Texture Synthesis in a Weighted Graph Using Nth-Order Voronoi Diagrams

by

Lester Mundt

A thesis submitted to the
Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science
The School of Computer Science
Carleton University
Ottawa, Ontario
April, 2019

©Copyright
Lester Mundt, 2019

The undersigned hereby recommends to the
Faculty of Graduate and Postdoctoral Affairs
acceptance of the thesis

**Texture Synthesis in a Weighted Graph Using Nth-Order
Voronoi Diagrams**

submitted by **Lester Mundt**

in partial fulfillment of the requirements for the degree of

Master of Computer Science

Professor David Mould, Thesis Supervisor

Professor Douglas Howe, Director,
School of Computer Science

Ottawa-Carleton Institute for Computer Science
The School of Computer Science
Carleton University
April, 2019

Abstract

We present a method to procedurally generate textures at variable resolutions using weighted graphs. Our method takes a number of input points that exist within a grid-like graph. Using the input points as site nodes the algorithm then assigns all the nodes in the graph to Voronoi regions by using geodesic rather than Euclidian distance. The edge weights are calculated by comparing properties of the target nodes with the site nodes of the respective regions. We have generated a number of variations of our basic algorithm that use this same mechanism to generate a range of textures.

Our method allows us to chain multiple algorithms. The output of one algorithm to be the input for a successive algorithm variation and successive outputs can be combined in a variety of ways.

Our contributions are as follows: We have demonstrated an algorithm using a weighted graph for higher order Voronoi cell determination. We present examples of properties that can be used for the weighting calculations providing varying patterns. We have produced a framework that allows us to combine successive runs of the algorithm using the output of the previous algorithm as an input to its successor. We have shown how we can use alternatives to single points as originating site for use in generating the higher order Voronoi diagrams. Additionally we have illustrated that our technique and accompanying framework can be used to synthesize textures that have real world analogues such as cracks in stone or other minerals.

Acknowledgments

First I would like to acknowledge my good friend and academic advisor Dr. David Mould. Without David's steadfast support and vast stores of knowledge this Thesis would never have come to completion.

Secondly I would like to thank my wife Sujin Park for her unwavering faith that I might, in fact, one day complete this Thesis.

Finally I would like to thank my dear friends and family who despite expressing agreement that I would indeed finish this Thesis, all had grave misgivings. Despite which they graciously accepted my missing of trips and other adventures with the simple explanation "I need to work on my Thesis".

Table of Contents

Abstract	iii
Acknowledgments	iv
Table of Contents	v
List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Problem Statement	2
1.2 Contributions	3
1.3 Suggestion to the Reader	4
1.4 Thesis Outline	4
2 Background	6
2.1 Previous work	6
2.1.1 Texture Description	6
2.1.2 Foundations	7
2.1.3 Texture Synthesis Overview	9
2.1.4 Geodesic Based Methods	19
3 Algorithm	22
3.1 Basic Algorithm	23
3.1.1 Graph Assembly	23
3.1.2 Site Assignment	24
3.1.3 Voronoi Diagram Generation	24

3.1.4	Normalization	30
3.1.5	Rendering	33
3.2	Framework	36
3.2.1	Combining	37
3.3	Variations	40
3.3.1	Site Weights	40
3.3.2	Edge Weight Calculation	42
3.3.3	Accidental Edge Weight Calculation	45
3.3.4	Site Locations	48
3.3.5	Stopping Conditions	49
3.3.6	Rendering	51
4	Results and Discussion	52
4.1	Weight Calculation Variations Results	52
4.1.1	Basic Weight Calculation Results	52
4.1.2	Guided Vector Weight Calculation Results	56
4.1.3	Accidental Weight Calculation Results	60
4.2	Combination Results	64
4.2.1	Brown Rocks	65
4.2.2	Dry Riverbed	66
4.2.3	Sparse	67
4.2.4	Sharp	68
4.2.5	Variety	70
4.3	Selective Site Placement Results	72
4.4	Naturalistic Results	77
4.5	Tiling and alternative texture application	81
4.6	Alternative Colourization	82
4.7	User Guide	83
4.8	Limitations	83
5	Conclusions	85
5.1	Summary	85
5.2	Future Work	86
	List of References	87

List of Tables

4.1	Runtimes of the algorithm in milliseconds	56
4.2	Variations, parameter and combination operations used to generate Brown Rocks	66
4.3	Variations, parameter and combination operations used to generate Dry Riverbed	67
4.4	Variations, parameter and combination operations used to generate Sparse	69
4.5	Variations, parameter and combination operations used to generate Sharp	70
4.6	Variations, parameter and combination operations used to generate Variety	72
4.7	Variations, parameters and combination operations used to generate Mahogany Obsidian	78
4.8	Variations, parameter and combination operations used to generate Orbicular Jasper	79
4.9	Variations, parameter and combination operations used to generate Rhyolite	80

List of Figures

1.1	The image on the left shows an non-textured scene, same image on the right shows a textured scene. [38] The difference in detail is significant.	1
1.2	Left: Euclidian 3rd order Voronoi diagram, Right: Example of our method	2
1.3	Left: Natural Mineral, Right: Approximation produced by our method	3
2.1	Voronoi diagram defined in R^2 on the Euclidean plane (left) and using the Manhattan distance (right).	8
2.2	Higher Order Voronoi diagram defined in R^2 starting on the left(top row) 1st, 2nd, 3rd, continuing from the left (bottom row) 4th and 5th orders	10
2.3	1. Input Image, 2. Input Mask, 3. Visualized GGDT; Image from Criminisi et al. [9]	21
3.1	1st and 5th order Voronoi diagrams	23
3.2	Our algorithm represented in a pipeline	24
3.3	Left: Random distribution, Right: Poisson distribution [1]	25
3.4	Left: The weight calculation from equation 3.3 Right: The weight calculation from equation 3.4	26
3.5	Simple graph depicting region growth	31
3.6	Left: 5th order Voronoi diagram pre-collapse; Right 5th order Voronoi diagram post collapse	32
3.7	Using existing path cost; left: first order, right: third order using k-tuples	34
3.8	Left: $F2 - F1$, Centre: $F2 + F1$, Right: $2F3 - F2 + F1$	35
3.9	Left: $F2 - F1$, Centre: $F2 + F1$, Right: $2F3 - F2 + F1$	35
3.10	Two type of normalization; left: global normalization, right: regional normalization	36
3.11	Left: global normalization, Right: regional normalization	37

3.12	Framework pipeline illustrating two algorithms	38
3.13	Each line: input graph, output graph, combination result; Top to bottom; Influence, Replacement, Additive, Subtractive	41
3.14	Two examples of the weighted site algorithm variation, left: $\zeta = 0.2$, $\eta = 0.1$; <i>right</i> : $\zeta = 0.2$, $\eta = 0.5$	43
3.15	Two examples of vector directed Voronoi diagrams	44
3.16	Two products of our accidental edge weight calculation	46
3.17	Left: mistakes uncorrected, Centre: mistake one corrected, Right: mistake two corrected	47
3.18	Left to right: Regions, edges, and confluences	48
3.19	Left: input graph, Middle: graph with selection sites, Right: output graph	49
3.20	Two examples of the region limited algorithm variation, the left restricts regions to 300 nodes and the left restricts regions to 900 nodes	50
3.21	Left input graph, Right confined subdivision	51
3.22	Voronoi diagram with different colour lookup tables	51
4.1	Left to right: $n = 1$, $n = 2$, $n = 3$, $n = 4$, $n = 5$ All: $s = 20$, $\gamma = 0.5$	53
4.2	Left to right: $s = 20$, $s = 100$, $s = 500$ All: $n = 1$, $\gamma = 0.5$	53
4.3	Left to right: $\gamma = 0.0$, $\gamma = 0.5$, $\gamma = 0.9$ All: $s = 20$, $n = 1$	54
4.4	Higher Order Voronoi Diagrams, left to right: First Order: $s = 50$, $\gamma = 0.25$; Second Order: $s = 91$, $\gamma = 0.06$ Third Order: $s = 42$, $\gamma = 0.2$	55
4.5	Left to right, : $i = 5.0$, $i = 15.0$, $i = 50$	56
4.6	Left to right, : $m = 0.1$, $m = 0.4$, $m = 0.8$, $m = 2.0$	57
4.7	Left to right, : $p = 1$, $p = 2$, $p = 7$, $m = 15$	57
4.8	Left to right, : $\gamma = 0.005$, $\gamma = 0.050$, $\gamma = 0.5$	58
4.9	Left to right: stratified, radial, central, All: $s = 80$, $p = 1.0$, $\gamma = 0.5$, $i = 0$, $\beta = 0$	59
4.10	Left to right: $p = 1.0$, $p = 7.0$, $p = 15.0$, All: $s = 80$, $p = 1.0$, $\gamma = 0.5$	59
4.11	Three interesting outputs of our guided vector weight calculation	60
4.12	Left to right, top to bottom: $\mu = 0.3$, $\mu = 0.4$, $\mu = 0.5$, $\mu = 0.6$, $\mu = 0.7$, $\mu = 0.8$, $\mu = 0.9$, $\mu = 1.0$	61
4.13	Left to right, top to bottom: $\gamma = 0.0$, $\gamma = 0.1$, $\gamma = 0.3$, $\gamma = 0.9$, $\gamma = 1.9$, $\gamma = 2.5$, $\gamma = 4.0$, $\gamma = 5.0$	62

4.14	Two textures reminiscent of marble	62
4.15	Two textures suggestive of frost.	63
4.16	Another example of frost: $s = 90$, $\mu = 0.437$, $\gamma = 0.525$	63
4.17	Three diverse image produced using the Accidental weight calculation	64
4.18	Variation 1: $s = 76$, $p = 7.0$, $m = 0.8$, $\gamma = 0.005$, $i = 5.0$; Variation 2: $s = 67$, $\gamma = 0.1$, max region size = 426	65
4.19	Variation 1: $s = 84$, $o = 2$, $\gamma = 0.175$; Variation 2: $s = 80$, $\gamma = 0.175$, max region size = 409; Variation 3: $s = 90$, $\gamma = 0.175$, max region size = 1640	67
4.20	Variation 1: $s = 60$, $\gamma = 0.413$, $o = 5$; Variation 2: $s = 31$, $\gamma = 0.005$, $\beta = 0.01$, $i = 15.0$, $p = 7.0$, $m = 0.8$	68
4.21	Variation 1: $s = 70$, $\gamma = 0.731$, $o = 3$; Variation 2: $s = 88$, $\gamma = 0.347$, $o = 1$	70
4.22	Variation 1: $s = 51$, $\gamma = 0.005$, $p = 7$, $i = 15$, $m = 0.8$ $\epsilon = 0.2$; Variation 2: $s = 70$, $\gamma = 0.005$, $p = 7$, $i = 15$, $m = 0.8$ $\epsilon = 0.2$; Variation 3: s $= 68$, $\gamma = 0.5$, $p = 15$, $\epsilon = 0.2$, ; Variation 4: $s = 28$, $\gamma = 0.2$, max region size = 3526	71
4.23	Both: Two paths selected as sites and widened into cracks	73
4.24	Left: Two regions selected to be seeded with sites, Right: two edges selected to be sites	74
4.25	Left: Input Voronoi diagram, Right: Region confined output	75
4.26	Left: Unconfined 1st Order Voronoi diagram, Middle: Unconfined 2nd Order Voronoi diagram Right: Unconfined 3rd Order Voronoi	75
4.27	Left: Input regions, Middle: Confluence based regions, Right: Final output	76
4.28	Left: Input regions, Middle: Confluence based regions, Right: Final output	76
4.29	Left: Confluence based third-order Voronoi diagram, Middle: Third- order Voronoi diagram, Right: Fifth-order Voronoi diagram	77
4.30	Left: Mahogany Obsidian, Right: Texture approximating Mahogany Obsidian	78
4.31	Left: Orbicular Jasper, Right: Texture approximating Orbicular Jasper	79
4.32	Left: Rhyolite, Right: Texture approximating Rhyolite	80
4.33	Tileable Voronoi diagram	81

4.34	Three multi-texture outputs	82
4.35	Example of high frequency mineral structure: white granite	84

Chapter 1

Introduction

Computer scientists have been interested in increasing of the details of their renderings throughout the history of computer graphics. Textures have long been used within computer graphics as a way to add visual detail to their synthetic creations. This detail is added without additional geometry or other information. Relying solely on geometry as a way to add detail would quickly prove computationally prohibitive. Textures can be thought of as a way of mapping each point of a surface to a pixel within a given texture. In practice, this provides for vastly superior visual interest in the final results.

Figure 1.1: The image on the left shows an non-textured scene, same image on the right shows a textured scene. [38] The difference in detail is significant.

Textures can be described with terms such as stationary, isotropic, regular, periodic. For our thesis the most important term is stationary; a stationary texture has similar characteristics throughout its entirety so that any two subsections of the texture would be close to indistinguishable.

Textures are either generated by artists or using algorithms. Artist's methods are painstakingly manual while generating textures with algorithms is a much quicker process. The process of generating them with an algorithm is commonly referred to

as texture synthesis. Texture synthesis is generally divided into two broad categories: example-based, and procedural. While example-based methods produce excellent results they are not capable of generating non-stationary textures and wouldn't be suitable for generating the non-stationary textures we intended to produce; so we looked to existing procedural methods as our starting point.

Figure 1.2: Left: Euclidian 3rd order Voronoi diagram, Right: Example of our method

This thesis is focused on presenting our new procedural texture synthesis algorithm. Our technique expands directly on the papers by Worley [48] and Mould [28], who had both demonstrated the utility of Voronoi diagrams for texture synthesis, so developing a method using Voronoi diagrams was a natural starting point for our research. Mould [28], Xu et al. [50], and Criminis et al. [9] demonstrated the flexibility of using a graph as an underlying data structure and how it allowed for manipulation of the geodesic distance. This manipulation would allow us to alter the characteristic cell-like structure of the Voronoi diagram into alternative structures improving the range of textures that could be synthesized.

1.1 Problem Statement

We were motivated to find a method that would allow us to generate realistic rock textures. We wanted to generate textures that would exhibit strong non-stationary

features. Imagine a cliff face where among the cracks of homogeneous stone are veins of a different mineral. In addition, we recognized that work has been done in cracking stone but we wanted to represent stone composed of a variety of minerals. The minerals are present in distinct areas within the primary material and are hallmarked with irregular boundaries and distinct colours. We wanted the user of our algorithm to be able to generate these textures with just the algorithm by manipulating the parameters provided.

Figure 1.3: Left: Natural Mineral, Right: Approximation produced by our method

1.2 Contributions

Our contributions can be summarized as follows:

- combining geodesic distances with higher-order Voronoi diagrams.
- an innovative weight calculation using values from sites and target nodes.
- an algorithm that allows us to generate non-stationary textures. The non-stationary features that our method provides include:
 - cracks spanning the entirety of the texture.
 - localized high density of regions.
- a framework that allows us to combine the output of successive variation of the algorithm.

We believe that our algorithm is the first texture synthesis approach to combined geodesic distances for use in generating higher-order Voronoi diagrams. This allowed us to distort the shape of the Voronoi regions allowing us to increase the variety of textures we can generate. This also allowed us to make the region boundaries more irregular making them look more like cracks in stone.

