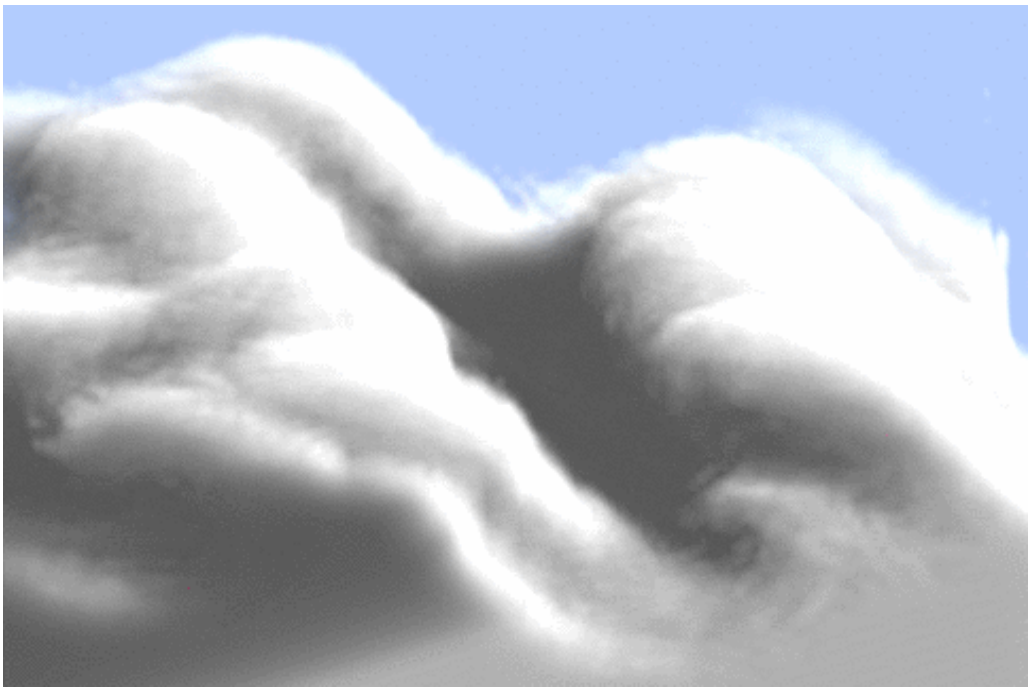
A journey in a procedural volume Optimization and filtering of Perlin noise

Reynald Arnerin

Master's thesis
M2R - Graphics Vision Robotics
june, 15 2009



Advisor : Fabrice Neyret, CNRS senior researcher
ARTIS - Laboratoire LJK
INRIA Rhône-Alpes - ZIRST
655 avenue de l'Europe, 38334 Saint Ismier Cedex

Acknowledgements

I would like to thank Fabrice Neyret for his numerous advises, is patience and his availability. I also want to thank the Artis team for their kindness and support, especially Cyril Crassin for his invaluable help on GPU matters as well as Pierre Benard and Pierre-Edouard Landes for sharing their office with me. I learned a lot during these 6 months in the team and I particularly enjoyed the experience.

Abstract

Perlin noise is the most widely used tool in procedural texture synthesis. It is a simple and fast method to enhance the quantity of detail or to render natural materials with no use of storage resources. However, this technique is very sensitive to aliasing artifacts, especially when composed with shape and color functions. Moreover, it is computationally intensive and can become slow, especially when generating procedural volumes of density in real time. This study aims at analyzing Perlin noise properties in order to control the apparition of artifacts and optimize the computational cost. We present a method for computing a maximum and minimum frequency threshold per noise component, we propose an idea to handle the case of non linear transforms of the noise, and show an optimization method for volume generation.

Contents

Chapter 1

Introduction

In the quest for realism, the amount and the credibility of details displayed in movie special effects or in video games is very important. It gives illusion that objects are constituted from actual materials and that they depict more complexity than noticeable. A poorly detailed object or phenomenon –particularly natural ones such as clouds, rocks, terrains, etc.– won't fool the spectator because it just strikes his eyes, used to the complexity of the real world. Apart from realism consideration, the sense of detail also give an artist a whole field of expression.
The quality of texture synthesis and volume rendering plays a major role in the ever growing visual quality of movies' special effects and 3D animation, but also in video games, in real-time simulator (*e.g.,* flight simulators, architecture design software) and scientific visualization (namely, medical imaging). However, graphics simulation of large environment including very complex shapes and natural phenomena such as avalanche, clouds, explosions or storms remain one of the most challenging goals in computer graphics. It requires a robust model to render volumes obeying to complex patterns and possibly animated. Achieving this high degree of detail in very large scenes while keeping spatio-temporal consistency in real-time is what motivates this project.
In this project we aim at analyzing noise functions in order make them more compliant with a real-time high quality exploration of procedural volumes. We will first introduce the general background of procedural textures and volumes. In a second time, we will address the main issues encountered with noise functions and we will survey the existing solutions. Afterwards, we will expose our analysis of the problem. Finally, a presentation of our solutions will be done.

Figure 1.1: Real-time rendering of a natural scene in Crysis (2008)

Figure 1.2: Storm special effect in The day after tomorrow, 2004

Chapter 2

Background

In order to properly introduce noise functions, a quick overview of the background is described in this chapter. The goal here is to justify the importance of procedural volumes generation.

2.1 Modelling of details

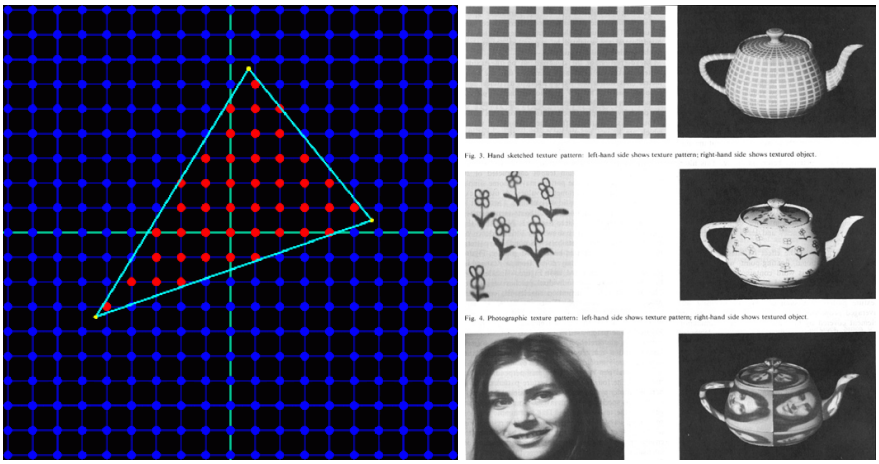
2.1.1 Texture mapping

Two levels of models are usually defined to provide a mesh with details : a shading model and a texturing model. The first describes the way the materials reflect the light whereas the second focus on the pattern and the details (*e.g.,* granularity, imperfections). Ever since the first computer generated texture by Catmull [Cat74] (Figure 2.1), texture synthesis became a major concern for the CG community.
In general, a texture is defined as an image which is mapped (*i.e.,* projected) on the surface of a 3D triangular mesh. It basically consists in attributing a color from the texture image to each pixel of the rasterized [1] triangle. Mapping any picture on a 3D surface was a revolution marked with the apparition of believable scenes (for the time) and lead to the emergence of many map-based techniques. These ones not only used pictures to color the surface but also to render other aspect like the apparent roughness with *Bump Mapping* [Bli78], transparency or the the shadow (*shadow maps* [Wil78]).

2.1.2 Higher dimension textures

Textures are not restricted to the 2D space though, they can be extended to 3D and even 4D (with dimension for time and/or space). Obviously, mapping 3D image onto a 3D surface can't be done as in 2D, and is somehow irrelevant for graphics. Nevertheless – and this is the point in using them – their different use overcome some issues resulted by strictly surfacic approaches . In 3D the data are a grid of voxels[2] valued with color and transparency ; it can be seen as a 3D generalization of a picture.

[1] *i.e.,* projected on screen then converted into pixels for display.
[2] *i.e.,* volume element, extension picture elements (pixels)



(a) rasterization (b) mapping

Figure 2.1: texture mapping, Catmull (1974)

Solid texture

A problem encountered with the two-dimensional texture, due to the mapping, is the distortion phenomenon (*e.g.,* tapestry on non flat surfaces,the poles of sphere). The dependency between the texture and the geometry makes it difficult to come up with a good surface parameterization as explained by Goldberg et al. [GZD08].
The purpose of solid texturing, introduced by both Peachey [Pea85] and Perlin [Per85], is to provide a full coherence of the texture over space by breaking this dependency. The idea is to parameterize the 3D texture directly with the coordinate of the point in object space, rather than defining intermediate 2D mapping coordinates. By analogy, we can consider mapping a texture like pasting a wallpaper whereas solid texturing is like carving the object into raw material (Figure 2.2). This technique is convenient to render objects actually carved from raw materials (*e.g.,* wood, marble) without spacial distortion.

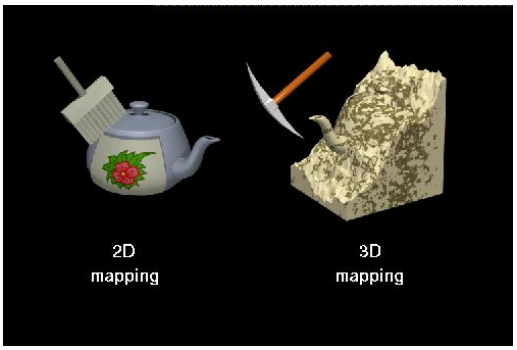


Figure 2.2: texture mapping VS *Solid texture* analogy, (Wolfe, DePaul University @ SIGGRAPH 97)

Volumetric textures

Another general issue, related to surface representation of 3D model, is the lack of accuracy due to the fact that the surface description (*e.g.,* meshes) alone cannot always approximate the complexity of the shape, especially for sparse natural models (*e.g.,* tree

leaves, grass, coral) because they look bushy or flurry. This problem is usually solved by using a volumetric representation of the object, which has the property to provide arbitrary complexity on half transparent voxels, suggesting the presence of very small details with no explicit storage. An alternative, called volumetric textures, was however introduced by Kajiya and Kay for fur rendering [KK89] and later extended by Neyret [Ney98]. Volumetric textures are a volume representation mapped in the neighborhood of the surface, designed to improve visual fidelity of the shape (Figure 2.3).

