Noisy Gradient Meshes
Procedurally Enriching Vector Graphics using Noise

Bachelor Thesis

Rowan van Beckhoven

Computer Science
Faculty of Science and Engineering
University of Groningen

July 2018

Supervisors:
Jiří Kosinka
Gerben Hettinga
Abstract

Vector graphics is a powerful approach for representing scalable illustrations. These vector graphics are composed of primitives which define an image. Existing high-level vector graphics primitives are able to model smooth colour transitions well and can be used to vectorise raster images which feature large regions with constant or slowly changing colour gradients. However, when applied to natural images, these methods often result in many small gradient primitives due to high frequency regions present in the image. Working with many gradient primitives becomes a tedious process and thus a solution has to be found in which a high level of detail can be achieved while the underlying mesh remains simple. Procedural noise has been used for texture synthesis before, but it is rarely used in combination with vector graphic primitives. In this research, a gradient mesh is combined with a procedural noise function.
# Contents

1 Introduction .................. 1  
  1.1 Gradient Mesh .................. 1  
  1.2 Noise ........................ 3  
    1.2.1 Perlin Noise .................. 4  
    1.2.2 Worley Noise .................. 4  
  1.3 Structure of this Thesis .................. 5  

2 Related Work .................. 6  

3 Problem Statement .................. 8  

4 Approach .................. 9  

5 Implementation .................. 10  
  5.1 Gradient Mesh Framework .................. 10  
    5.1.1 Halfedge Structure .................. 10  
    5.1.2 Framework Extensions .................. 11  
  5.2 Pipeline .................  
    5.2.1 Noise Field Generation .................. 13  
    5.2.2 Distortion .................. 14  
    5.2.3 Filtering .................. 15  
    5.2.4 Colour Mapping .................. 17  
    5.2.5 Colour Blending .................. 17  

6 Evaluation .................. 18  
  6.1 Methods .................. 18  
  6.2 Results .................. 19  
    6.2.1 Noise Functions .................  
      6.2.1.1 Perlin Noise .................. 19  
      6.2.1.2 Worley Noise .................. 19  
    6.2.2 Distortion .................. 21  
    6.2.3 Filtering .................. 22  
    6.2.4 Examples .................. 23  
  6.3 Discussion .................. 23
## CONTENTS

### 7 Conclusion and Future Work  
#### 7.1 Contributions of this Thesis  
##### 7.1.1 Tool  
#### 7.2 Limitations  
#### 7.3 Conclusion  
#### 7.4 Future Work  

### A Appendix  
#### A.1 Functions  
##### A.1.1 OpenGL Functions  
##### A.1.2 Filter Functions  
#### A.2 Colour Blending Modes
Texture generation using procedural noise functions has been around since Ken Perlin invented his world-famous *Perlin noise*. This noise has been used for a multitude of purposes, ranging from modelling natural phenomena [1] to computer generated cities in games [2]. Since then, a large number of procedural noise functions have been invented. One of the advantages of using procedural approaches to generate texture is that it is generally very fast. Using a noise function for this approach also allows for patterns with a theoretically unlimited amount of detail. Lagae et al. evaluate multiple procedural noise functions in their research [3]. They describe a number of properties that a procedural noise function should have:

- the function is continuous
- it is non-periodic
- it is parametrized
- it is randomly accessible

These properties are the reason why procedural noise functions are extremely suitable to use in a pixel shader. These noise functions are often used as a first step in a longer pipeline of functions. In this research, two different procedural noise functions are combined with a gradient mesh, a commonly used vector graphics primitive: Perlin noise [4] and Worley noise [5]. Additionally, a distortion block, filtering block and colour blending options are implemented which allow for extremely customizable pattern generation.

### 1.1 Gradient Mesh

The gradient mesh primitive can be used by designers to create a 2D mesh on which colours can be assigned to vertices. These colours are smoothly transitioned between vertices, allowing to
CHAPTER 1. INTRODUCTION

represent images with smooth colour transitions. The gradient mesh consists of polygons, often referred to as patches. Traditionally, a gradient mesh is represented as a rectangular array of Ferguson patches \[6\]. These patches are constructed by vertices, which are used as control points in the gradient mesh. Each vertex consists of two tangent vectors, which are used to control the curve between two vertices. The curves which finally make up the 2D grid are cubic Bézier curves which are formulated by

\[B(t) = (1 - t)^3 P_0 + 3(1 - t)^2 t P_1 + 3(1 - t)t^2 P_2 + t^3 P_3\]

where \(P_0\) and \(P_3\) are the vertices, \(P_1\) the tangent vector from \(P_0\) and \(P_2\) the tangent vector from \(P_3\) (See Figure 1.2). Patches are represented in a parametric surface form. A Ferguson patch is a commonly used parametric representation.