Our weight calculation allows values for the sites to form regions that have that are directly influenced by the sites. One example, is that we used this weight calculation to guide the shape and direction of the individual Voronoi regions.

To achieve our non-stationary features we use a weighted graph for Nth order Voronoi region determination. We present examples of properties that can be used for the weight calculations allowing us to alter the shape of the generated Voronoi regions. We are also able to create Voronoi regions that are long and thin or sharp and angular quite unlike Euclidian based Voronoi diagrams. We have also shown Voronoi regions with vastly varying sizes in a manner uncharacteristic to Voronoi diagrams with very small regions completely encompassed by much larger regions.

The framework we have produced also allows us to combine the output of successive variation of the algorithm and use the output of the previous algorithm as an input to its successor. This allows us to coarsely select subsequent site placement and allows us to generate some of the non-stationary features. We have also shown how we can use alternatives to a single point as an originating site for use in generating Voronoi diagrams.

Finally, we have illustrated that our algorithm and accompanying framework can be used to synthesize textures that have real-world analogues such as the cracks in stone or a variety of minerals.

1.3 Suggestion to the Reader

The images in this thesis are best viewed digitally as there can be a loss of detail when printed.

1.4 Thesis Outline

The rest of this thesis is divided into the following chapters. The second chapter provides some background in the form of related work and basic concepts. The third

chapter will describe the algorithm and how it works, this will, in turn, be followed by a section attempting to justify design decisions made during the algorithm's development. The fourth chapter provides the results of the algorithm and contrasts them with the papers predecessors. Finally the fifth and final chapter we present our conclusions and proposals for future work.

Chapter 2

Background

2.1 Previous work

In this chapter, we provide background for the area of computer science referred to as texture synthesis. Broadly interpreted texture synthesis encompasses methods for generating textures. We will focus on two groups: example-based methods and procedural methods. In this chapter, we start by presenting a brief taxonomy for texture description. This is followed by some information on a mathematical construct called a Voronoi diagram which is occasionally leveraged within texture synthesis and is central to our research in particular. Finally, we provide a thorough background on texture synthesis describing a number of research papers. This background information is intended to aid in understanding the rest of the thesis and to provide insight into the history of this area of computer science.

2.1.1 Texture Description

To aid in the understanding of the merits of different approaches to texture generation it is best to establish the dimensions in which textures can be described. Liu et al. [27] defined the term *near-regular* textures to describe textures that exhibit a strong tendency towards visual regularity and symmetry but, imperfectly presented. As an example one can visualize a brick wall with the spacing between the bricks not quite uniform and the bricks not all identical.

A number of papers [8,16,32,37] use spectral analysis to help characterize textures. The frequency domain is used to describe texture in the terms of phase and magnitude across ranges of frequencies.

One of the more complete glossaries of texture characteristics is provided by Choy [7]. This glossary provides dimensions with which one can describe a texture. These dimensions are as follows:

Stationarity - a texture is stationary if the statistical properties of the texture do not vary regardless of translation or scaling. That is, given any subset of a texture of a particular size, the subset would seem to come from the same underlying process. Repeating dot patterns where the dots are uniformly distributed and identically sized would be considered stationary patterns. Conversely, a pattern consisting of dots increasing in size as they progress in a single direction would be considered non-stationary.

Isotropy - a texture is isotropic if rotating the texture would not change the statistical qualities of the texture. Many stripe patterns are not isotropic but are anisotropic because all stripes would follow a particular direction in the texture. In an example of an anisotropic texture, one could envision the stripes of a zebra which run vertically along the zebra's torso.

Periodicity - a texture is periodic if there is a detectable pattern which reoccurs at a regular frequency. This frequency or interval is referred to as the period. In an example a checkerboard pattern is an example of a pattern which is periodic. When trying to generate natural analogues periodicity is considered an undesirable trait in texture synthesis.

Regularity - intimately related to the concept of periodicity, regularity refers to the perceived randomness of a texture. Textures that one might describe as random or chaotic and thus exhibiting little or no regularity are referred to as Stochastic. Stochastic textures, unlike their regular counterparts, exhibit little to no symmetry or periodicity. Near regular textures would fall between stochastic and regular.

2.1.2 Foundations

We worked to expand Worley's [48] work on cellular basis function for texture synthesis. As both Worley's and our own work leverages Voronoi diagrams we provide some background on this flexible mathematical structure.

A Voronoi diagram is a structure that divides the points of R^n into regions where all the points with a given region are closest to the same site. The aforementioned site is one of an arbitrary set of points within R^n . To continue with a formal definition in R^2 a Voronoi diagram defined as V and given a set of sites $S = \{s_1, s_2 \dots s_n\}$ is the union of all the regions V_i as defined by the following equations:

$$\begin{aligned} V_i &= \{x \in R^2 : \|s_i - x\| \leq \|s_j - x\|, \forall \{j\} \neq \{i\}\} \\ V &= \{V_0, \dots, V_i, \dots V_n\} \end{aligned} \quad (2.1)$$

The above equation (2.1) allows us to subdivide a plane into regions where the points in the regions are closer to the site that defines the region than any other site.

Voronoi diagrams can be calculated outside of Euclidean space. A simple alternative distance calculation is the Manhattan distance which rather than calculating the distance between two points as a straight line calculates the distance by adding the components. This alternative distance calculation causes the regions of the Voronoi diagram to look more square.

Figure 2.1: Voronoi diagram defined in R^2 on the Euclidean plane (left) and using the Manhattan distance (right).

As can be seen in Figure (2.1) the change in how the distance is calculated can have a profound visual change on the resulting Voronoi diagram.

Higher Order Voronoi Diagrams

Higher order Voronoi diagrams divide a space into regions but rather than all points within the regions being closest to a single originating site the regions are composed of points that are closest to unordered tuples of sites. Thus, for a second order Voronoi diagram, the points in a given region will be closest to pairs of sites; in a third order the points will be closest to triples of sites; and so on. This can be seen formalized in equation 2.2 originally defined by Fischer et al. [14]; given a set of points $S = \{s_1, s_2, \dots, s_n\}$ where $n \geq k$ and $x \in R^2$ then V is the k -order Voronoi diagram represented by the union of the regions defined as follows:

$$V(i_1, i_2, \dots, i_k) = \{x : \|s_{i_1} - x\| + \dots + \|s_{i_k} - x\| \leq \|s_{j_1} - x\| + \dots + \|s_{j_k} - x\|\},$$

$$\forall \{i_1, i_2, \dots, i_k\} \neq \{j_1, j_2, \dots, j_k\}$$
(2.2)

In Figure 2.2 we can see an example of a graph with 15 sites. The order increases from left to right and top to bottom and we can see the increasing fragmentation of the regions. Additionally, we can observe an increasing variety of region size and shape, but it should be noted at orders above the fifth the Voronoi diagrams begin to resemble each other.

In addition to the distance calculations and using higher order Voronoi diagrams there are a variety of ways to alter the generation of Voronoi diagrams [29]. Instead of single point sites being used to determine the Voronoi regions, lines or polygons could be used. Another alternative would be instead of the sites growing equally the sites could be weighted. We use a variety of these techniques as well as some others of our own and explore each in turn later in this thesis.

2.1.3 Texture Synthesis Overview

Texture synthesis can be divided into two broad categories: example-based, and procedural. Further, in this chapter, we will present a cross-section of example-based methods presented in chronological order. Finally, we will present another cross-section of the research in procedural algorithms for texture synthesis.

Figure 2.2: Higher Order Voronoi diagram defined in R^2 starting on the left (top row) 1st, 2nd, 3rd, continuing from the left (bottom row) 4th and 5th orders

Example-Based Texture Synthesis

Example based methods use an input texture to copy from or perform analysis on to produce a new texture with similar characteristics to the first. Example or exemplar-based techniques [2, 10–12, 16, 18, 26, 27, 33, 36, 39, 45] have one key feature in common; they all require a texture as an input, commonly referred to as the exemplar. These algorithms are intended to produce an arbitrarily sized output with similar characteristics as the exemplar while avoiding undue repetition and other undesirable artifacts. These algorithms can be divided into four subgroups; pixel-based, patch-based, statistical and machine-learning. Pixel-based methods grow the image one pixel at a time. Patch-based methods grow the output texture by generating patches from the exemplar with special care along the border allowing them to be placed together like tiles but without visual artifacts highlighting this process. Statistical methods which perform a statistical analysis of the texture and produce a texture with statistically similar properties. Finally, machine-learning methods are presented. We discuss each in turn.

Pixel-based Methods

Pixel-based methods work using an exemplar and growing it one pixel at a time. Early work with pixel-based methods were only capable of synthesizing stochastic stationary textures and tended to be slow like most example-based methods.

Efros and Leung’s paper *Texture Synthesis by Non-parametric Sampling* [12] was the progenitor of pixel-based texture synthesis. The method uses a Markov random field model to synthesize a texture one pixel at a time and attempts to improve the results of statistical synthesis methods like De Bonet’s [10]. It works by seeding the new texture with a small patch from the exemplar. The method then grows the patch a single pixel at a time; the pixel is determined using a neighbourhood search process. The method is also notable for ease of use; it has a single parameter, the neighbourhood size. The size needs to be large enough to capture features that were desired for the output. The results can be quite good but sometimes produce degenerate output consisting of direct copies or garbage from getting stuck in a few pixels.

Wei and Levoy [45] worked on improving the speed of Efros and Leung. Wei and Levoy used a tree search with a fixed size neighbourhood and grew the texture in scanline order. The authors found the fixed size allowed for an increase in speed but showed a tendency to introduce noise. The same paper introduced synthesizing temporal textures which had an additional dimension and the slices of it could be replayed like the frames of an animation. Ashikhmin [2] built on Wei and Levoy’s work that improved on preserving structure in textures. Unfortunately, the results tended to be more blurry than in previous work. Tong et al. [40] leveraged the work of Wei and Levoy but added k-coherence as a technique to preserve structure. Sabha et al. [36] built upon previous work trying to extend the variety of texture that would synthesize without artifacts by developing a technique that reduces the incidence of high-frequency discontinuities.

We now present the next major classification of example-based methods: patch-based.

Patch-based Methods

Patch-based methods operate in a similar manner to pixel-based methods but rather than adding a pixel at a time to the synthesized texture, the methods add entire

patches. This naturally allows for speed improvements and allows for the more successful synthesis of near-regular textures. One of the primary challenges of this area of research is how to avoid discontinuities between the patches. We present a brief history of this area of research.

Praun et al. presented *Lapped Textures* [33], perhaps the earliest paper using a patch-based texture synthesis. Praun et al.’s approach to avoiding visual discontinuities was twofold; the first the title implies was to allow the patches to overlap and the second was to irregularly shape the patches. Praun et al.’s method also had the ability to synthesize textures directly onto a model’s surface. Praun et al.’s addition of localized vectors allowed the adjustment of orientation of the growing patches providing signification variation with anisotropic exemplars. Liang et al. [26] avoids visual discontinuities by blending the edges of the patches. Efros and Freeman [12] produced another patch-based method that like Praun et al.’s [33] allowed the textures to overlap. Efros and Freeman generated a number of square patches from the original exemplar. Squares are selected randomly into the new texture where the overlap is adjusted to minimize discontinuities. Finally, the edge of the new square is made irregular by traversing it as a least cost path removing the pixels exterior to the path. Kwatra et al. [22] used an algorithm similar to Efros and Freeman but added irregularities to the square using a graph cut method. Wu and Yu [49] augmented existing patch-based methods with the addition of a feature map; this helped to preserve structure from the exemplar.

While allowing for some success with near-regular textures as a class patch-based methods are generally incapable of producing stationary textures.

Statistical-based Methods

Next we present some statistical methods of example-based texture synthesis. As a group these methods are not good at preserving structure; generally they synthesize unstructured stationary textures.

Heeger and Bergen [18] presented a method for synthesizing textures using a statistical method. The paper proposed a method that utilized a number of images generated from sub-bands of wavelet decomposition and assembled into a pyramid from higher to lower frequencies while handling the colour content separately. This method is primarily focused on stochastic textures as regular and near-regular textures tend to lose structure. This method generated a lot of successive algorithms

that had additional constraints in attempts to preserve more structure allowing for the synthesis of more regular textures.

De Bonet’s paper *Multi-resolution Sampling Procedure for Analysis and Synthesis of Texture* [10] was one of the statistically based texture simulation methods to impose a perception model developed from psychology studies on human perception of texture. The author used a two phased approach; the first phase analyzed the exemplar by computing the joint occurrences of the features from the cognitive models. This was done across different resolutions generating a pyramid of texture tiles with psychologically significant features. The second phase generates the output texture by an ordered sampling of the spatial frequency bands of the exemplar, subject to the same conditions imposed the join occurrence from all the lower frequencies; which is to say the various resolution tiles are reassembled together in a psychologically salient manner. De Bonet’s method seems to work best on stochastic textures but has some limited success on exemplars with more regularity thus improving upon Heeger and Bergen’s work.

Simoncelli and Portilla [32, 37] expand upon the work of Heeger and Bergen [18] by adding in second order properties. Simoncelli and Portilla also used the properties of joint wavelet coefficients. The changes by Simoncelli and Portilla improve on the results of previous statistical methods regarding regular texture preserving more structure than Heeger and Bergen [18]. The method still has difficulty with highly structured patterns as it is unable to correctly synthesize the high-frequency information.

Galerne et al. presented *Micro-texture Synthesis by Phase Randomization* [16] demonstrates a method which features a Random Phase Noise algorithm which randomizes the Fourier phase of an input texture. The method is limited to textures characterized by their Fourier modulus which the author refers to as *microtextures*. Micro-textures are the homogeneous areas of an image composed of elements that are individually indistinguishable. An example would be the sand in picture of a heavily populated beach; the individual grains of sand are not apparent but that part of the image is continuous and homogeneous. The method is very quick, avoids visual artifacts and avoids convergence problems.

Machine Learning Methods

There has recently been an explosion of interest in machine learning for texture synthesis [19, 20, 35, 42, 43, 51]. I have added them to the exemplar-based methods as they use an input texture. We briefly present two methods we found particularly compelling.

Zhou et al. presented *Non-stationary Texture Synthesis by Adversarial Expansion* [51] used machine learning to develop an exemplar-based texture synthesis method. The system takes non-stationary images as input, the system takes a small subset of the input and uses that as an exemplar. One generator network then tries to produce a larger image from the exemplar and the second network, the discriminator network evaluates the result using the original image for comparative purpose. As with all adversarial networks, the results of both networks improve over time. The method Zhou et al. developed is respectful of structure and can reproduce near-regular textures and non-stationary features, vastly improving previous machine learning exemplar-based methods.

Isola et al. [19] also use machine learning for texture synthesis. The authors use conditional adversarial networks to address image to image translation problems in a general manner. The image to image translations includes edge drawings and labelled diagrams as inputs. The system then uses generative adversarial networks with one network producing results and another evaluating the output so the system is self-learning. The results are impressive suggesting that this area will continue to be explored for the purposes of texture synthesis.