Figure 2.3: Volumetric texture of grass, Neyret (1996)

2.1.3 Volume representation

Aside from geometrical constraints, purely volumetric properties of the materials are heterogeneity of the density and the optical behavior (Figure 2.4). Typically, translucent shapes with complex light interactions like smoke, gases but also countless materials generally assumed to be solid (*e.g.,* vegetation or fur at distance). Finally, the model granting the greater control on the spacial complexity is the volume representation because it defines the optical properties of any point in space. The main advantage of the volume representation model over the surface one is the possibility to implement more complex physical models of lighting taking into account volumetric optical interaction (*e.g.,* multiple anisotropic volume scattering [Max94]).

Of course, the rendering algorithms for this representation are completely different than for meshes.

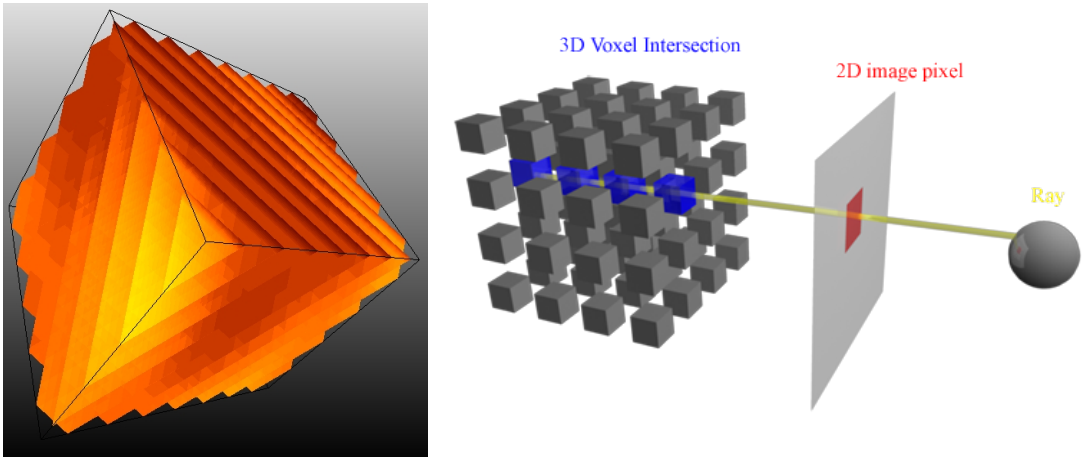
Figure 2.4: Volume representation of a cloud with complex light interactions, Bouthor (2008)

Volume Rendering

Several methods exist to render volumes, but overall, two main concepts stand out.

An early approach to render volumes was to draw a series of semi-transparent parallel slices such as in shear warp [LL94] (Figure 2.5(a)). These slices are planar polygons parallel to the point of view which are textured using solid texture mapping (paragraph 2.1.2). The results obtained with this rendering method are acceptable in the context of scientific visualization (*e.g.,* it is widely used to render medical scans), but offers poor visual quality in terms of accuracy. Firstly, the number of slices is limited (usually to 256) due to the cost of transparent geometric primitives to be blended together.

Another approach, known as *ray-marching* (Figure 2.5(b)), directly renders the volume by casting rays on the volume. A ray is first "shot" from the view point through each pixel of the image plane, then the light contribution of sample points is integrated along the ray. This is usually done in multiple passes like in [KW03] and the integration stops when the opacity reach a fixed threshold.



(a) Volume slicing (b) Ray-marching

Figure 2.5: Volume rendering methods

Volumes data structure

Rendering volumes requires many access to the voxel data, especially in the case of *ray marching* in which each voxel along the ray must be evaluated. Accessing to those data can quickly become costly if they are not structured properly, because of the large amount of informations. This is also a problem for graphic cards, where data must be fetched on the limited memory of the hardware, implying heavy memory transfers. In order to optimize the memory management for real-time *ray marching*, several data structures where introduced.

The "standard" data structure for *ray marching* on graphic cards is the regular grid [PBMH02]. It ensures a constant access time for each voxel of the grid. Moreover, this structure has good properties on memory locality (*i.e.,* two voxels close to each others are also close in memory), which reduces the amount of memory transfer stated earlier.

Another convenient data structure is the k-dimensional tree, a binary tree recursively partitioning the space in half spaces. Latest implementations of KD-trees for real-time *ray-marching*(*e.g.,* in [HSHH07]), fully take advantage of the hardware architecture to optimize data access. The interest of this kind of hierarchical data structure are more and more looked forward. Crassin et al. [CNLE09], for instance, explored the possibilities offered by N^3*-trees* (*i.e.,* hierarchical subdivision in N cubic subspaces ; generalization of the octree) to implement a real-time rendering of very large volumes thanks to a thorough data organization and optimized memory access on current hardware.

2.2 Procedural approach

Although increasing the resolution of textures and volumes indeed allows more elaborate scenes to be rendered, handling so much complex data is a real issue. Firstly, it makes the data storage prohibitive when resolution gets very high, particularly in the case of volumes reaching several gigabytes. Secondly, more details means more data to create : the complexity and the size of environments require a lot work from designers. In addition, even if a scene is expected to contain many similar objects and patterns (*e.g.,* the trees of a forest, the waves of the sea), human visual system is particularly good at spotting identical and regular samples. Letting the application take care of the hard work is then more and more required.

Figure 2.6: Trees modeled and textured procedurally, Dave Jerrard

2.2.1 Main idea

Overall, procedural techniques allow the generation of 3D meshes, textures or animations using a set of rules and functions. The main interest of these techniques is that the computation can be done on the fly just specifying relevant parameters (*e.g.,* the number of branches of a fractal-generated tree), which uses far less memory than explicitly storing

every details. Another benefit is the simplification in the design process since the hardest of the work – the amount of details and the global coherence – is taken care of by an algorithm. The use of procedural techniques for textures generation opened a brand new field of research and allowed the synthesis of many kind of textures like wood, bricks or water surface.
A distinction is usually made between implicit and explicit procedural textures. An implicit procedure is a function which, given any point of the space, computes its color. It fits perfectly with the renderer's philosophy because it delivers the necessary information whenever needed, independently of everything else than the evaluated pixel. An explicit procedure, however, must be seen as a texture builder. It computes the data in a fixed order and usually requires storing them in an image buffer for a later access. Implicit procedural textures are therefore better suited for a generation on the fly using the fragment shader of the GPU[3].

2.3 The Graphical Processing Unit

In order to render 3D scenes in interactive frame rate (*i.e.,* number of image computed per second), it is mandatory to maximize the speed of the rendering pipeline. GPU were designed to discharge the CPU from a certain number of procedures heavy to compute, taking advantage of the fact that the pipeline is particularly prone to parallelization (*e.g.,* operations on triangle, operation on pixels). Progressively, graphics processors became more and more powerful while handling more and more rendering operations (*e.g.,* transformations, illuminations). However, these operation were hard-coded in the graphic card, making it difficult for the developer to control this power. The programmability of the GPU later allowed developers to take advantage of the high parallelism of its architecture to execute specialized programs known as *shader programs*.

The evolution of the graphic pipeline (Figure 2.7) provides control at different step of the renderin on GPU. In particular, per-pixel operations can be programmed to arbitrarily attribute a color to a pixel. It permits, among other, the implementation of efficient real-time illumination algorithm computed for each pixel and advanced texture synthesis, including procedural textures.

2.4 The aliasing problem

Aliasing is the nemesis in computer graphics : it can ruin the most gorgeous picture with unexpectable patterns and artifacts. Aesthetics and fidelity being the main concerns, an uncontrolled deformation of the final picture is not acceptable especially when the accuracy of the details is important. Aliasing phenomenon can occur whenever the screen resolution is unable to reconstruct the original data (assuming a single sample per pixel) (Figure 2.8). In the case of high definition texture mapping, any fine details will cause aliasing when seen from afar. To be accurate, the aliasing problem occurs when the **Nyquist-Shannon sampling theorem** is not satisfied.

[3]Graphic Processing Unit as opposed to CPU, main processor of the computer. Graphic cards are embedded with a GPU

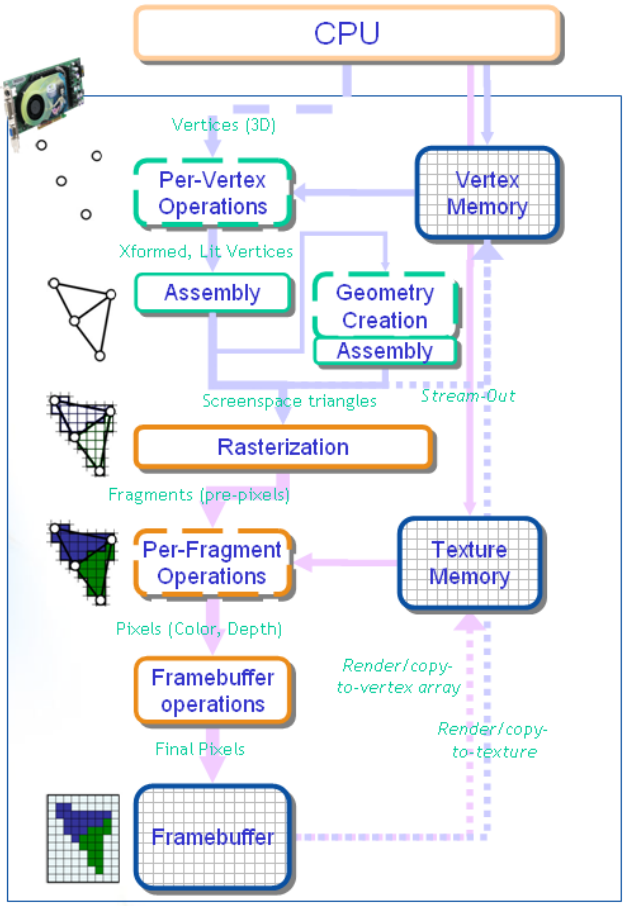


Figure 2.7: Graphics pipeline

Albeit anti-aliasing methods are a major subject of research in many domains related to signal processing, there is no miracle solution : the implementation must be chosen carefully in function of the situation, and considering the trade off between quality and computation time. The solutions, surveyed by Heckbert in [Hec89] and [Hec86] in the context of texture synthesis, consists in avoiding the frequencies above the *Nyquist frequency*[4]. This is traditionally done either by sampling at a higher frequency or by cutting high frequency with a filter.

