throughout this chapter and finding a matching Maya's built-in materials to compare with. We will now focus mainly on the performance side of using the noise shaders. For this purpose, we have created a simple scene containing a red torus partially surrounded by a volumetric fog (see figure 1.20). To give the fog some structure, its density will be guided by various volume noises:

1. Constant - fog density constant (0.5) in every point. Will be used as a rough measurement of how much time was spend on just rendering the scene (excluding noise evaluation)
2. Perlin (Maya) - Maya Perlin noise
3. Perlin (Non-SSE) - Our non-vectorized Improved Perlin noise (MinGW version)
4. Perlin (SSE) - Our vectorized Improved Perlin noise (MinGW version)

5. Billow (Maya) - Maya Billow noise. Even though it provides noise with very different features than our Perlin-based Billow noise, we will have a look at how it performs
6. Perlin (Infl, Maya) - Maya Perlin noise with "Inflection" parameter enabled
7. Billow (Non-SSE) - Our non-vectorized Billow noise (MinGW version)
8. Billow (SSE) - Our vectorized Billow noise (MinGW version)
9. RidgedMulti (Non-SSE) - Our non-vectorized RidgedMulti noise (MinGW version)
10. RidgedMulti (SSE) - Our vectorized RidgedMulti noise (MinGW version)

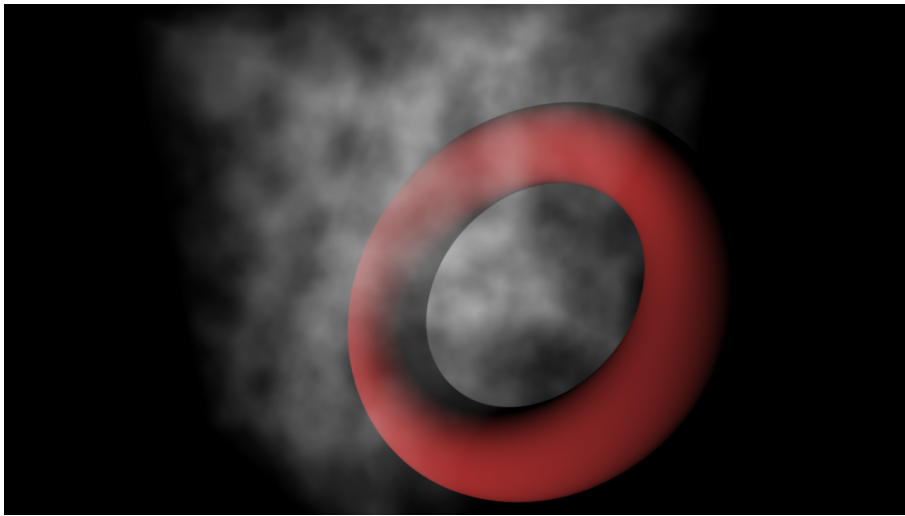


Figure 1.20: Volumetric fog scene

Each noise will be parametrized with the same values as presented in section 1.4.2, with the exception of *Frequency*, which will be set to 8.0. The scene will be rendered at 720p HD resolution. Time required for the rendering will be measured 3 times for each noise and the average of these 3 measurements will be compared.

Results

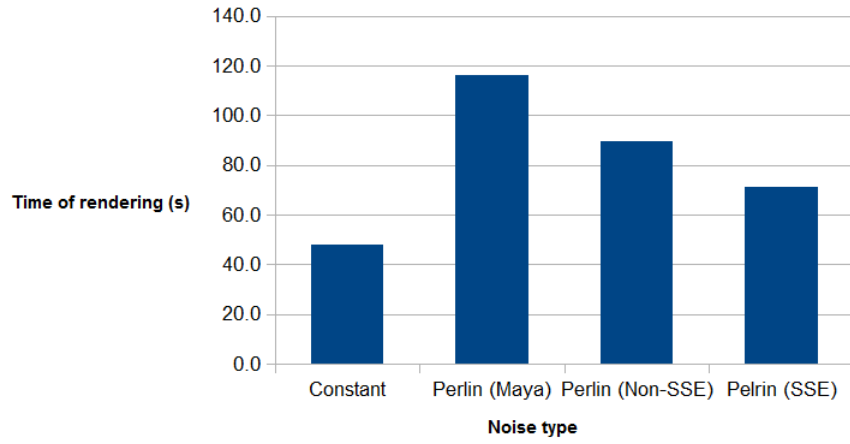


Figure 1.21: Perlin volumetric fog scene rendering time measurement results

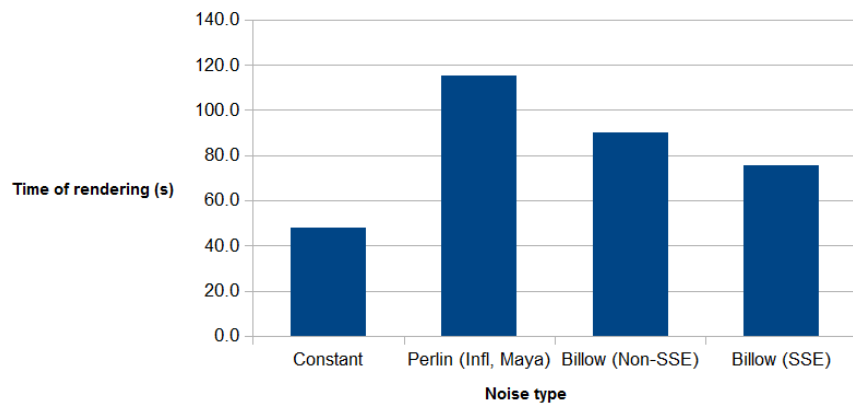


Figure 1.22: Billow volumetric fog scene rendering time measurement results

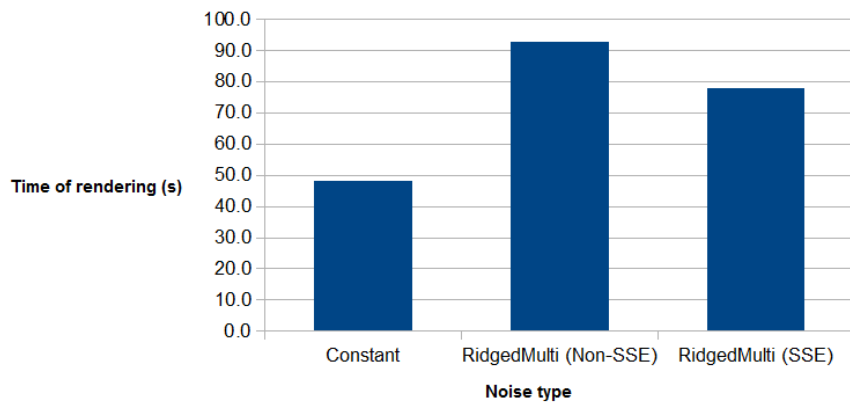


Figure 1.23: RidgedMulti volumetric fog scene rendering time measurement results

The measurements (see figures 1.21, 1.22, 1.23) show that the built-in noise shader in Maya is slower even than our non-vectorized version (Non-SSE). However, Maya’s noise is more feature-rich, which may have impact on its performance. The Billow Noise type of the Maya’s ”Volume noise” shader was not included in the charts, as it took around 14 minutes to render the scenes with it. It would thus make the charts hard to read. This, along with the comparison images from section 1.4.2 suggests, that Maya’s Billow noise is based on an entirely different algorithm. The table 1.6 provides rendering durations of all the noise types used (minutes:seconds).

Noise type	Rendering time (m:s)
Constant	0:48
Perlin (Maya)	1:56
Perlin (Non-SSE)	1:29.3
Perlin (SSE)	1:11.3
Perlin (Infl, Maya)	1:55
Billow (Maya)	14:01.3
Billow (Non-SSE)	1:30
Billow (SSE)	1:15.3
RidgedMulti (Non-SSE)	1:32.7
RidgedMulti (SSE)	1:17.7

Table 1.6: Average rendering times of all volumetric noise fog scenes

2. Gabor noise

2.1 Introduction to Gabor noise

Gabor noise is a relatively new addition to the family of algorithms for procedural noise generation, introduced by Ares Lagae, Sylvain Lefebvre, George Drettakis and Philip Dutré in 2009 [4] and inspired by sparse convolution noise introduced by J. P. Lewis in 1989 [5]. We will first describe, analyze and vectorize the 2D Gabor noise and later extend on the work to provide Gabor surface noise, suitable for integration into 3D modelling environments as a surface material.

2D Sparse convolution noise is a random pulse process Y which is defined a sum of randomly weighted and positioned pulses

$$Y(x, y) = \sum_i w_i h(x - x_i, y - y_i) \quad (2.1)$$

where h is the pulse, the random locations $\{x_i\}$, $\{y_i\}$ are guided by a Poisson distribution with mean λ (called *Impulse Density*) and weights $\{w_i\}$ are realizations of a random variable W zero mean. In the Gabor noise, the pulse (now named g) is a Gabor kernel [1] of the cosine type, which is a multiplication of a Gaussian envelope and a harmonic. Thus, 2D Gabor kernel g is defined as

$$g_{K,a,\omega_0,F_0}(x, y) = K e^{-\pi a^2(x^2+y^2)} \cos[2\pi F_0(x \cos \omega_0 + y \sin \omega_0)] \quad (2.2)$$

where K and a are the magnitude and width of the Gaussian envelope and F_0 and ω_0 are the magnitude and orientation of the frequency of the harmonic. With this, two versions of the Gabor noise can be defined - anisotropic

$$N_{K,a,\omega_0,F_0}(x, y) = \sum_i w_i g_{K,a,\omega_0,F_0}(x - x_i, y - y_i) \quad (2.3)$$

and isotropic

$$N_{K,a,F_0}(x, y) = \sum_i w_i g_{K,a,F_0}(x - x_i, y - y_i, \omega_{0,i}) \quad (2.4)$$

where the frequency orientations $\{\omega_{0,i}\}$ are realizations of the random variable W . These two definitions can be further generalized into a noise with controllable band limits which has a power spectrum that covers a specific region of frequency space

$$N_{K,a}(x, y) = \sum_i w_i g_{K,a}(x - x_i, y - y_i, F_{0,i}, \omega_{0,i}) \quad (2.5)$$

where the frequency magnitudes $\{F_{0,i}\}$ are realizations of the random variable W . Furthermore, if we sample $\{F_{0,i}\}$ and $\{\omega_{0,i}\}$ from intervals $[F_{0,min}, F_{0,max})$ and $[\omega_{0,min}, \omega_{0,max})$, we obtain a noise with power spectrum equal to a region of frequency space defined by $[F_{0,min}, F_{0,max}) \times [\omega_{0,min}, \omega_{0,max})$. With the generalized Gabor noise defined, the anisotropic noise can be obtained by $(F_{0,min} = F_0, F_{0,max} = F_0, \omega_{0,min} = \omega_0, \omega_{0,max} = \omega_0)$ and the isotropic by $(F_{0,min} = F_0, F_{0,max} = F_0, \omega_{0,min} = 0, \omega_{0,max} = 2\pi)$.

To achieve procedural evaluation of the Gabor noise, a quadratic grid is introduced and the properties of Gabor kernels are generated "on-the-fly" using a pseudo-random number generator. Size of the cells is to the radius of the truncated Gabor kernels, since Gabor kernels have infinite radius. The truncation occurs at points where the influence of the Gaussian envelope reaches 5% of its peak value K . Because of this, Gabor kernels are evaluated only in the cell that contains the sample point and all neighbouring cells. Consistency is achieved by proper seeding of the pseudo-random number generator with a seed that is always the same for a given grid cell. For a non-periodic noise, the seed for a cell $C_{i,j}$ obtained using the Morton order [8].

2.1.1 Base algorithm

Let us have a look at the base implementation of the Gabor noise, which is based on the sample implementation provided on the website of the Gabor noise project¹. In the entry point of the algorithm, represented by function *Gabor* (see algorithm 6), the sample point $P = (x, y)$ is first normalized to the size of the Gabor kernel. Afterwards, cell in the quadratic grid is located that contains the sample point and the point's position is transformed into the local space of the cell. For each cell that contains Gabor kernels that could influence the sample value, a *cell* function is called that returns sum of all such influences. Contributions from all these cells are summed and scaled with variance of the Gabor random process. The result is the Gabor noise value.

The *cell* function (see algorithm 7) is responsible for summing the contributions of all the Gabor kernels that influence the sampled point. First, proper seeding for the cell's pseudo-random number generator is found using the *morton* functions, which represents the Morton order. Afterwards, the number of impulses for the given cell is calculated. For each such impulse, pseudo-random parameters for the Gabor kernel are sampled. If the point is influenced by a Gabor kernel, the kernel is sampled and the value is added to total noise value. It needs to be noted that to retain consistency across multiple sample points, all Gabor kernel parameters need to be sampled for every impulse, not just for impulses that influence the sample point.

In algorithm 8 we list the supplemental functions that are used throughout the calculation of the Gabor noise. Two functions are worth a closer look - *uniformNormalized* and *poisson*, as these will be discussed further when vectorizing the algorithm.

¹<http://graphics.cs.kuleuven.be/publications/LLDD09PNSGC/>

Algorithm 6 Gabor noise entry point

```
1: struct GaborProps
2:   K : float // Gaussian envelope magnitude
3:   a : uint // Gaussian envelope width
4:   F0 : float // harmonic magnitude
5:   F0Spread : float // spread of the harmonic magnitude
6:   omega0 : float // harmonic orientation
7:   omega0Spread : float // harmonic orientation spread
8:   numberOfImpulses : float
9:   seed : integer
10: end struct

11: function GABOR(x, y: float; props: GaborProps)
12:   x  $\leftarrow x / \text{kernelRadius}(\text{props})$ 
13:   y  $\leftarrow y / \text{kernelRadius}(\text{props})$ 
14:   floorX : float  $\leftarrow \lfloor x \rfloor$ 
15:   floorY : float  $\leftarrow \lfloor y \rfloor$ 
16:   fracX : float  $\leftarrow x - \text{floorX}$ 
17:   fracY : float  $\leftarrow y - \text{floorY}$ 
18:   i : integer  $\leftarrow \text{floorX}$ 
19:   j : integer  $\leftarrow \text{floorY}$ 
20:   noise : float  $\leftarrow 0.0$ 
21:   for di = -1  $\rightarrow$  +1 do
22:     for dj = -1  $\rightarrow$  +1 do
23:       noise  $\leftarrow \text{noise} + \text{cell}(i + di, j + dj, \text{fracX} - di, \text{fracY} - dj, \text{props})$ 
24:     end for
25:   end for
26:   return noise / (3.0 * sqrt(variance(props)))
27: end function
```