Collectively, example-based methods tend to be slow and limited to stationary textures. Zhou et al. had some success at producing non-stationary textures but still relies on an exemplar. With our goals of producing non-stationary textures without an input texture, we looked to procedural methods for inspiration.

This concludes our brief background on machine learning methods and we now present a brief background on procedural texture synthesis methods.

Procedural Texture Synthesis

Procedural texture synthesis produces a procedural texture, which is a texture created using a mathematical description rather than directly stored data [46]. Procedural texture synthesis methods also provide parameters to modify the output and generally avoid input textures unlike exemplar-based methods. Procedural methods also have

a number of other useful properties and tend to have small memory footprints, and are relatively quick to compute.

In this section, we will start with one of the earliest and perhaps best known procedural methods, Perlin Noise [30]. This is followed by a number of other noise-based procedural methods. We then present several non noise-based procedural methods.

Perlin Noise and successive papers

One of the earliest examples of procedural texture synthesis was Perlin’s *An Image Synthesizer* [30] paper which demonstrated a function approximating white noise that could be used to generate textures. Perlin noise has since been used for cloud generation [34], fire [15] and water effects in addition to more traditional texturing applications like marble effects. Perlin noise has left an indelible mark on the industry with many graphics frameworks including functions to generate Perlin noise.

Perlin presented a framework that generated “solid” textures. Perlin used the term solid to indicate textures that were three dimensions and could be mapped directly to geometry defined within the boundary of the encompassing solid texture. This direct mapping was without distortion induced by traditional mapping techniques as the solid texture would be queried directly. More interesting was Perlin’s proposal to define the solid texture using a simple traditional two-dimensional texture and a noise function. The noise function would work on any input point in the texture space and would perturb the input texture to produce a value for that point.

Perlin did this by first defining a cubic lattice with pseudo-random gradient vectors assigned at every corner. Points within these cubes were interpolated from the eight corner gradients. Perlin used a splined interpolation to avoid artifacts associated with linear interpolations. The gradients are pseudo-random as they are selected from 12 vectors that correlate to the directions from the center of a cube to its edges. The noise function did exhibit periodicity but Perlin also demonstrated a turbulence function that added together octaves of the noise avoiding the artifacts from the repetition.

Perlin’s work helped to define the properties of noise functions for use with texture synthesis. The first was a small storage size as all that is stored are seeds for the noise function. The second is that noise functions are continuous which is to say they can be used to produce noise of any resolution. Thirdly the noise function is

not periodic which allows the output to be any resolution. Finally, the functions are parameterized, allowing for changes to the patterns generated.

Perlin’s work sparked a huge volume of research dedicated to noise functions and has since become a very well researched sub-area of texture synthesis. We present some of the more immediate antecedents later in this section. Lagae et al. [23] provides a more exhaustive summary of noise-based texture synthesis methods.

Perlin also improved upon his own work in 2002 [31] by improving upon the sub-optimal gradient calculation and addressing visual discontinuities in second-order interpolations.

Kensler et al. [21] worked on improving Perlin noise using three discrete methods. The first was to include an alteration to the hash function combined with a gradient table. The second improved the band-limits by changing the reconstruction kernel. The third was an alternative projection when mapping the resulting texture into the 2D space. Taken together these three methods improved the spectral properties significantly with a closer approximation to white noise.

We continue with an exploration of non-Perlin based noise functions.

Other Noise Functions

Spot Noise was developed by Van Wijk [44] as a way to generate stochastic textures for visualizing both scalar and vector fields. The technique used randomly weighted and positioned spots and performs a convolution on spots. The method also provides localized control to the user by allowing for variation of the spots.

Wavelet Noise Wavelet noise as presented by Cook and DeRose [8] highlighted some shortfalls of Perlin’s approach; aliasing and detail loss. Cook and DeRose cited Perlin’s weak band limitations as the cause of these issues and developed a method that would be truly band-limited. The method used a preprocessing step that generated a number of tiles of noise coefficients. This was done first filling a tile with noise, then down-sampling, followed by an up-sample and finally subtracting. The method was truly band-limited and helped minimize aliasing but was slower and took more memory than Perlin noise.

Gabor Noise Gabor noise was introduced by Lagae et al. in the paper *Procedural Noise using Sparse Gabor Convolution* [24]. Lagae et al. built upon the work of

Lewis [25] developing a texture synthesis method using a sparse convolution and featuring the Gabor kernel. Lagae et al.’s method create band-limited anisotropic noise using a pulse process with the Gabor kernel as a pulse. This approach provides spectral control using limited parameters that naturally map to the power spectrum domain.

Non Noise-based Procedural Methods Within the area of procedural texture synthesis, there have been methods that don’t use noise functions which we refer to collectively as non noise-based procedural methods. In this section we briefly present several of these alternative methods including Worley’s *Cellular Based Texture Function* [48], and Turk’s Reaction Diffusion [41] paper.

Worley Noise Worley’s paper *Cellular Based Texture Functions* [48] introduced a new texture basis function which could be used to generate cellular patterns which could, in turn, be used to generate textures.

Worley does this by seeding a number of sites or “feature points” in space then defines functions such as $F_1(x)$ so that the value returned is the distance to the closest feature point relative to the point defined by x . This effectively divides the space into cells where every point within a given cell is closest to the same feature point. This is the definition of a first order Voronoi diagram. Worley then defines additional distance functions $F_n(x)$ so that the distance returned is to the n th closest feature point. This does deviate from the definition of higher-order Voronoi diagrams which uses n -tuples of points to define higher-order Voronoi diagram. Worley then assembles these functions into linear combinations eg. $2F_3(x) + 4F_2(x) - 3F_1(x)$. Worley used the values of the linear combinations colours and normal-displacement to produce textures.

This approach could be used wherever noise methods had been used as it also provides a continuous mapping of points to scalar values while providing a very different appearance.

Reaction Diffusion Turk’s paper *Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion* [41] was modelled after a natural process called reaction diffusion where chemicals diffuse and react at unequal rates to form spots and stripes. Turk’s work demonstrated how multiple iterations of the reaction-diffusion simulation with each successive iteration using the output of the previous iteration as input could

produce a larger variety of patterns. Turk also demonstrated how this process could be done directly on the surface of objects avoiding using projections and the associated issues. Turk’s method starts by randomly distributing points across the surface of an object. Then a relaxation is performed upon the points until a somewhat uniform distribution is achieved. At this point, a Voronoi diagram is generated using the relaxed points as sites and the Voronoi regions are used to produce a mesh. The reaction-diffusion is performed on this mesh using the edge lengths of the regions as the diffusion coefficients. Finally, the texture is rendered directly on the mesh using a weighted sum of the mesh values. One other detail to note is Turk found he could use a vector field to make the Voronoi regions anisotropic and this anisotropy would be transferred to the diffusion process, this allowed Turk to generate stripes as well as spots. In the same year that Turk’s method was published Witkin and Kass [47] published *Reaction-Diffusion Textures* which presented two methods to simulate reaction diffusion; the first used a convolution and the second used a hierarchy of successively finer grids.

Graph-based Methods Caron and Mould’s [6] paper *Partition of Unity Parametrics for Texture Synthesis* presented a procedural method that synthesized textures over a graph. Caron and Mould’s method uses partition-of-unity parametrics to interpolate between data values seeded across a plane. The graph is a triangle mesh with data stored at the nodes. Values are interpolated across the triangle faces using projection, normalization and a weighted sum. Caron and Mould’s method provides two main degrees of freedom; distribution of the nodes and assignment of the stored data. This approach allowed for a large range of outputs that can approximate Perlin Noise and Worley Noise as well as producing progressive textures.

Smooth Signed Distance Field Textures by Ehren Choy [7] proposed another procedural texture synthesis method that used a graph. Like Worley’s work, the process starts by seeding points in space but, rather than generating Voronoi diagrams with the seeded sites Choy’s method applies labels to the points and assembling them into groups sharing the same label. These groupings are considered implicit regions. The regions then have interior and exterior contours defined and a smooth signed distance field is constructed from these contours. Finally, a texture is generated by blending together regions for all groups. The use of two contours to construct the smooth signed distance field avoids discontinuities. Choy’s method is capable of generating

textures resembling patterns found in nature.

We now present the papers that we built on most closely for our work. While all are not within the area of texture synthesis they all are methods that use geodesic distances.

2.1.4 Geodesic Based Methods

Three papers, in particular, influenced this work. The first was *Image Guided Fracture* [28]. The second was *Procedural Tree Modeling with Guiding Vectors* [50]. The third was *Geodesic Image and Video Editing* [9]. In this section, we present these three papers. All three of these papers developed graph-based algorithms that used the geodesic distance in their calculations.

Image Guided Fracture Mould’s paper *Image Guided Fracture* [28] presented a method to generate realistic cracked stone textures and provided the ability to guide the placement of some of the cracks.

Like Worley’s [48] method Mould’s starts by seeding a number of points into a space. Also like Worley’s method, the points are then assembled into a Voronoi diagram, however rather than using the Euclidian distance Mould uses the geodesic distance calculated over a 4-connected lattice. Each node in the graph has an associated noise value which is used in calculating the edge weight. Mould is able to guide the cracks using an input image with dimensions matching the graph. Mould also allows for the widening of cracks by relating the crack width to the distance from the originating site. Finally, Mould creates multi-resolution cracks by iterating the same process within the confines of a subset of the regions. These techniques combine to make some convincing cracking effects. Mould’s paper should be considered our work’s most immediate predecessor, and we differ our approach in two ways: first, we can generate non-stationary textures without an input texture by using an edge weight calculation that uses a value from the originating site; second, we use Nth order Voronoi diagrams. These two differences allow us to vary the region shape significantly for the previous methods.

Guiding Vectors Xu and Mould’s *Procedural Tree Modelling with Guiding Vectors* [50] presented a procedural framework for tree modelling. The framework used graph

based tree synthesis techniques and augmented them with guiding vectors. In graph-based tree modelling systems the trees are the least-cost paths within a graph. Xu and Mould’s technique associated with guiding vectors with each node in a weighted directional graph. The guiding vectors are calculated for each node when they are first visited. This calculation is a product of the preceding node’s guiding vector and a small rotation. When calculating the weight of a given edge; edges that are most similar in direction to the guiding vector will have a reduced cost. This results in paths through the graph with edges that correspond to the guiding vectors being the shortest. Synthesizing the tree was performed by seeding a number of nodes as endpoints, using Dijkstra’s algorithm to search for the shortest path to the endpoints while performing the guiding vectors based edge weight calculation. Once the shortest paths to all the endpoints are found this process is repeated a number of times using the old endpoints as new starting points and selecting new endpoints. To vary the appearance of the generated trees Xu and Mould developed a number of degrees of freedom including different methods to select endpoints, and different rules and parameters to alter the rotation of each new guiding vector.

Geodesic Images Criminisi et al. presented *Geodesic Image and Video Editing*, detailing a unified approach to perform de-noising, image segmentation, texture flattening and painterly effects. The algorithm presented was very efficient allowing the technique to apply to video in real-time. This efficiency was achieved by avoiding energy minimization and using parallel programming techniques which allowed the authors to leverage modern multi-core CPUs. Inspired by the work of Bai and Sapiro [3] the algorithm uses a Geodesic Distance Transform (GDT). Traditionally a GDT use a binary mask to indicate the object to which the distance would be calculated. Criminisi et al. work uses soft masks that introduces an a degree of likelihood in the mask and which the authors use in the geodesic calculations. The authors referred to this as a Generalized Geodesic Distance Transform (GGDT).

To efficiently compute the GGDT the authors discretized the image domain using an eight-connected graph and using a raster-scan method of calculation [4] which uses kernel-based operations that are applied in multiple passes over the image. This allows them to calculate the geodesic distance d as a chain of pixels between a and b such that $(x_0 = a, x_1, \dots, x_{n-1}, x_n = b)$ where $\mathbf{x} = a$ and $\mathbf{x}' = b$ thus d can be defined as follows:

$$d(\mathbf{x}, \mathbf{x}') = \sum_{k=1}^n [|\mathbf{x}_k - \mathbf{x}_{k-1}|^2 + \gamma^2 |I(\mathbf{x}_k) - I(\mathbf{x}_{k-1})|^2]^{\frac{1}{2}}, \quad (2.3)$$

In equation 2.3 γ is the geodesic factor that weights the contribution.

Figure 2.3 comes directly from Criminisi et al.'s paper and provides insight into a GGDt and the input image and mask that was used to compute it. The left-most image is the input image. The middle image is a (soft) mask with the darkness of the pixel indicating the likelihood of it being part of the mask. The right-most image provides a visualization of the generalized geodesic distance with darker areas indicating smaller distances.

Figure 2.3: 1. Input Image, 2. Input Mask, 3. Visualized GGDt; Image from Criminisi et al. [9]

This concludes our background chapter. In the next chapter, we present the algorithm we developed and its variations.

Chapter 3

Algorithm

We wanted to develop a procedural algorithm that would be controllable and with the ability to develop non-stationary textures. We also intended for the resulting textures to be reminiscent of natural geological formations such as rock containing mineral veins. Additionally, we wanted to avoid using input textures; this eliminated the exemplar-based methods. Existing procedural methods could produce cracked rock surfaces [28], but they could not produce stochastic features such as veins without an input texture. We built upon Worley [48] and Mould [28] by using Voronoi regions as the basis of our texture generation. Like Mould we believed that using a geodesic distance rather than the normal Euclidean calculations to generate the Voronoi regions would be flexible enough to produce the desired visual artifacts.

We also believed that using higher order Voronoi diagrams would allow us to generate a broader variety of output than first-order Voronoi diagrams and that these higher order Voronoi diagrams would be indicative of natural formations such as fractures in rock. First order Voronoi diagrams have a distinct cellular pattern somewhat similar to a honeycomb. In higher order Voronoi diagrams, the cells tend to become more irregular and the honeycomb becomes markedly less noticeable. These differences can be seen in Figure 3.1 while the 1st order Voronoi diagram clearly demonstrates cellular appearance and the fifth order produces longer thinner regions and a larger variety in region size. These two characteristics combine to produce a non-cellular appearance. The flexibility to produce these markedly different Voronoi diagrams would allow us to produce more diverse outputs.

The chapter is organized as follows: we present a breakdown of our algorithm, describing the initial graph generations, the calculation for determining the distance

Figure 3.1: 1st and 5th order Voronoi diagrams

within the graph, details of the Voronoi diagram generation, the stopping conditions, and the rendering. We also describe the framework we developed to allow us to do successive runs of the algorithm using the output of previous runs of the algorithm as input to successive ones. Finally, we will describe the variations of the algorithm.

3.1 Basic Algorithm

Our algorithm pipeline is shown in Figure 3.2. Our basic algorithm consists of the following steps. First, we assemble the graph and assign unique values to each node. The values that we assign will be used in later calculations. In the second stage of our pipeline, we promote some nodes into sites, which are the defining point of a Voronoi region. In the third stage, the graph is passed to the Voronoi diagram generation stage where geodesic distances are calculated and used to generate a Voronoi diagram. The fourth stage prepares the graph for output as a texture. The final stage is the rendering stage. In this stage, each node in the graph is used to produce a colour value for a corresponding pixel in the output texture.