A Ferguson patch uses bicubic (Hermite) interpolation in the form of

\[m(u, v) = UCQC^T V\]

where \(U\) and \(V\) are the \((u, v)\) coordinates as vector representations given by:

\[U = \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix}, V = \begin{bmatrix} 1 & v & v^2 & v^3 \end{bmatrix}\]

\(Q\) the matrix of the control points of the patch

\[Q = \begin{bmatrix} m_0 & m_1 & m_0^u & m_1^u \\ m_2 & m_3 & m_2^u & m_3^u \\ m_0^u & m_1^u & 0 & 0 \\ m_2^u & m_3^u & 0 & 0 \end{bmatrix} \]

and \(C\) the matrix of the Hermite basis function coefficients.

\[C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3 & 3 & -2 & -1 \\ 2 & -2 & 1 & 1 \end{bmatrix}\]
1.2 Noise

Procedural noise functions come in a number of classifications. These classifications are based on methods that are used in a noise function.

**Lattice noise**

Lattice noises are based on an integer lattice. By interpolation of random values at lattice points, a noise field is created. Different methods are used for interpolation, varying from linear to cubic interpolation methods. Cubic interpolation has a continuous first and second derivative, resulting in a noise field with smooth transitions. *Value noise, Gradient noise* and *Value-gradient noise* use values, gradients and both values and gradients, respectively.

Lattice noises often have the deficiency that artifacts appear in line with the lattice, resulting in the noise being anisotropic.

The most well-known lattice gradient noise is Perlin noise (see Section 1.2.1). Other examples are *Simplex noise* and *Wavelet noise*.

**Convolution noise**

Convolution noises are generated by convolving a filter, which is radially symmetrical. Lattice noise can also be generated using convolution noise, in which case the filter kernels are assigned at lattice points.

Sparse convolution noises are generated by scattering random impulses and assigning the kernel to these impulses. This technique of generating a sparse field of impulses is used in multiple procedural noise functions, like Worley noise (see Section 1.2.2).

Known convolution noise implementations are *Gabor noise* and *Spot noise*.

**Explicit noise**

Unlike implicit procedural noise functions, explicit noise functions cannot be used directly during rendering. Noise values are generated explicitly and are stored in a table or a texture. To use these explicit noises during rendering, this table or texture has to be generated beforehand.

Interesting methods like the inverse Fourier transform of a frequency spectrum can be used, which allows for filtering of frequencies. This technique is known as *Fourier spectral synthesis*. 
Procedural noise functions generally generate band-pass noises [3], meaning that the noise function will generate a pattern which has a limited frequency range. This frequency can be controlled by scaling the noise. By generating and combining multiple octaves of noise, a larger spectrum of noise can be achieved, which is also known as fractal noise (see Figure 1.3). This spectrum consists of multiple bands of noise which can be assigned weights in order to control the spectrum.

1.2.1 Perlin Noise

Ken Perlin introduced Perlin noise [4] in 1985, which was the first procedural noise function. This noise is based on a uniformly spaced grid. A point is sent to the function which places the point in a two-dimensional integer lattice. For each lattice point, a pseudo-random gradient vector is computed. The dot product of this gradient vector and the vector from the lattice point to the coordinates of the point is computed for all lattice points, which finally are used for a bicubic spline interpolation at each point. This spline interpolation uses a 5th degree polynomial to provide $C^1$ continuity:

$$x = 6t^5 - 15t^4 + 10t^3$$

This principle can be extended to any dimension, but the most common use-cases are with either two- or three-dimensional Perlin noise.

1.2.2 Worley Noise

Similar to Perlin noise, Worley noise [5] uses a uniformly spaced grid. Each square on the grid contains zero or more feature points. To compute a noise scalar, a function $F(n)$ is defined which computes a scalar based on the $n^{th}$ nearest feature point. Computing $F(n)$ is done by taking the eight adjacent squares and the square in which the point itself lies and computing the feature points in these squares. Computing $F(n)$ up to $n = 4$ produces varying patterns, whereas higher values of $n$ start to look similar (see Figure 6.2)[5]. Linear combinations of $F(n)$ result in very interesting patterns, each with cellular-like characteristics (see Figure 6.3).

Since $F(n)$ is based on distance, different metrics can be used to gain varying results. For instance instead of using the Euclidean metric, the Manhattan distance can be used.

By combining layers of different noise frequencies, a fractal noise pattern can be achieved, although this does somewhat break the cellular characteristic of the noise. The cellular pattern that Worley noise creates can be used to represent numerous patterns that occur in nature, making this noise extremely useful.

Taken all these characteristics and variations into account, Worley noise is a suitable noise function to apply to a gradient mesh.
1.3 Structure of this Thesis

The structure of the remainder of this thesis is the following:

- In Chapter 2, the approach of related research is described
- In Chapter 3, the research problem is described in detail
- In Chapter 4, the thought process behind decisions is explained as well as the general approach for finding and evaluating solutions to the problem
- In Chapter 5, a detailed description of the implementation is given including the methods used for this
- In Chapter 6, results are evaluated and the methods used to evaluate these results are described
- In the final chapter, Chapter 7, the thesis is concluded and future work is discussed
Jeschke et al. [7] describe a similar problem as described in this research; Diffusion curves are similar to a gradient mesh in the sense that both representations consist of control points which make up an image. Diffusion curves however are able to construct both smooth transition regions as well as sharp edges, whereas a gradient mesh generally is used only for smooth transitions.