Algorithm 7 Gabor noise cell function

```
1: function CELL(i, j: integer; x, y: float; props: GaborProps)
2:   s : unsignedinteger  $\leftarrow$  morton(i, j) + props.seed
3:   if s == 0 then
4:     s  $\leftarrow$  1
5:   end if
6:   impulsesPerCell : float  $\leftarrow$  impulseDensity(props) *
   kernelRadius(props) * kernelRadius(props)
7:   impulses : integer  $\leftarrow$  poisson(s, impulsesPerCell)
8:   noise : float  $\leftarrow$  0.0
9:   for i = 0  $\rightarrow$  numberOfImpulses do
10:    xi : float  $\leftarrow$  uniformNormalized(s)
11:    yi : float  $\leftarrow$  uniformNormalized(s)
12:    wi : float  $\leftarrow$  uniformRange(s, -1.0, +1.0)
13:    F0 : float  $\leftarrow$  uniformRange(s, props.F0 -
   props.F0Spread, props.F0 + props.F0Spread)
14:    omega0 : float  $\leftarrow$  uniformRange(s, props.omega0 -
   props.omega0Spread, props.omega0 + props.omega0Spread)
15:    xix : float  $\leftarrow$  x - xi
16:    yiy : float  $\leftarrow$  y - yi
17:    if ((xix * xix) + (yiy * yiy)) < 1.0 then
18:      noise  $\leftarrow$  noise + wi * gabor(props.K, props.a, F0, omega0, xix *
   kernelRadius(props), yiy * kernelRadius(props))
19:    end if
20:  end for
21:  return noise
22: end function

23: function GABOR(K, a, F0, omega0, x, y )
24:   gaussianEnvelope : float  $\leftarrow$  K * exp(- $\pi$  * (a * a) * ((x * x) + (y * y)))
25:   sinusoidalCarrier : float  $\leftarrow$  cos(2.0 *  $\pi$  * F0 * ((x * cos(omega0)) + (y *
   sin(omega0))))
26:   return gaussianEnvelope * sinusoidalCarrier
27: end function
```

Algorithm 8 Gabor noise supplemental functions

```
1: function MORTON(i, j: integer)
2:    $z : \text{unsignedinteger} \leftarrow 0$ 
3:   for  $i = 0 \rightarrow (\text{sizeof}(\text{uint32}) * \text{CHAR\_BIT})$  do
4:      $z \leftarrow z | ((x \& (1 \ll i)) \ll i) | ((y \& (1 \ll i)) \ll (i + 1))$ 
5:   end for
6:   return  $z$ ;
7: end function

8: function KERNELRADIUS(props: GaborProps)
9:   return  $\text{sqrt}(-\log(0.05)/\pi)/\text{props}.a$ 
10: end function

11: function IMPULSEDENSITY(props: GaborProps)
12:   return  $\text{props}.numberOfImpulses / (\pi * \text{kernelRadius}(\text{props}) * \text{kernelRadius}(\text{props}))$ 
13: end function

14: function VARIANCE(props: GaborProps)
15:    $\text{integralGaborFilterSquared} \leftarrow ((K * K) / (4.0 * a * a)) * (1.0 + \exp(-(2.0 * \pi * F0 * F0) / (a * a)))$ 
16:   return  $\text{impulseDensity}(\text{props}) * (1/3) * \text{integralGaborFilterSquared}$ 
17: end function

18: function UNIFORMNORMALIZED(s: out, unsigned integer)
19:    $s \leftarrow s * 3039177861$ 
20:   return  $s / \text{max}(\text{unsignedinteger})$ 
21: end function

22: function UNIFORMRANGE(s: out, unsigned integer; min, max: float)
23:   return  $\text{min} + (\text{uniformNormalized}(s) * (\text{max} - \text{min}))$ 
24: end function

25: function POISSON(s: out, unsigned integer; mean: float)
26:    $g : \text{float} \leftarrow \exp(-\text{mean})$ 
27:    $em : \text{unsignedint} \leftarrow 0$ 
28:    $t : \text{unsignedint} \leftarrow \text{uniformNormalized}(s)$ 
29:   while  $t > g$  do
30:      $em \leftarrow em + 1$ 
31:      $t \leftarrow t * \text{uniformNormalized}(s)$ 
32:   end while
33:   return  $em$ 
34: end function
```

2.2 Algorithm vectorization

We will stay true to the way we described vectorization of the Improved Perlin noise in section 1.2. We will begin by analyzing the scalar version of the Gabor noise. Then we will try to divide the algorithm into smaller parts with well-defined interfaces and identify portions suitable for vectorization. The transformation will be described incrementally. Charts representing speed gains will be provided throughout the process of vectorization.

2.2.1 Initial performance analysis

With the 2D Gabor noise algorithm defined, let's take a look at how it performs the base, non-vectorized version. The measurement will be done with a simple application that calculates the sum of 1 million noise values. As with the Improved Perlin noise, three versions will be tested, compiled by three different compilers:

1. GNU MinGW G++ cross-compiler for 64-bit, version 4.7.2
 - compilation optimization options: -O3
 - -O3 - full optimization, function inlining
2. Microsoft MSVC 2012 cl compiler for 64-bit, version 15.00.21022.08
 - compilation optimization options: /Ox /Ob2 /Oi /Ot
 - /Ox - full optimization
 - /Ob2 - inline any suitable function
 - /Oi - enable intrinsic functions
 - /Ot - favor fast code (as opposite to small size)
3. Intel C++ Composer XE 2013 icc compiler for 64-bit, version 13.1.1.171
 - compilation optimization options: /O3 /Ob2 /Oi /Ot
 - /O3 - highest optimizations
 - /Ob2 - inline any suitable function
 - /Oi - enable intrinsic functions
 - /Ot - favor fast code (as opposite to small size)

The execution measurements will be done on a system with these properties:

- OS: Windows 8, version 6.2
- CPU: Intel Core i5 3317U, 1.7 GHz, Turbo boost to 2.6 GHz, 2 cores, 4 threads
- Memory: 4 GB DDR3 1600 Mhz, CL11-11-11-28

These compiler parameters and system properties will be used for all subsequent measurements.

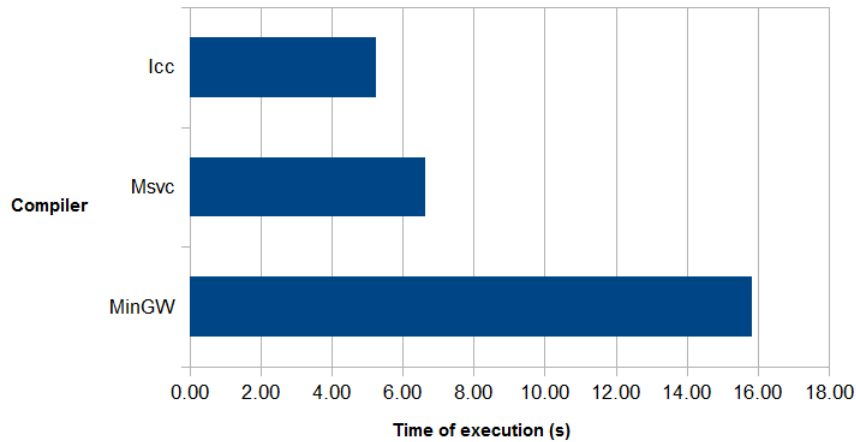


Figure 2.1: Execution time of the base 2D Gabor noise algorithm

To obtain the measurement data, each calculation was performed 7 times for each compiler version. The fastest and slowest results were discarded and the remaining 5 measurements were averaged. Unless stated otherwise, this procedure will be used for all subsequent time measurements.

As can be seen in figure 2.1, the base algorithm performs very differently across various compilers. This was, however, not the case with Improved Perlin noise, where the base algorithm versions performed similarly. The main difference between the Gabor noise and Improved Perlin noise is in the use of the standard library. In Gabor noise, functions for calculation of sine, cosine and exponential are used. Thus, to more accurately describe the results we have obtained, the base algorithm performs very differently across various implementation of the standard C++ library

We need to normalize the implementation of the trigonometric and exponential functions. Later in this chapter, vectorized version of these functions will be provided from a project² based on the Cephres library³. We took this vectorized implementation, rewrote it back into a scalar form and used in the base version of the Gabor noise. First, let us take a look at how the implementation of these functions performs compared to their counterparts in the standard library. We will have a look at the exponential and the cosine functions by calculating sum of 160 million values:

²<http://gruntthepeon.free.fr/ssemath/>

³<http://www.netlib.org/cephres/>

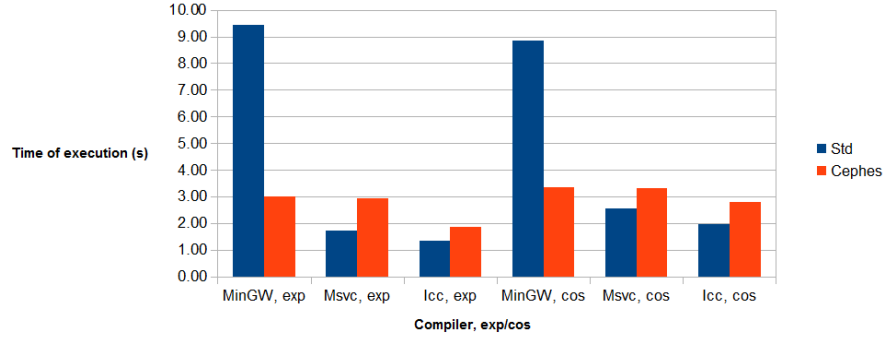


Figure 2.2: Exp/Cos "Cephes" vs standard library performance

The results in figure 2.2 establish the relationship between the speed of execution of the standard library and the our "Cephes" implementation. We will use this implementation in the base, non-vectorized Gabor noise algorithm. The "Cephes" implementation also offers algorithm that calculates sine and cosine of an angle at once, with only slight overhead. This feature is also employed when calculating sine and cosine values of the *omega0* angle in the *gabor* function (see algorithm 7, line 25). Here is how it performs:

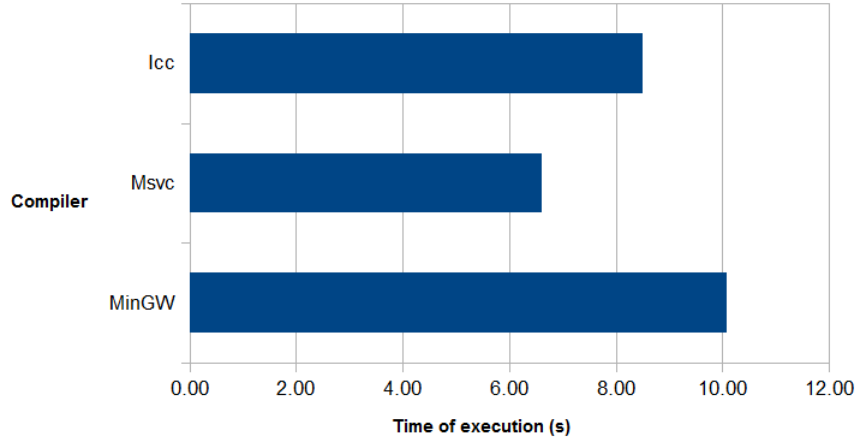


Figure 2.3: Execution time of the base 2D Gabor noise algorithm with "Cephes" implementation

The results in figure 2.3 are partially expected, partially surprising. As expected (from the results in figure 2.2), the Intel Icc version performs a bit worse than with the standard library implementation and the MinGW version performs better. Surprising is the fact that the Microsoft MSVC compiled version performance stayed the same. We have rechecked and made sure we use the "Cephes" implementation for this version too. We analyzed both Intel Icc and Microsoft Msvc versions with inlining turned off using the Intel VTune analyzer. With the inlining turned off, they perform almost equally - 13.46s for Intel Icc and 13.60s for Microsoft Msvc. We thus have to conclude that the Microsoft Msvc compiled version achieves such performance levels with full optimization.

We have already mentioned the Intel VTune analyzer, so let us look at the hotspot analysis of the Intel Icc compiled version, which will guide us in the following sections when vectorizing the algorithm (see table 2.1).

Function	Total time inside	Self time inside
<code>gabor2</code>	13.63s	0.002s
<code>-GetValue</code>	13.63s	0.06s
<code>-cell</code>	13.50s	1.81s
<code>—gabor</code>	5.29s	0.93s
<code>—uniformRange</code>	3.16s	1.75s
<code>—poisson</code>	1.11s	0.45s
<code>—uniformNormalized</code>	0.84s	0.66s
<code>—morton</code>	0.79s	0.79s
<code>-variance</code>	0.06s	0.02s

Table 2.1: Hot-spot analysis of the base Gabor noise algorithm

With the Gabor noise algorithm defined, resolved issue with various sine/cosine/exp implementations and performance and hotspot analysis of the base algorithm, we now have everything necessary to begin with the code vectorization.

2.2.2 Algorithm vectorization - Iteration 1

When vectorizing the Improved Perlin noise (see section 1.2.2), we have found out the best way to approach the vectorization is to do it bottom-to-top. We will try to follow this route with the Gabor noise as well, but we will do a slight exception right at the beginning to make the description easier to handle.

In the *GetValue* function, which represents the entry point into the algorithm, contribution from 9 cells are summed to obtain the noise value. We will provide 2 vectorized algorithms for the cell evaluation with the goal to achieve high SSE utilization. One of them will evaluate 4 cells at once, the other will vectorize evaluation of one cell. Preparing the input values for the *cell* function is a simple calculation so will not spent much time describing it. The only thing worth mentioning is that the order of the cell evaluation is guided by two static arrays which hold the relative indices of the central and neighbouring cells. In the first iteration, this order is kept the same as in the base algorithm, however, in later sections, we will focus on this aspect of the calculation as well.

Single cell evaluation

As can be seen in the hotspot analysis (see table 2.1), evaluation of the Gabor kernels (represented by function *gabor*) takes more than 1/3 of the total execution time. Looking at the code (see algorithm 7, lines 23-27), with use of the SSE implementation of the Cephes library, vectorizing the evaluation of the Gabor kernels is very straightforward. Our primary focus is thus providing the input values for the Gabor kernels in sets of four.

When evaluating single cell at a time, none of the code, up to the point of the Gabor kernel sampling, is very suited for the use of SSE. Thus, at lines 17 and

18 of the algorithm 7, instead of evaluating the kernel, we store the input values and impulse weight into a temporary array. When we have a set of 4 values (or we have obtained input values for the last impulse), we input the values stored in the temporary array into the vectorized *gabor* function and sums the results. This represent the noise contribution for this cell.

Evaluating 4 cells at once

To vectorize evaluation of 4 cells at once, in the first iteration, we chose to divide the algorithm into two parts. The first part will calculate, sample and store input values and weights for the Gabor kernels for all impulses across all 4 cells. The second part then goes through the stored input values, one impulse at a time and checks if it influences the sample point. If it finds 4 such impulses (or if all impulses were checked), it inputs them into the vectorized *gabor* function and sums the results to obtain the noise contribution for the 4 cells.

Let us look in more detail on the first part. Calculating 4 Morton order values at once using SSE is straightforward, as all the operations have their equivalent SSE instructions. The rest of the code samples the pseudo-random number generator to obtain the number of impulses for each cell, impulse position, impulse weight and the $F_{0,i}$ and $\omega_{0,i}$ Gabor kernel input values. However, we need to achieve consistency of the seeding of the pseudo-random number generator (PRNG from now on) across sampling of multiple points.

Obtaining the Morton order value, which serves as the initial seed for the PRNG, does not represent a problem in this regard. Finding the number of impulses does. It is governed by the Poisson distribution, represented in the function *poisson* (see algorithm 8). For each cell (each with different initial seed of the PRNG), different number of iterations of the *while* cycle is realized and each such iteration changes the seeding of the PRNG through the call to the *uniformNormalized* function. To obtain four Poisson distribution values at once, we need to make sure that for each cell, the *uniformNormalized* function is called proper amount of time and that the seeding remains consistent.