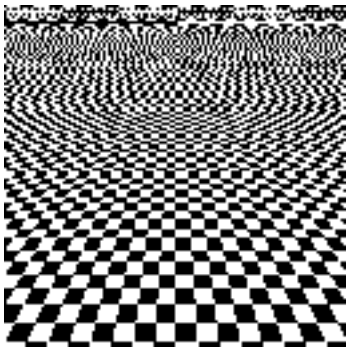


Figure 2.8: Aliasing artifacts on a texture

2.4.1 Prefiltering vs *Supersampling*

- The **prefiltering** of an image is done before the sampling. This implies that the input signal is completely known and, more importantly, that it can be processed.

[4]Reminder : The Nyquist frequency, defined as half the sampling frequency, is the maximum frequency for a signal to be fully reconstructed.

The prefiltering consists in decreasing the maximum frequency of the signal by applying a low pass filter :
With s the input signal, f the low pass filter and $*$ the convolution operator :

$$s' = s * f$$

with s' band limited with no frequency higher than the filter cutoff.
As a result, aliasing can be removed by applying a low pass filter cutting the frequencies above the Nyquist frequency. In practice however, convolution with a filter can be costly and working in a finite space may alter the quality of a theoretically ideal filter.

- ***Supersampling*** is a post filtering method which consists in taking several samples per pixel to perform a discrete integration of the value of the pixel. This is similar to output the image at a higher scale (*i.e.,* increase the *Nyquist frequency* and thus reduce aliasing), filter the reconstructed signal and then downscale the image to the wanted resolution. However, this method is very costly in terms of performance, since many more pixels must be evaluated. Moreover, *supersampling* signals which are not band-limited (*e.g.,* a sharp edge) won't reduce aliasing much. This is because high frequencies were never really filtered : integrating several samples over the pixel is equivalent to filter the reconstructed signal, but this reconstructed signal is not necessarily equal to the input signal.

Consequently, choosing the filtering method is very tricky since they cannot be used in every cases. Prefiltering is only possible when the input signal is known, whereas supersampling is inefficient on non band-limited signal.

2.4.2 MIP-mapping

Introduced by Williams [Wil83], MIP-mapping is an alternative filtering methods in which the texture is precomputed at several resolution scales. Usually, the pyramid contains multiple versions of the texture at resolution from 2^0 to 2^n, n being the maximum resolution wanted. The idea behind this, is to map the version of the texture relevant to the scale of the surface, *i.e.,* choose the prefiltered texture according to the sampling frequency (Figure 2.9). This is done, for each pixel, by computing the radius of its projection on the surface to deduce the local Nyquist period, then fetch the value from first non-aliased texture. However, in order to avoid the effect of pop-up as the distance between the camera and the sampled point varies, it is better to linearly interpolate the value between the blurry and the aliased texture (Figure).

2.4.3 Anisotropic filtering

Antialiasing methods may actually fail because of the projective geometry. Indeed, the Nyquist frequency is computed locally, according to the radius of the projection of the pixel – the footprint– on the surface. But the wrong assumption in this method, is to consider that the footprint is well defined by "a radius". The footprint is not necessarily isotropic : it shape is elliptical when the projection of the pixel is not orthogonal to the surface (almost always).
Heckbert ([Hec89]) proposed a set of elliptical filters and extended the MIP-Map pyramid with rectangular versions of the texture.

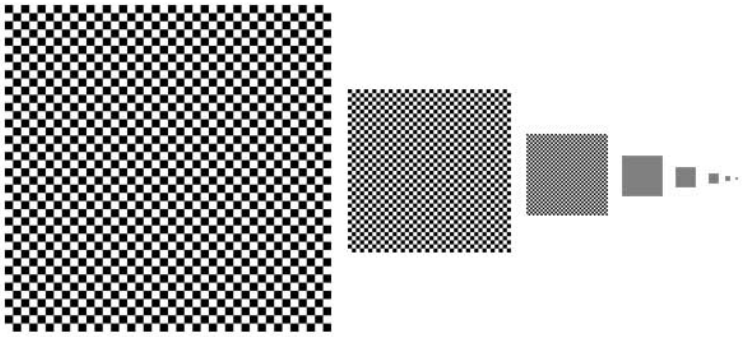


Figure 2.9: Illustration of the MIP map pyramid

Chapter 3

Context and previous works

3.1 Procedural noise

As seen in section 2.2, procedural functions are essential to generate complex, repetitive and coherent content in computer graphics. Yet, bringing irregularity to the patterns requires a stochastic approach in order to break the noticeable monotony. The introduction of noise primitives by Ken Perlin [Per85] brought the foundation of the most widely used class of procedural textures. Their purpose is to generate irregular values – randomness being a necessary condition to synthesize convincing natural materials – while keeping interesting properties for the designer. Rather than generating a *white noise*[1] which would cause aliasing, Peachey defined a set of properties for an ideal noise [EMP+02]:

- repeatability
- continuity
- band-limited spectrum with no frequency higher than 1
- no obvious periodicities, *i.e.,* no noticeable copies of the pattern
- stationarity and isotropy of the distribution

These properties assure the good controllability, smoothness and spacial consistency wanted. Spectral properties, in particular, are essential to generate textures with the needed amount of details while keeping the maximum frequency below the nyquist frequency.

3.1.1 Perlin Noise

The development of Perlin Noise, also called gradient noise, has allowed computer graphics artists to better represent the complexity of natural phenomena in visual effects for the motion picture industry. It is largely used by the CG community for its efficiency and implemented in the standard library of most of the modeling and rendering softwares (*e.g.,* Blender, Renderman, Maya, 3DSMax). Furthermore, the performances achieved by graphics hardware in the past few years now makes it possible to use noise functions to enrich detail in real time applications.

[1] white noise is a completely random distribution with infinitely high frequencies

Perlin noise is built as a sum of $\mathbb{R}^{1..4}$ wavelet functions centered on lattice points (*i.e.,* here, point with integer coordinates). Each wavelet is the product of a weight function with a linear function of pseudo-random[2] gradient (Figure 3.1). The process on $\mathbb{R}^3$ (the generalization is direct) is the following :

- let the symmetric cubic spline :

$$\begin{cases} spline(t) = 1 - 3|t|^2 + 2|t|^3 \\ spline(t) = 0 & \forall |t| > 0 \end{cases}$$

  The weight function with compact support of radius 1 is :

$$W(u,v,w) = spline(u) * spline(v) * spline(w)$$

- A linear function of gradient G is obtain with the dot product :

$$G \cdot [u,v,w]$$

  The value of G is important for the randomness of noise. Indeed, a pseudo-random gradient G_{ijk} is computed for each lattice point using a hash function on the coordinates i,j,k. This hash function uses a permutation table of 256 unique entries guaranteeing the non repeatability of the distribution over 256 values in each dimension. Meaning that the noise function is mathematically periodic of period 256.

- Now, the value of a wavelet centered on $[i,j,k]$ is given by

$$wavelet_{ijk}(x,y,z) = W(x-i, y-j, z-k)(G_{ijk} \cdot [x-i, y-j, z-k])$$

  Note that at [i,j,k] the wavelet is zero and its gradient is G_{ijk}. By construction, it has compact support of radius 1, it integrates to zero and is $\mathscr{C}^1$.

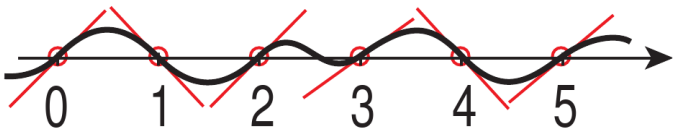
- Finally, the noise function is defined as the sum of all the wavelets :

$$noise(x,y,z) = \sum_{i,j,k \in \mathbb{Z}^3} wavelet_{ijk}(x,y,z)$$

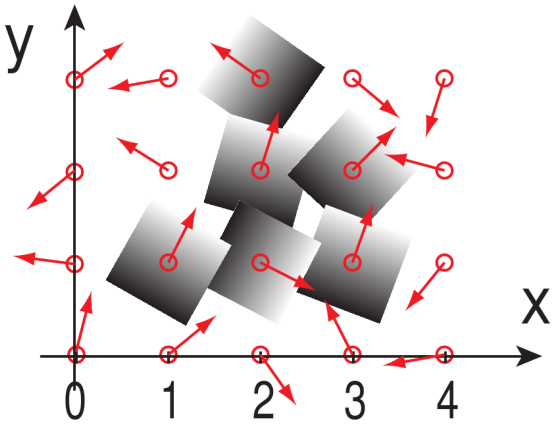
  When implemented, the result of the function is normalized to fit between -1 and 1, for the sake of intuitivity.