A key difference is that Jeschke et al. try to solve a similar problem, to represent details by means of procedural noise, but for the purpose of vectorising a raster graphics image. To vectorise an image, a simplified diffusion curve structure is generated on which a noise field is added based on the input image. This technique of generating noise based on an input image is called *noise by example*.

The second contribution of their paper consists of extending the diffusion curve primitive to be able to store noise parameters. For the purpose of *noise by example*, Gabor noise is a good fit for this approach [8].

The Gabor noise parameters are automatically derived by performing a Gabor filter analysis on the entire image. These noise parameters can be modified later on. Their approach of combining the noise parameters with the existing vertices is similar to the approach used in this research. Jeschke et al. limit their approach by having a separate grid on which the Gabor noise is generated. This way, when editing the curve network, the Gabor noise grid remains unchanged, resulting in the noise pattern to change instead of moving with the curves. This makes their approach infeasible for animating. Besides this, Gabor noise itself is already limited since it is a basic noise function. As described in *Texturing and Modelling: A Procedural Approach* [9], noise as basis function for texture generation is often insufficient.

The gradient mesh has been a topic for multiple researches. Sun et al. [6] use image vectorization to vectorise a raster graphics image to a gradient mesh. Considering that doing this results in highly dense gradient mesh structures, they propose a method to optimize this gradient mesh.
This results in a simplified gradient mesh structure which represents the raster graphics image with a high level of similarity.

Barendrecht et al. [10] extend on the traditional gradient mesh primitive by introducing a method for local refinement, allowing for adding details in specific regions. Furthermore, they introduce branching, which allows for a wider range of mesh topologies. Finally, they present a feature which allows for sharp colour transitions. By being able to represent both smooth transitions as well as sharp edges, the gradient mesh becomes similar to the diffusion curve representation.

The gradient mesh primitive was first introduced in Adobe Illustrator, after which it has been implemented in other vector graphic tools like CorelDRAW and Inkscape. These gradient mesh tools are used by 2D graphic designers to (re)create art or photo-realistic images. Although these tools have an option to add noise to an existing mesh, this often resolves to a very limited solution in which a user can add simple grainy noise which is not bound to any underlying grid, making it infeasible to use during a design process.

Procedural noise functions have been used for texture generation for over 30 years. Various implementations exist which create texture using procedural noise functions, however these are not linked to any topology but rather produce a square texture which can be used for texture mapping.

One interesting implementation is Filter Forge [11]. Filter Forge is used to procedurally generate filters with an immense level of customizability. The user has an enormous range of function blocks which can be used. These function blocks act as nodes which have defined inputs and outputs. By combining these nodes, a composition of multiple function blocks can be achieved. Although these filters cannot be combined with a gradient mesh, similar techniques can be used in a gradient mesh tool.
The goal of this thesis is to find a method to combine a procedural noise function with a gradient mesh in order to provide a high level of detail in the mesh while preserving a simple underlying structure.

In order to combine the noise function with the mesh, noise parameters should be combined with existing vertices which are then interpolated in a shader. Furthermore, the grid on which the noise is generated should be linked to the mesh in such a way that any geometrical change in the mesh structure also affects this grid.
The general approach to this problem is finding parameters which control the noise. To do this, the impact of different parameters on the final noise field is evaluated. These parameters can be implemented globally or locally. Global parameters are not bound to the mesh structure and therefore easy to edit, whereas local parameters are bound to vertices and therefore provide more control, but have to be set for all vertices.

Theoretically, it would be possible to have local parameters for each noise parameter. For example, weights of different types of noise can be combined in order to achieve a more complex pattern. These weights can then be locally defined, allowing for varying noise patterns in different regions. Although this sounds great, this quickly increases the amount of work needed to construct a noise texture. By limiting the amount of local parameters to only vital noise control parameters, the usability of this implementation is improved. Another argument against localizing every single parameter is that it would be much easier to simply be able to have multiple layers of different noise functions when multiple noise patterns are required.

To control the frequency spectrum of the noise, control over the frequency bands can be given as shown in Lagae et al. [3, Figure 7]. However, this method is not applicable to the solution of this thesis; having to define a frequency spectrum for every vertex is extremely unpleasant. A simple optimized egg yolk mesh already consists of 270 patches [6, Figure 1]. To define local noise frequency in a simple way, the fractal noise spectrum is used. This leaves many options such as using a Gaussian function to define the weights of the octaves, or by defining a linear function. A Gaussian function would be similar to using a band-pass filter on the frequency spectrum, whereas a linear function would leave the lower or higher bands intact. For this implementation, the latter is used.
This chapter is split into two sections. First the original implementation of the gradient mesh framework with its extensions is described. Secondly, the implementation of the fragment shader code is described.