To achieve this, we backup the seeding for all 4 cells and keep track of the number of iterations for each cell. During each iteration of the *while* cycle, we do the following:

1. backup the current seeding of the PRNG for every cell
2. call *uniformNormalized* with the current seeding
3. blend the backup seeding with the new seeding, where key is the comparison between the t and g terms. Where t is greater than g for a given cell is zero, select the backup seeding, otherwise, select the new seeding
4. increment the number of iterations for cells where the new seeding was selected
5. if t is not greater than g for all cells, break the cycle

The figure 2.4 tries to visualize the vectorized calculation of the Poisson distribution. The middle column represents the mask for blending backup seeding with the new seeding and for incrementing the number of iterations (left column). The

right column contains the seeding pair - top one is the seeding backup and bottom one is the new seeding after calling *uniformNormalized*. These two seeding vectors are then blended to obtain the seeding backup for next iteration.

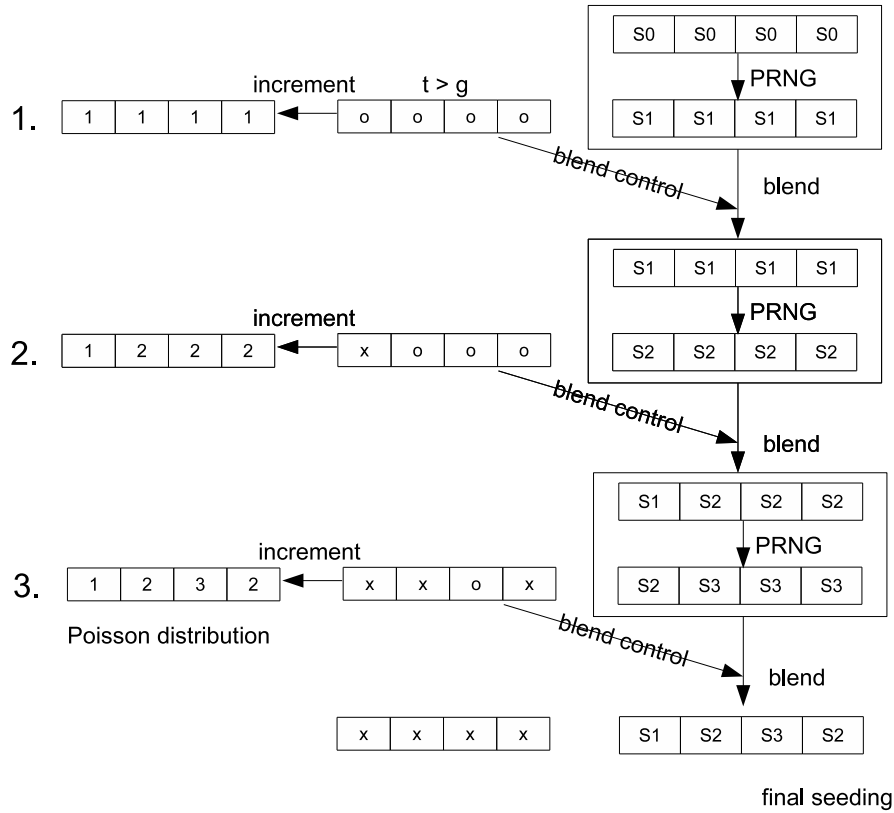


Figure 2.4: Visualization of vectorized calculation of Poisson distribution

With the number of impulses known for each cell, we can start sampling the Gabor kernel properties, positions and weights. These values will be generated 4 impulses at once, one for each cell. This approach does not threaten the consistency of the seeding of the PRNGs. The generated kernel properties are stored into temporary arrays big enough to hold values for all the impulses. To avoid costly allocation of the arrays on the heap, space for them was allocated on the stack. What problems this brings will be discussed in the "Results" section 2.2.2.

Results

Let us see how the code performs after the first iteration of vectorization. The implementation uses instructions from SSE, SSE2 and SSE3.

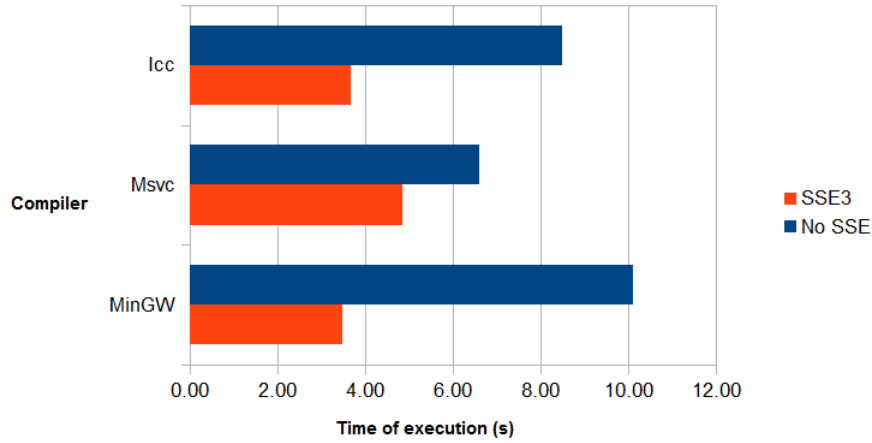


Figure 2.5: Speed measurement of the Gabor noise after the first iteration of vectorization

The results show we were able to do the vectorization fairly efficiently, as almost the whole algorithm was transformed into a SIMD execution model - except for the first part of the single cell evaluation, which is called only once, and the algorithm entry point. In absolute values, it took the GNU MinGW compiled version 3.49s to calculate the sum, 3.66s for the Intel Icc version and 4.85s for the Microsoft Msvc. Compared to their base non-vectorized versions, these represents speedups of 189%, 132% and 36%, for MinGW, Icc and Msvc, respectively.

We could accept this version of the algorithm as the final one, however, there is still one unresolved problem, which was already mentioned. The temporary arrays are created with a variable size (based on the number of impulses per cell) and on stack (to avoid heap allocation). This is, however, not supported by the C++ language and requires compiler or platform-specific extensions⁴. Apart from not being part of the standard, it also creates a potential threat of a stack overflow if it needs to allocate larger amounts of space. In the following sections, we will provide two possible ways of modifying the algorithm to avoid these allocations.

2.2.3 Algorithm vectorization - Fixed-sized local arrays

The first way of removing the need for variable-sized local arrays is a straightforward modification of the algorithm - instead of pre-calculating impulse properties for all the impulses and all cells, we will sample 4 at a time. From this group of 4, we will check which ones influence the sample point and if we find 4 of them, we call the vectorized Gabor kernel evaluation with them.

Results

At this point, the library supported various levels of SSE, we will provide measurements for all of them. For compiler parameters, see section 1.2.3.

⁴[http://msdn.microsoft.com/en-us/library/5471dc8s\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/5471dc8s(v=vs.80).aspx)

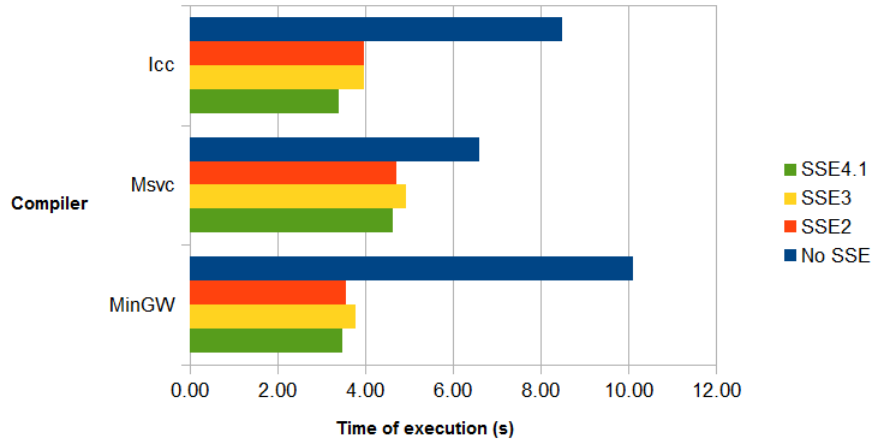


Figure 2.6: Speed measurement of the Gabor noise with fixed-sized local arrays

As we can see in figure 2.6, the algorithm with fixed-sized temporary arrays performs equally to the version with variable-sized arrays. We did not change they way the vectorization of the algorithm is done and the results reflect this fact.

2.2.4 Algorithm vectorization - No arrays, merging

Initial version

While vectorizing Improved Perlin noise (see chapter 1), we have found out that using temporary arrays makes the vectorized algorithm not very efficient. Even though our SIMD version of the Gabor noise fares better in this regard, we will try to remove the need for arrays completely.

We start the same way as before, calculating 4 Morton order values, 4 Poisson distributions and sampling 4 impulse parameters for the 4 cells (all vectorized). At this point, instead of storing the parameters for the impulses that influence the sample point into an array, we evaluate the 4 Gabor kernels defined by these parameters right away and afterwards sum the noise contributions of the influencing impulses (see figure 2.7).

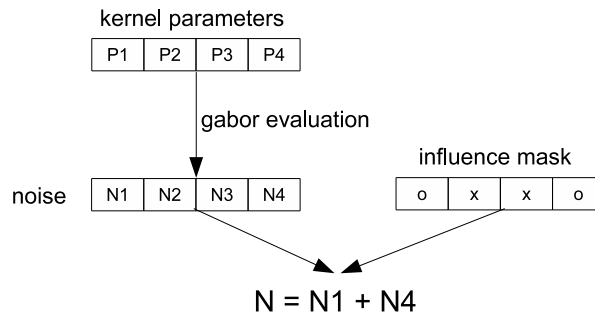


Figure 2.7: Gabor kernels noise contribution sum without arrays

Let us see how it performs:

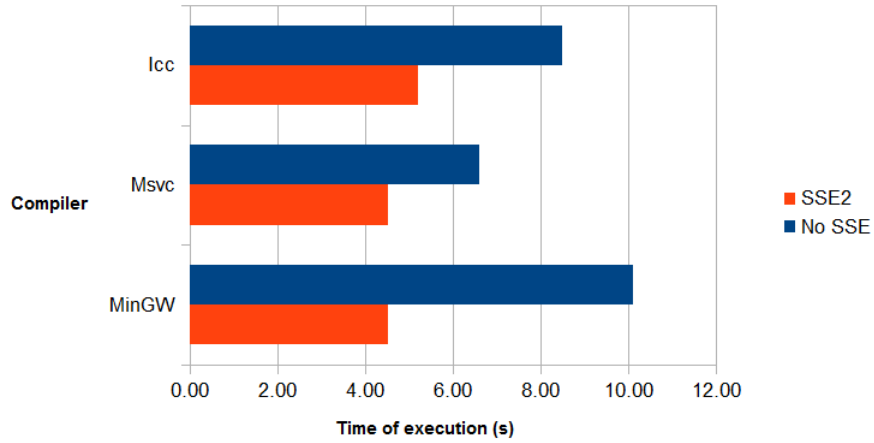


Figure 2.8: Speed measurement of the Gabor noise with no temporary arrays

As we can see in figure 2.8, the overall performance of this version is worse than the one which uses temporary arrays. However, there are ways to improve the algorithm.

Gabor kernel parameters accumulation

We know, from the analysis of the base algorithm (see table 2.1), that evaluating Gabor kernels is the single most time consuming part of the application. We also know that we are evaluating Gabor kernels also for all the impulses, not just the ones that influence the sample point (in contrast to the base version and temporary-array version). Finding the ratio of how many impulses influence the sample point will give us an idea of how much time we are wasting evaluating unneeded kernels.

We have modified the algorithm a bit to provide us with the count of how many Gabor kernels were evaluated and how many of these results were actually used when calculating 4 cells at once. In our measurement application, which calculates sum of 1 million noise values, 168 million Gabor kernels were evaluated and only 51 million of these were influencing the sample point - that is ratio of 3.27:1. To improve the performance, we need to lower the ratio get it to 1:1 as close as possible.

We will try to achieve that by accumulating the kernel parameters. Instead of evaluating the Gabor kernels right away, we will merge the vectors containing the parameters and weights with previous set of these parameters. We know which impulses influence the sample point before the Gabor kernels are evaluated and this information drives the merging. If the merge cannot be done cleanly (meaning that both the current and previous impulse of one particular cell influence the sample point), we evaluate Gabor kernels for impulses that could be merged and rest will be kept for the next iteration (see figure 2.9).

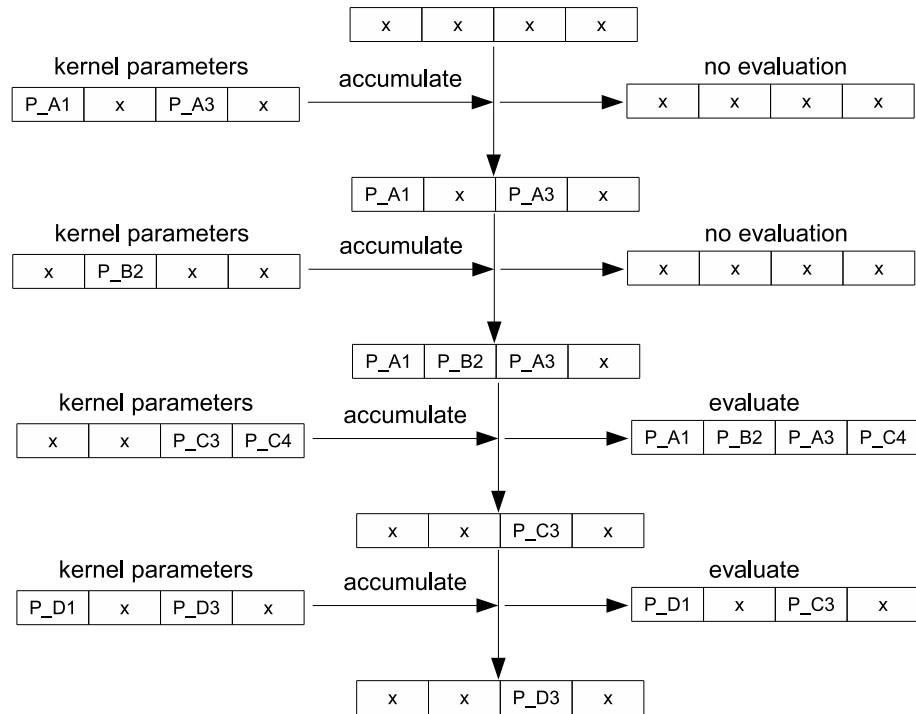


Figure 2.9: Merging of the Gabor kernel parameters

Here are the results with the improvements:

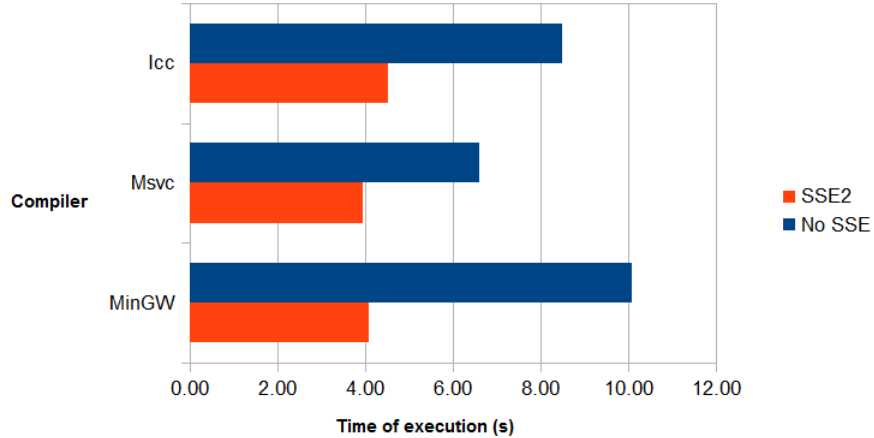


Figure 2.10: Speed measurement of the Gabor noise with merging

The figure 2.10 shows we were able to gain considerable improvements, although the Intel Icc and MinGW compiled versions still trail behind the version with temporary arrays. The Msvc version, however performs better. With these improvements, we were able to achieve ratio of all/influencing impulses at 2.22:1.