Using such wavelets guarantees a relatively null mean and an almost band-limited spectrum, and the compactness of the support allows efficient optimizations. Indeed, the relative distance between 2 lattice is greater than or equal to the "radius" of the wavelets : if we consider a single cell of the grid, it is immediate that only the wavelets centered on the corners of the cell might contribute to the value. The noise function is actually periodic, and thus will expose regularities on the pattern at large scale. However, the period is long enough not to be noticed in usual cases. We'll also see later that the lack of precise information on the spectrum is an issue.

[2]Note that the noise functions are not actually random but deterministically constructed for implementation reason and to guarantee the repeatability.



(a) fixed value and gradient on lattices



(b) orientation of the wavelets

Figure 3.1: Illustration of the Perlin noise construction at lattice points, (Stefan Gustavson, 2005)

3.1.2 Other noises

The success of the original Perlin noise in 1985 motivated different approaches for noise generation. Therefore, several alternative noise functions were created.

Improvement and alternatives

Some of these function are very similar to Perlin noise, only differing in the way the interpolation is done. Such noise functions based on interpolation of values at lattice points are called *lattice noise*. For instance, the *value noise* functions interpolate pseudorandom values generated at lattice points rather than gradient, thus avoiding risks of bias when generating gradients. Various interpolation function can be used depending on the statistical and spectral properties aimed at, and keeping in mind in what extent the interpolation function is can be optimized.

Different implementations of *gradient noise* also exist, with modifications tending to fix the gradient generator and the interpolation function. Ken Perlin himself brought some improvements to his noise in 2001 [Per01] and 2002 [Per02] ; the improvement process was latter clarified by Stefan Gustavson [Gus05] and renamed *simplex noise*. The simplex noise introduces 3 major improvements :

1. A new interpolation polynomial :
   The original perlin noise used the Hermite function $f(t) = 3t^2 - 2t^3$ for the interpolation to be $\mathscr{C}^1$. However, the derivative of noise functions is often used in texture synthesis (*e.g., bump mapping*, *displacement mapping*), and since the derivative be-

comes $\mathscr{C}^0$, discontinuities can be noticed at cell boundaries.
This is overcame by using a new polynomial with a zero second derivative at $t = 0$ and $t = 1$. The function chosen is a fifth degree polynomial with a similar shape : $f(t) = 6t^5 - 15t^4 + 10t^3$

2. A better gradient generator : Although the gradient generation was uniformly distributed over a sphere, the very disposition of the wavelets on lattice points introduced biases, hence anisotropy. The reason for that is the non-uniform contribution of the wavelets according to the position of the evaluated point on the grid cell. This means that a points evaluated along the axis of the grid are more likely to take high values than those at the center of the cell (because farther from each corners). As a result, one can guess the orientation of the grid from the noise pattern : this is anisotropy.
The solution to this problem is to better choose the gradients so that wavelets' maximum contribution are not directed toward grid axis. By the way, Perlin noticed that 12 different gradient directions suffice to provide enough "noise randomness" in the 3D case. The intuition is that since a cell is fully defined by its 8 corners' wavelets then the hash function generates enough unique cell with 12 different gradients.

3. A simplicial grid (Figure 3.2): When generalized to higher dimensions, the original Perlin noise's complexity became rapidly intractable ($\mathscr{O}(2^N)$). This was due the grid partitioning of the space implying 2^N corner by cell. The new approach use simplex grid to restrain the number the number of corner of the cells to N.

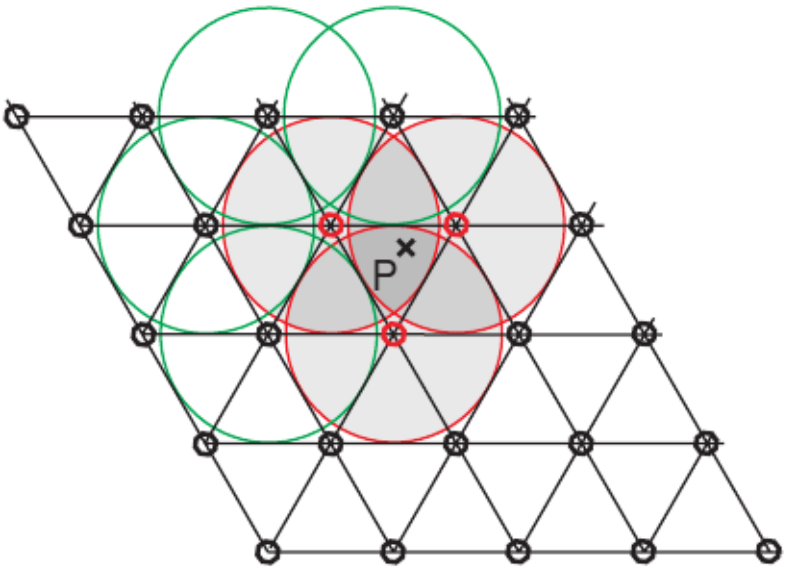


Figure 3.2: Fewer contribution of wavelets make the simplex grid more efficient (Gustavson, 2005)

Apart from simplex noise which is better and faster than the original noise, the various implementations of lattice noise functions are to be chosen in function of the properties required for a situation.

Different concepts

Other types of noise functions, not based on lattice noise made their apparitions :

Worley Noise [Wor96] is based on Voronoi diagrams. It randomly generates cells pattern like skin, scales or tiles.

Other implementations based on Fourier transform were developped, but these functions are explicit procedures.

3.1.3 Noise implementation on fragment shader

At first, fragment shader was used to access and interpolate a 3D precomputed texture of noise [Per04], mainly to overcome the computational cost of shader instructions. The results using this solution are honest, but it betrays the main concept of implicit procedural textures and more importantly, has gradient discontinuity and doesn't handles 4D textures. Despite the fact that a full implementation is still costly on modern GPU (processing time is highly precious), it tends to be affordable and finally opens the gates of real-time generation of procedural textures [Gre05].

3.2 Extended use of noise

Noise functions are designed to be used as basic primitives to add irregularities to the procedural techniques. The possibilities provided by these functions gave birth to many models.
In general, the construction follow this pipeline :

$$noise(x) \longrightarrow f(noise(x)) \longrightarrow \sum_{i=0}^{N-1} \frac{f(noise(b^i x))}{a^i} \longrightarrow pattern\left(\sum_{i=0}^{N-1} \frac{f(noise(b^i x))}{a^i}\right) \longrightarrow LUT\left(pattern\left(\sum_{i=0}^{N-1} \frac{f(noise(b^i x))}{a^i}\right)\right)$$

3.2.1 Fractal construction

Fractal noise was introduced by Perlin in the same time as his noise function. Actually, people talking about Perlin noise usually refer to the fractal summation of Perlin noise. The distinction was made, because every noise function can be fractally summed. The most important reason why the noise functions are designed to be band-limited is to be summed at multiple scales (Figure 3.3).
The Fractal noise is defined as a sum of scaled base functions called **octaves** :

$$Fnoise_N(\boldsymbol{x}) = \sum_{i=0}^{N-1} b_i(x)$$

With $b_i(x) = \dfrac{noise(b^i x)}{a^i}$ (usually with a=b=2) the i-th octave of the fractal noise

This method can be seen as the procedural version of the MIP-mapping : each term of the sum adds a new scale of detail.

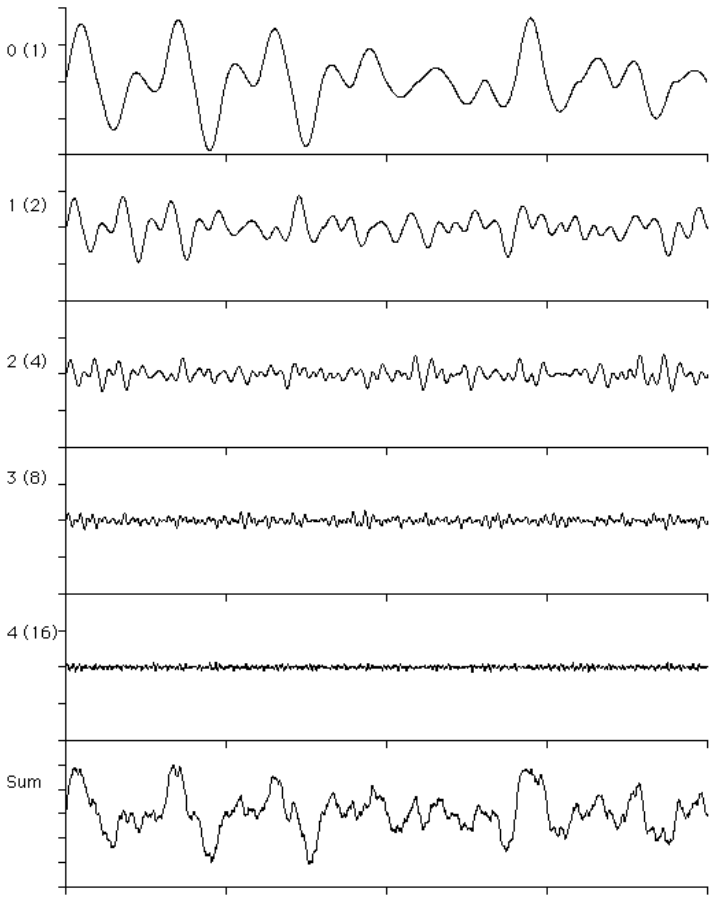


Figure 3.3: Base function are summed at different octaves to provide more details

3.2.2 *Compound noise*

In practice, noise function are rarely used in their pure form (*i.e.,* as direct output), their original purpose is to perturb a pattern so that it looks irregular. Mathematically speaking, the noise functions are composed with "pattern" functions :

$$material(x) = pattern(noise(x))$$

In this report, we will call "*Compound noise*" such noise functions.