5.1 Gradient Mesh Framework

The gradient mesh framework, provided by the second supervisor Gerben Hettinga, is written in C++, using the Qt framework. Therefore, this implementation was continued in the same language. The Qt framework uses OpenGL for rendering, which allows for the use of the OpenGL pipeline. In this pipeline, the OpenGL Shader Language (GLSL) is used [12]. The properties of a procedural noise function mentioned in Chapter 1 make it possible and feasible to use a noise function in the fragment shader.

The patches are sent as quads to the GPU which sends the vertex parameters from the vertex shader directly to the tessellation shader. During tessellation, the vertex parameters are interpolated using Hermite interpolation (see Section 1.1). Finally these parameters are used in the fragment shader, in which the noise function is implemented.

5.1.1 Halfedge Structure

The gradient mesh framework uses an underlying halfedge structure. This structure is not a requirement of a gradient mesh, but is a practical structure to traverse the mesh. Each patch consists of four halfedges, each pointing to the next and previous halfedge. A halfedge also has a twin, which is the halfedge on the other side of the adjacent patch (see Figure 5.1).
Each halfedge points to a vertex, which is used to retrieve the vertex parameters while traversing the halfedge structure. Using this halfedge structure also allows for parameters to be assigned to the halfedges instead of vertices, resulting in sharp edges (see Figure 5.2).

5.1.2 Framework Extensions

To combine the gradient mesh framework with the noise functions, both local and global noise parameters are added. Local parameters are stored in the vertex. Global parameters are stored in a settings class in the mesh.

To be able to persist the gradient mesh, a simple implementation to load and save mesh data is added. By obeying to the .OBJ standards [13], the mesh object is saved as a list of coordinates, indices and faces. Besides this, the global parameters and the halfedge structure are parsed.

The control of the tangent vectors is extended to provide \( C^1 \) (Ctrl + Click) and \( G^1 \) (Shift + Click) continuity while dragging. This is done by determining which tangent vector is opposite to the tangent vector being dragged.

For \( C^1 \) continuity, the opposite tangent vector \( \vec{x} \) is replaced by the negated vector of the tangent
vector being dragged $\vec{y}$:

$$\vec{x} = -\vec{y}$$

For $G^1$ continuity, the length of the opposite tangent vector is multiplied with the negated normalized tangent vector:

$$\vec{x} = -\vec{y} \ast \|x\|$$

One of the key elements to combine the noise functions with the gradient mesh is that the lattice grid is mapped to the gradient mesh structure. To do this, the $(u,v)$ coordinates are added to the vertices. During tessellation, these coordinates are interpolated bilinearly

$$u = \begin{bmatrix} 1 - x & x \end{bmatrix} \begin{bmatrix} u_0 & u_3 \\ u_1 & u_2 \end{bmatrix} \begin{bmatrix} 1 - y & y \end{bmatrix}$$

$$v = \begin{bmatrix} 1 - x & x \end{bmatrix} \begin{bmatrix} v_0 & v_3 \\ v_1 & v_2 \end{bmatrix} \begin{bmatrix} 1 - y & y \end{bmatrix}$$

where $x$ and $y$ are the tessellation coordinates and $u$ and $v$ the $(u,v)$ coordinates. These $(u,v)$ are then used in the noise functions. By doing this, manipulating the gradient mesh will also affect the underlying $(u,v)$ grid, which is used for the noise functions, and therefore affect the generated noise field (See Figure 6.10).

## 5.2 Pipeline

A key concept described in *Texturing and Modelling: A Procedural Approach* [9] is functional composition. This concept entails that functions can be used as building blocks with one or more inputs and one or more outputs. Using functional programming, this can be achieved easily. However, due to limitations of GLSL, this technique cannot be used. Therefore, a more imperative approach is used in this implementation, in which the pipeline of functions is already defined.

For this implementation, the final result is computed in a fragment shader. In this fragment shader, a pipeline structure is used, meaning that each output from the previous step is used in the next. The pipeline consists of five steps:

1. Noise Field Generation
2. Distortion
3. Filtering
4. Colour Mapping
5. Colour Blending


CHAPTER 5. IMPLEMENTATION

5.2.1 Noise Field Generation

By passing the \((u, v)\) coordinates into either the Perlin or Worley noise function, a noise field is generated. This field consists of pseudo-random scalars ranging in \([0, 1]\). This noise field can be used as a final result. However, this is very limited and only allows for cloud- or cellular-like textures. To achieve different results, the noise field is only used as a base function.

The noise field is generated based on a global frequency, a local frequency and the number of octaves \(m\). The global frequency is used to multiply the \((u, v)\) coordinates which are sent to the noise function. The local frequency is an interpolated value which determines how much each next octave is contributing to the final scalar \(N\). This is done using

\[
N = \frac{\sum_{i=0}^{m} p^i F(\vec{x} f 2^i)}{\sum_{i=0}^{m} p^i}
\]

where \(p\) is the local frequency, ranging in \([0, 1]\), \(\vec{x}\) the \((u, v)\) coordinates, \(f\) the global frequency and \(F\) being the noise function.