Gabor kernel parameters shifting and accumulation

The ratio achieved in the previous section shows that there is still room for improvements. Our next steps will be guided by the fact that impulses of a cell have higher probability of influencing the sample point the closer this point is

to the cell's borders. This also means that such cells have higher probability of generating two influencing impulses in a row, forcing a kernel evaluation. To try to prevent this situation, before we do the kernel parameters merging, we shift the elements in vectors containing the previous set of parameters to the right by one and then proceed with merging. This operation will guarantee that if a cell generates two influencing impulses in a row, this fact alone will not force kernel evaluation (see figure 2.11).

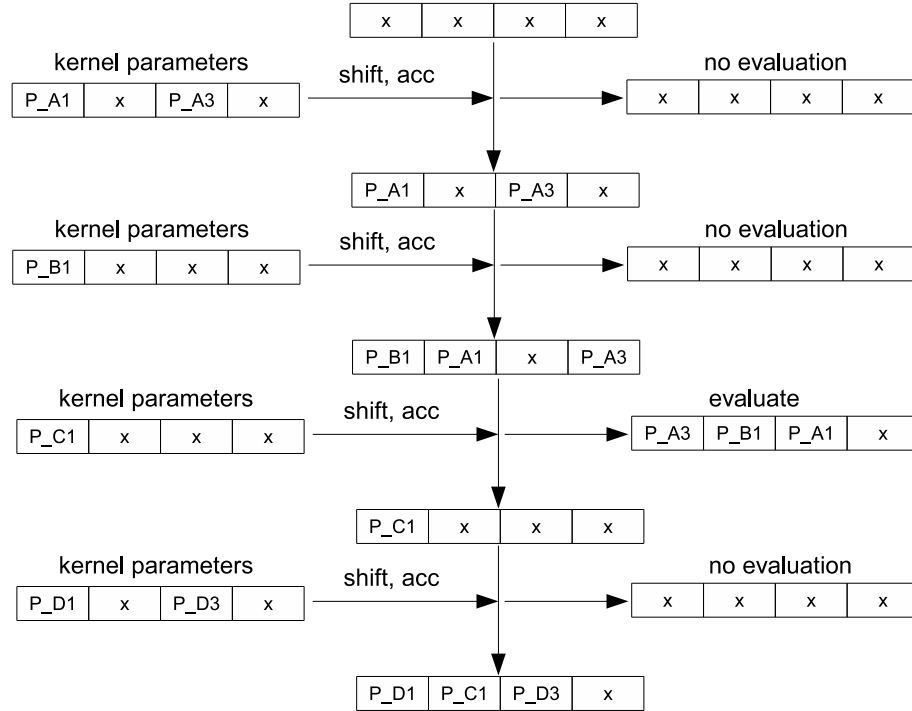


Figure 2.11: Shifting and merging of the Gabor kernel parameters

Here are the results with the shifting improvements:

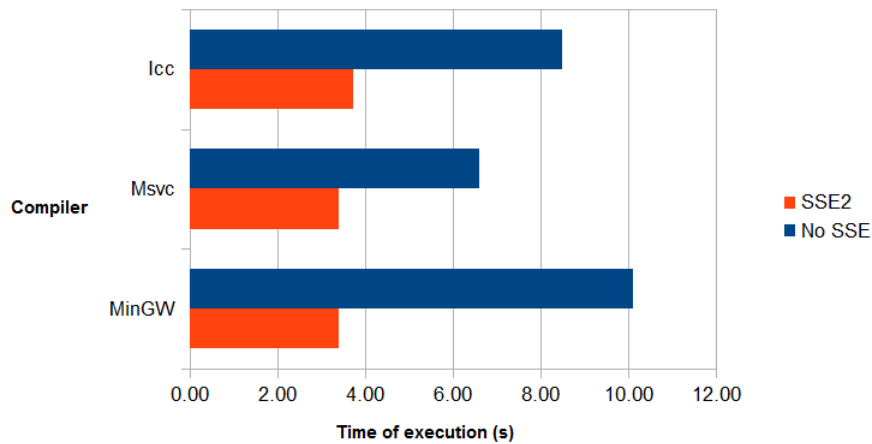


Figure 2.12: Speed measurement of the Gabor noise with shifting and merging

The results in figure 2.12 shows that we were able to gain another substantial

speed improvements thanks to lowering the ratio of total evaluated kernels to influencing kernels to 1.32:1.

Cell evaluation ordering

Our final attempt at lowering the ratio will not focus on the algorithm itself, but on the order in which cells are processed. For the 2D Gabor noise, 9 cells are evaluated - 8 cells in 2 groups of 4 plus 1 single cell evaluation. We the way we shift and merge the kernel parameters, the ordering will be guided by the probability of a cell generating an influencing impulse. Figure 2.13 shows schematic representation of such probabilities for all the 9 cells (lighter blue represents lower probability).

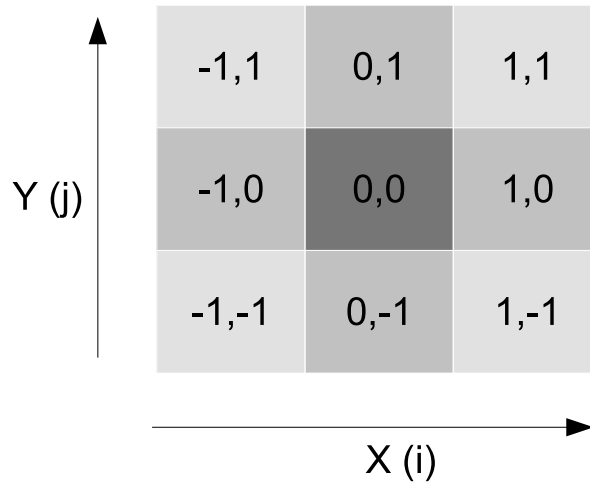


Figure 2.13: Influencing impulse generation probability map

Even further, depending on the position of the sample point in one of the quadrants of the central cell, the probabilities will increase for the cells in that corner and lower for the cells in the opposite corner. With this in mind, we group the cells following these rules:

1. The first 4 cells are the central cell and the 3 cells neighbouring with the quadrant of the sample point. Kernel evaluations will be frequent, but since all the cells have high probability of generating influencing impulse, most of these evaluations will be useful.
2. The second 4 cells are made up of the rest of the cells excluding the one that is opposite the sample point's quadrant. Kernel evaluations will be less frequent, but due to sparsity of the influencing kernels, the shift and merge algorithm should achieve high ratio.
3. The cell opposite the sample point's quadrant has the lowest probability of generating an influencing impulse and thus has potentially the highest ratio of total impulses/influencing impulses. It is thus evaluated using the single cell evaluation algorithm.

Let us see how the cell ordering effects the algorithm's performance:

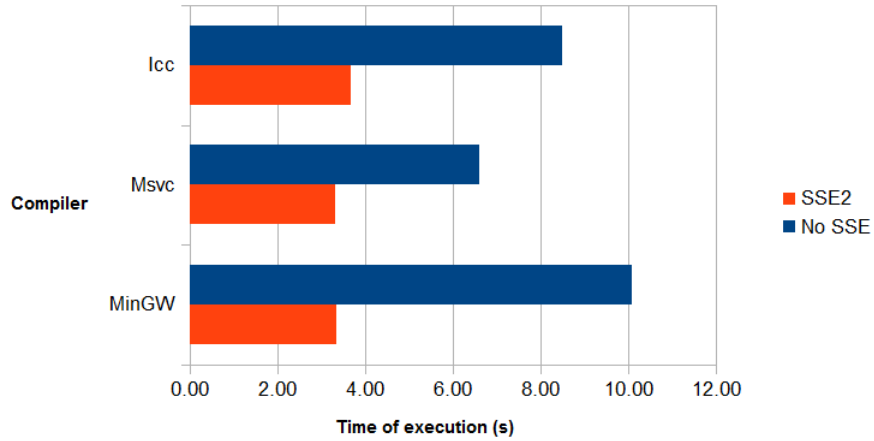


Figure 2.14: Speed measurement of the Gabor noise with guided cell ordering

With the guided cell ordering, we were able to achieve a very slight performance increase and a ratio of 1.3:1.

Comparing the two approaches

Let us now compare the two approaches for solving our variable-length array issue - fixed-sized arrays approach (FA) and shifting/merging approach (SM). All levels of SSE support in the library will be measured.

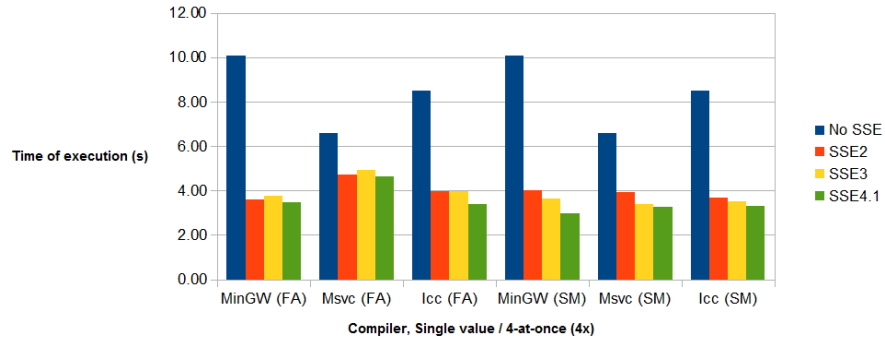


Figure 2.15: Gabor noise, comparing two vectorization approaches

In the figure 2.15 we see that the version with merging of kernel parameters performs overall better than the version with temporary arrays. In terms of speedup, that is 240% (MinGW), 157% (Icc) and 103% (Msvc) in comparison to 189% (MinGW), 150% (Icc) and 42% (Msvc). Even though more complicated, due to the overall speed of the merging version, we chose this as the implementation of the vectorized 2D Gabor noise algorithm.

2.2.5 Algorithm vectorization - 4 values at once

Vectorizing the calculation of 4 Gabor noise values is partially based on previous work. Sampling 4 noise values means we need to evaluate 36 cells. We will do

that by employing modified version of the 4-cells-at-once algorithm we developed in the previous sections. However, each one of the 4 cells belongs to a different sampling point, therefore we cannot use the shifting optimization to reduce the ratio of usefully evaluated Gabor kernels. As a matter of fact, in the case of sampling 4 noise values at once, the ratio is guided by the spatial relation of the sampling points. If they are located in the same cell, all 4 points are in every iteration checked against the same impulse. Furthermore, if the sample points are close together, there is a high probability that if one of the points is influenced by the impulse, all the rest of the sample points will be too. The algorithm therefore employs none of the ratio-optimizing features developed for the single noise value evaluation (except that it does not use any temporary arrays). This makes the calculation of 4 noise values at once suitable for systematic sampling of continuous noise surfaces. We will test the algorithm by evaluating and summing a continuous patch of 2D noise of size $(-256, 256) \times (-256, 256)$ and sample point distance of 0.5. This represents 1 million samples.

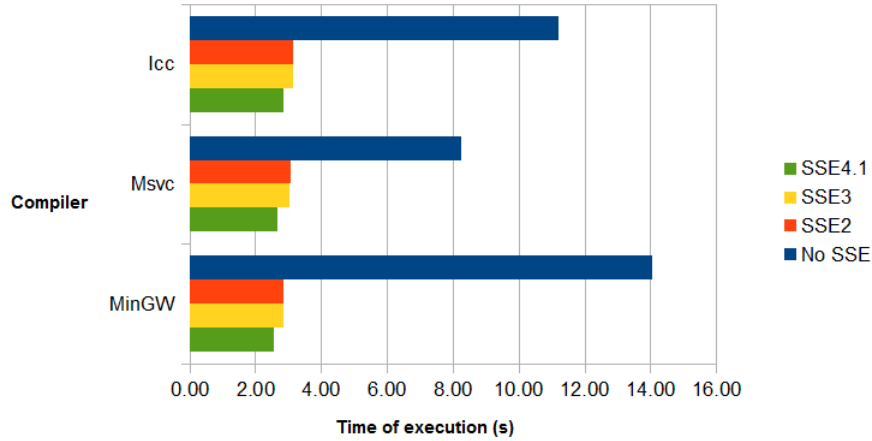


Figure 2.16: Speed measurement of the Gabor noise, 4 samples at once

The figure 2.16 shows the speed gains compared to non-vectorized version using the "Cephes" implementation of the trigonometric and exponential functions. Some of the speedups (mainly for the MinGW-compiled version) are beyond the theoretical speed improvement of using SSE, which suggests the MinGW compiler does not produce efficiently optimized non-vectorized code (in this case). Due to the sampling of a continuous patch of 2D space, we were able to achieve ratio of useful Gabor kernel evaluations of 1.1:1.

2.3 Gabor surface noise

The purpose of the Gabor surface noise is to provide a truly procedural texturing primitive without the need for any texturing-related parametrization of the rendered surface. Detailed description of the approach can be found in [4], section 5.1. This goal is achieved by a few modifications to the 2D Gabor noise. First of all, impulses are generated in a cubic cell grid, where the sample point's containing cell is surrounded by 26 neighbouring cells. Thus, impulse contributions from

27 cells need to be evaluated. Each generated impulse is projected onto a tangent plane of the rendered surface at the point of sampling. Only impulses inside a cylinder of radius equal to the Gabor kernel radius and height equal twice the radius. Distance of the impulse to the tangent plane defines the weight of the impulse.

2.3.1 Implementation and vectorization

Apart from the sample point's position in 3D space, we need to provide the Gabor surface noise implementation also with the normal vector of the tangent plane. Full implementation of the Gabor surface noise requires only relatively minor changes to the algorithm. Morton order of a 3D point needs to be calculated. 3D impulse position is sampled from the PRNGs. These points are then orthographically projected onto the tangent plane. From this projection, 2D sample point of a 2D Gabor kernel is obtained, along with the weight of the impulse. From this point on, the algorithm is the same as for the 2D Gabor noise.

Vectorizing the Gabor surface noise is thus straightforward as well, since the 3D Morton order and orthographic projection for 4 points at once can be vectorized with basic SSE instructions. We thus provide a modified function calculating contribution from for 4 cells at once. We call it 7 times, where the last call takes contribution only from 3 cells (totalling $4 * 6 + 3 = 27$ cells).