3.1.1 Graph Assembly

We use a uniform four-connected graph in R^2 . One benefit of this decision is the graph is a direct analog for the texture we are generating: there is a node for every

Figure 3.2: Our algorithm represented in a pipeline

pixel in the output texture. Additionally, during the graph assembly stage, we assign a unique value to each node. This value is used later in the edge weight calculations. In our basic algorithm, the value we assigned was a random three-dimensional vector. Each component of this vector was randomly determined and each component was restricted to the same range; this is presented in Equation 3.1 where V is the random three-dimensional vector.

$$\left. \begin{aligned} V &= (v_1, v_2, v_3) \\ \{v &\in \Re | v \in [0, 1]\} \end{aligned} \right\} \quad (3.1)$$

3.1.2 Site Assignment

We next place the sites which will be used to generate the Voronoi regions. We implemented two methods for seeding the sites. The first method of site distribution randomly distributes the sites throughout the graph and the second distribution is a Poisson disc sampling algorithm [5]. The primary property of the Poisson disc algorithm is that no two points will be closer than a specified radius [5]. We use the Poisson disc sampling algorithm to produce a specified density of pixels without noticeable periodicity. The Figure 3.3 shows the differences between the two methods with the random method having both denser clusters of sites and sparser areas and the Poisson distribution have a more uniform distribution.

Next, we present the Voronoi diagram generation stage of our pipeline.

3.1.3 Voronoi Diagram Generation

We use the initial sites previously generated to assign all the nodes in the graph into an N-th order Voronoi regions. We can differentiate our work by looking at

Figure 3.3: Left: Random distribution, Right: Poisson distribution [1]

some of the most similar predecessors. Worley [48] used standard Euclidian distance calculations to subdivide the space into higher-order Voronoi regions. Mould [28] used a weighted graph, and determined the edge weights by using noise values associated with the frontier and target nodes. Our work derives the weights of the edges in the graph with calculations contrasting values from the the region's originating site and the target node. This is demonstrated by the equations (3.2) and (3.3). The equation (3.2) shows how we calculate the weight for the edge between a frontier node and a target node along the frontier of a newly forming region.

In equation (3.2) we denote v_s a vector of the originating site that was assigned during graph assembly. We denote v_t as the vector of the target node. We then subtract the vector in the target node v_t from the vector in the originating site v_s which produces Δv which is then used in the calculation of w_e as shown in equation (3.3).

$$\Delta v = v_s - v_t \quad (3.2)$$

In equation (3.3) we take the absolute value of the maximum of the components of v_Δ then add γ a term that guarantees the minimum weight of the edge. This is important as all edges must be greater than zero to guarantee the algorithm will finish. This provides w_e which is the weight of the edge between the node and the target node back to the originating site.

$$w_e = \max(|\Delta v_x|, |\Delta v_y|, |\Delta v_z|) + \gamma \quad (3.3)$$

An alternative calculation to 3.3 is presented in 3.4. In the alternative calculation,

we take the magnitude of the differences between the two vectors.

$$w_e = ||v_s - v_t|| + \gamma \quad (3.4)$$

In comparison, we preferred the slightly more regular characteristics of the output produced by equation 3.3. These subtle differences can be seen in Figure 3.4 where we have highlighted the more significant of the differences.

Figure 3.4: Left: The weight calculation from equation 3.3 Right: The weight calculation from equation 3.4

We then use the well known Dijkstra’s algorithm [17] to perform a breadth-first search to calculate the regions of the Voronoi diagram much like Mould [28] and Erwig’s [13] Parallel Dijkstra Algorithm. We start by seeding the algorithm with sites from the site generation stage. As the algorithm progresses the regions form around the sites. The nodes constituting the edges of expanding regions can be thought of the ”frontier”. These frontier nodes all have an associated path cost which is the geodesic distance from the originating site. The path cost is the smallest cumulative edge weights from the originating site of the regions to this frontier node. As per Dijkstra’s algorithm during each successive iteration, we take the node with the smallest path cost; we will refer to this node as f . The frontier is stored in a min-heap which has the property of having the smallest weighted object at the top of the heap which guarantees we are getting the node with the smallest path cost with each removal operation. We examine the neighbours of f which we refer to as target nodes or t , we then calculate the edge weight using 3.3. We then add the new

edge weight to the path cost of f , and if this new path cost is less than the path cost already associated with t , then t is added to the region and will, in turn, become part of the frontier and inserted into the min-heap. This continues until there are no more nodes in the frontier at which point every node will be associated with a region. It should be noted that there are ways this process can be varied so that not all nodes are assigned and we will discuss this later in the chapter.

The basic algorithm is provided in the pseudocode below. Note that the pseudocode does not contain alternate stopping conditions and is only demonstrative of how we generate first-order Voronoi diagrams. The function $weight[n, f, s]$ calculates the weight of between a target node n , a frontier node f using values from an origination site s . The function $region[n]$ returns the site node that defines a region for a

given node n .

Result: Graph of nodes that represent a Voronoi diagram

N = set of nodes in graph;

S = set of sites;

M = min-heap of nodes;

for *each* n *in* N **do**

$\text{dist}[n] = \text{INFINITY}$;

$\text{region}[s] = \text{UNDEFINED}$;

end

for *each site* s *in* S **do**

$\text{dist}[s] = 0$;

$\text{region}[s] = s$;

 insert s into M ;

end

while *While* M *is not empty* **do**

$f = \text{extract-min from } M$;

$s = \text{region}[f]$;

for *each neighbour* n *of* f **do**

$e = \text{weight}[n, f, s]$;

$d = \text{dist}[f] + e$;

if $d < \text{dist}[n]$ **then**

$\text{region}[n] = s$;

$\text{dist}[n] = d$;

if n *in* M **then**

 decrease-key of n in M ;

else

 insert n into M ;

end

end

end

end

Algorithm 1: Our Basic Algorithm

Using the same nodes the process is then repeated for each of the higher order Voronoi diagrams that we want to calculate, with one small difference; the smallest path cost must be for a region that isn't already associated with f . The

first time the process is repeated this will result in adding an association and storing the associated path cost for the region that is second closest and so on for each higher order.

We implemented the collective frontiers as a minimum heap, which has the essential property of always having the node with the smallest cost path at the top. Since any given node has at most four neighbours it can only ever be reassigned to a new region a maximum of four times. The properties of a min-heap ensure that it is never greater than $\log(n)$ and all operations have an upper bound of $\log(n)$. Thus processing of all nodes has an upper bound of $(4n)\log n$ and is thus $O(n\log n)$. We repeat the process for each higher order Voronoi diagram that we care to calculate, in our experiments we restricted the algorithm to the fifth order so the upper bound would remain as $O(n\log n)$. In our case, n is tied to the resolution of our output texture and directly corresponds to the number of pixels.

To help demonstrate that the upper bound of our algorithm is $(4n)\log n$ we will provide an example. A few properties that hold are:

- a node can be added to the frontier a maximum of four times, once for each edge connecting it to the graph.
- the geodesic distance can only increase as the weights are non-negative and greater than zero.
- any node removed from the min-heap containing the frontier will be the smallest geodesic distance calculated.
- only the frontier nodes can change their associated geodesic distance when they change region membership.

The fourth property merits more discussion: This property holds as a node inside the frontier has previously been removed from the heap. A node is only removed from the node if it has smallest geodesic distance. As we don't allow positive non-zero edge weights path costs can only increase thus no later path can have a smaller geodesic distance. Finally as the frontier from all sites are contained in the same heap this is the smallest geodesic distance to any site node.

The result of these properties can be observed in Figure 3.5 depicting a simplified example. In Figure 3.5 we present a two-connected graph with five nodes and two site nodes. Frontier nodes are in pink and site nodes are in red, unassigned node are

white without a letter. The min-heap is seeded with the two sites each with a weight of zero. To keep the example clear we are using integer weights but our algorithm works with real numbers. We present a description of the sequence as follows:

- Line 1: Site A is removed from the heap, the edge weight is calculated to be 10 and the node is added to the heap with a geodesic distance of 10. The second node from the left is added to the heap and is part of the frontier of site A.
- Line 2: Site B is removed from the heap, the edge weight to the adjoining node is calculated to be 11 to the node from the right and is added to the min-heap as part of the frontier of site B.
- Line 3: As the second node has the smaller geodesic distance of 10 it is removed from the top of the heap and the next edge weight is calculated to be 20. So the middle node has a geodesic distance of 30 and is stored in the min-heap.
- Line 4: The second node from the right is at the top of the heap with a geodesic distance of 11. The edge from it to the middle node is calculated to have a weight of one. So the middle node is placed on the min-heap with a geodesic value of 12. This causes site B to replace both the association and the geodesic distance of the middle node. This demonstrates how the frontier can change their association.
- Line 5: The node second from the right is removed from min-heap and evaluated we calculate an edge weight of one which gives a geodesic distance of 13 from site B, but the nodes current geodesic distance is 10 so its association to A remains. Since we didn't put another node on min-heap the algorithm completes. This demonstrates how the nodes outside the frontier will have the shortest geodesic distance by the time they leave the frontier.

In the example depicted in Figure 3.5 highlights how the properties of minimum-heap ensure only the frontier can change region association which keeps the upper bound to $O(n \log n)$ for our algorithm.

3.1.4 Normalization

The normalization stage is intended to provide an output graph where every node is at most associated to single site. This requires that we select just one of the higher

Figure 3.5: Simple graph depicting region growth

order Voronoi diagrams produced in the previous stage. When the Voronoi diagram generation is completed we do not have a k-tuple higher order Voronoi diagrams as presented earlier and again presented in equation 3.5 where given a set of points $S = \{s_1, s_2, \dots, s_n\}$ where $n \geq k$ and $x \in R^2$ then we will define the k-order Voronoi diagram by the union of the regions defined as follows:

$$V(i_1, i_2, \dots, i_k) = \{x : \|s_{i_1} - x\| + \dots + \|s_{i_k} - x\| \leq \|s_{j_1} - x\| + \dots + \|s_{j_k} - x\|\},$$

$$\forall \{i_1, i_2, \dots, i_k\} \neq \{j_1, j_2, \dots, j_k\}$$
(3.5)

Upon completion of the Voronoi diagram generation step the nodes' associations to regions are the k-closest (first, second, third) sites rather than regions defined by k-tuples as per 3.5; we can see the marked difference between the two Voronoi diagrams. In Figure 3.6 the Voronoi diagram on the left uses cyan to highlight all the regions that are associated with the site in green. One can observe how a single site's region becomes highly fragmented.

Figure 3.6: Left: 5th order Voronoi diagram pre-collapse; Right 5th order Voronoi diagram post collapse

To “collapse” these regions into the definition presented in (2.2) we iterate over all the nodes in the graph. We determine the k^{th} order for every node by grouping the nodes with the same k closest site into new regions representing the k-tuple.

This simple process is provided in the pseudocode below. The function *SiteForOrder*[n, o] returns the o -closest site associated to node n . The function *Get*[H, k] retrieves a set of nodes that represent a single region in a higher-order k -tuple Voronoi diagram from the hash H using the k -tuple k as the key. The function *Hash*[H, k, r] stores a set of nodes r in the hash H using the k -tuple k as the key.

Result: Graph of nodes that represent a higher-order k -tuple Voronoi diagram

```

N = set of nodes in graph;
H = Hash of sets of nodes r hashed by k-tuple k;
for each  $n$  in  $N$  do
    k = new k-tuple;
    for each order  $o$  of  $N$  do
        s = SiteForOrder[n,o];
        add s to k;
    end
    if  $H$  contains  $k$  then
        r = Get[H,k];
        add n to r;
    else
        r = new set of nodes;
        add k to r;
        add n to r;
        Hash[H,k,r]
    end
end

```

Algorithm 2: Our Collapse Algorithm

Finally, the inputs to the algorithm include the specification of the desired k -order Voronoi diagram so a graph with nodes associated to the regions that compose the desired k -order Voronoi diagram are passed on to the rendering stage.

3.1.5 Rendering

The rendering stage allows us to turn our graph of nodes into a texture. Until this point any visualization of the graph would just show the regions; we intended for the rendering stage to provide additional freedom to vary the output texture. To provide

for this we mapped each node to a value in a colour lookup table. This mapping allowed for a colour to be added to each pixel of the newly generated texture.

We first needed to create a function that assigns a colour to each node. The colours available would come from a colour lookup table that could be changed providing additional freedom to vary the output texture. First, we contemplated using the existing path cost from the site of each region. While this had a pleasing visual effect this only worked for first-order Voronoi diagrams as higher-order Voronoi diagrams defined by k -tuples don't have a single site for which to calculate the path cost.

Figure 3.7: Using existing path cost; left: first order, right: third order using k -tuples

We also tested out Worley's rendering method [48] of using linear functions as a basis function. These basis functions are derived from none k -tuple higher order Voronoi diagram and we refer to them as (F1, F2, ...) according to their respective order. Figure 3.8 shows how undesirable discontinuity errors appear when we tried to map the Worley style basis function to the regions generated by the higher order Voronoi diagram while using a per region normalization. This was due to the regions for which we were calculating the normalizing factor being "collapsed" into k -tuple higher order Voronoi regions and the distances stored during Voronoi diagram generation were from the none k -tuple higher order Voronoi regions. One alternative would be to combine the distances to each site composing the k -tuple but we decided to pursue an approach similar to what Worley had used.

Figure 3.8: Left: $F2 - F1$, Centre: $F2 + F1$, Right: $2F3 - F2 + F1$

We also used Worley’s rendering method using a global normalization. As can be seen in Figure 3.9 this alternative normalization has the beneficial effect of reducing discontinuities and looks much like Worley’s original work.

Figure 3.9: Left: $F2 - F1$, Centre: $F2 + F1$, Right: $2F3 - F2 + F1$

A method to avoid the discontinuities is one that assigns values to each node contained within the region based off of the nodes path cost from the region’s perimeter. This method had the benefit of working for all higher order Voronoi diagrams as well as providing an interesting aesthetic.

We determined the path cost of each regional node by again using Dijkstra’s algorithm backed by a heap. We performed this algorithm to determine the path cost for each region separately. We seeded the algorithm with all the perimeter nodes of the region and all the perimeter nodes are considered as the same site, this causes the frontier to expand inward. The process stops as the edges of the frontier converge in the center again terminating when the frontier is empty.

Upon the termination of Dijkstra’s, we have the path cost for each node from the

perimeter of the region. These values are not in the range of zero to one and thus need to be normalized. To provide the most variation of the output the rendering stage of our algorithm allows for two types of normalization; per region and a global normalization.

Figure 3.10: Two type of normalization; left: global normalization, right: regional normalization

This was accomplished during the regional calculation of the path costs; recording the largest path cost per region and the largest path cost for the entire Voronoi diagram.

With a normalized value, it becomes trivial to map each node to a pixel of a specified colour map. With a colour associated with each node in the graph we can output these colour values as our final texture. Figure 3.11 demonstrates a colour map assigned to two very different Voronoi diagrams.

3.2 Framework

Natural looking rock formations are often composed of a number of different rocks and minerals and these different materials exhibited different characteristics. Some examples contain veins of minerals visibly distinct from the surrounding rock. Other examples contained flecks and small patches of minerals. One way of obtaining such

Figure 3.11: Left: global normalization, Right: regional normalization

effects would be to combine the outputs of different variations of the algorithm, variation that had their parameters altered as to provide unique and desirable characteristics in their respective outputs. The output of one algorithm would become the input to successive algorithms. This would allow us to select features from the input as originating sites for the subsequent algorithm.