**Perlin noise**

The shader implementation of Perlin noise by Gustavson [14] uses a 2D version of Perlin noise. The Perlin noise function accepts a point as an argument. From this point \((u, v)\), the four lattice coordinates are computed by taking the integer part from this point

\[
\vec{P}_i = \begin{bmatrix} \lfloor u \rfloor & \lfloor v \rfloor & \lfloor u \rfloor & \lfloor v \rfloor \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}
\]

Instead of storing the lattice coordinates in four pairs, a vector containing four values is used. By combining elements from \(\vec{P}_i\), all four lattice coordinates can be formed.

The decimal part is used to compute the distance vectors from these coordinates to the point (See Appendix A.1.1 for \(\text{fract}\))

\[
\vec{P}_f = \begin{bmatrix} \text{fract}(u) & \text{fract}(v) & \text{fract}(u) & \text{fract}(v) \end{bmatrix} - \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}
\]

The lattice coordinates are then hashed using a quadratic permutation polynomial of the form

\[
x = (34x^2 + x) \mod 289
\]

The hash values are used to construct vectors, which range in \((-1, 1)^2\) after being normalised. The dot product is then taken of each vector with its corresponding distance vector. Finally these four values are interpolated. By using a quintic fade function, the interpolation is smooth (See Section 1.2.1).

**Worley noise**

The shader implementation of Worley noise by Gustavson [14] is slightly different from the original implementation by Steven Worley [5]: Instead of treating the square on the grid separately,
each row of three squares is treated as a vector. This vector contains exactly one feature point in
each square, which is shifted by a distance scalar and a direction vector. This distance scalar is
referred to by \textit{jitter}.

Reasons for choosing this implementation are mostly efficiency and that the original Worley noise
implementation does not translate well to GLSL.

Similar to the Perlin noise function, the Worley noise function also accepts a point of which
the integer coordinates and decimal part are stored. The hash values are computed in a similar
manner. These hash values are used to construct the direction vector. This vector is then used to
calculate a scalar value by using a distance metric. This results in nine distances from the point.

By sorting these coordinates, the nearest four points are returned.

Six predefined $F(n)$ combinations are implemented: $F_1, F_2, F_3, F_4, F_2 - F_1, F_2^2 - F_1^2$ (see
Figure 6.2) as well as two metrics: Euclidean distance $x^2 + y^2$ and Manhattan distance $|x| + |y|
(see Figure 6.3).

5.2.2 Distortion

To implement distortion in a gradient mesh, it should be parametrized locally, controlling the
distortion. In this implementation, a simple value ranging in $[0, 1]$ is used to control the distortion
amount. This value is interpolated during tessellation. In this research, three types of distortion
have been implemented and evaluated:

\begin{itemize}
  \item Coordinate distortion
  \item Distortion using a sine function
  \item Distortion by multiplying the resulting scalar (Wood distortion)
\end{itemize}

\textbf{Coordinate Distortion}

The noise field is generated based on $(u, v)$ coordinates, ranging in $[0, 1]^2$. These $(u, v)$ coor-
dinates are passed into the noise function, resulting in a scalar. This scalar can also be used to
offset the \((u, v)\) coordinates, causing the \((u, v)\) coordinate map to be non-linear (see Figure 5.3). This scalar however, should not be used to offset both the \(u\) and \(v\) coordinates, since this will result in the noise function to be anisotropic. To avoid this anisotropy, the \((u, v)\) coordinates are offset with two different scalars. To achieve this, the noise function is simply called twice; first using the normal \((u, v)\) coordinates and the second time using the same coordinates, offset in one direction using the first noise scalar.

**Sine Distortion**

Using a sine function to distort a noise map is a common technique, which is also used for the well-known 'Marble vase' by Ken Perlin [4, Figure Marble Vase]. The angle \(\alpha\) of the sine function can be controlled, as well as the frequency \(f\).

Using the \((u, v)\) coordinate system, a point \((x, y)\) can be converted to an angle/phase \(\varphi\).

\[
\varphi = \begin{bmatrix} x \\ y \end{bmatrix} \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \end{bmatrix}
\]

By offsetting this phase using a noise scalar \(N\) and the distortion amount \(d\), the sine function will be distorted.

\[
N = \frac{\sin(2\pi f(\varphi + Nd)) + 1}{2}
\]

**Wood distortion**

By multiplying the noise scalar with the distortion amount \(d\) and taking the decimal part of this result, a wood-like texture can be achieved.

\[
N = \text{frac}(Nd)
\]

### 5.2.3 Filtering

Although noise can be regarded as a power spectrum and filtering is usually performed on a frequency level, for this implementation the filtering is performed directly on the noise field. The reason for this decision is mostly efficiency: a Fourier transform has to be applied on the noise field to filter in the frequency domain. In this implementation, five different filtering functions have been defined (see Figure 5.4, Appendix A.1.2 for implementation).