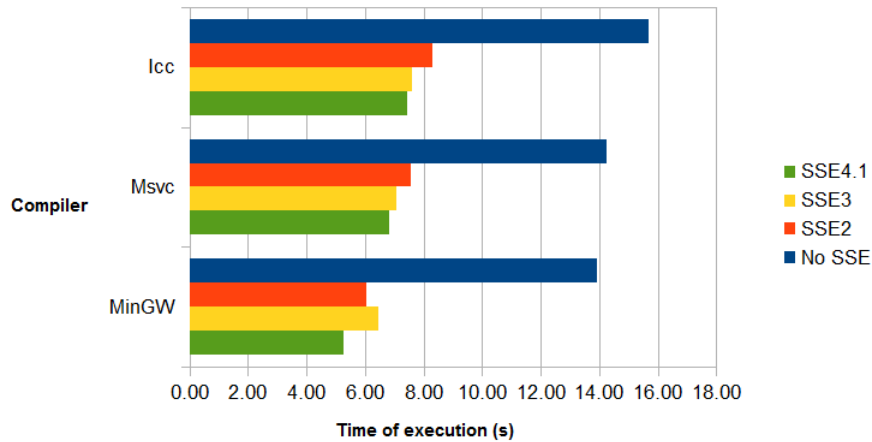


Figure 2.17: Speed measurement of the Gabor surface noise

The figure 2.17 shows speed measurements of the Gabor surface noise across all the SSE levels supported by our library. Relative speedups from the base, non-vectorized implementation are 164% for MinGW, 111% for Intel Icc and 109% for Microsoft Msvc. We achieved ratio of useful Gabor kernel evaluations of 1.46:1.

2.4 Integration to Mental-Ray, Maya

As with the Improved Perlin noise and Perlin-based noises, we will integrate the Gabor surface noise into the Maya 3D modelling and animation studio with built-in Mental-Ray ray-tracing rendering engine.

2.4.1 Exposing the noise

Due to its focus on providing truly procedural, setup-free approach to texturing materials, we have chosen to expose the Gabor surface noise in form of a Mental-Ray texture shader. Inspired by the section 3.3 about interactive noise design in [4], we have provided the ability to define up to 3 sets of parameters for the random pulse process and Gabor kernels, so called "widgets". The exposed texture shader provides the following parameters:

- NoiseType: type of noise encoded in a 2-digit integer (more details below)
- K: magnitude of the Gaussian envelope
- a: width of the Gaussian envelope
- F0: magnitude of the harmonic
- frequencySpread: spread of the harmonic's magnitude
- omega0: orientation of the frequency of the harmonic, in radians
- angularSpread: spread of the orientation of the frequency of the harmonic. 0.0 for anisotropic noise, 3.14 for isotropic noise.
- numberOfImpulses: number of impulses per cell
- include2ndWidget: enables the second widget
- K2, a2, F02, frequencySpread2, omega02, angularSpread2, numberOfImpulses2: set of parameters for the second widget
- include3rdWidget: enables the third widget
- K3, a3, F03, frequencySpread3, omega03, angularSpread3, numberOfImpulses3: set of parameters for the third widget

As mentioned in the list above, the noise type is encoded using a 2 digit integer $d_c d_s$, where:

- d_c - encodes code to use based on the compiler that generated it
 - 1 - Microsoft MSVC
 - 2 - Intel Icc
 - 3 - GNU MinGW
- d_s - encoded whether to use version with SSE enabled or disabled
 - 0 - SSE disabled
 - 1 - SSE enabled

2.4.2 Rendering performance

We will measure the rendering performance of the Gabor surface texture on a simple scene (see figure 2.18) containing a torus with applied texture that tries to match the textile cushion texture in figure 7 in [4]. The scene will be rendered in 720p HD resolution and all versions compiled by all 3 compilers will be tested, both non-vectorized and vectorized. Each such combination will be measured 3 times and the averages will be compared. Measurement of a torus without any texture applied will be provided for reference.

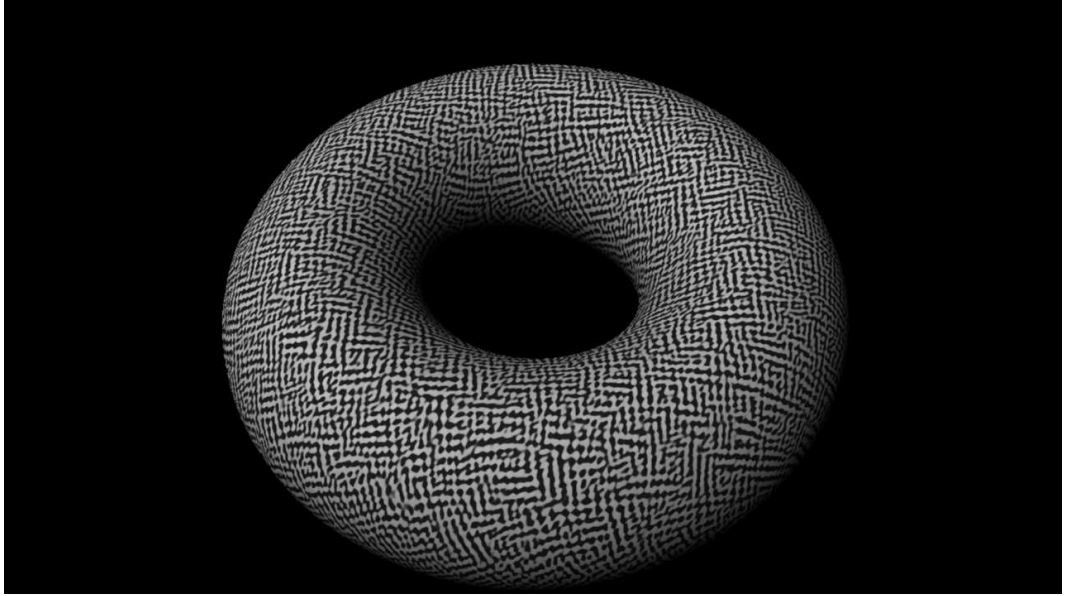


Figure 2.18: Torus with textile cushion texture

The measurement results are provided in figure 2.19.

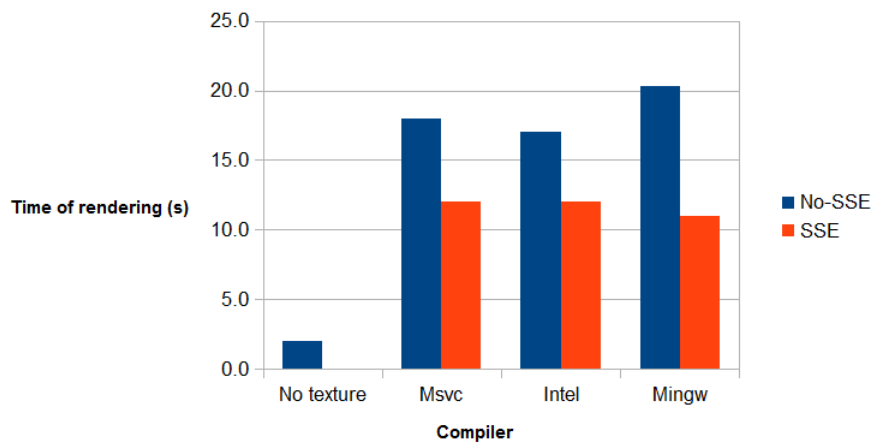


Figure 2.19: Gabor surface rendering time measurement

We also provide parameters used to setup the MentalRay Gabor surface noise texture (see table 2.2) for completeness:

Parameter	Value
Noise Type	...
K	0.72
a	0.07
F0	0.3
frequencySpread	0.03
omega0	1.5
angularSpread	1.57
numberOfImpulses	64
include2ndWidget	true
K2	1.02
a2	0.08
F02	0.17667
frequencySpread2	0.0
omega02	-0.72
angularSpread2	0.0
numberOfImpulses2	64
include3rdWidget	true
K3	1.0
a3	0.0887
F03	0.2
frequencySpread3	0.0
omega03	0.78
angularSpread3	0.0
numberOfImpulses3	64

Table 2.2: Gabor surface texture parameter values

3. Implementation details

3.1 Design and implementation goals

The initial idea to base our implementation on the libnoise library¹ was mainly due to its popularity in the world of C++ noise generating libraries and its clean, simple programming interface. In the core of the libnoise library stands an object called *Module*, which has three main characteristics:

1. it provides a *GetValue* method, which takes a 3D point as an input and outputs single value
2. it can be parametrized with zero or more parameters which are persistent across several *GetValue* calls
3. it can be linked to zero or more other *Modules*, called *SourceModules*. Output of the *GetValue* method of these *SourceModules* can be used during the execution the *Module's* *GetValue* method for whatever purpose necessary.

Thanks to the 3rd characteristic of a *Module*, complex graphs of interconnected *Modules* can be created to model various materials, phenomena, etc. An Improved Perlin noise, for example, is one specific instance of a *Module*.

However, the libnoise library and its API also has some deficiencies. The most notable of these are:

- inability to evaluate modules for other than 3D points (no 1D, 2D or 4D)
- hard-coded "double" floating point precision
- no support for SIMD features of CPUs

With this in mind, the design and implementation of our noise generating library was guided by these goals:

- provide public API that as closely resembles that of the libnoise library as possible. Ideally, existing programs using the libnoise library should be able to use our noise library with no or only very few code changes and a full program recompilation.
- allow evaluation of the *GetValue* method across multiple dimensions - 1D, 2D, 3D and 4D
- allow to select the precision of the floating point values and calculations ("float" and "double" precisions)
- support multiple levels of SIMD processor features.
- provide the library as a header-only C++ library. Reasons for this goal are mainly:

¹<http://libnoise.sourceforge.net>

- easier integration into projects - only thing required for a project to use the library will be to include proper header file
- it gives greater flexibility to the application developers when integrating our library to their project
- delegates the decision of what level of SSE support to use to the library user

3.2 Implementation

In order to fulfill our design goals, the implementation of the library was divided into 3 logical layers, each with its own responsibilities:

1. Public API layer - the main responsibilities of this layer is to keep the resemblance to the original libnoise library public API and to provide framework for implementation of specific *Module* instances
2. Specific *Modules* implementation layer - the purpose of this layer is to provide *GetValue* method implementations of specific *Module* instances. It is also responsible for providing proper vectorization algorithms if SIMD support is enabled.
3. Vector Math layer - the purpose of this layer is to provide common vector Math API to the *Module* implementation layer that hides SSE implementation details - both across floating point precision levels and various SSE instruction sets

3.2.1 Public API layer

Since one of the main responsibilities of the Public API layer is to provide an API to the library user that as closely resembles that of the original libnoise library, some implementation details need to be taken over from (or influenced by) the libnoise library:

- the whole library is defined inside *noise2* namespace. Namespace *noise* is reserved to provide compatibility with the original libnoise API
- everything related to *Modules* is defined inside *noise2::module* namespace
- the base class that defines common *Module* characteristics is named also *Module*. All specific implementations inherit from this class
- class names of specific *Module* implementations are carried over from the libnoise library (e.g., class representing Improved Perlin noise is named *Perlin*)

Module base class

Let us take a look at the core *Module* class and show its primary declaration right away.

```
template< typename FloatType, unsigned int Dimension >
class Module;
```

The primary *Module* class declaration only establishes that the *Module* is a template class parametrized with a *FloatType* type (specifying the floating point precision) and an unsigned int *Dimension* (specifying dimension). It defines no interface at all. This responsibility is passed to partial specializations of the *Module* template class. Partial specialization defining a 3D *Module* class looks like this (only important bits listed):

```
template< typename FloatType >
class Module< FloatType, 3 >:
    public ModuleBase< Module< FloatType, 3 > >
{
    ...
public:

    virtual
    FloatType
    GetValue( FloatType x, FloatType y, FloatType z ) = 0;

    virtual
    void
    GetValue4( const FloatType* inputX,
               const FloatType* inputY,
               const FloatType* inputZ,
               FloatType* output )
    {
        for( uint8 i = 0; i < 4; ++i )
        {
            output[ i ] = GetValue( inputX[ i ],
                                    inputY[ i ],
                                    inputZ[ i ] );
        }
    }
    ...
};
```

As can be seen, the partially specialized 3D *Module* template class defines a pure virtual *GetValue* method that, with proper *FloatType* template parameter, looks exactly the same as the *GetValue* method of the *Module* class from the original libnoise library. Every specific *Module* implementation needs to implement it. The *GetValue4* method is our addition to the *Module* interface. It is defined as a virtual method with base implementation, so specific *Modules* can

opt to not implement any optimized logic for calculation of 4 values. Partial specializations for other dimensions look accordingly.

The *ModuleBase* class that the *Module* template class' partial specializations inherit from provides interface and implementation for linking to *SourceModules*. Since this part of the *Module* characteristics is common for *Modules* of all dimensions and floating point precisions, logic that defines it was moved to a separate class to reduce code duplication.

With the way the base *Module* template class and its specializations are defined, we have achieved several of our design goals at once:

1. provide the same/similar public interface as the one of the libnoise library. The only standing issue is the parametrization of the *Module* template class. This is solved by defining "typedefs" to fully specialized specific *Module* implementations inside the *noise* namespace.
2. provide a way for the *GetValue* method to be evaluable across multiple dimensions
3. provide a way to select floating point precision

Specific *Modules*

Specific *Modules* follow a somewhat similar pattern. We will look at it using the *Perlin* specific *Module*.

```
template< typename FloatType, unsigned int Dimension >
class Perlin;
```

We begin by declaring the a template parametrizable by the same values as the bas *Module* template class. However, instead of defining partial template specialization as before, we declare another template class, responsibility of which is to implement the *GetValue* method (and *GetValue4* method, if desired).

```
namespace perlin
{
    template< typename FloatType,
              unsigned int Dimension,
              unsigned int VectorSize >
    class PerlinImpl;
}
```

Based on the level of SSE instruction set selected we then define the template *Perlin* class to inherit from the *PerlinImpl* class with proper template parameters, like this:

```
#if defined( LIBNOISE2_AT_LEAST_SSE2 )
template< typename FloatType, uint8 Dimension >
class Perlin:
    public perlin::PerlinImpl< FloatType,
                              Dimension,
                              4 >
```

```

{
};
#else
template< typename FloatType, uint8 Dimension >
class Perlin:
    public perlin::PerlinImpl< FloatType,
                              Dimension,
                              1 >
{
};
#endif

```

This is where the line between the first layer (Public API) and the second layer (Specific *Module* implementation) gets a bit blurry. The third template parameter of the *PerlinImpl* template class specifies vectors of how many elements should be used for the vectorized algorithms (where 1 is scalar, non-vectorized algorithms). Knowing this at this point requires knowledge of the various algorithms are vectorized. This is thus not fully encapsulated in the inner parts of the library implementation. However, this allows more control when choosing specific implementations based on the level of SSE support and the floating point precision. We could, for example, develop different algorithm for double precision *Perlin* module, which would take advantage of the fact that double support in SSE is provided by 2-element vectors only. We could thus cleanly separate this implementation into some other partial (or full) specialization of the *PerlinImpl* class.

It is also necessary to mention that the implementations of *PerlinImpl* class are responsible for inheriting from the *Module* class and thus making the *Perlin* class part of the *Module* ecosystem.

The Public API layer provides one last set of classes for the specific *Module* implementations. In the context of the *Perlin* module, this class is called *PerlinBase* (inside *noise2::module::perlin* namespace). It provides API and implementation for setting all the exposed *Perlin* module parameters (2nd characteristic of a *Module* in section 3.1). All the implementations of the *PerlinImpl* class inherit from the *PerlinBase* class to get access to these parameters and to make its public interface part of their public interface.

GaborSurface

The *GaborSurface* noise implementation is handled differently. First of all, it is not defined as a specific implementation of a *Module* at all. It is thus separated in a different namespace than all the *Modules* - *noise2::proto*. It also does not provide the ability to specify dimension and does not provide *GetValue* method with the common interface as its purpose is very specific. Apart from these differences, the way it handles *Gabor*-specific parameters and selection of vectorized implementation follows the same pattern as the specific *Module* implementations.