Pattern functions used for compound noise can be arbitrarily complex : designers can experiment many possibilities to come up with new procedural textures. However, several classes of compound noise functions stand out :

- $Fnoise(\boldsymbol{x}) = \sum_{i=0}^{N-1} \frac{noise(b^i x)}{a^i}$ or $Tnoise = \sum_{i=0}^{N-1} |\frac{noise(b^i x)}{a^i}|$
  Fnoise is the Fractal noise described earlier. Tnoise, or turbulence, is a fractal sum of absolute value of noise.
  Usage : They are rather used in other functions than alone.

- $f(dist(x) + Fnoise(x))$
  The perturbation of a distance function (*i.e.,* a function parameterized with a distance function (namely, a sphere)
  Usage : It is often used to model density volumes (*e.g.,* clouds, smoke)

- $pattern(x + k * Fnoise(x))$
  Perturbs the support of the pattern function.
  Usage : It is widely used to add randomness to patterns. A famous example is the marble, where the lines are displaced to act like veins of marble.

- $\nabla Fnoise(x)$
  The gradient of noise function.
  Usage : it is often used to perturb the normals of a surface in order to simulate irregularity (bump mapping)

This list is non exhaustive, but is representative of the various applications of noise.

3.2.3 Look Up Tables

In computer graphics, a color is defined by a vector of 4 values : red, green, blue and alpha (*i.e.,* the opacity). A direct consequence in the context of procedural textures, is that the function must return 4 parameters (or at the RGB) in order to output a colored result. Instead, it is more convenient to use a lookup Table.
A look-up table (LUT) is a mechanism used to transform a range of input values into another range of precomputed values. Therefore, the can be used transform the value of a scalar function into 4-dimensional vector, and thus a color. The noise functions usually returning a scalar, it is a simple method to obtain colored patterns (colored compound noise are obtained this way). However, this introduce a new type of aliasing, called *index aliasing* by Worley [EMP+02].

Chapter 4

Problem analysis : Properties of noise models

Using noise functions without causing artifacts, especially when composing functions, is very tricky. The spectrum of the final function is very difficult to guess without a thorough analysis beforehand. So the designers have very few control over the aliasing and the general behavior of the functions. In the following, we will focus our analysis on Perlin noise because it is the most used noise function. We will also only consider the filtering for isotropic cases, since the footprint is spherical in our context of procedural volumes of density.

4.1 Definition of the problem

We saw in section 3.2 that noise functions can be fractally summed to obtain *fractal noise*, arbitrarily composed with various functions, then used as index of a Lookup Table or to distort patterns. The applications are numerous but designers lack of solution to control artifacts issues.
Noise has a certain computational cost of its own, so adding a heavy filtering process must be avoided as much as possible. By specifying the lower and the higher octave of fractal noise, one can obtain a low cost kind of MIP-mapping, with a scale adapted to the sampling rate. However, it appeared that the base noise has more high frequencies than ideally stated. As a result, the Perlin noise is postfiltered in practice [Ste01].
Most often, an unstated assumption is often made when filtering compound noise function:

$$f(noise(x)) * filter = f(noise(x) * filter)$$

Which is false for f non linear.

These problems were more or less ignored until the possibility to render the noise in real-time emerged. Postfiltering is indeed the last ressort in real-time because of its cost (whereas it is affordable offline). The recent alternative solutions (see [LLDD09]) focus on offering better spectral control in order to address the issues of aliasing, while remaining fast on GPU. They provide band-limited functions, particularly compliant to the fractal noise, as well as anisotropic filtering solutions. Nevertheless, no solutions exist to efficiently control the aliasing in the case of compound noise functions and LUT.

A filtering method described by Peachy [EMP+02], was to evaluate the lower and the higher octave of the fractal sum. But the base noise was assumed to have no frequencies higher than 1, and the extension to compound noise was totally ignored. Moreover, higher octaves of Perlin noise still contains low frequencies, thus erasing details when this octave is cut off making.

Yet, the workflow of noise construction process is well known and should be used to take care of the problem. If we analyze the properties of the noise functions at each stage of the noise pipeline, we can fix the existing methods to address these problems. Our purpose is to take advantage of the properties of the functions used in the process, in order to erase all unnecessary evaluation of the base noise.
We denote an unnecessary evaluation each time the result of the base noise function has a null or negative effect on the final image. Negative effects are, of course, the aliasing artifacts we described earlier, whereas null effects occurs whenever the color of the sampled point is independent from its evaluation. Since, base Noise is always at least called in a fractal sum, minimizing the number of terms in the sum is an equivalent goal.
The interest of this "lazy evaluation" approach is twofold : Firstly, it optimizes the rendering, since only the relevant terms are evaluated. Secondly, it reduces the aliasing as well as the loss of details.

4.2 Analysis

4.2.1 Fractal analysis

We showed the principle of Perlin noise earlier in section 3 and we noted among several important properties that the base noise functions should be bounded between -1 and 1. We can use this property to study the boundary of the fractal sum of the noise.

let $x \in \mathbb{R}^{1..4}$, and $b_i(x) = \dfrac{noise(2^i x)}{2^i}$ the i-th octave of the fractal noise $Fnoise_N(x)$,

We known that

$$|noise(x)| \leq 1$$

Then the scaled octave is

$$|b_i(x)| \leq \frac{1}{2^i} \tag{4.1}$$

Consequently, the fractal sum is bounded by the sum of the octave boundary

$$|Fnoise_n(x)| \leq S_{n-1} \tag{4.2}$$

With $S_n = 2 - 2^{-n}$ the geometric series of $n+1$ terms and ratio of $\dfrac{1}{2}$

Let $Nmax$ the maximum octave to be summed, we have :

$$|Fnoise_n(x)| \leq S_{Nmax-1} \quad \forall n < Nmax \tag{4.3}$$

Since $S_n < 2 \; \forall n$, this means that the fractal noise never exceed a maximum amplitude of 2. But in practice, a $Nmax$ will always be defined, so it is safer to express the boundary

with S_{Nmax-1}, which is a constant.

We'll see that knowing this boundary is important for detecting octaves with null effect. Also, this boundary can be used to normalize the fractal noise to achieve better control.

4.2.2 Spectral analysis

Our goal is to characterize the bandwidth of noise functions. To do so, we want to compute the minimum frequency for which we can neglect aliasing (*i.e.,* consider it non visible), and the maximum frequency for which we can neglect low frequencies (*i.e.,* no visible information are displayed) (Figure 4.1). Since we work in discrete spaces, the analysis of the spectrum will be done with the discrete Fourier transform.

The Fast Fourier Transform is known for its efficiency to compute the Discrete Fourier Transform of a signal. But this tool must be handled with care : the signal must be enough sampled, and must be long enough to characterize the overall frequency behavior. For instance, during the early stage of our analysis, we noticed that artifacts in the spectrum which were due to a bias in the noise function. Kensler [SKB09] explained this was caused by the hash function embedded in the Perlin noise function. The permutation table actually induces a correlation between the dimensions.

Choice of a metric

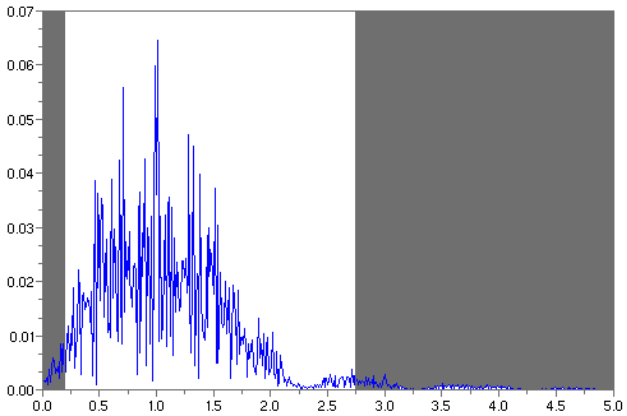


Figure 4.1: Characterization of bandwidth on the 2^{nd} octave of fractal noise. "Most" of the energy is contained between the grey parts, making it "almost" band-limited. We aim at actually measuring bandwidth, given an energy threshold (*e.g.,* 95%).

In order to neglect frequencies, we have to ensure that their contributions to the signal are low. For instance, if we want to neglect the high frequencies of a noise function, because they are too low to generate noticeable aliasing, then we can split the function :

$$noise = noise_{low} + noise_{high}$$

And make sure that the contribution of $noise_{high}$ is below a certain value.

Measuring a function requires to choose a metric. In our study, we consider the L^2 – *norm*, since the mean-square minimization is the usual method for this kind of problem. We also use this metric because it appears to be the most meaningful. The spectra are traditionally displayed in the Real domain by the magnitude spectrum (*i.e.,* the modulus of of the Fourier Transform), or the Power Spectral Density (PSD) (*i.e.,* the square of the magnitude). The PSD allows us to the use the **Parseval's theorem** :

$$\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2 \tag{4.4}$$

where $X[k]$ is the Discrete Fourier Transform of $x[n]$, both of length N

This is exactly the identity we need so as to compute $||noise_{high}||^2$ from the spectral domain :
we have

$$noise = noise_{low} + noise_{high} \tag{4.5}$$

by linearity :

$$\mathscr{F}\{noise\} = \mathscr{F}\{noise_{low}\} + \mathscr{F}\{noise_{high}\} \tag{4.6}$$

Using 4.4, and the orthogonality of $\mathscr{F}\{noise_{low}\}$ and $\mathscr{F}\{noise_{high}\}$ we have that :

$$\sum_{n=0}^{N-1} |noise_{high}[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |\mathscr{F}\{noise_{high}[k]\}|^2 \tag{4.7}$$

Therefore, bounding the energy $||noise_{high}||^2$ is equivalent to bound the norm of the PSD of $noise_{high}$. As a result, to characterize the bandwidth of the noise functions, an energy threshold must be defined in order to decide which part of the spectrum are not visually significant. In our example, the minimum frequency for which we can neglect aliasing, based on an energy threshold T, is the minimum frequency for which $\frac{1}{N}\sum_{k=0}^{N-1} |\mathscr{F}\{noise_{high}[k]\}|^2 < T$.