A key element is that each function is \(C^1\) continuous. This results in the edges on the cutoff value to be smooth. Each function uses one or two control values which define the lower and upper cutoff value.

**Pulse**

The pulse function is a simple function with a low and a high cutoff value. Each value between these cutoff values is mapped to 1. This is particularly useful for creating patterns with thick lines.
CHAPTER 5. IMPLEMENTATION

Figure 5.4: Filtering functions

**High-pass**
The high-pass filter removes the lower values from the noise field, using a high cutoff. This can be used to create small highlights or reflections.

**Low-pass**
The low-pass filter removes the higher values from the noise field, using a low cutoff. With these lower values, a subtle texture can be generated.

**Band-pass**
The band-pass filter combines a high- and a low-pass filter, resulting in a noise field in which both high and low frequencies can be filtered out.

**Normalized Band-pass**
The regular band-pass filter is limited in the sense that the output value is limited by the high cutoff value. By using cubic interpolation between the low and high cutoff value and mapping this to 0 and 1, the whole spectrum of noise values is regained while maintaining the $C^1$ continuity. This is extremely useful for creating sharp and bright edges.
5.2.4 Colour Mapping

The noise scalar is used as a value to interpolate between a primary and a secondary colour. This interpolation is done linearly by

\[ C(t) = (1 - t)C_1 + tC_2 \]

where \( C_1 \) and \( C_2 \) are the primary and secondary colour respectively and \( t \) the noise scalar. This transforms the scalar field to a field of colour vectors.

5.2.5 Colour Blending

The final colour is achieved by combining the noise colour with the colour of the gradient mesh. The blending of two colours can be performed in numerous ways, e.g. averaging between the two, or taking either colour value based on some logic. Using colour blend techniques by Jamie Owen [15], 25 different colour blending modes are established (see Appendix A.2).
During the evaluation, the building blocks are evaluated based on important aspects. One of the goals is to provide local parameters controlling the noise on vertices. To preserve the properties of a noise function as described in Chapter 1, having different parameter values at the vertices should still result in a smooth noise field. Another important aspect is usability, meaning that controlling a parameter should feel natural and each parameter should have a noticeable impact on the final result. To achieve this usefulness, each building block consists of a minimal number of parameters and each parameter is evaluated based on impact.

### 6.1 Methods

It is difficult to measure the quality of the solution for reasons that it consists of multiple sub-solutions and that there exists no clear definition of what is considered good and what is considered better. Taking into account the entire pipeline, there exist 13 different noise base functions (see Section 5.2.1), three distortion functions (see Section 5.2.2) and five different filter functions (see Section 5.2.3), resulting in 195 different combinations of functions. Therefore, to evaluate the results, the building blocks are evaluated separately by using a single patch of Perlin noise.

Distortion and local frequency are both mapped to a single parameter and can therefore be evaluated in the same manner: Each vertex on the patch is assigned a different parameter value; the top right vertex is set to 1.0, top left to 0.5, bottom left to 0.0 and bottom right to 0.25. This produces a field in such a way that both low and high values can be evaluated as well as different slopes of interpolation. This patch is compared to a basic patch on which all parameters are set to 0 (the zero patch).
6.2 Results

In this section, the results from the noise functions, building blocks and combinations are described and evaluated.

6.2.1 Noise Functions

Each noise function is evaluated based on the effect of local frequency.

6.2.1.1 Perlin Noise

The basic Perlin noise function creates a field of distinguishable noise structures. By increasing the local frequency, more bands become visible, creating a smooth transition between low and high frequency areas (see Figure 6.1).

The structures of the base octave remain visible in higher frequency areas.

6.2.1.2 Worley Noise

The Worley noise shader implementation is slightly different than the original Worley noise implementation [5], yet the results are sufficiently similar.

The Worley noise function itself is already extremely versatile: By varying $F_n$, various patterns can be achieved, although $F_3$ and up result in mainly high values (See Figure 6.2). By varying the jitter parameter or the distance metric, an even larger pattern variation is possible (See Figure 6.3).

The fractal version of Worley noise creates a similar effect as with Perlin noise: The base noise structures remain visible on which layers of higher frequency noise are created.
Figure 6.2: Worley noise functions
6.2.2 Distortion

To draw conclusions on the effect of a distortion function, the original noise field is compared to the distorted noise field, of which the absolute difference is taken.
CHAPTER 6. EVALUATION

Coordinate distortion
Coordinate distortion (see Figure 6.5) causes the original noise structures to be distorted, creating a ripple effect. This ripple effect occurs in both high and low value noise structures, destroying the underlying noise values. Comparing the difference map to the original noise field, it can clearly be seen that the biggest differences often occur at the dark parts of the noise field. At high parameter values, the ripples become of an extremely high frequency, potentially creating aliasing artifacts (See Figure 7.2). Interpolating the distortion amount causes no irregularities in the resulting noise field. The distorted noise remains continuous. Low parameter values barely distort the original noise field, whereas increasingly higher parameter values clearly gradually distort the noise field more. Since the origins of the ripple are centered near the center of the noise structures, the effect of this distortion is heavily affected by the frequency of the noise.