Specifying levels of SSE support

The final responsibility of the Public API layer is to provide the users with a way to select levels of SSE instruction sets that should be used for the vectorized al-

gorithms and for the library implementation to be able to act upon this selection. This is achieved through the use of the C++ preprocessor. Defined in the file "include/noise2/Platform.hpp", checks are made if some predefined preprocessor macros are defined by the user. If they are, another set of macros are defined which are used inside the library to get information which SSE instruction sets should be used. Here is an example for SSE3 level:

```
#if defined( LIBNOISE2_USE_SSE3 )
#define LIBNOISE2_AT_LEAST_SSE1
#define LIBNOISE2_AT_LEAST_SSE2
#define LIBNOISE2_AT_LEAST_SSE3
#endif
```

As can be seen, if the user specifies the use of SSE3 level of instructions, the library is informed that it can use all instructions from the SSE, SSE2 and SSE3 instruction sets. Support for the rest of the SSE levels is done in the same way.

3.2.2 Specific *Module* implementation layer

We will now take a look at the way specific *Module* implementations are realized. As was described in section 3.2.1, the actual implementations of the *GetValue* methods (and if desired, *GetValue4* methods) are defined in partial or full specialization of the *Impl* template classes.

Improved Perlin, Billow and RidgedMulti noises

Since these three noises are very similar (they differ only in the way they modulate the noise value calculated for each octave), we will cover them all together and on the example of the *Perlin* specific *Module*.

The whole implementation is located inside *noise2::module::perlin* namespace. The non-vectorized version is defined in 4 partial specializations of the *PerlinImpl* class. We provide a listing of the 2D version as an example:

```
template< typename FloatType >
class PerlinImpl< FloatType, 2, 1 >:
    public Module< FloatType, 2 >,
    public PerlinBase< FloatType >
{
    ...
public:
    virtual
    FloatType
    GetValue( FloatType x, FloatType y ) { ... }
};
```

Lets look at the partial specialization parameters for a while. We declare that the 2D (2nd *PerlinImpl* template parameter), non-vectorized (3rd template parameter) *Perlin* module implementation is defined for any floating point

precision. We are able to do this because "float" and "double" types in C++ support the same operations. No special code needs to be written for either of them. This is true for all dimensions and also for the Billow and RidgedMulti noises. These thus follow the same pattern when defining their implementations.

Internally, all these noise types use the same Improved Perlin noise algorithm for calculation of every octave noise value. This is realized in two classes - *NoiseGen* and *NoiseGenImpl*. The relation between the two is the same as the relation between the *Perlin* and *PerlinImpl* classes.

```
namespace perlin
{
    template< typename FloatType, unsigned int Dimension >
    class NoiseGen;

    template< typename FloatType,
              unsigned int Dimension,
              unsigned int VectorSize >
    class NoiseGenImpl;

    ...

    #if defined( LIBNOISE2_AT_LEAST_SSE2 )
    template< typename FloatType, uint8 Dimension >
    class NoiseGen:
        public NoiseGenImpl< FloatType,
                             Dimension,
                             4 >
    {
    };
    #else
    template< typename FloatType, uint8 Dimension >
    class NoiseGen:
        public NoiseGenImpl< FloatType,
                             Dimension,
                             1 >
    {
    };
    #endif
}
```

The same can be said about the *NoiseGenImpl* template class and its partial or full specializations as about the *PerlinImpl* template class. The only thing necessary to mention is the reason we chose to remove the 3rd template argument from the *NoiseGen* class the introduce it again in the *NoiseGenImpl* classes. In the future (and in line with the original libnoise library), these Improved Perlin noise generating algorithms may be more publicly exposed in the library. The user of the library should not be forced to specify the vector element count, as this is an implementation detail that should be hidden.

The vectorized implementations of the *PerlinImpl* partial implementations (and also *NoiseGenImpl*) are too defined in 4 classes. Lets take a look at the 2D version:

```
template< typename FloatType >
class PerlinImpl< FloatType, 2, 4 >:
    public Module< FloatType, 2 >,
    public PerlinBase< FloatType >
{
    ...
public:
    virtual
    FloatType
    GetValue( FloatType x, FloatType y )
    { ... }

    virtual
    void
    GetValue4( const FloatType* inputX,
               const FloatType* inputY,
               FloatType* output )
    { ... }
};
```

It differs from the non-vectorized version only in partially specialized 3rd template argument and the fact that it also provides its own implementation of *GetValue4* method. Again, we declare that we provide the same implementation no matter the floating point precision. However, the SSE vectors for "float" floating point precision and "double" floating point precision differ in the number of elements - 4 and 2, respectively. This fact is however hidden in inside the Vector Math layer, which provides a common interface for 4-element vector operations regardless of the floating point precision. This allows us to write the vectorization code only once and support both precisions. However, as can be seen in the chapter 4, this approach does not provide very good results in terms of speed-up. The *NoiseGen* and *NoiseGenImpl* classes for the vectorized version follow the same logic.

Gabor and GaborSurface noises

The Gabor and GaborSurface noises follow the same route as the Perlin-based noise when providing their implementation, which is located in the namespace *noise2::module::gabor* for the Gabor noise and *noise2::proto::gaborSurface* namespace for the GaborSurface noise. Both *GaborImpl* and *GaborSurfaceImpl* template classes are defined the same way as their Perlin counterparts with the exception of the *GaborSurfaceImpl* class not providing the *Dimension* template parameter (in accordance to the *GaborSurface* module). For the *GaborImpl* template class, partial specializations are provided only for the 2D version, as the library does not provide implementation for other dimensions. Common logic for

the pseudo-random number generator is provided in classes *Prng* (for the non-vectorized version) and *PrngVector* (for the vectorized version). GaborSurface-specific logic for Gabor kernel projection is provided in classes *Projection* and *ProjectionVector* located in the *noise2::proto::gaborSurface* namespace for the non-vectorized and vectorized version, respectively.

3.2.3 Vector Math layer

The main purpose of the Vector Math layer is to provide common interface for vector operations across both "float" and "double" floating point precisions. Also, it is responsible to hide the SSE specific implementation details and provide increasingly efficient implementations of certain operations with increasing level of SSE support. The Math API is accessible through *Math* class, located inside *noise2::math* namespace, which is defined as follows:

```
template< typename ValueType >
class Math: public LIBNOISE2_SUPPORT_MATH_IMPL< ValueType >
{
};
```

The *LIBNOISE2_SUPPORT_MATH_IMPL* macro defines the actual class the provides the implementation. For example, with SSE3 level of support, the macro is defined like this:

```
#if defined( LIBNOISE2_AT_LEAST_SSE3 )
#include <noise2/math/Math_SSE3.hpp>
#undef LIBNOISE2_SUPPORT_MATH_IMPL
#define LIBNOISE2_SUPPORT_MATH_IMPL      Math_SSE3
#endif
```

For each level of SSE that the library supports, a *Math_** class exists that provides vector implementations with features from that and all previous SSE features. This is done by inheriting a *Math_** class of previous level:

```
template<>
class Math_SSE3< float >: public Math_SSE2< float >
{
    ...
public:
    using Math_SSE2::reduce;

    static inline
    Vector4F
    reduce( const Vector4F& l, const Vector4F& r )
    {
        return _mm_hadd_ps( l, r );
    }
    ...
}
```

The code snippet above shows, apart from how *Math_** classes are inherited, how methods providing some vector operation are overridden to provide implementation that takes advantage of features from that particular SSE level. It also shows what type represents the vectors - in this case, *Vector4F* represents vector of 4 floating point elements. This type is defined for both *Mathjfloatj* and *Mathjdoublej*:

```
template<>
class Math_SSE1< float >: public MathBase< float >
{
public:
    typedef __m128    Vector4F;
    ...
}

template<>
class Math_SSE2< double >: public MathSSE1< double >
{
public:
    typedef struct
    {
        __m128d lo;
        __m128d hi;
    } Vector4F;
    ...
}
```

This allows the specific *Module* implementations work with just one type of floating point vector through *MathjFloatTypej::Vector4F* and automatically provide support for both "float" and "double" floating point precisions without writing any specialized code. The library provides two other vector types - *Vector4I*, storing 4 32-bit integers, and *Vector4I64*, storing 4 64-bit integers. They are defined as follows:

```
template< typename ValueType >
class Math_SSE2_Integer: public Math_SSE1< ValueType >
{
public:
    typedef __m128i    Vector4I;
    typedef struct
    {
        __m128i lo;
        __m128i hi;
    } Vector4I64;
    ...
}
```

4. Measurements

4.1 Improved Perlin and Perlin-based noises

In this section we provide speed measurements for all the Improved Perlin and its derived noises for 1D, 2D, 3D and 4D, both when calculating single noise value at a time and 4 noise values at a time will be measured as well. Also both "float" and "double" floating point precisions will be measured. The measurement application is calculating sum of 16 million noise values. The application will be compiled with 3 different compilers - Microsoft MSVC, Intel Icc and GNU MinGW. Measurements will be done 7 times for each version, the slowest and fastest times will be discarded and the average of the remaining 5 measurements will be presented. The compilation for individual compilers and SSE levels will be as follows:

- GNU MinGW G++ cross-compiler for 64-bit, version 4.7.2
 - No SSE: -O3 -DLIBNOISE2_USE_CEPHES
 - SSE2: -O3 -DLIBNOISE2_USE_SSE2
 - SSE3: -O3 -msse3 -DLIBNOISE2_USE_SSE3
 - SSE4.1: -O3 -msse4.1 -DLIBNOISE2_USE_SSE4.1
- Microsoft MSVC 2012 cl compiler for 64-bit, version 15.00.21022.08
 - No SSE: /Ox /Ob2 /Oi /Ot /DLIBNOISE2_USE_CEPHES
 - SSE2: /Ox /Ob2 /Oi /Ot /arch:SSE /DLIBNOISE2_USE_SSE2
 - SSE3: /Ox /Ob2 /Oi /Ot /arch:SSE /DLIBNOISE2_USE_SSE3
 - SSE4.1: /Ox /Ob2 /Oi /Ot /arch:SSE /DLIBNOISE2_USE_SSE4.1
- Intel C++ Composer XE 2013 icc compiler for 64-bit, version 13.1.1.171
 - No SSE: /O3 /Ob2 /Oi /Ot /DLIBNOISE2_USE_CEPHES
 - SSE2: /O3 /Ob2 /Oi /Ot /arch:SSE /DLIBNOISE2_USE_SSE2
 - SSE3: /O3 /Ob2 /Oi /Ot /arch:SSE /DLIBNOISE2_USE_SSE3
 - SSE4.1: /O3 /Ob2 /Oi /Ot /arch:SSE /DLIBNOISE2_USE_SSE4.1

In the measurement charts, first 3 column groups represent calculation of a single noise value at a time for each compiler (one compiler per column group), last 3 column groups represent calculation of 4 noise values at once.