Notice :
We chose this metric because of its good properties for the characterization of frequencies contribution in the whole spectrum. However, in the context of computer images, the perception of these contributions depends on the many other considerations such as contrast or color distribution.

Chapter 5

Contributions

Our previous analysis of the boundary of the fractal noise and the spectral analysis of the base noise functions allowed us to better understand the behavior of noise. In the following we present our works on the possibilities offered by a good characterization of the noise. We first describe an optimization method based on the fractal properties of the boundary. Then we introduce a smart filtering by optimization of the bandwidth. Finally, we present our experimentations in compound noise filtering.

5.1 Incremental bounding

Sometimes, the LUT returns constant colors for a certain range of values (*e.g.,* negative values). If the compound noise function returns a value in that range whatever the evaluation of the fractal sum, then it can be considered useless.

5.1.1 Method

The analysis of the boundary of $Fnoise_N$ allows us to come up with an optimization algorithm detecting these cases. The equation (4.2) show that $Fnoise_N$ is bounded by a series. However, we can recursively refine this boundary using equation 4.1 :

$$|Fnoise_{n+1}(x)| = |Fnoise_n(x) + b_{n+1}(x)| \tag{5.1}$$

$$|Fnoise_{Nmax}(x)| = |Fnoise_n(x) + \sum_{i=n+1}^{Nmax-1} b_i(x)| \tag{5.2}$$

Hence

$$|Fnoise_{Nmax}(x)| \leq |Fnoise_n(x) + \sum_{i=n+1}^{Nmax-1} \frac{1}{2^i}| \tag{5.3}$$

$$|Fnoise_{Nmax}(x)| \leq |Fnoise_n(x) + S_{Nmax-1} - S_n| \tag{5.4}$$

Consequently, we can obtain a better estimation of boundary when the fractal sum is incrementally evaluated. At each step, we recompute the boundary with an offset of the current value. In practice, since the Fractal noise is implemented in a loop, this is particularly suited for an optimization.

The optimization consists in testing if the set of possible values of the fractal noise evaluated at x is included in the set of values for which the final color is constant.

5.1.2 Results

We tested the method on the distance function (Figure 5.1):

$$distNoise(p) = CLAMP(a * d + 1 + Fnoise(p))$$

with $d = ||p||$, the distance to the center, a the slope and CLAMP(x) returns 0 for x<0 and 1 for x>1.

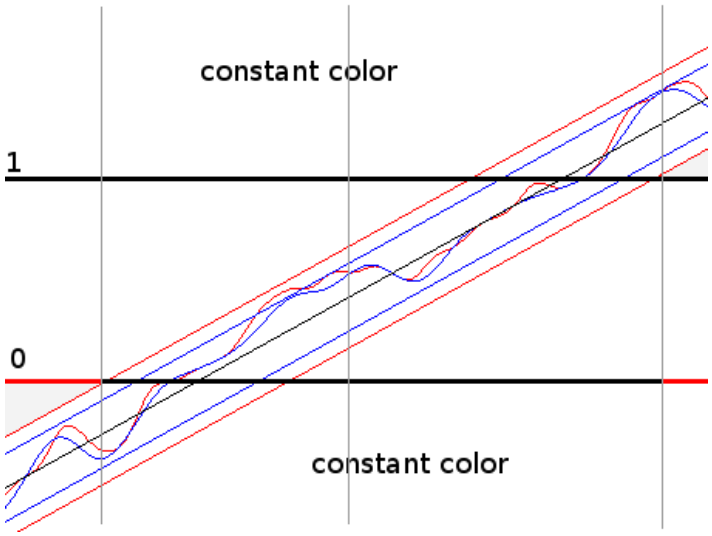


Figure 5.1: Distance Noise. The area betwen 0 and 1 returns non constant color. Each octave added extends the boundary of the possible values of the function. The values in the red areas are guaranteed to return a constant color whatever the value of the function.

Estimation of the cost without optimization for N octaves in 2D :
$\mathbb{O}(n^2.N)$
Estimation of the cost with our optimization for N octaves in 2D:
$\mathbb{O}(\sum_{i=0}^{N} \pi.n^2((R+(S_N-S_i))-(R-(S_N-S_i)))$

Our method is far less sensitive to the number of octave : since $S_N - S_i$ decreases quickly, the surface on which the octaves are evaluated decreases too as seen on Figure 5.3. The gain of performance is more than noticeable (see table 5.1, especially when the number of octave rises.

5.2 Spectral bounding

In the previous chapter (4.2.2) we justified the choice of the $L^2 - norm$ as a metric to measure the contribution of a signal on the screen. We showed that thanks to the parseval theorem, it is possible to measure this quantity directly in the spectral domain, using the PSD of the signal. In the following, we describe a method to compute the "relevant" bandwidth of the base function (*i.e.,* the band of non-neglectable frequencies) ; then use this information to efficiently filter the fractal noise.

number of octave	Incremental bounding	original implementation
0	19.1 ms	54.1 ms
1	40.4 ms	71.9 ms
2	49.8 ms	117.5 ms
5	64.6 ms	298.8 ms
7	69.5 ms	460.0 ms
10	89.4 ms	1069.1 ms

Table 5.1: Performances comparition in milliseconds. Computed on the smoke ball (Figure 5.2) rendered by ray-marching in a unit-cube subspace. Screen resolution 512*512 ; integration step 0.01, (integration stops at sufficient opacity (*e.g.,* 99%))

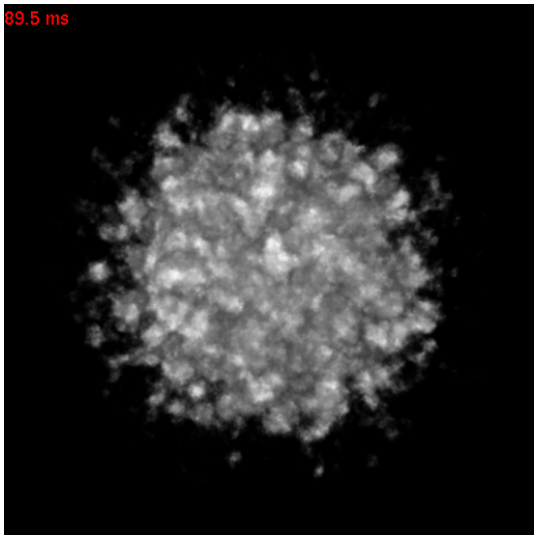


Figure 5.2: Smoke ball, 10 octaves

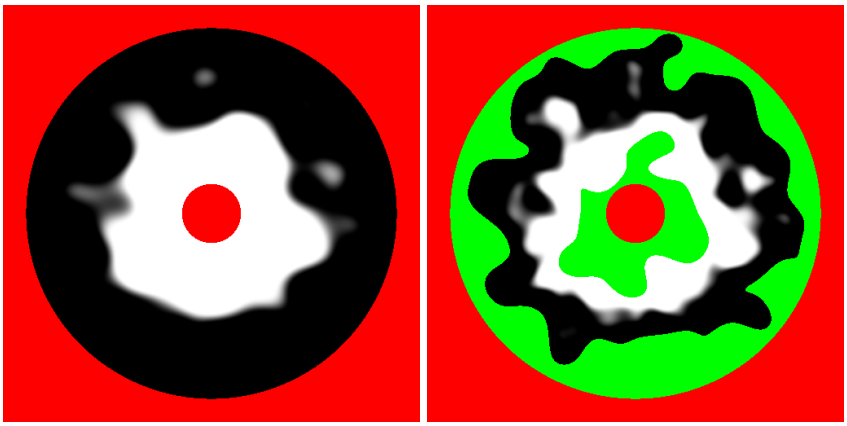
5.2.1 Method

In order to efficiently compute fractal noise without aliasing nor loss of details (when used in its pure form), we have to :

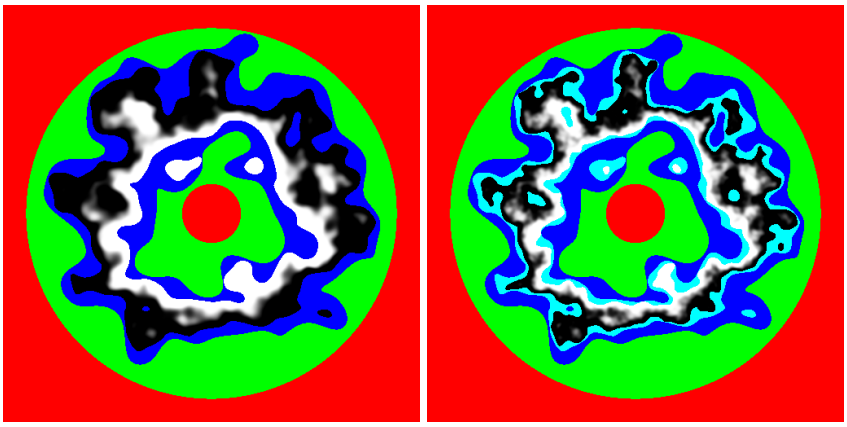
- **keep** all the terms with no frequency above Nyquist
- **cut** all the terms with no frequency below Nyquist (*i.e.,* entirely aliased)
- **filter** all the terms with frequency both above and below nyquist (*i.e.,* only keep the frequencies below)

This is a major optimization since the actual filtering is targeted on a limited number of octaves. That is, we truly use the band-limited properties of the octaves (Figure 5.4).

Now, we need a method to determine the bandwidth of each octave. That is, we use the metric defined earlier to decide which frequencies can be neglected. Note that we restricted ourself in the 1D case for this analysis, but considering isotropic functions, this restriction should be safe. Moreover, an extension to higher dimension should be direct.