Sine distortion
By enabling Sine distortion (see Figure 6.6), the noise field is replaced by the sine. This however does mean that when the distortion amount is set to zero, there is no noise visible. This could be tackled by defining a second local parameter to control how much of the distorted sine field is mixed with the original noise function, which could be impractical. The sine function produces a field that periodically changes from 0 to 1, which remains present at lower parameter values. At higher values, the shape of the original noise field becomes visible again while the periodic sine structure remains. Looking at the difference map, it can clearly be seen that the difference appears to be equal over the entire patch. For higher values, the periods of the sine function become more dense, generating thinner lines. Overall, the resulting noise field remains smooth.

Wood distortion
Wood distortion (see Figure 6.7) produces a result, comparable to coordinate distortion. Since the decimal part of integers larger than one is taken, the result is a sharp transition between extreme values. Similar to the coordinate distortion, the denser these lines are, the higher chance of aliasing artifacts. Even more so regarding the fact that on these lines, values switch straight from 1 to 0. This also means that the resulting noise field is no longer continuous. Parameter values have a similar effect on wood distortion as coordinate distortion: the lines become more dense the higher the parameter values.

6.2.3 Filtering
Filtering functions (see Figure 6.8) provide a useful method to filter out unwanted noise values.
6.2.4 Examples

By applying combinations of the functions on a gradient mesh, interesting results can be achieved which normally would require the mesh to be extremely complex (see Figure 6.10). One example, using a mesh of a vase constructed by only 12 patches (see Figure 6.9), is split up into the mesh itself, the unprocessed noise, processed noise and the combination of both, which displays the simplicity of the mesh, the high level of detail added by the noise function and the contribution of the distortion and filtering functions. It can clearly be seen how the noise field is attached to the gradient mesh structure as it follows the curves of the patches.

6.3 Discussion

Frequency is achieved using the concept of fractal noise and an exponential function that defines the impact of each octave. By providing more control over the noise spectrum, a larger set of base noise functions can be achieved. However, this is difficult to realise in a gradient mesh by simply using a single scalar.

Local parameters are limited to frequency, opacity and distortion amount. More parameters could be extracted to local parameters, creating more options for local refinement, but increasing the complexity of manipulating the mesh.

The non-continuity of wood distortion could be resolved by applying a Gaussian function on the distortion, blurring the sharp transitions, although the sharp edges might be considered to be a desired property of the distortion. This last argument applies to many cases; What one may desire, might be completely unwished for another. Obviously a line has to be drawn somewhere to define what is considered to be good and what is completely wrong.
CHAPTER 6. EVALUATION

(a) Basic Perlin noise
(b) Sine

Figure 6.4: Basic Perlin noise & Sine

(a) Coordinate Distortion
(b) Difference map

Figure 6.5: Coordinate distortion of Perlin noise & Difference

(a) Sine Distortion
(b) Difference

Figure 6.6: Sine distortion of Perlin noise & Difference
Figure 6.7: Wood distortion of Perlin noise & Difference
(a) Basic Perlin noise  
(b) Pulse filter  
(c) Low-pass filter  
(d) High-pass filter  
(e) Band-pass filter  
(f) Normalized Band-pass filter  

Figure 6.8: Filter functions applied to Perlin noise.
(a) Colour

(b) Unprocessed noise

(c) Processed noise

(d) Colour + noise

Figure 6.9: Vase mesh (12 patches)
CHAPTER 6. EVALUATION

(a) Butterfly (6 patches)

(b) Banana (6 patches)

(c) Watermelon (4 patches)

(d) Watermelon (perturbed) (4 patches)

(e) Strawberry (4 patches)

Figure 6.10: Other examples
Conclusion and Future Work

This thesis is concluded by listing the contributions of this project and by drawing conclusions based on the evaluation. Finally, some future ideas are proposed.

7.1 Contributions of this Thesis

The main contribution of this project is the extension of the gradient mesh tool.

7.1.1 Tool

The existing gradient mesh framework has been extended such that it supports the following features:

- Continuity while modifying tangents ($C^1$ & $G^1$)
- Parameter values can be assigned to halfedges, creating sharp-edge transitions (Figure 5.2)
- Loading & saving the mesh

During this project, a method has been developed to combine the existing gradient mesh with a noise function. This noise function is strictly bound to the gradient mesh and can be modified using distortion and filter functions.