4.1.1 Improved Perlin noise, "float" precision

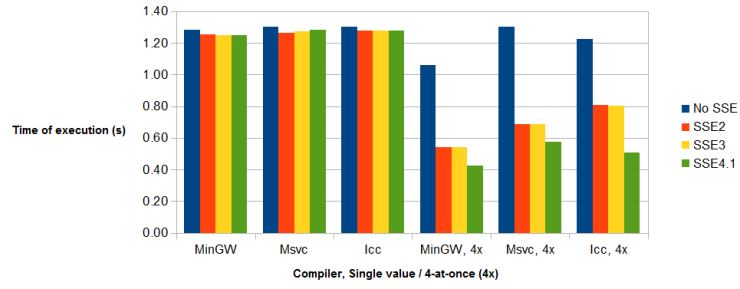


Figure 4.1: 1D Improved Perlin noise, "float" precision

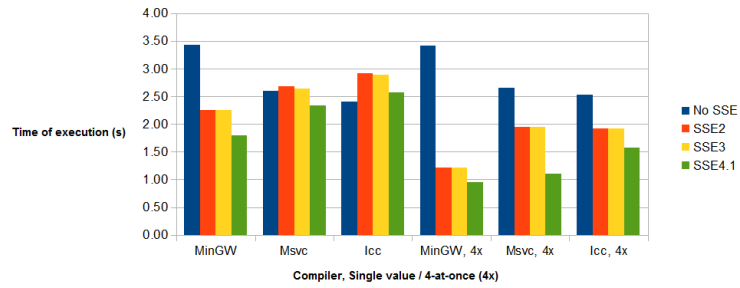


Figure 4.2: 2D Improved Perlin noise, "float" precision

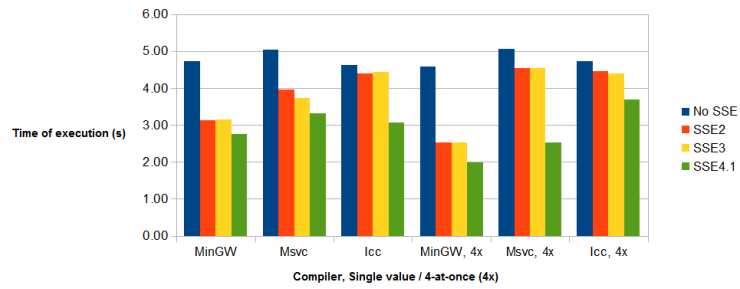


Figure 4.3: 3D Improved Perlin noise, "float" precision

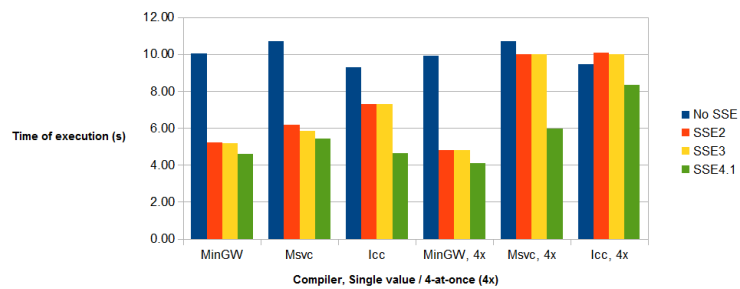


Figure 4.4: 4D Improved Perlin noise, "float" precision

4.1.2 Improved Perlin noise, "double" precision

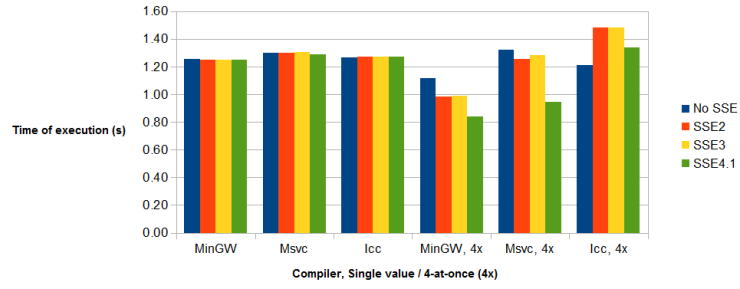


Figure 4.5: 1D Improved Perlin noise, "double" precision

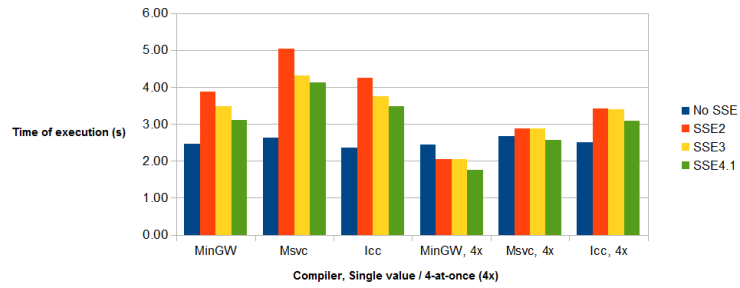


Figure 4.6: 2D Improved Perlin noise, "double" precision

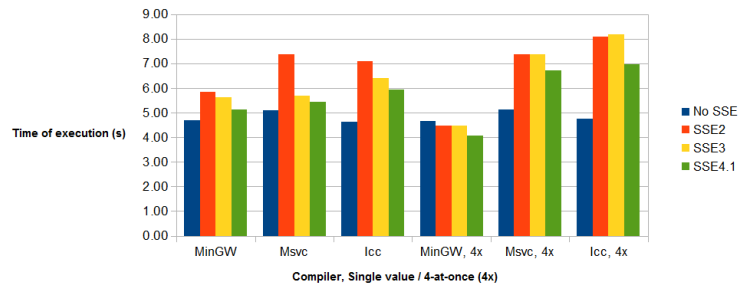


Figure 4.7: 3D Improved Perlin noise, "double" precision

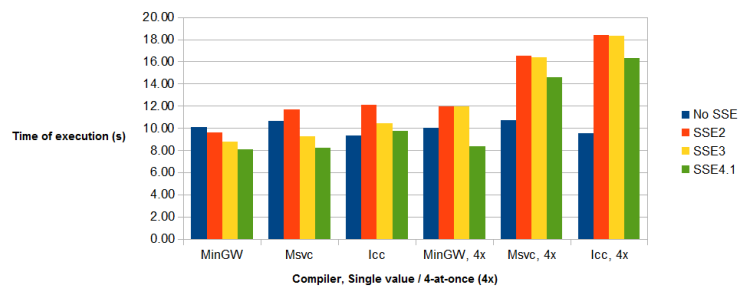


Figure 4.8: 4D Improved Perlin noise, "double" precision

4.1.3 Billow noise, "float" precision

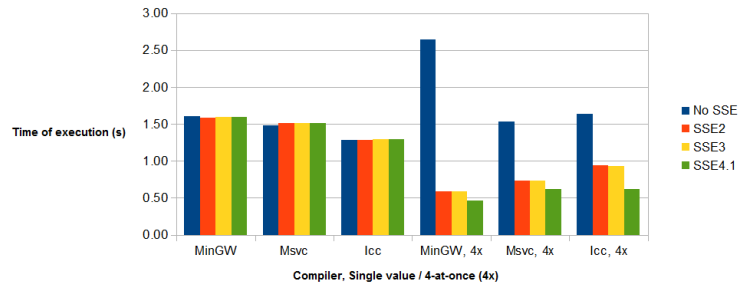


Figure 4.9: 1D Billow noise, "float" precision

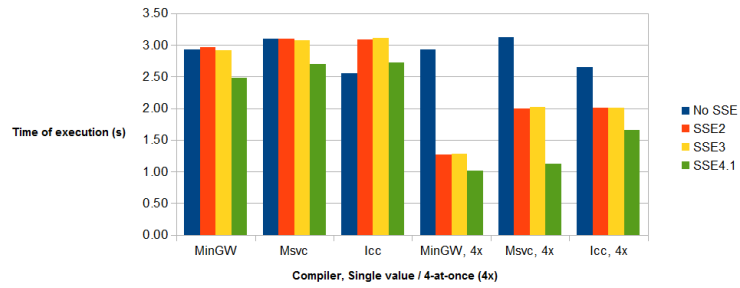


Figure 4.10: 2D Billow noise, "float" precision

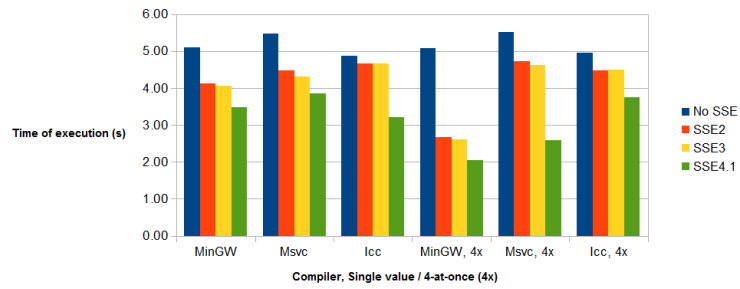


Figure 4.11: 3D Billow noise, "float" precision

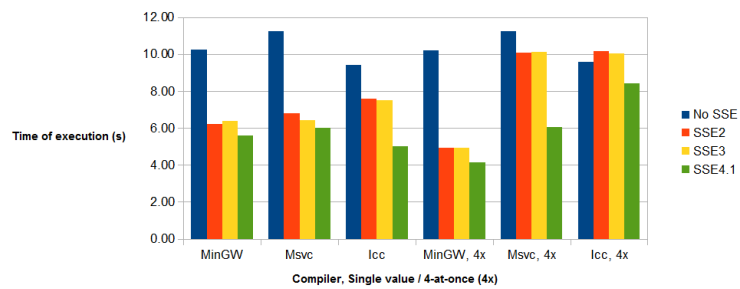


Figure 4.12: 4D Billow noise, "float" precision

4.1.4 Billow noise, "double" precision

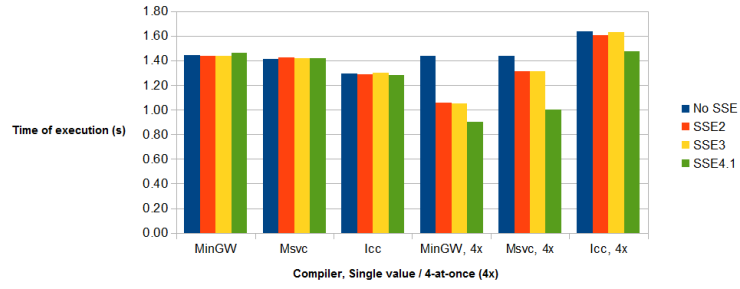


Figure 4.13: 1D Billow noise, "double" precision

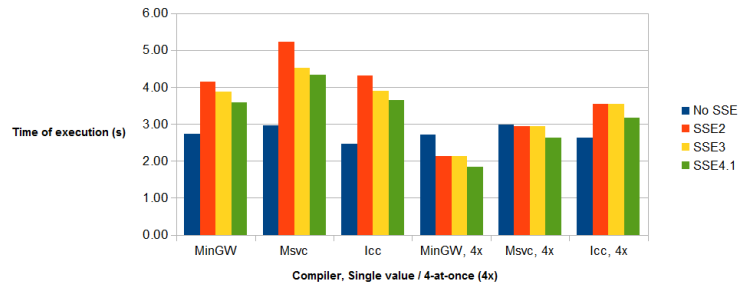


Figure 4.14: 2D Billow noise, "double" precision

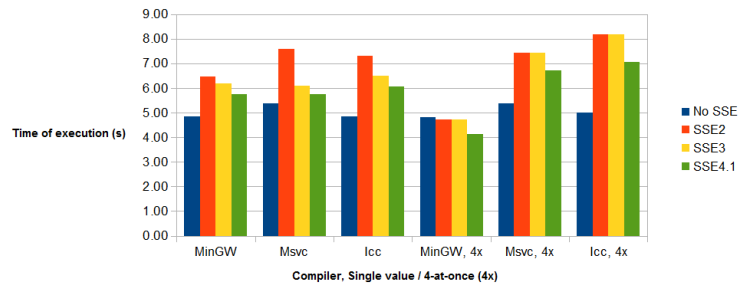


Figure 4.15: 3D Billow noise, "double" precision

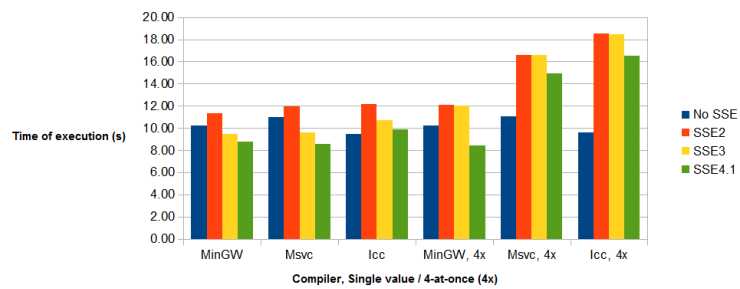


Figure 4.16: 4D Billow noise, "double" precision

4.1.5 RidgedMulti noise, "float" precision

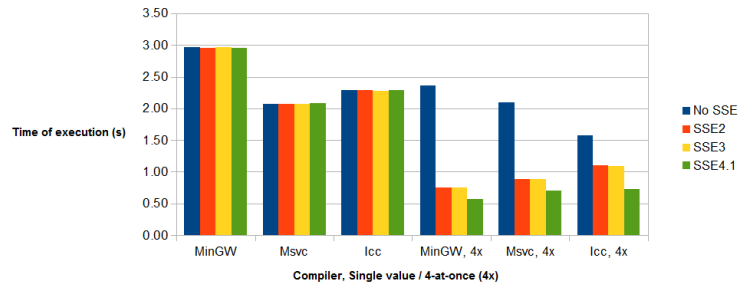


Figure 4.17: 1D RidgedMulti noise, "float" precision

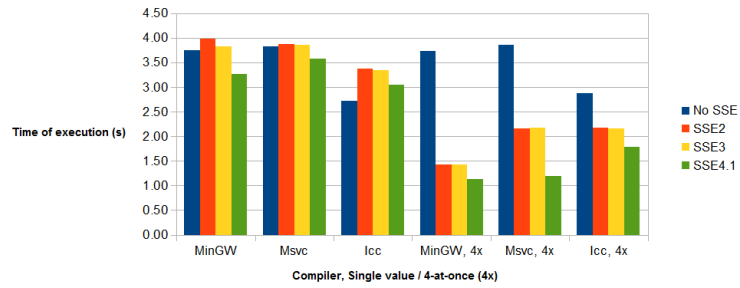


Figure 4.18: 2D RidgedMulti noise, "float" precision

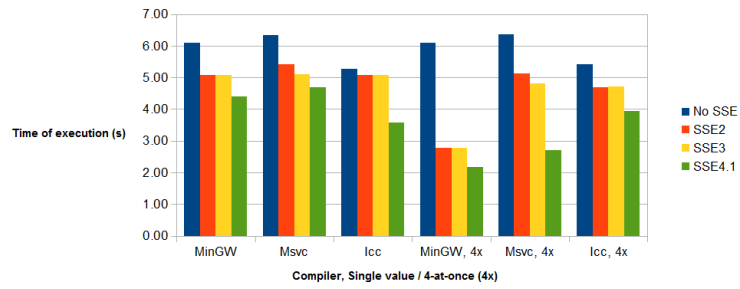


Figure 4.19: 3D RidgedMulti noise, "float" precision

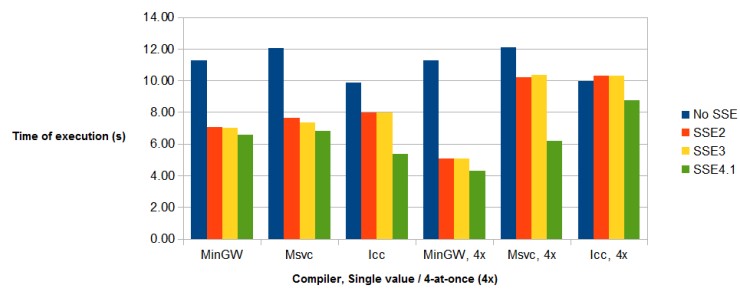


Figure 4.20: 4D RidgedMulti noise, "float" precision

4.1.6 RidgedMulti noise, "double" precision

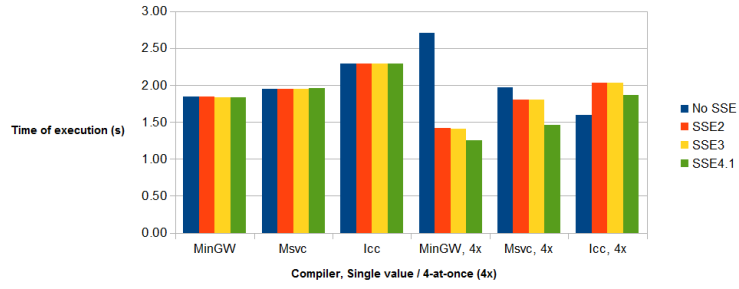


Figure 4.21: 1D RidgedMulti noise, "double" precision

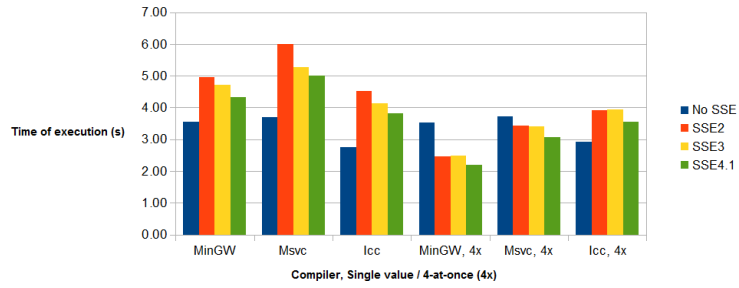


Figure 4.22: 2D RidgedMulti noise, "double" precision

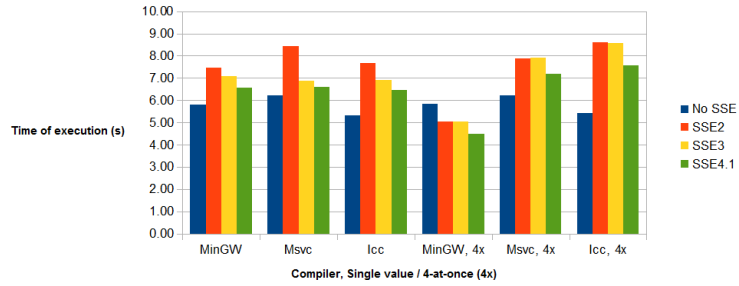


Figure 4.23: 3D RidgedMulti noise, "double" precision

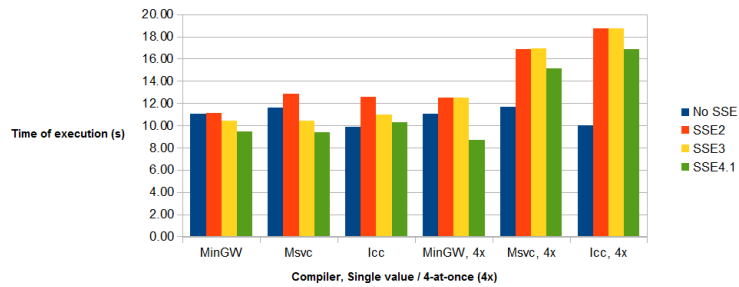


Figure 4.24: 4D RidgedMulti noise, "double" precision

4.2 Gabor noise and Gabor surface noise

In this section we provide speed measurements for our Gabor noise and Gabor surface noise implementations. In case of the Gabor noise both calculation of single noise value at a time and 4 noise values at a time will be measured as well. Also, both "float" and "double" floating point precisions will be measured. The measurement application is calculating sum of 1 million noise values over a patch of continuous surface. The application will be compiled with 3 different compilers - Microsoft MSVC, Intel Icc and GNU MinGW. Measurements will be done 7 times for each version, the slowest and fastest times will be discarded and the average of the remaining 5 measurements will be presented. The compilations parameters for the compilers and SSE levels are the same as listed in section 4.1.