We then decided to create a framework that would allow running a series of variations of the algorithm and to use the output of each previous variation to be used as input in each subsequent run of the algorithm.

Figure 3.2 shows how the post-process of the first algorithm feeds the successive algorithm.

Next, we describe the operations we used to combine the output of the successive algorithm variation.

3.2.1 Combining

The framework we developed allows us to run algorithm variations multiple times and allows us to use the output of previous algorithms to be used as input to successive variations. This is, in turn, allowed us to develop new variations to the algorithm that could capitalize on having an input. The region confinement variation was a method that allowed the input to influence the output of the successor but the regions of the earlier runs were replaced by the successive algorithm's output. We decided that

Figure 3.12: Framework pipeline illustrating two algorithms

preserving the regions from previous algorithms would allow for more variation in the final output. It was decided that this could be accomplished by creating operations that would combine the input and the output in a variety of ways.

The output of a previous algorithm variation is a graph, every node in that graph can have one of two states. Either the node has an association to a site and thus belongs to a Voronoi region or lacks an association and is considered unassigned. A node may be unassigned due to alternate stopping conditions such as those imposed by region size limitations which we will discuss later. Each successive variation generates an output graph and every node in it will either be associated with a site created by the successive variation or unassigned. During this generating the successive algorithm can use the input graph from the previous variation to influence the generation of the new Voronoi diagram. It is during the normalization step that the input graph from the previous algorithm variation is combined with the output graph of the successive variation. Upon completion of this step, each node in the graph can be unassigned, associated with a site and corresponding Voronoi region from the successive algorithm or it can have preserved its association with a site and corresponding Voronoi region from the previous variation. The combination operation determines which of these three states each node in the output graph will take.

The combination operations we defined were *influence*, *replacement*, *additive*, and *subtractive*. We refer to the nodes from the input graph as *incoming*, and the nodes in the output graph of the successor algorithm as *outgoing*.

Influence does not directly alter the output node irrespective of the associations of the input nodes. This does allow for the input graph to influence the output during the Voronoi diagram generation step but does not directly alter any associations in the output graph.

Replacement allows for nodes from the outgoing graph that have been assigned to a region to replace nodes in the input graph if those nodes are unassigned; thus any node that remains unassigned by the successor algorithm will maintain their previous association. Inversely stated the replacement operation will overwrite the association of an input node if the corresponding outgoing node has been assigned a site by the successive algorithm and any unassigned areas will show the regions from the incoming graph.

Additive allows for nodes that are unassigned in the incoming graph to be replaced with values of the corresponding outgoing nodes. This combination operation can be thought of as a way to fill in unassigned areas in the incoming graph with regions from the successive algorithm.

Subtractive allows for nodes from the outgoing graph that have not been assigned to a region to replace the associations in the input graph. Thus only the unassigned nodes from the successor algorithm contribute to the output. In summary, the subtractive operation can only create unassigned nodes in the output graph.

Figure 3.13 shows an example of combining two output graphs together using different combination operations. Each line of the figure uses the same input and output graphs. Additionally, each line of the figure also contains the result of the different combination operation. The input graph contains twenty size limited Voronoi regions. The output graph contains forty smaller size limited Voronoi regions. The red areas are composed of nodes that have not been assigned to a region, this is due to the alternative stopping condition of the size limited variation. We now describe the output graphs top to bottom.

The first is the *influence* combination; this operation subtly influences the output changing the shape of larger conjoined regions in the top left and the bottom right.

The second is the *replacement* combination, this allows for the smaller regions to exist completely within the confines of larger regions from the input graph.

The third is the *additive* combination that only adds regions from the second graph to unassigned areas of the first. Finally, the fourth image on the bottom is the *subtractive* combination where only nodes that were assigned in both graphs remain.

Each combination has produced a unique output with the same inputs. This demonstrates how the combination operations can be used to alter the final output and can be used to generate features that a single algorithm variation could not.

Our next sections present the variations of the algorithm that we developed.

3.3 Variations

3.3.1 Site Weights

Further variations of the texture can be obtained by adding weights to the sites used to generate the Voronoi diagram. We suggest two approaches. The first approach is to give the sites starting weights other than zero. As discussed earlier in the chapter the geodesic distance is the sum of edge weights, and in our calculation, every node associated with a region will have this distance from the node to the originating site of the region. We present the geodesic distance as follows: y is the number of edges from a site to a given node, n is a node, e_k is the weight of the k th edge on the path and this produces g the geodesic distance from a specific site to a specific node.

$$g = \sum_{k=1}^y e_k \quad (3.6)$$

When we add an initial weight to the sites, represented as ρ the geodesic equation is modified to equation 3.7. One can observe initial site weight is only added once to the entire geodesic distance.

$$g = \rho + \sum_{k=1}^y e_k \quad (3.7)$$

This alteration to the equation will cause a delay in the growth of the sites with larger weights causing the resulting Voronoi regions to be smaller.

The second approach gives the sites a weight factor or *gravity* that is used in the edge weight calculations. When the weight for a new edge is calculated the weight is multiplied by the gravity of the originating site of the region. We represent the weight factor as σ in equation 3.8.

Figure 3.13: Each line: input graph, output graph, combination result; Top to bottom; Influence, Replacement, Additive, Subtractive

$$g = \rho + \sum_{n=1}^y \sigma e_n \quad (3.8)$$

Looking closely at 3.8 it can be observed that σ is a factor for every edge weight and can have a profound effect on the geodesic distance. Using a fractional weight factor or *gravity* will cause the geodesic distance to increase slowly so the regions associated will grow faster than sites that don't have a fractional gravity. The results of this behaviour can be observed in Figure 3.14.

A simple way to think of these two approaches is as follows: the former could be considered additive and the later multiplicative.

Having used uniformly weighted sites that tend to grow at approximately the same rate we wanted to explore sites that would grow much faster. While there are innumerable ways to assign values for ρ and σ to site node we only explored one. We varied weight factors by introducing a second factor for a subset of the sites; thus the weight factor variation provided two degrees of freedom in the form of two variables. First, there is the number of sites that will have a weight assigned, which we will refer to as ζ . ζ will be expressed as a percentage of the total number of sites. The second variable is the weight factor of the weighted sites that we will refer to as η .

The two images in Figure 3.14 illustrate two runs of the same algorithm. The image on the left has a ζ of 0.2 and a η of 0.1. This means that twenty percent of the site nodes will have a weight of one-tenth the other eighty percent. This means this minority of sites will grow at a much faster rate than the remaining sites. This can be observed in the leftmost image. The image on the right has a ζ of 0.2 and a η of 0.5. This means that twenty percent of the site nodes will have a weight that is half the weight of the remaining nodes. As one observes in the right image this means that twenty percent of the nodes grow at twice the rate of the remaining nodes.

This ends our brief discussion on site weights.

3.3.2 Edge Weight Calculation

Varying the edge weight calculation can also alter the output. We did this by adding values to the sites and other nodes that could be used for the edge weight calculation. We also altered the edge weight calculation; adding new variables and terms.

Figure 3.14: Two examples of the weighted site algorithm variation, left: $\zeta = 0.2$, $\eta = 0.1$; right: $\zeta = 0.2$, $\eta = 0.5$

We generated two alternate edge weight calculation which we refer to as the guiding vectors edge weight calculation and the accidental edge weight calculation. These two weight calculation is described in the following sections.

Guiding Vectors Edge Weight Calculation

Looking at rock formations for natural phenomena that we would want to add to our texture synthesis we observed how many rock formation have differing materials running throughout their entirety. These materials often took forms that could be described as strata or veins. We wanted to be able to have striations that would cross through the texture decreasing the stationary quality of our resulting textures. These striations would dramatically reduce how static the texture would be perceived and illustrate the range of textures we can synthesize.

With the idea of creating these long flat regions, we looked to leverage the work of Xu and Mould [50]. Xu and Mould [50] used a technique they referred to as guiding vectors to procedurally grow trees. The basic premise is that an originating node in a weighted graph has a vector assigned to it and the weight of each edge is calculated based on the difference in the originating node's vector and the target edge's vector. Additionally integral to the algorithm; the child's vector is an incremental rotation of its predecessor's.

Figure 3.15 shows two runs of the guided vector based algorithm. In these examples the vectors of the site nodes all point towards the centre of the space. We can see how this introduction of guiding vectors influences the regions thus diminishing the honey-comb that tends to present in Voronoi diagram based algorithms.

Figure 3.15: Two examples of vector directed Voronoi diagrams

Where the guiding vectors Voronoi diagram generation step differs from the basic algorithm Voronoi diagram generation step is in the calculation of the edge weight and the addition of a guiding vector for this purpose. We mentioned in earlier sections that there is an intensity vector stored in all nodes during the graph assembly stage; call this vector i . Also during graph assembly, a vector indicating the location of the node is stored in all nodes; we refer to this vector as l . Additionally, there is the guiding vector that is initially only assigned to site nodes; call this vector g . Three nodes are used in the calculation of a given edge weight: the originating site node, the node at the frontier of the region and the target node. These nodes will be referred to with s , f and t respectively.

First we calculate the intensity difference Δi between the originating site and the target node:

$$\Delta i = i_s - i_t \quad (3.9)$$

We then calculate the difference between the locations of the originating site and the location vector using their respective stored location vectors producing Δl .

$$\Delta l = l_s - l_t \quad (3.10)$$

The third thing we do is calculate the guiding vector for the target node which we will refer to as g . We noted earlier that initially, only site nodes have guiding vectors. The guiding vector has been calculated stored in target nodes as they are added to a region. The calculation takes the vector stored in the frontier node and perturbs it by rotating by β . β are randomly selected from a given range. Let $r(x, y)$ be a function that takes vector (x) and rotates it by a specified number of degrees (y) is this case.

$$g = r(g_f, \beta) \quad (3.11)$$

With the three vectors calculated we perform the weight calculation (3.12). As in the weight calculation for the higher order Voronoi diagrams, we use the largest magnitude component from the calculated intensity vector. We also introduce the term ϵ to avoid a divide by zero error.

$$w_e = |\max(i_x, i_y, i_z)| * (1/(|l \cdot g|^p + \epsilon)) \quad (3.12)$$

This weight calculation allows us to grow the Voronoi regions along the direction of the guiding vectors.

We now present another edge weight calculation.

3.3.3 Accidental Edge Weight Calculation

We also experimented with mild variations of the basic algorithm that would use more than one edge weight calculation switching when prescribed logical conditions were met; during this process, we created an algorithm that had very interesting regions with several notable features.

Figure 3.16: Two products of our accidental edge weight calculation

In Figure 3.16 one can observe a variety of region sizes; some are very large and others barely larger than the originating site. The second detail is the irregularity of the region's boundary; many regions have peninsula-like features extending from the bulk of the region.

The primary difference with this algorithm was that as the regions initially expand one formula is used for the edge weight calculation and a different formula is used to calculate the weight of an edge when the target node is already assigned to a region.

In the initial edge weight calculation presented in equation 3.13 we define the edge weight as w_{e1} ; we let the path cost to the frontier node be represented as τ and let γ be a term to add a minimum cost for the edge. The term v_f will denote the vector value assigned to the frontier node and the term v_t will denote the vector value initially assigned to the target node. Equation 3.13 is used when calculating the weight to a node that hasn't been assigned to a region.

$$w_{e1} = \tau + ||v_f - v_t|| + \gamma \quad (3.13)$$

The secondary edge weight equation, Equation 3.14 is used to calculate an edge weight when the target node has been assigned to a region. The term v_s will denote the vector value assigned to the site node and the term v_t will denote the vector value initially assigned to the target node, both of which are unit vectors. Finally, we use

the term μ to allow us to vary the output, in practice it allows us to alter the size of the generated regions. These terms are then used as shown in equation 3.14.

$$w_{e2} = (1 - ||v_t \cdot v_s|| + \mu) * w_{e1} \quad (3.14)$$

The algorithm also involved two significant departures from the original algorithm. The first departure was in equation 3.14 where we include τ in the edge weight calculation which causes an initial geometric growth of the path cost. The second departure was that nodes belonging to a region could be reevaluated and have their path cost reduced while remaining in the same region.

Thus equation 3.14 produces a value from zero to one, adds μ and multiplies it by the initial weight calculation which includes the path cost of the target node. This guarantees that some large regions will consume smaller regions that have perpendicular vectors assigned to their site nodes.

While these two changes may not seem significant; Figure 3.17 illustrates what happens if remove these differences. If we correct the first change that causes the geometric growth of the path cost the result is very much like a normal Voronoi diagram. If we remove the second departure and stop the reevaluation of the path cost we can see that the irregularity of the region boundaries diminishes.

Figure 3.17: Left: mistakes uncorrected, Centre: mistake one corrected, Right: mistake two corrected

This edge weight calculation is referred to as the accidental algorithm, because this weight calculation was coded by accident while transforming an implementation of the basic algorithm into one that would use guiding vectors.

3.3.4 Site Locations

The framework we developed allows for runs of the algorithm to be used as input for the subsequent algorithm this provided us with the capability to be more selective when placing the sites. Instead of just relying on various distributions for the sites of successive generations the site locations can be more structured. This would allow us to put sites in specific size regions, or at the junction of regions, or along edges between regions. We believed this would allow us to widen “cracks” between regions. We would also need to support the idea of a site being more than a single node, this way all the nodes that are part of an edge would be considered the same site and the successive site would expand in a uniform manner.

The first part of this was to expand the functionality of the site assignment stage. As we discussed earlier the input graphs are in a normalized form with each node containing its own site assignment. To augment this information the selection based site distribution algorithm was created. The selection based site distribution assemble the graph into regions, edges and *confluences*. We define a confluence as a node that has neighbours in more than two different regions. The nodes that are part of an edge have neighbours in exactly two different regions.

In Figure 3.13 we show an example where an input graph has been processed into regions, edges and confluences. The confluences have been enlarged to make them more visible.

Figure 3.18: Left to right: Regions, edges, and confluences

After we process the input graph we then select the subset of elements that will be used to generate the new sites. This could be done randomly, this could be based on attributes of the artifacts themselves such as selection the longest edges or the smallest regions.

In Figure 3.19 we show an example input graph. The image in the middle shows the ten edges that were selected. These new sites were then used in a region limited

basic algorithm which kept the new regions relatively small. Finally, the image on the right is the output with new regions.

Figure 3.19: Left: input graph, Middle: graph with selection sites, Right: output graph

3.3.5 Stopping Conditions

We also looked at ways to terminate the algorithm without emptying the entire frontier. One method that emerged from this line of inquiry was the idea of limiting the growth of regions. We anticipated that this could also help model some natural phenomena that occur in natural rock namely rock formations having small deposits or flecks of other minerals distributed throughout.

The images in Figure 3.20 below provide indicative examples of the region limited variation. The random distribution provides interesting clusters of regions. In these images, we also removed the sites for clarity.