(a) N=1 (b) N=2



(c) N=3 (d) N=4

Figure 5.3: Inner and outer boundaries of a disk perturbed with Fractal noise at N octaves. At each step, the area out of the boundary (*i.e.,* with a guaranteed constant value) is colored. A different color is displayed according to the number of octaves evaluated to confirm that the sample is out of bound

First, we choose a threshold T, the maximum energy for which the signal has no noticeable impact on the final image. Therefore, we will assume in this method, that the variance of the signal has a major impact of the color intensity perceived (*i.e.,* an image with a total energy below T will appear almost black).
Since the color intensity of a pixel is coded on [0..255], we will choose the threshold accordingly :
If we consider that a color intensity of $2/255$ is barely visible, then

$$T = (2/255)^2$$

is such a threshold.

Secondly, we compute the PSD the base noise function. It gives the energy of each frequency.
In a third time, we compute

$$\Gamma(\lambda) = \int_{\lambda}^{+\infty} PSD(f)df$$

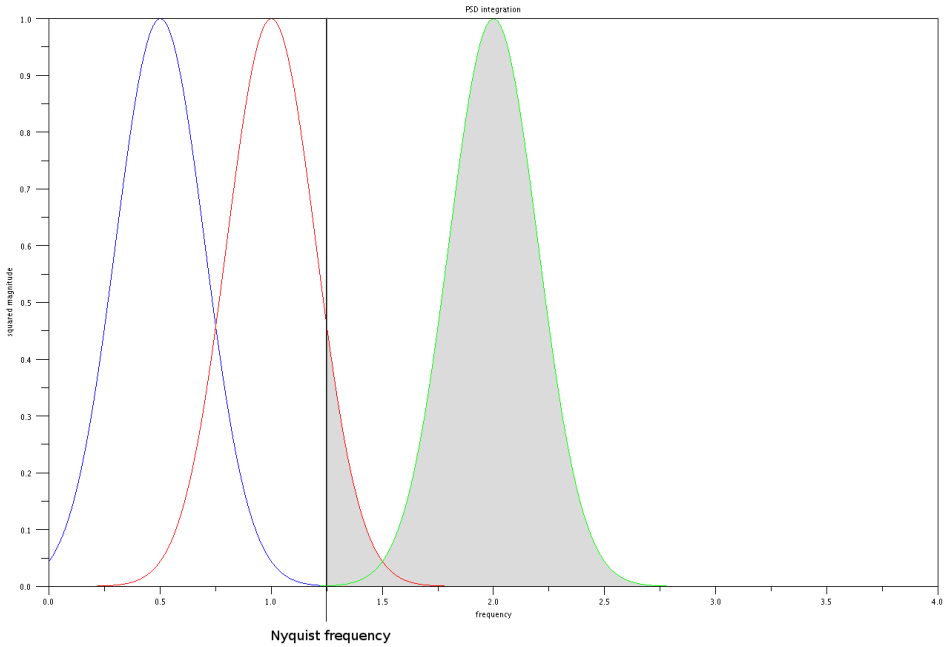


Figure 5.4: illustration of our optimization process for a spectrum being the sum of 3 spectra. The green spectrum is above Nyquist and must be **cut** to prevent aliasing ; the red spectrum has some of its frequency above Nyquist and must be **filtered** to remove the high frequency part; the blue curve is below Nyquist and must be **kept**

This function is a direct application of the **Parseval's theorem**, and returns the energy of the frequencies above λ.

Finally, we compute

$$F_{sup} = \Gamma^{-1}(T)$$

to obtain the minimum frequency for which we can neglect aliasing : the superior boundary of our bandwidth.
The computation of the inferior boundary, the maximum frequency for which we can neglect low frequencies, is done similarly, with

$$F_{inf} = \Gamma^{-1}((\sigma - \sqrt{T})^2)$$

where σ the standard deviation of the signal (because $\int(PSD) = mean(signal)^2 + var(signal)$).

As a result, $[F_{inf}, F_{sup}]$ is the relevant bandwidth of the spectrum (see Figure 5.5). And we can keep, cut or filter the signal according to the Nyquist frequency.

5.2.2 Results

We carried out a test of our method on GPU, using the numerical computational software Scilab to do a spectral analysis of the Perlin noise function.

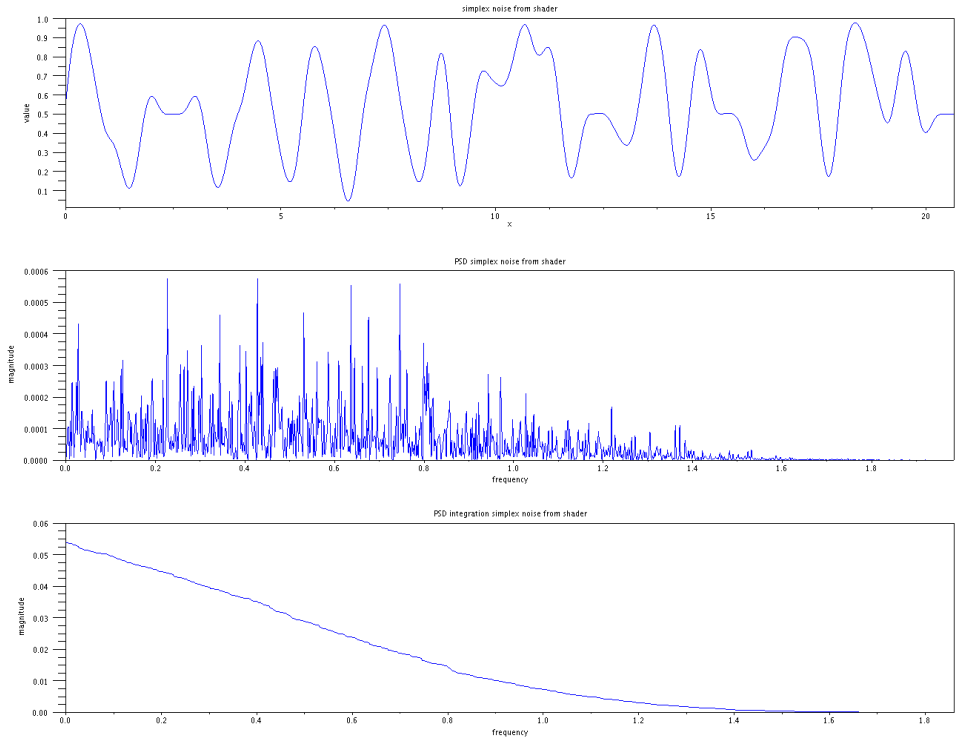


Figure 5.5: Perlin noise ; PSD of the function ; $\Gamma(\lambda)$. Here we obtain $[F_{inf}, F_{sup}] = [0.0019, 1.7460]$ with $T = (2/255)^2$

Importing the signal

To analyze the signal on Scilab, we wrote a script to import the image generated by the shader as a 1D signal :

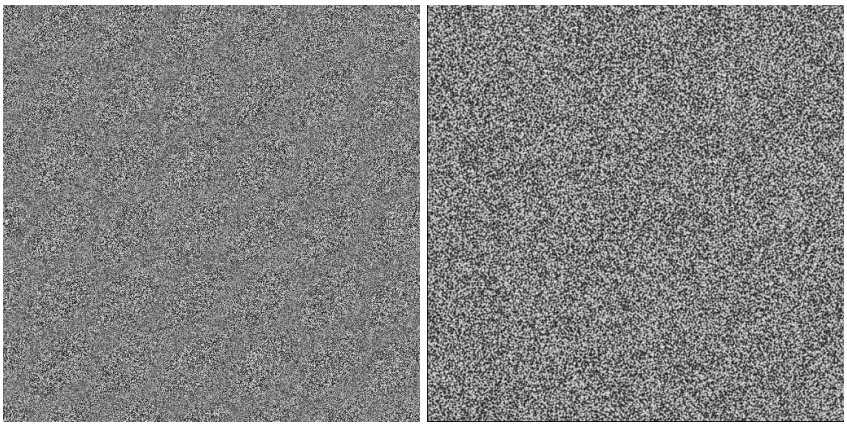
1. We modified the output of the shader to encode a 1D signal on multiple lines (Figure 5.6) :

Figure 5.6: A long sampling of the signal is evaluated in 1D, then split into blocks and displayed on screen

2. The value of each sample is hashed into the 3 color channels of the pixel, in order to avoid loss due the fact that a color is coded on $[0..255]$ (Figure 5.8 (a)). The program on GPU is :

```
//the output value (in range 0..1) is scaled to fit
//the range 0..256*256*256 (possible number of value
//coded in 3 chanels)
int val = noise(p)*16777216;
//the value is hashed : each color carries different
//bit weight of the value
color.r=(val/65536)/256.0;
color.g=(val/256-(val/65536)*256)/256.0;
color.b=(val%256)/256.0;
```

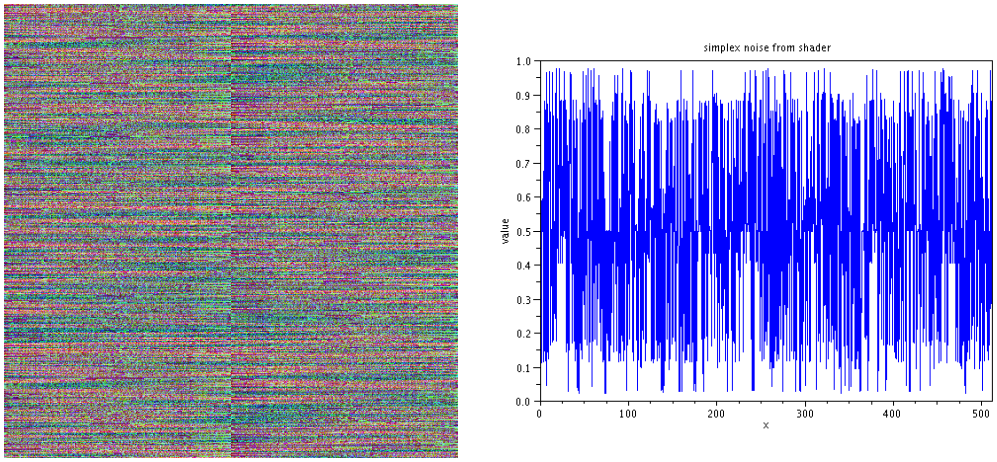
3. The obtained picture is then read by Scilab, which reconstruct the signal with the inverse operation (Figure 5.8 (b)).