4.2.1 Gabor noise

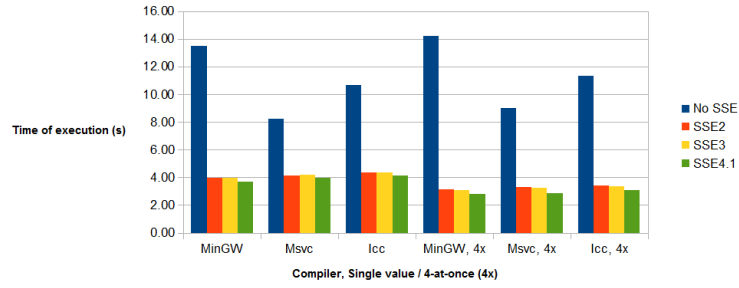


Figure 4.25: 2D Gabor noise, "float" precision

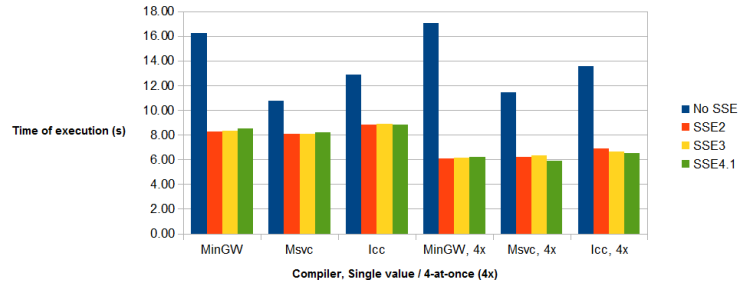


Figure 4.26: 2D Gabor noise, "double" precision

4.2.2 Gabor surface noise

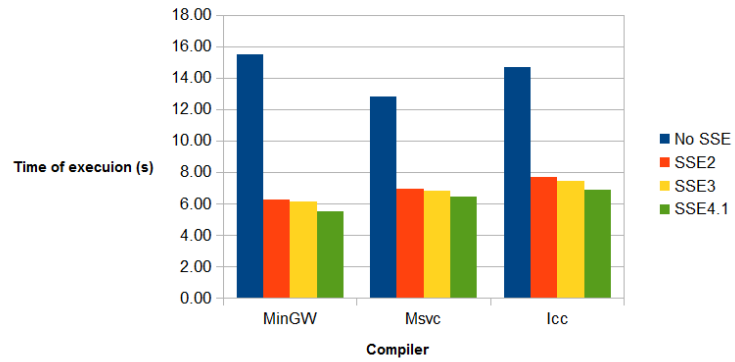


Figure 4.27: Gabor surface noise, "float" precision

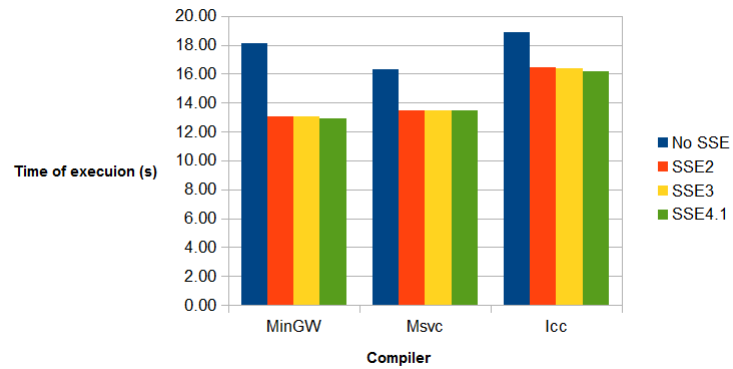


Figure 4.28: Gabor surface noise, "double" precision

Conclusion

Procedurally generated noise textures are an important field in the world of computer graphics. Due to their high computational requirements, achieving efficient use of hardware resources by developing optimized algorithms with the use of the features modern hardware is very beneficial. During the course of this work, we have vectorized two types of noise - the common and popular Improved Perlin noise (and its derivatives Billow and RidgedMulti noise) and a new Gabor noise. This process was, at its core, similar to both noises. We began by gaining a deeper understanding of the base, non-vectorized implementation by analyzing the source code, identifying parts with well defined interfaces and by hotspot analysis of the run-time execution of the algorithm. We then proceeded to the vectorization, following a bottom-to-top approach. The code transformation was done in several iterations, beginning with vectorization of the parts that represent the largest execution hotspots and gradually increasing the total amount of vectorized code. In case of the Gabor noise, two versions of the optimized algorithm were developed and the one with overall better performance was chosen. The final versions of the optimized noise implementations were integrated into the Mental-Ray ray-tracing rendering engine and their performance compared to the built-in MentalRay shaders (if available) and to the base, non-vectorized algorithms.

We chose to base our optimized implementation on the SSE instructions sets, which are commonly available in almost all currently used lines of processors. This means that the theoretical limit to the achievable speed-up is 300% (4-times faster) for single-precision floating points and 100% (2-times faster) for the double-precision floating points. In practice, however, the achievable speed-up greatly depends on the amount of vectorized code, effectiveness of the implementation and the ability of the compiler to generate properly optimized code. Our single-precision implementation of the Improved Perlin and Perlin-based noises was able to achieve speed-ups of up to 150+%. Our Gabor noise implementation relative speed-ups vary greatly across the compilers (100% - 250%), but this is mainly due to various performance of the base, non-vectorized algorithms, against which we calculate the effectiveness of the vectorized code. In absolute numbers, the compilers perform comparatively.

The double-precision implementation, however, shows very different results. We have chosen to hide the details of the SSE double-precision functionality behind a common vector math interface shared with the single-precision implementation. This enabled us to write the vectorized algorithm only once for both single- and double-precision math. In case of the Gabor noise, this approach delivers overall good results, in line with what could be expected with double-precision SSE. However, in case of the Improved Perlin noise, the results are not satisfactory and in many cases, the vectorized versions perform worse than the base, non-vectorized versions.

Nevertheless, in general, our vectorized algorithms perform very well and show that carefully handwritten optimized code can provide great benefits in the efficient use of the available hardware resources, which was the primary goal of this work. We also show that these benefits can also be taken advantage of when integrated into a rendering system (MentalRay, in our case) and that depending

on the scene can provide real improvements in the rendering time. Last, but not least, our implementation is provided in form of a public library¹, which can be easily integrated into various rendering systems and projects.

The work done in this thesis also offers opportunities for future improvements. The latest line of modern processors offer a new set of vectorized instructions, AVX², which can theoretically provide twice as much speed-up as the SSE instruction sets. The implementation of the double-precision algorithm can be modified for more direct use of the available instructions with all their peculiarities. Many other noise types with various properties exist that would also benefit from optimized implementations.

¹<https://github.com/dalibor-frivaldsky/libnoise2>

²AVX - Advanced Vector eXtensions

Bibliography

- [1] GABOR, D. *Theory of communication. Part 1: The analysis of information*, In: *Electrical Engineers - Part III: Radio and Communication Engineering, Journal of the Institution of*, volume 93. IEEE, 1946, pp. 429-441. DOI: 10.1049/ji-3-2.1946.0074.
- [2] INTEL® *Introducing the 4th Generation Intel® Core™ Processor (code-named Haswell)*. 2013. [cit. 2013-07-18]. Available at: <http://software.intel.com/sites/default/files/introduction-to-intel-4th-generation-core-processor.pdf>
- [3] INTEL® *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 1: Basic Architecture*. June 2013. [cit. 2013-07-18]. Available at: <http://download.intel.com/products/processor/manual/253665.pdf>
- [4] LAGAE, A., LEFEBVRE, S., DRETTAKIS, G., DUTRÉ, P. *Procedural Noise using Sparse Gabor Convolution*, In: *SIGGRAPH '09 ACM SIGGRAPH 2009 papers*. New York, NY, USA: ACM, 2009, pp. 54:1-54:10. ISBN 978-1-60558-726-4. DOI: 10.1145/1576246.1531360.
- [5] LEWIS, J. P. *Algorithms for solid noise synthesis*, In: *SIGGRAPH '89 Proceedings of the 16th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1989, pp. 263-270. ISBN 0-89791-312-4. DOI: 10.1145/74333.74360
- [6] LINARUNA. *SSE intrinsics optimizations in popular compilers*. [Www.linaruna.com](http://www.linaruna.com) [online]. [cit. 2013-07-18]. Available at: <http://www.linaruna.com/sse-intrinsics-optimizations-in-popular-compilers/> Also available at: <http://web.archive.org/web/20130501041751/http://www.linaruna.com/sse-intrinsics-optimizations-in-popular-compilers/>
- [7] MAKAROV, V. N. *The integrated register allocator for GCC*, In: *Proceedings of the GCC Developers' Summit*. Ottawa, Ontario, Canada: 2007, pp. 77-90
- [8] MORTON, G. M. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*, IBM, 1966. Available at: <http://books.google.cz/books?id=9FFdHAAACAAJ>
- [9] PERLIN, K. *ACM SIGGRAPH 84 conference*, course in "Advanced Image Synthesis." Minneapolis: 1984.
- [10] PERLIN, K. *An Image Synthesizer*, In: *SIGGRAPH '85 Proceedings of the 12th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1985, pp. 287-296. ISBN 0-89791-166-0. DOI: 10.1145/325334.325247.
- [11] PERLIN, K. and HOFFERT, E. *Hypertexture*, In: *SIGGRAPH '89 Proceedings of the 16th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1989, pp. 253-262 ISBN 0-89791-312-4. DOI: 10.1145/74333.74359.

- [12] PERLIN, K. *Improving noise*, In: *SIGGRAPH '02 Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 2002, pp. 681-682 ISBN 1-58113-521-1. DOI: 10.1145/566570.566636

List of Figures

1.1	Perlin sample point surrounded by integer cubic lattice vertices	4
1.2	Execution time of the base 3D Improved Perlin noise algorithm	9
1.3	Two <i>haddps</i> instructions to get a single dot product	11
1.4	Efficient use of <i>haddps</i> instruction	12
1.5	Speed measurement of the Improved Perlin noise, 1st iteration	13
1.6	Breadth-first calculation of gradient dot products	15
1.7	Depth-first calculation of gradient dot products	16
1.8	Speed measurement of the Improved Perlin noise, 2nd iteration	17
1.9	Gathering of gradient vectors from the gradient vector table	19
1.10	Moving gradient vector values to group them according to axes	20
1.11	Speed measurement of the Improved Perlin noise 4-at-once version	20
1.12	Speed measurement of 3D Billow noise	23
1.13	Speed measurement of 3D RidgedMulti noise	23
1.14	Speed measurement of 3D Billow noise, 4 values at once	24
1.15	Speed measurement of 3D RidgedMulti noise, 4 values at once	24
1.16	2D Perlin noise renderings	26
1.17	2D Billow noise renderings	27
1.18	2D Billow and Perlin with inflection renderings	28
1.19	2D RidgedMulti noise rendering	29
1.20	Volumetric fog scene	30
1.21	Perlin volumetric fog, rendering time measurement	31
1.22	Billow volumetric fog, rendering time measurement	31
1.23	RidgedMulti volumetric fog, rendering time measurement	31
2.1	Execution time of the base 2D Gabor noise algorithm	39
2.2	Exp/Cos "Cephes" vs standard library performance	40
2.3	Execution time of the base 2D Gabor noise with "Cephes"	40
2.4	Visualization of vectorized calculation of Poisson distribution	43
2.5	Speed measurement of the Gabor noise, 1st iteration	44
2.6	Speed measurement of the Gabor noise, fixed-sized arrays	45
2.7	Gabor kernels noise contribution sum without arrays	45
2.8	Speed measurement of the Gabor noise, no arrays	46
2.9	Merging of the Gabor kernel parameters	47
2.10	Speed measurement of the Gabor noise, merging	47
2.11	Shifting and merging of the Gabor kernel parameters	48
2.12	Speed measurement of the Gabor noise, shifting and merging	48
2.13	Influencing impulse generation probability map	49
2.14	Speed measurement of the Gabor noise, cell ordering	50
2.15	Gabor noise, comparing two vectorization approaches	50
2.16	Speed measurement of the Gabor noise, 4-at-once	51
2.17	Speed measurement of the Gabor surface noise	52
2.18	Torus with textile cushion texture	54
2.19	Gabor surface rendering time measurement	54
4.1	1D Improved Perlin noise, "float" precision	67
4.2	2D Improved Perlin noise, "float" precision	67

4.3	3D Improved Perlin noise, "float" precision	67
4.4	4D Improved Perlin noise, "float" precision	67
4.5	1D Improved Perlin noise, "double" precision	68
4.6	2D Improved Perlin noise, "double" precision	68
4.7	3D Improved Perlin noise, "double" precision	68
4.8	4D Improved Perlin noise, "double" precision	68
4.9	1D Billow noise, "float" precision	69
4.10	2D Billow noise, "float" precision	69
4.11	3D Billow noise, "float" precision	69
4.12	4D Billow noise, "float" precision	69
4.13	1D Billow noise, "double" precision	70
4.14	2D Billow noise, "double" precision	70
4.15	3D Billow noise, "double" precision	70
4.16	4D Billow noise, "double" precision	70
4.17	1D RidgedMulti noise, "float" precision	71
4.18	2D RidgedMulti noise, "float" precision	71
4.19	3D RidgedMulti noise, "float" precision	71
4.20	4D RidgedMulti noise, "float" precision	71
4.21	1D RidgedMulti noise, "double" precision	72
4.22	2D RidgedMulti noise, "double" precision	72
4.23	3D RidgedMulti noise, "double" precision	72
4.24	4D RidgedMulti noise, "double" precision	72
4.25	2D Gabor noise, "float" precision	73
4.26	2D Gabor noise, "double" precision	73
4.27	Gabor surface noise, "float" precision	74
4.28	Gabor surface noise, "double" precision	74

List of Tables

1.1	Hot-spot analysis of the base Improved Perlin noise algorithm . . .	10
1.2	Hot-spot analysis of the Improved Perlin noise, 1st iteration . . .	14
1.3	Our Perlin noise to Maya Perlin noise parameter mapping	27
1.4	Our Billow noise to Maya Perlin noise parameter mapping	28
1.5	RidgedMulti noise parameter values	29
1.6	Average rendering times of all volumetric noise fog scenes	32
2.1	Hot-spot analysis of the base Gabor noise algorithm	41
2.2	Gabor surface texture parameter values	55

List of Abbreviations

- FPS - Frames Per Second
- HD - High Definition
- SIMD - Single Instruction, Multiple Data
- SSE - Streaming SIMD Extensions
- SISD - Single Instruction, Single Data
- PRNG - Pseudo-Random Number Generator
- AVX - Advanced Vector eXtensions

Attachments

Attachment 1: DVD

Attached to this thesis is a DVD with all the materials, source codes and resources used to develop this work. The DVD contains:

- Text of the thesis in electronic form, along with the $\text{\LaTeX}$ source code, images, data and charts it was created from
- Source code of both non-vectorized and vectorized versions of the noises developed in this work, encapsulated in a header-only C++ library
- Source code and distribution package of the MentalRay shaders
- Maya scenes used to measure the performance of the MentalRay shaders
- Source code of the measurement applications with scripts used to obtain the measurement data
- MinGW 64-bit cross-compiler suite for the Windows OS used to create the MinGW compiled versions of the algorithms

Detailed structure of the attached DVD is described in the accompanying "Readme.txt" file, located in the root of the DVD

Attachment 2: DVD copy

Attached to this thesis is also a copy of the DVD from Attachment 1.