This variation is implemented in a very simple manner; first detail is that we maintain a list of each region's size. After calculating the path cost to a target node there are two conditions for the target node to join the region to which the frontier node belongs. First, the newly calculated geodesic distance must be smaller than the target node's stored path cost and second, the region must not have already met the size limit.

Region Confinement Variation

Another variation that we experimented was region confinement. It was designed to allow for the new sites of a subsequent algorithm variation to grow new regions without

Figure 3.20: Two examples of the region limited algorithm variation, the left restricts regions to 300 nodes and the left restricts regions to 900 nodes

exceeding the bounds of confining regions from the previous algorithm variation. We anticipated that this variation would allow us to subdivide specific regions and help produce more varied output much like Mould [28] did in his guided fractures.

In Figure 3.21 we can see two images; the image on the left shows an input graph generated for the basic algorithm with the first order as the output. The image to the right was a successive run that or the higher order algorithm. The original regions are still preserved but most of the larger regions have been subdivided into a number of constrained regions.

The region confinement variation was added to the Voronoi diagram generation stage of the algorithm pipeline. This variation was implemented as follows: before we calculate the weight of an edge to a target node we compare the region membership of the frontier's analogue node in the input graph and the region membership of the target node's analogue; if these two nodes do not have the same membership then we conclude the edge in question would be crossing out of the encompassing region that the frontier had been expanding in. As this crossing of borders is disallowed we disregard this node as a potential target.

Figure 3.21: Left input graph, Right confined subdivision

3.3.6 Rendering

Another way we realized we could vary the output was by varying a colour lookup table. Using various lookup tables we could introduce banding, bias the output texture towards light or dark, or any number of other visual effects.

Figure 3.22 shows the same Voronoi diagram with different textures used as the colour lookup table. The left and middle images use the same tonal range but the leftmost has a light blue bias. The middle texture used a colour lookup table with an even split between light and dark blue. The rightmost introduces bands of purple in the colour lookup table which almost create a topographic effect in the output texture.

Figure 3.22: Voronoi diagram with different colour lookup tables

Chapter 4

Results and Discussion

In the previous chapter, we presented our algorithm and a number of variations that can be used to modify the original algorithm; in this chapter, we present the results from our algorithm and its variations. We first look at the results from the variations of the edge weights calculations in isolation. We then present the results from the algorithms in conjunction by leveraging our framework. In principle, we could have altered the edge weight distribution but we have not explored this much.

4.1 Weight Calculation Variations Results

The following section presents some of the variations achieved with the variations of the weight calculations. We first present results from the basic weight calculation by varying a number of inputs. We then present results from the guided vector weight calculation. Finally we present results from the accidental weight calculation.

4.1.1 Basic Weight Calculation Results

The following section presents some results from varying the inputs of our basic weight calculation. The inputs to the algorithm are as follows:

- n order of the Voronoi diagram we wish to produce
- s number of sites
- γ the minimum edge weight used in the weight calculation

Figure 4.1 illustrates the difference between the first to fifth order. As the order

increases, we see increasing diversity in the size and shape of the regions. The differences diminish as the order increases and this diminishing return is why we have limited our experimentation to fifth order.

Figure 4.1: Left to right: $n = 1, n = 2, n = 3, n = 4, n = 5$ All: $s = 20, \gamma = 0.5$

Figure 4.2 shows the effects of varying the number of sites. The three images feature Voronoi regions increasing in number but the individual regions maintain similar characteristics even as their total number increases.

Figure 4.2: Left to right: $s = 20, s = 100, s = 500$ All: $n = 1, \gamma = 0.5$

Figure 4.3 illustrates the effects of changing γ in the basic edge weight calculation. In Figure 4.3 one can observe the irregularity of the edges decreases as γ increases while the shapes of the regions become increasingly blocky.

Figure 4.3: Left to right: $\gamma = 0.0$, $\gamma = 0.5$, $\gamma = 0.9$ All: $s = 20$, $n = 1$

The output of our basic edge weight calculation with a higher order Voronoi diagram chosen can be quite striking with the addition of a colour table. The three images in Figure 4.4 all demonstrate the variety of Voronoi regions sizes. The three images were all generated with randomly distributed sites which tend to produce veins composed of smaller regions that start form into a lattice in the higher order Voronoi diagram; this is most notable in the third image. This feature was not present in Mould’s [28] guided fracture work but may have been possible in Worley’s [48] even though it wasn’t demonstrated in the example images.

Table 4.1 provides insight into the speed of the algorithm. The measurements were captured when generating the images in 4.1. All the images were generated with 20 sites and a 600 x 600 output size. All times are in milliseconds and are an average over a thousand runs of the algorithm. The table breaks the running time into the time to create the graph and initial sites which is in the Site Generation row. The time to generate the Voronoi diagram is in the Voronoi Generation row. Finally, the time to complete the post-processing is listed which in this case is the time to “collapse” higher order Voronoi regions. It can be observed that the Voronoi Generation row increases a little approximately thirteen hundred milliseconds per order while the other two rows are approximately constant.

In the next subsection, we present some results of the guided vector weight calculation.

Figure 4.4: Higher Order Voronoi Diagrams, left to right: First Order: $s = 50$, $\gamma = 0.25$; Second Order: $s = 91$, $\gamma = 0.06$ Third Order: $s = 42$, $\gamma = 0.2$

Texture Synthesis Runtimes					
Stage	1st Order	2nd Order	3rd Order	4th Order	5th Order
Site Generation	57	54	57	61	59
Voronoi Generation	286	1350	2657	3746	5319
Post Processing	396	581	620	712	807

Table 4.1: Runtimes of the algorithm in milliseconds

4.1.2 Guided Vector Weight Calculation Results

The following section present some of the results when we vary the algorithm to use guiding vectors. The inputs to the algorithm are as follows:

- s number of sites
- i initial weight of sites
- p power to which we raise the dot product of the two vectors
- m maximum for the edge weight
- ϵ a value to prevent a divide by zero error
- γ the minimum edge weight used in the weight calculation
- β variation in degrees of change for the guiding vector at each edge weight calculation

Figure 4.5 illustrates how the size of the Voronoi regions are effected by varying i can produce.

Figure 4.5: Left to right, : $i = 5.0$, $i = 15.0$, $i = 50$

Figure 4.6 illustrates the effects of changing m . It can be observed in Figure 4.6 how the overall shape, size and edge irregularity can be altered by changing m . It is easy to note that as the value of m increases the regions elongate and decrease in size. This is because by limiting the maximum edge weight the region can consume nodes that would not belong to that particular Voronoi region otherwise.

Figure 4.6: Left to right, : $m = 0.1$, $m = 0.4$, $m = 0.8$, $m = 2.0$

Figure 4.6 illustrates the effects of changing p . It can be observed that increasing the p causing the resulting Voronoi regions to elongate.

Figure 4.7: Left to right, : $p = 1$, $p = 2$, $p = 7$, $m = 15$

Figure 4.8 illustrates how changes in γ directly affect the size of the resulting Voronoi regions. It can be observed that increasing the size of γ decreases the size of the resulting Voronoi regions.

The guiding vectors weight calculation can exhibit more structured output if the vector of the sites are more structured even while the locations of the sites remain random. We tested with three structured site assignments; the first had all the site's vectors assigned parallel values. The second structured site assignment pointed the vector of each site along a radial with each radial being $360/n$ degrees more than the last. The third structured site assignment had each site's vector pointing towards the centre of the texture. For convenience, we refer to these structure assignments as stratified, radial, and central respectively.

Figure 4.8: Left to right, : $\gamma = 0.005$, $\gamma = 0.050$, $\gamma = 0.5$

Additionally, it should be noted that we adjusted the Guided Vector Weight calculation slightly for use with these structured inputs. We no longer divide one by the dot product but subtract the dot product from one. Additionally, we now use ϵ as an addition to the edge weight as we no longer risk a divide by zero error. We also set β to zero avoiding perturbations of the vector and the m is unlimited. These changes are captured in the equation 4.1.

$$w_e = |\max(v_{i_x}, v_{i_y}, v_{i_z})| * (1 - (|(v_l \cdot v_g)^p| + \epsilon)) \quad (4.1)$$

In Figure 4.10 we can observe results of various structured site value assignments. The stratified assignment produces parallel regions as one might anticipate. The radial assignment produces regions that point in a variety of directions with some almost swirling around each other. The central assignment produces regions that radiate from the centre.

Figure 4.9: Left to right: stratified, radial, central, All: $s = 80$, $p = 1.0$, $\gamma = 0.5$, $i = 0$, $\beta = 0$

In Figure 4.10 the regions in all three images exhibit a subset of regions that pinch at the middle. To reduce this effect we changed the value of p . Figure 4.10 shows how we can reduce the undesirable pinching effect resulting in smoother regions.

Figure 4.10: Left to right: $p = 1.0$, $p = 7.0$, $p = 15.0$, All: $s = 80$, $p = 1.0$, $\gamma = 0.5$

We can also vary the shape quite significantly allowing for very interesting textures to be generated. Figure 4.11 show three images illustrating how varied the output can be when using this weight calculation. The leftmost image is non-stationary where the Voronoi regions closer to the center increase in density as they decrease in size. In this image, the sites were along the outer edge and had the structured site input pointed towards the centre using our central input. The values of ϵ and p restrict region growth so that they can only grow perpendicular to the site's vector and towards the centre of the image. In this image $\gamma = 0.2$, $p = 13.5$, $\epsilon = 0.01$ and $s = 40$. The middle image shows an implementation of our radial structured site value assignment. Again we can observe a non-stationary texture composed of a variety of non-uniform sized Voronoi regions. In this image $\gamma = 0.5$, $p = 15.0$, $\epsilon = 0.2$ and

$s = 93$. The rightmost image demonstrates another example of a texture generated using our central structured site value assignment. One can observe how different the shapes of the Voronoi regions are from the leftmost image while preserving the non-stationary quality of the texture. In this image $\gamma = 0.5, p = 15.0, \epsilon = 0.2$ and $s = 30$.

Figure 4.11: Three interesting outputs of our guided vector weight calculation

In the following subsection, we present some results generated using the accidental weight calculation.

4.1.3 Accidental Weight Calculation Results

The following section presents some results from the accidental weight calculation. As we described in the previous chapter the inputs to the algorithm are as follows:

- s number of sites
- γ the minimum edge weight used in the weight calculation
- μ variable used within the edge weight calculation

Figure 4.12 shows how the μ effects the output; all the equations used to generate the Voronoi diagrams in Figure 4.12 had the same 50 initial sites and used a γ of 0.5. As was explained in the previous chapter the distinct characteristics of the accidental weight calculation are primarily because of the use of two different edge weight calculations. The weight calculation that is introduced when expanding regions come in contact with each other uses μ as a primary determinant as to whether

Figure 4.12: Left to right, top to bottom: $\mu = 0.3$, $\mu = 0.4$, $\mu = 0.5$, $\mu = 0.6$, $\mu = 0.7$, $\mu = 0.8$, $\mu = 0.9$, $\mu = 1.0$

the current region will start to consume the adjacent region. The other significant determinant is the vector of the origination site defining the adjacent region.

Examining Figure 4.12 one can observe that when μ has a value of 0.3 the output is degenerate, containing no visible Voronoi regions. One would suspect the cause of this is because one site has expanded surrounding all other sites. This suspicion is confirmed in the output where μ is 0.4 and we can observe that a few sites that must contain almost perpendicular vectors produce visible regions. Correspondingly as μ increases, the regions become more uniform in size as the threshold of similarity between the vectors associated with the sites becomes smaller. One can also observe that when μ reaches a value of 1.0 that the output no longer exhibits the features characteristic of the accidental weight calculation as the weight can no longer reduce the path cost of a target node in an adjacent region.

We also examine the effects of changing γ in the accidental weight calculation. Figure 4.13 shows how the γ effects the output; all the equations used to generate the Voronoi diagrams in Figure 4.12 had the same 50 initial sites and used a μ of 0.7.

Examining the Voronoi diagrams in Figure 4.13 one can observe that both the irregularity of the regions and the size of the regions are affected by γ . One can observe that with our other values that when γ exceeds a value of 5.0 it has reached a degenerate state allowing for a single region to have consumed all the others. We cannot set γ below 0.0 as the Dijkstra's Algorithm does not allow for negative edge

Figure 4.13: Left to right, top to bottom: $\gamma = 0.0$, $\gamma = 0.1$, $\gamma = 0.3$, $\gamma = 0.9$, $\gamma = 1.9$, $\gamma = 2.5$, $\gamma = 4.0$, $\gamma = 5.0$

weights.

This weight calculation produces some textures that are suggestive of natural materials. The images in Figure 4.14 are reminiscent of marble. The observer can notice the veins running across the entire output; this non-stationary feature is generally not found in exemplar-based methods.

Figure 4.14: Two textures reminiscent of marble

The colour table can help alter the viewer's interpretation of the texture. Using a different colour table the textures from Figure 4.14 can be seen in Figure 4.15 as

Figure 4.15: Two textures suggestive of frost.

having a resemblance to frost as one might see on a window.

The algorithm is capable of making textures far more convincing of frost as can be seen in Figure 4.16. This texture has regions that radiate out with an almost fractal quality reminiscent of how frost forms. Also in this particular example, the number of gently rounded regions are reduced.

Figure 4.16: Another example of frost: $s = 90$, $\mu = 0.437$, $\gamma = 0.525$

The accidental weight calculation can produce a large variety of results with interesting features. The images in Figure 4.17 appear with diverse features. The leftmost image is composed of two distinct types of Voronoi regions; the first type is small and self-contained while the second is very large and encompass some of the smaller

regions. This results in a non-stationary texture. In this image the variables are $s = 12$, $\mu = 2.867$, $\gamma = 0.287$. The middle image from the figure exhibits elements of normal Voronoi diagram based textures and is very similar to the result from Worley [48] however there are large dark cracks that run from edge to edge again which is a non-stationary feature. In this image the variables are $s = 52$, $\mu = 0.104$, $\gamma = 0.889$. The image on the right of Figure 4.17 contains very similarly shaped Voronoi regions in a variety of sizes; the regions are so similarly shaped they almost have a fractal-like appearance. In the rightmost image the variables are $s = 97$, $\mu = 0.713$, $\gamma = 0.889$.

Figure 4.17: Three diverse image produced using the Accidental weight calculation

We continue our results chapter in the next section where we present the results of the algorithm chained together by our framework.

4.2 Combination Results

In the previous section, we presented some of the results individual runs of the algorithm can generate. In this section, we present what successive runs of the algorithm can produce.

The results we present in the following section were generated from a framework that automatically varied the inputs to our algorithm, the edge weight calculation, the number of times the algorithm ran, and the combination operations. We examined the results choosing the textures that had interesting characteristics. Finally, we tweaked the settings and present the results in this section.

4.2.1 Brown Rocks

The image in Figure 4.18 is the product of two runs of the algorithm. The first run used the guided vector edge weight calculation. This run of the algorithm is responsible for the longer Voronoi regions that are oriented towards the centre of the texture.

The second run uses the basic weight equation with a small maximum region size. The second run of the algorithm uses the replacement combination operation caused by the site associated with the new small areas to overwrite the existing site associations.

The colour tables used by the respective runs of the algorithm combine to give this texture a natural appearance of brown rocks with flecks of a golden mineral.