(a) Original signal ; apparition of unwanted patterns due to aliasing

(b) Normalized signal ; no artifacts because no noticeable frequencies above Nyquist's

Figure 5.7: Comparition between the original and spectrally normalized Perlin base noise. Signal sampled at $2*2pixel/unit^2$.



(a) split and hashed signal displayed on screen

(b) Reconstructed signal

Figure 5.8: 1D noise split into lines and hashed on the color channels, then reconstructed

Once the signal loaded and reconstructed by Scilab, we run a script computing the bandwidth like described in section 4.2.2. The script we wrote is automated and requires

2 inputs : the energy threshold, and the noise signal. Note that we used Scilab because it contains the packages necessary to our analysis, but a direct implementation in the rendering application would be easy to carry out.

interpretation of the results

A first use of this information, is the normalization of the base function's spectrum with a precomputed F_{sup}, in order to fix the max frequency of the 1^{st} octave to 1 :

$$Nnoise(x) = noise(\frac{1}{F_{sup}} \cdot x)$$

This simple normalization make the base noise closer to the ideal noise criteria, and therefore easier to handle (Figure 5.7). However, when summed together the high frequencies neglected might eventually show up (see Table 5.2).

signal	F_{inf}	F_{sup}
1^{st} octave	0.0019531	1.7460938
2^{nd} octave	0.0292969	2.765625
$Fnoise_2$	0.0019531	3.1738281

Table 5.2: Computed bandwidth of Fractal noise and its octaves. The bandwidth of the fractal noise should be the minimum and maximum boundaries of its octaves. It is not the case because the base functions are not purely band-limited

A second idea then, is to precompute and store Γ for the base function, in order to sum the spectrum at different scales, and/or apply any other operation, to compute the bandwidth of more complex functions.
Anyway, Γ^{-1} can be computed for any signal, thus enabling the use of our method in the general case of adaptive filtering. Although it would require an extension to higher dimension.

5.3 An approach on compound noise

Filtering compound noise is very difficult to perform (apart from brute-force supersampling) due to the arbitrarily complex functions composed with noise functions. This causes the spectrum of the compound noise to have a completely different bandwidth than the Fractal noise. Therefore, filtering either f or g doesn't guarantee that $f \circ g$ will be filtered correctly. So, to know how to filter the fractal noise in order to filter the compound noise, a spectral analysis must be done.

In the previous section, we proposed a method to compute the bandwidth of the noise functions. Computing a non biased Fourier transform for every functions can be a bit slow, because a spectrum should be computed for each N of $Fnoise_N$. So, we rather deduce an expression of the final spectrum from the base spectra. A better knowledge of the spectral properties of the base noise functions is a first step in the resolution of the

compound noise filtering problem.
The coumpound noise functions are usually in this form :

$$Cnoise(x) = pattern(Fnoise(x))$$

We notice that the support of the pattern function is restricted to the range of values of $Fnoise(x)$. This means that, knowing the boundary B of $Fnoise(x)$, we can use whatever pattern function f such as

$$f(x) = pattern(x) \quad \forall x \in B$$

Therefore, a local approximation of the pattern function could be useful to simplify the complexity of spectral operations. Indeed, restricting the pattern function to simple operations in the spacial implies simple operations in the spectral domains (*e.g.,* additions, convolutions).

An idea is to interpolate the pattern function with a polynom in order to restrict operations to sums and convolution of basic spectra. For instance :

$$Cnoise(x) = a \cdot Fnoise(x)^2 + b \cdot Fnoise(x) + c$$

will result in spectral domain in

$$\mathscr{F}\{Cnoise(\lambda)\} = a \cdot \mathscr{F}\{Fnoise(\lambda)\} * \mathscr{F}\{Fnoise(\lambda)\} + b \cdot \mathscr{F}\{Fnoise(\lambda)\} + c \cdot \delta(\lambda)$$

Where $\delta(\lambda)$ is the Dirac function, and $*$ denotes the convolution operator.

Therefore, a marble pattern (Figure 5.9) could be written as

$$\begin{aligned} marble(x) &= \sin(a \cdot x + b \cdot Fnoise(x)) \\ &= \sin(a \cdot x) \cdot \cos(b \cdot Fnoise(x)) + \cos(a \cdot x) \cdot \sin(b \cdot Fnoise(x)) \end{aligned}$$

The Taylor series are particularly suited to local interpolation. A 1^{st} order development would give

$$marble(x) \approx \sin(a \cdot x) - \frac{\sin(a \cdot x) \cdot (b \cdot Fnoise(x))^2}{2} + b \cdot \cos(a \cdot x) \cdot Fnoise(x)$$

This Fourier transform of this expression is easier to compute than the original. The products become convolutions in the Fourier domain, and sin and cos functions become shifted diracs. As a result, computing the bandwidth of the compound noise function could be done with simple operations on the Fractal noise spectrum.

This first approach could overcome the computational cost of the method of our spectral bounding methods in the case of Compound noise functions. However, sereval points have to be clarified : the validity of the approximated spectrum with respect to the actual one ; the performances achieved ; and the eventual convenience of using such a method.

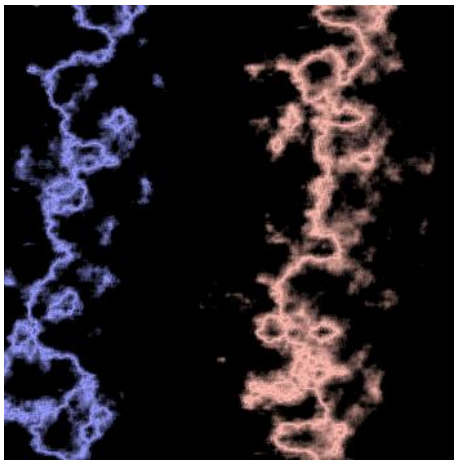


Figure 5.9: example of marble noise

Chapter 6

Conclusion and future works

We showed that the optimizations based on the fractal noise are really important for a proper use of noise functions. Firstly, it allows an important gain in performances in the context of procedural volumes, where every unnecessary evaluation have a dramatic impact. Secondly, the repartition of details in each octave of the fractal noise implies that a careful evaluation of the octaves is really suited for the filtering. However, these optimizations are not possible without a good characterization of the noise properties, especially the bandwidth. The characterization of the bandwidth permits an evaluation and a filtering, only when necessary.
Unfortunately, we couldn't test the efficiency of the proposed filtering methods more deeply during this master's thesis. Therefore a comparition of the different approaches with sereval base noise is necessary to draw conclusions. Furthermore, our first idea on the filtering of compound noise must be studied more deeply.

Also, many concerns on noise are still unsolved : from the point of view of expressiveness and features, the user would like to have more control over noise: for example, to follow an outline, or to allow inclusions (knots of wood). Similarly, the extension of 2D noise to 3D is not always so easy for the user: it is sometimes difficult to reproduce some patterns yet natural in 2D: "spikes", "filaments network" (like trabecular). Using noise in animations, for instance to enrich the apparent resolution of a physical simulation, requires to fulfil some constraints (velocity, curl, etc.).

Bibliography

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, 1978. 3

[Cat74] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974. 3

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, feb 2009. 7

[EMP+02] David S. Ebert, Kenton F. Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing & Modeling: A Procedural Approach, Third Edition (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, December 2002. 12, 18, 20

[Gre05] Simon Green. Implementing improved perlin noise. In Matt Pharr and Randima Fernando, editors, *GPU Gems 2*, chapter 26. Addison Wesley Professional, March 2005. 16

[Gus05] S. Gustavson. Simplex noise demystified, 2005. 14

[GZD08] Alexander Goldberg, Matthias Zwicker, and Frédo Durand. Anisotropic noise. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–8, New York, NY, USA, 2008. ACM. 4

[Hec86] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6:56–67, 1986. 9

[Hec89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Technical report, Berkeley, CA, USA, 1989. 9, 10

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM. 7

[KK89] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. *SIGGRAPH Computer Graphics*, 23(3):271–280, 1989. 5

[KW03] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2003. 6

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, 1994. 6

[LLDD09] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. Procedural noise using sparse gabor convolution. *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)*, 28(3), August 2009. 19

[Max94] Nelson L. Max. Efficient Light Propagation for Multiple Anisotropic Volume Scattering. In *Fifth Eurographics Workshop on Rendering*, pages 87–104, Darmstadt, Germany, 1994. 5

[Ney98] Fabrice Neyret. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):55–70, 1998. 5

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002). 6

[Pea85] Darwyn R. Peachey. Solid texturing of complex surfaces. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 279–286, 1985. 4

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985. 4, 12

[Per01] Ken Perlin. Noise hardware. course notes, 2001. 14

[Per02] Ken Perlin. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, New York, NY, USA, 2002. ACM. 14

[Per04] Ken Perlin. Implementing improved perlin noise. In *GPU Gems*, chapter 5. Addison Wesley Professional, 2004. 16

[SKB09] Josef B. Spjut, Andrew E. Kensler, and Erik L. Brunvand. Hardware-accelerated gradient noise for graphics. In *GLSVLSI '09: Proceedings of the 19th ACM Great Lakes symposium on VLSI*, pages 457–462, 2009. 21

[Ste01] Ian Stephenson. Antialiasing perlin noise. SIGGRAPH sketch, 2001. 19

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, 1978. 3

[Wil83] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, 1983. 10

[Wor96] Steven Worley. A cellular texture basis function. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 291–294, New York, NY, USA, 1996. ACM. 16