The following features have been added to the gradient mesh tool for this purpose (see Figure 7.1):

- Noise type selection (Perlin, Worley & variations)
- Base noise frequency input
- Maximum number of octaves
CHAPTER 7. CONCLUSION AND FUTURE WORK

7.1 GUI

- RGB mix of the noise field
- Three noise blending modes (Colour, Noise, Colour + noise)
- 26 colour blend modes [15]
- Three distortion functions (Basic, Sine, Wood)
- Five filter functions (Pulse, Low-pass, High-pass, Band-pass, Normalized Band-pass)
- Primary & secondary colour mapping
- Vertex & halfedge controls for frequency, opacity & distortion amount

7.2 Limitations

One of the limitations of this tool is that it is not dynamic because of its fixed pipeline structure. This limits the user from trying out different combinations of functions. By creating presets of
function combinations, this problem can be partially avoided.
A defect that appeared during the evaluation of this tool is that due to high frequency noise, aliasing effects can occur (see Figure 7.2). Normally, by multi-sampling [16] anti-aliasing can be achieved. Unfortunately, 64 samples is not always sufficient for extremely detailed noise.

7.3 Conclusion

The implementation can be considered to be successful, despite the forementioned limitations; Two noise functions have been combined with the gradient mesh in such a way that limitations presented by previous work [7] have been resolved. The tool offers a large variety of function combinations which allow for versatile texture generation. By controlling the noise function locally, details can be added by means of noise parameters instead of gradient mesh primitives, thus keeping the mesh structure simple. To keep the design process simple, each function block is restrained to a minimal amount of parameters.

Distortion
All three distortion functions result in a dissimilar distortion pattern, each with different use cases. With a single local parameter, both Wood and Coordinate distortion can be manipulated in a desirable manner. Sine distortion would require another parameter to be effective in the same manner. Coordinate distortion is both smooth and parametrized by a single value. Sine distortion is smooth, but requires at least one more local parameter to be effective. Wood distortion is parametrized by a single value, but is non-continuous. The difference map used to evaluate the distortion also creates an interesting pattern.
CHAPTER 7. CONCLUSION AND FUTURE WORK

Filtering
The addition of filtering functions allows the user to filter out unwanted features. All filters have different purposes and by providing the user with 5 filtering functions, a lot of control is given to the user.

Interesting results have been generated using the tool, which clearly show that the combination of a gradient mesh with noise functions allows for creating a high level of detail on a simple mesh structure.

7.4 Future Work

During the research, a lot of ideas came to mind, but due to time limitations and other limitations like language constraints, these ideas have not been implemented or researched. Considering procedural texture generation using noise functions is endless, this section will be limited to suggestions that drastically change the current implementation or that are worth researching.

A key concept which would improve the customisability by a lot would be composition. This results in, instead of having a fixed building block pipeline, each function (noise, distortion, filtering) is considered as a separate building block with possibly multiple inputs and outputs which can be controlled by the user. Basically, using this method, two types of distortion building blocks can be removed, since they consist of multiple functions. Combined with composition, another feature to improve customisability would be layering. Combining multiple layers of noise would allow the user to create multiple noise patterns within the same mesh. This would be similar to Adobe’s layering functionality. This introduces problems as to how these layers should be mixed. Again, this could be solved the same way Photoshop deals with this; by using layer priorities and alpha blending.

The current tool utilizes the noise field in the colour domain. Besides this, common techniques are to use the noise scalar for a bump map, normal map or an ambient occlusion map. These techniques improve the realism of the resulting image.

The current tool can only be used for texture generation, meaning that every detail has to be created from scratch. An extremely interesting concept, used by Jeschke et al. [7] to estimate noise parameters and generally known as noise by example. This concept is mainly used in combination with vectorizing raster images. Although this concept is identical to Jeschke et al., this thesis proposes improvements such that limitations mentioned in Chapter 2 are reduced or even solved.
A.1 Functions

A.1.1 OpenGL Functions [17]

```c
genType smoothstep(float edge0, float edge1, genType x){
    genType t; /* Or genDType t; */
    t = clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);
    return t * t * (3.0 - 2.0 * t);
}

genType clamp(genType x, genType minVal, genType maxVal){
    return min(max(x, minVal), maxVal);
}

genType fract(genType x) {
    return x - floor(x);
}
```

A.1.2 Filter Functions

```c
float pulse(float lowCut, float hCut, float x){
    return smoothstep(lowCut-0.1, lowCut, x) -
           smoothstep(hCut, hCut+0.1, x);
}

float lowpass(float cutoff, float x){
    return x * (1 - smoothstep(cutoff, cutoff+0.1, x));
}
```
APPENDIX A. APPENDIX

```c
float highpass(float cutoff, float x) {
    return x * smoothstep(cutoff-0.1, cutoff, x));
}

float bandpass(float lowCut, float hCut, float x) {
    return x * pulse(lowCut, hCut, x);
}

float bandpassNormal(float lowCut, float hCut, float x) {
    return smoothstep(lowCut, hCut, x) - smoothstep(hCut, hCut + 0.1, x);
}
```

A.2 Colour Blending Modes

- Add
- Average
- Color Burn
- Color Dodge
- Darken
- Difference
- Exclusion
- Glow
- Hard Light
- Hard Mix
- Lighten
- Linear Burn
- Linear Dodge
- Linear Light
- Multiply
- Negation
- Normal
- Overlay
- Phoenix
- Pin Light
- Reflect
- Screen
- Soft Light
- Subtract
- Vivid Light
Bibliography