The most notable feature is the structured shape of the brown Voronoi regions from the guided vectors site weight calculation. The significantly different sizes and shapes of the Voronoi regions generated by the two runs of the algorithms are the next most notable features.

Figure 4.18: Variation 1: $s = 76$, $p = 7.0$, $m = 0.8$, $\gamma = 0.005$, $i = 5.0$; Variation 2: $s = 67$, $\gamma = 0.1$, max region size = 426

We now continue with the result we refer to as dry riverbed.

Brown Rocks						
Guided Vector	s = 76	$\gamma = 0.005$	$\beta = 0.01$	p = 7	i = 5.0	m = 0.8
ADDITIVE						
Basic	s = 67	$\gamma = 0.1$	n = 1	max = 426		

Table 4.2: Variations, parameter and combination operations used to generate Brown Rocks

4.2.2 Dry Riverbed

The image in Figure 4.19 bears some resemblance to an overhead view of a dry riverbed; the yellow-brown muck of the river drying and cracking and various sized stones normally at the bottom visible for all to see.

This image was constructed by three variations of the algorithm combined using our framework. The first variation was our basic algorithm and configured to produce a second-order Voronoi diagram. This first variation produces the yellow areas with the brown border. One can observe both larger Voronoi regions and the smaller regions that form lattice-like structures in higher orders. These two features combine to produce naturalistic features in the texture.

The second variation is the basic algorithm with the first order and maximum size restrictions. Additionally, the variation is configured to respect the region's boundaries from the previous variation which helps to confine the new region growth creating greater irregularity to their own boundaries. This variation produces the smaller grey Voronoi regions. This variation is combined using the replace combination operation allowing these new regions to directly overwrite existing regions.

The third variation is the same as the second but with the maximum size, restriction increased and again is restricted from crossing boundaries of either of the previous variations. This variation is responsible for the larger grey Voronoi regions.

This mixture of varying regions shapes and sizes would likely to be non-reproducible using Worley's [48] technique as it only uses a basis function formed from higher order Voronoi diagrams and would be unable to have regions of significantly differing characteristics. Mould's method could produce something akin to the yellow-brown areas using successive rounds of Voronoi diagram generation but would require a separate input texture to create the smaller Voronoi regions.

Figure 4.19: Variation 1: $s = 84$, $o = 2$, $\gamma = 0.175$; Variation 2: $s = 80$, $\gamma = 0.175$, max region size = 409; Variation 3: $s = 90$, $\gamma = 0.175$, max region size = 1640

Dry Riverbed				
Basic	$s = 84$	$\gamma = 0.1753$	$o = 2$	
ADDITIVE				
Basic	$s = 80$	$\gamma = 0.175$	$o = 1$	max = 409
SUBTRACTIVE				
Basic	$s = 90$	$\gamma = 0.175$	$o = 5$	max = 1640

Table 4.3: Variations, parameter and combination operations used to generate Dry Riverbed

The strategy of using successive region restricted first-order Voronoi diagrams could be used to generate textures resembling pebbles or walls of rubble.

We now move to a result that we refer to simply as Sparse.

4.2.3 Sparse

The image in Figure 4.20 is most notable for the large areas without detail punctuated by small isolated regions with high frequency detail. This combination is suggestive of fungal growth on a smooth surface.

This texture was formed by two variations. The first algorithm variation uses our a basic edge weight calculation with the fifth order Voronoi diagram selected as the output. This produces the green areas that host a large variety of differently sized

Voronoi regions.

The second variation uses the guided vector edge weight calculation. This variation is responsible for the boundaries of the green regions. This variation generated Voronoi regions with the irregular shape but since the second variation was using the subtract combination operation the Voronoi regions from the first variation are preserved where the second variation had generated Voronoi regions. The rest of the space has its site associations removed. This produces the featureless red area.

We use red to indicate areas in the space that are not associated with a site. The red area could become associated with further algorithm variations or we could have associated a colour table with it if we didn't want the red area to be featureless.

Worley's work might be incapable of similar output. There are examples of Worley's that have large featureless areas but they don't contain smaller areas with a lot of detail as can be seen in Figure 4.20. This type of output is unlikely to be achievable with Mould's work either; as Mould's technique divides the entire space into Voronoi regions.

Figure 4.20: Variation 1: $s = 60$, $\gamma = 0.413$, $o = 5$; Variation 2: $s = 31$, $\gamma = 0.005$, $\beta = 0.01$, $i = 15.0$, $p = 7.0$, $m = 0.8$

Next, we present another result that we refer to as sharp depicted in Figure 4.21.

4.2.4 Sharp

The image in Figure 4.21 has the appearance of a number of clusters of small sharp crystals embedded in another substance. The non-crystal areas run throughout the

Sparse						
Basic	s = 60	$\gamma = 0.413$	o = 5			
ADDITIVE						
Guided Vector	s = 76	$\gamma = 0.005$	$\beta = 0.01$	p = 7	i = 15.0	m = 0.8

Table 4.4: Variations, parameter and combination operations used to generate Sparse

entire texture in one non-stationary region. We left this texture in black and white to emphasize its sharp boundaries. This texture was produced by combining two variations.

The first variation is responsible for the boundary shape of the crystal clusters as well as the homogeneous area that surrounds them. This was done by the first variation use the basic edge weight calculation to produce a third order Voronoi diagram.

The second variation actually uses the same basic edge weight calculation to generate a first order Voronoi diagram but with boundary crossing restricted so that the new areas are confined to the first variations Voronoi regions. The clustering is caused when several sites from the second variation were seeded into a single Voronoi region from the first variation which caused the clustering that we observed. The second variation combined with the first variation using the replacement combination operation. This causes any point that was contained within a Voronoi region to be replaced by the association from the second variation.

The observer should note that it is the same edge weight calculation used in both variations and this result highlights the diversity that the framework enables.

While Worley's work [48] can produce straight edged Voronoi regions they don't produce the irregular almost serrated edges that can be seen in 4.21. Mould's work [28] using a similar approach as us namely a weighted graph can produce the irregular boundaries present in our Voronoi regions it is unable to produce the continuous region surrounding the clusters of irregular regions.

We now present the next result which we refer to as Variety.

Figure 4.21: Variation 1: $s = 70$, $\gamma = 0.731$, $o = 3$; Variation 2: $s = 88$, $\gamma = 0.347$, $o = 1$

Sharp			
Basic	$s = 70$	$\gamma = 0.731$	$o = 3$
REPLACEMENT			
Basic	$s = 88$	$\gamma = 0.347$	$o = 1$

Table 4.5: Variations, parameter and combination operations used to generate Sharp

4.2.5 Variety

The image in Figure 4.22 is notable for the variety of differently shaped regions in this texture. These differently shaped regions combine the output from four algorithm variations to produce another non-stationary texture.

The first variation uses the guided vectors edge weight calculation. This produces a number of irregularly shaped regions.

The second variation also uses the guided vectors edge weight calculation but we restrict the second variation from crossing boundaries from the input graph. We then use the replacement combination operation, this in combination with the boundary crossing restriction has two effects. First, the regions from the first variation are be subdivided when two or more sites from the second variation are confined within the same region. The second is that new Voronoi regions that grow outside the confines of the previous regions will be retained.

The third variation is the guided vectors with structured site locations and combines the output using the additive combination operation. The use of this combination algorithm ensures this algorithm variation only fills the unassigned space left from the previous algorithms. This variant generates some of the long centrally oriented regions one can observe in the output textures.

The fourth and final variation uses the basic edge weight calculation with a maximum region size. This variation is using the replacement operation which allows it to overwrite existing regions. This variation is responsible for the round white coloured areas in the output texture.

The diversity of shapes is unlikely to be reproducible by Mould's [28] method without a manually created image to guide the Voronoi generation away from its natural cellular shape. In the output we have observed, it would seem that Worley's [48] method does not mix Voronoi region shapes and thus is unlikely to reproduce an output similar to ours that has both round and elongated regions.

Figure 4.22: Variation 1: $s = 51$, $\gamma = 0.005$, $p = 7$, $i = 15$, $m = 0.8$ $\epsilon = 0.2$; Variation 2: $s = 70$, $\gamma = 0.005$, $p = 7$, $i = 15$, $m = 0.8$ $\epsilon = 0.2$; Variation 3: $s = 68$, $\gamma = 0.5$, $p = 15$, $\epsilon = 0.2$; ; Variation 4: $s = 28$, $\gamma = 0.2$, max region size = 3526

This concludes the section on combination results. The next section briefly highlights some of the selective site placement results.

Variety						
Guided Vector	s = 51	$\gamma = 0.005$	$\epsilon = 0.2$	p = 7.0	i = 15.0	m = 0.8
REPLACEMENT						
Guided Vector	s = 70	$\gamma = 0.005$	$\epsilon = 0.2$	p = 7	i = 15.0	m = 0.8
ADDITIVE						
Guided Vector	s = 68	$\gamma = 0.5$	$\epsilon = 0.2$	p = 7	i = 15.0	m = 0.8
SUBTRACTIVE						
Basic	s = 28	$\gamma = 0.2$	max = 3546			

Table 4.6: Variations, parameter and combination operations used to generate Variety

4.3 Selective Site Placement Results

As mentioned in the previous chapter using the output of algorithm variations to be input into successive variations allows for selective site placement. We do this by inspecting the input for specified characteristics and placing the sites of the subsequent algorithm in accordance with the selection criteria. This gives us a measure of control that we can use to influence the output texture.

The first placement technique is the one used to generate the images in Figure 4.23. This placement technique looks for paths along boundary edges from one side of the input graph to the other. In both the images, we have configured the selection to produce two paths. The start and end points were randomly selected from the available edges that terminate at the edge of an image, then the shortest path between these points was generated. All the nodes in the resulting paths become sites for the subsequent algorithm variation. The paths were selected from the edges of the regions of a previous variation which produces the Voronoi regions surrounding the cracks providing the brown and red Voronoi regions in the respective images. The subsequent algorithm variation used the basic algorithm weight edge weight calculation with maximum regions sizes to expand the paths into the white cracks in each image.

This technique allows us to add non-stationary features to the final texture. This is not something that Worley's [48] work is likely capable of producing. Mould's [28] could certainly provide similar features by adding this feature to the image used to guide the fractures, this would be manually done however and changes in the number

of cracks would require the manual image to be edited.

Figure 4.23: Both: Two paths selected as sites and widened into cracks

As mentioned in the previous chapter we can identify regions, edges and confluences in a Voronoi diagram contained within an input graph and our placement technique allow selection of edges, regions and confluence.

The images in Figure 4.24 illustrates selections of regions of regions and edges. It is easily observed that both images share the same input graph. The left image is a product of an algorithm that selects the two largest Voronoi regions from the input graph and then places ten sites within each region. The regions are then grown using the basic edge weight calculation with a maximum region size. The image on the right was configured so that the two longest regions are selected and the nodes of the respective edges are set as sites. The site is then used in an algorithm variation using the basic edge weight calculation with a maximum region size.

These images demonstrate a level of user direction not available using Worley's method [48]. Mould's method [28] automatically widens the cracks corresponding to their length but does not allow for the specification of the number of edges to widen due to its indiscriminate nature. Mould's method does allow for the selection of Voronoi regions for further partitioning, however, our framework allows for any of our algorithm variations to be used post-selection allowing for more a variety of alternative to simple Voronoi diagram partitioning.

To further demonstrate the non-stationary properties that selective site placement can achieve we present Figures 4.25 and 4.26. The images in Figure 4.26 consist

Figure 4.24: Left: Two regions selected to be seeded with sites, Right: two edges selected to be sites

of an input Voronoi diagram on the left and the output Voronoi diagram from a successive algorithm on the right. The input Voronoi diagram was used for site selection, with the largest two regions being selected two site placements and the third and fourth largest regions selected for forty site placements. To generate the output Voronoi diagram in the image on the right we used the basic algorithm with the region confinement variation as the successive algorithm. We can examine this non-stationary output; see how the larger two regions were bisected by the new Voronoi regions generated within their confines and the two smaller regions have become quite fragmented by the forty regions generated within them. The result in the second image is very close to Mould's [28] approach as both used a first order Voronoi diagram for the new regions. Our approach allows us to produce far more diverse outputs as can be seen in Figure 4.26.

In Figure 4.26 we present several outputs of an algorithm variation that is unconfined and uses selective site placement. To generate these three images we use the input from 4.25, but we remove the region confinement variation. The same selective site placement was used causing the same number of sites to be placed in the same regions from the input Voronoi diagram. This was done for each of the output images, for which each of the successive algorithms was configured to generate either the first, second or third order Voronoi diagrams in the respective generations. In these images, we can see the selective site placement and the higher order Voronoi diagram calculation combine to produce non-cellular images that are highly non-stationary.

Figure 4.25: Left: Input Voronoi diagram, Right: Region confined output

These images are also very distinct from what Mould's [28] technique can generate without an input texture.

Figure 4.26: Left: Unconfined 1st Order Voronoi diagram, Middle: Unconfined 2nd Order Voronoi diagram Right: Unconfined 3rd Order Voronoi

In the previous chapter, we defined confluences as a node that has neighbours associated with three or more unique regions. Figure 4.27 demonstrates this using confluence for site placement. The left image depicts a first order Voronoi diagram we produced from a variation of our algorithm that is passed to a successive algorithm variation. The middle image shows the sites that were the confluence of the leftmost image and the resulting regions from the successive algorithm variation. The rightmost image shows the texture produced from the two algorithm variations depicted. One can observe how the Voronoi regions elongated and present more diversity in the

Figure 4.27: Left: Input regions, Middle: Confluence based regions, Right: Final output

sizes of the regions after using the confluence based site placement technique.

Figure 4.28 illustrates the same technique with a larger number of initial sites. This again produces some promising variation in both shape and size of the regions but the critical observer would note that the output is beginning to resemble higher order Voronoi Diagrams.

Figure 4.28: Left: Input regions, Middle: Confluence based regions, Right: Final output

Figure 4.29 explores the similarity of our successive approach using confluence based site placement. The leftmost image shows the results of our algorithm when we generate a third order Voronoi diagram using sites that were the confluences of a first-order Voronoi diagram. The middle image is a third-order Voronoi diagram. The rightmost image is a fifth-order Voronoi diagram. One can observe that the image of the confluence based site placement more closely resembles the fifth-order Voronoi diagram than the third-order. Upon examination, the confluence site placement didn't produce characteristics that a simple higher-order Voronoi diagram wouldn't produce

and we felt additional experimentation unwarranted.

Figure 4.29: Left: Confluence based third-order Voronoi diagram, Middle: Third-order Voronoi diagram, Right: Fifth-order Voronoi diagram

The next section will present some results exhibiting features approximating those found in real rocks and minerals.

4.4 Naturalistic Results

Our algorithm allows us to produce natural looking rock and mineral textures. This section presents several images of natural stones that present interesting features. Alongside these images, we present some of the textures our framework has generated that contains some similar features. These textures were generated by first looking through results from our framework to find textures that exhibited features similar to the chosen minerals. We then manually varied the inputs to the framework after inspecting the output textures to improve the results. In this section, we will present three natural stones alongside our texture analogues, Mahogany Obsidian, Oribicular Jasper and Ryolite.

Figure 4.30 shows a piece of Mahogany Obsidian and a texture from our algorithm. Mahogany Obsidian produces irregularly shaped regions in a broad variety of sizes. These regions are one of two colours; dull red and deep dark brown heading towards black. Our texture shows similar irregularity along the edges of the regions. Our texture also exhibits similar differences in sizes of the regions with some being very small and others being quite large. Finally, our framework allowed us to divide the regions almost into red and black much like the actual stone. One unfortunate side

effect of our technique is that grey boundary edges visible in the read areas of the texture, this does not appear in the actual stone.

Figure 4.30: Left: Mahogany Obsidian, Right: Texture approximating Mahogany Obsidian

Table 4.7 presents the algorithm variations we used to generate the Orbicular Jasper texture in Figure 4.30. Each row presents either the weight calculation variation and accompanying parameter or the combination operation that was used to combine the successive algorithm variation with its predecessor.

Mahogany Obsidian				
Accidental	$s = 30$	$\gamma = 0.5$	$\mu = 0.3$	
REPLACEMENT				
Basic	$s = 9$	$\gamma = 0.1$	$n = 1$	$\max = 60000$

Table 4.7: Variations, parameters and combination operations used to generate Mahogany Obsidian

4.31 shows Orbicular Jasper. Orbicular Jasper is an interesting stone because of the “round” or “orbicular” regions of minerals that form within the stone. These regions appear as concentric rings of different coloured minerals. Additionally one can see irregularly shaped regions. These features of Orbicular Jasper can be seen in Figure 4.31 alongside a texture approximation generated by our algorithm. In our texture, one can observe similar “orbicular” regions and irregularity of the regions.

One can observe that the light grey areas in the Orbicular Jasper are noisier while in our texture they are reproduced as a smooth gradient.

Figure 4.31: Left: Orbicular Jasper, Right: Texture approximating Orbicular Jasper

Table 4.8 presents the algorithm variations we used to generate the Orbicular Jasper texture in Figure 4.31. Each row presents either the weight calculation variation and accompanying parameter or the combination operation that was used to combine the successive algorithm variation with its predecessor.

Orbicular Jasper						
Basic	s = 43	$\gamma = 0.1$	n = 1	max = 2687		
ADDITIVE						
Guided Vector	s = 10	$\gamma = 0.005$	$\beta = 0.01$	p = 7	i = 15	m = 0.8
REPLACEMENT						
Accidental	s = 30	$\gamma = 0.8$	$\mu = 3.46$			
REPLACEMENT						
Basic	s = 23	$\gamma = 0.1$	n = 1	max = 6000		

Table 4.8: Variations, parameter and combination operations used to generate Orbicular Jasper

Rhyolite is shown in 4.32. Rhyolite has irregularly shaped patches of alternate coloured mineral throughout, additionally, these patches have a white boundary. The

same features can be seen in the output texture in the right image of the figure. One failure to note is how in the real mineral the dark green areas have higher frequency yellow spots. Our technique reproduces these areas as ridges following the midway point between the smaller regions.

Figure 4.32: Left: Rhyolite, Right: Texture approximating Rhyolite

Table 4.9 presents the algorithm variations we used to generate the Orbicular Jasper texture in Figure 4.32. Each row presents either the weight calculation variation and accompanying parameter or the combination operation that was used to combine the successive algorithm variation with its predecessor.

Rhyolite						
Guided Vector	$s = 59$	$\gamma = 0.005$	$\beta = 0.01$	$p = 7$	$i = 15$	$m = 0.8$
SUBTRACTIVE						
Basic	$s = 44$	$\gamma = 0.1$	$n = 1$	$\max = 927$		
REPLACEMENT						
Guided Vector	$s = 23$	$\gamma = 0.005$	$\beta = 0.01$	$p = 7$	$i = 15$	$m = 0.8$
ADDITIVE						
Basic	$s = 9$	$\gamma = 0.1$	$n = 1$	$\max = 60000$		

Table 4.9: Variations, parameter and combination operations used to generate Rhyolite

4.5 Tiling and alternative texture application

This section briefly discusses our algorithms capability to generate tile-able textures. The Voronoi diagrams elsewhere in this document have not been tile-able. In these examples one can observe how the regions along the edges of the texture are unbound and don't line up with the regions along their opposite edges. This is due to the graph with which we generated the Voronoi diagram. One can visualize the graph as a rectangle and can conclude that the nodes along the edges are not four-connected. However to make a tile-able texture we just need to connect the sides of the rectangle together so that we will have only four-connected nodes. Our algorithm still terminates as all conditions still hold and we will have regions that are not unbound along the edges of the graph. If we then treat the nodes in the graph as a rectangle again we will have a texture that can be tiled.

This behaviour can be seen in Figure 4.33 where we show a Voronoi diagram that has been tiled into a cross.

Figure 4.33: Tileable Voronoi diagram

Tiling is one way to avoid discontinuities as we enlarge a texture or for mapping to simple geometry but another issue is distortion. Distortion can occur when we map a texture to arbitrary geometry. Our texture synthesis method could be applied to arbitrary geometry if we distribute the graph nodes densely over the surface of

the model. Once our graph was in place we could essentially generate our Voronoi diagram on the surface of the model. After that mapping the resulting texture to the model would be trivial and distortion free.

4.6 Alternative Colourization

This section briefly presents one alternative to our distance-based colourization which we refer to as multi-texturing. We investigated using textures the same size as our output texture to provide the colour look-up table for the rendering stage of the algorithm. We used a different texture for each algorithm variation; this allows the regions produced by a given algorithm variation to correspond to a pixel in one of the input textures. Our method with this alternative colourization could be considered a hybrid with example-based methods.

Figure 4.34 shows some result output textures. The same algorithm variations and inputs as we used for the Mahogany Obsidian texture were used in generating all three of the images. The left and right images both resemble stone composed of two distinct minerals. The central image resembles an aerial view of a body of water surround by a rocky shoreline. One benefit of this approach is the regions from the different algorithm variations don't exhibit regions boundary artifacts as we saw in Figure 4.30. Another benefit is that we can get high-frequency areas in the output texture that our algorithm otherwise isn't capable of producing.

Figure 4.34: Three multi-texture outputs

4.7 User Guide

In this section, we include some coarse guidelines on how to use our framework.

- If you require small discrete regions, use the basic algorithm with the region confinement variation.
- If you need elongated regions, use the guided vector weight calculation.
- If you desire regions that all point in a single direction, use the guided vector alternate weight calculation.
- If you need cracks running the entirety of the texture, you can use the selective site placement with a preceding algorithm.
- If you need to have a localized density of regions, use selective site placement on a successive algorithm having selected a suitable number of regions and use region confinement variation.
- If you need varying sizes of regions, either use weighted sites or the accidental algorithm alternate weight calculation

More ideas can be found by examining the examples in the previous sections.

We briefly summarize some of the limitations we have already described in the following section.

4.8 Limitations

Our algorithm is unable to reproduce high-frequency mineral structure such a granite an example of which is shown in Figure 4.35. These types of textures are more suited to other techniques such as wavelet-based micro texture techniques. It is possible that changing the colourization technique that we use in the rendering stage could add higher frequency noise as opposed to our distance-based colour look-up table.

In our algorithm, the placement of colours is a design decision without much granularity as currently colour tables are applied per algorithm variation. The implication of this is that every region produced by a particular algorithm variation will use the

Figure 4.35: Example of high frequency mineral structure: white granite

same colour table. Some variations produce diverse enough regions that it would be preferable to colour a subset of them differently.

Our algorithm provides a number of parameters that can be used to generate a variety of effects. However, selecting and configuring the parameters when trying to produce a specific effect can be quite difficult.

The combination operations that we developed to combine the outputs of the algorithm variations are simple to understand and flexible enough to provide diverse effects on the output texture. There are times where it is insufficient. An example of this would be if we wanted to combine all the regions into a single region to remove the boundary artifacts visible in the final output.

Chapter 5

Conclusions

5.1 Summary

We presented a method capable of generating rock textures containing non-stationary textures. We used a four-connected graph and selected a subset of the nodes as sites. We then generated a Voronoi diagram using the geodesic distance rather than the Euclidian distance. This was done with a novel weight equation for use in calculating the geodesic distance. We also developed a framework that allows for successive variations of the algorithm to be combined. In addition to combining the successive outputs, we allow for the output of a predecessor algorithm variation to be used as the input to the successive algorithm variation. Finally, we generate the texture using a colour table and a function to determine the distance of the pixel in question from the region's boundary.

We distinguish our work with its immediate predecessors; Worley's *Cellular Basis Function* [48] and Mould's *Guided Fractures* [28].

Worley's *Cellular Basis Function* [48] produces textures that are characterized with either smooth or straight boundary regions while our method can create irregular boundaries and have a distinctly non-cellular appearance. Our method is capable of produce long thin regions and non-stationary features that can run the width of the texture.

Mould's *Guided Fractures* [28] is unable to produce non-stationary features without an input texture to influence the output. Mould also only used first-order Voronoi diagrams limiting the regions shapes the standard cellular region shapes easily recognize in Voronoi diagrams, this is particularly useful for modelling cracks in the pavement or drying mud. Our method, due to our weight calculation and our use of

higher order Voronoi diagrams produces a significant increase in the variety of the region shapes. This allows us to produce textures with the appearance of a broader variety of stones and minerals.

While our method conforms to several of the characteristics of a good procedural texture synthesis algorithm: the storage of our method is small; all we need are the sites and the colour table, and it is non-periodic. One drawback is that our current implementation is slow. We are confident our method could be sped up but that was not a focus of our research.

We are able to create textures that are analogues of real minerals and to produce non-stationary textures.

5.2 Future Work

Our course our research leaves areas for further exploration. While our work has been solely R^2 it could easily be extended to R^3 . Due to speed constraints, a sparse graph could also be explored at that time. This could result in improved performance and could be used as a *solid* texture. As our method is discretized we would also need to be able to interpolate between the nodes of the graph to be used as a solid texture.

We demonstrated how Voronoi regions produced by our method could be mapped to textures. In the example, we provided the regions from each algorithm were mapped to a different texture. As this is analogous to label based texture methods such as the machine learning method Pix2Pix [19] we anticipate our method could be used in conjunction with machine learning methods.

We also feel like the distributions of both sites and the initial properties could be explored producing more diverse outputs.

Finally, our selection based inputs could be extended in a number of ways. We could remove all regions without favourable characteristics before passing it to the successive algorithm variation. We feel this would also allow for more diverse outputs and provide more control to the user.

List of References

- [1] *Poisson disc sampling in Python*, Apr 2017. [Online]. Available: <https://scipython.com/blog/poisson-disc-sampling-in-python/>
- [2] M. Ashikhmin, “Synthesizing natural textures,” in *Proceedings of the 2001 symposium on Interactive 3D graphics*. ACM, 2001, pp. 217–226.
- [3] X. Bai and G. Sapiro, “Distancecut: Interactive segmentation and matting of images and videos,” in *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, vol. 2. IEEE, 2007, pp. II–249.
- [4] G. Borgefors, “Distance transformations in digital images,” *Computer vision, graphics, and image processing*, vol. 34, no. 3, pp. 344–371, 1986.
- [5] R. Bridson, “Fast poisson disk sampling in arbitrary dimensions.” in *SIGGRAPH sketches*, 2007, p. 22.
- [6] J. Caron and D. Mould, “Partition of unity parametrics for texture synthesis,” in *Proceedings of Graphics Interface 2013*. Canadian Information Processing Society, 2013, pp. 173–179.
- [7] E. Choy, “Smooth Signed Distance Field Textures,” Master’s thesis, Carleton University, Ottawa, Ontario, 2016.
- [8] R. L. Cook and T. DeRose, “Wavelet noise,” in *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3. ACM, 2005, pp. 803–811.
- [9] A. Criminisi, T. Sharp, C. Rother, and P. P’erez, “Geodesic image and video editing,” *ACM Trans. Graph.*, vol. 29, no. 5, pp. 134:1–134:15, Nov. 2010.
- [10] J. S. De Bonet, “Multiresolution sampling procedure for analysis and synthesis of texture images,” in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 1997, pp. 361–368.
- [11] A. A. Efros and W. T. Freeman, “Image quilting for texture synthesis and transfer,” in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001, pp. 341–346.
- [12] A. A. Efros and T. K. Leung, “Texture synthesis by non-parametric sampling,” in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 2. IEEE, 1999, pp. 1033–1038.

- [13] M. Erwig, “The graph voronoi diagram with applications,” *Networks: An International Journal*, vol. 36, no. 3, pp. 156–163, 2000.
- [14] I. Fischer and C. Gotsman, “Fast approximation of high-order voronoi diagrams and distance transforms on the gpu,” *Journal of Graphics Tools*, vol. 11, no. 4, pp. 39–60, 2006.
- [15] A. R. Fuller, H. Krishnan, K. Mahrous, B. Hamann, and K. I. Joy, “Real-time procedural volumetric fire,” in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. ACM, 2007, pp. 175–180.
- [16] B. Galerne, Y. Gousseau, and J.-M. Morel, “Micro-texture synthesis by phase randomization,” *Image processing on line*, vol. 1, pp. 213–237, 2011.
- [17] M. T. Goodrich and R. Tamassia, “Algorithm design,” *Wiley India*, 2002.
- [18] D. J. Heeger and J. R. Bergen, “Pyramid-based texture analysis/synthesis,” in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. ACM, 1995, pp. 229–238.
- [19] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-image translation with conditional adversarial networks,” *arxiv*, 2016.
- [20] N. Jetchev, U. Bergmann, and R. Vollgraf, “Texture synthesis with spatial generative adversarial networks,” *CoRR*, vol. abs/1611.08207, 2016.
- [21] A. Kensler, A. Knoll, and P. Shirley, “Better gradient noise,” in *Tech. Rep. UUSCI-2008-001, SCI Institute*, 2008.
- [22] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick, “Graphcut textures: image and video synthesis using graph cuts,” *ACM Transactions on Graphics (ToG)*, vol. 22, no. 3, pp. 277–286, 2003.
- [23] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D. S. Ebert, J. P. Lewis, K. Perlin, and M. Zwicker, “A survey of procedural noise functions,” in *Computer Graphics Forum*, vol. 29, no. 8. Wiley Online Library, 2010, pp. 2579–2600.
- [24] A. Lagae, S. Lefebvre, G. Drettakis, and P. Dutré, “Procedural noise using sparse gabor convolution,” *ACM Transactions on Graphics (TOG)*, vol. 28, no. 3, p. 54, 2009.
- [25] J.-P. Lewis, “Algorithms for solid noise synthesis,” in *ACM SIGGRAPH Computer Graphics*, vol. 23, no. 3. ACM, 1989, pp. 263–270.
- [26] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum, “Real-time texture synthesis by patch-based sampling,” *ACM Transactions on Graphics (ToG)*, vol. 20, no. 3, pp. 127–150, 2001.
- [27] Y. Liu, W.-C. Lin, and J. Hays, “Near-regular texture analysis and manipulation,” *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 368–376, 2004.

- [28] D. Mould, “Image-guided fracture,” in *Proceedings of Graphics Interface 2005*. Canadian Human-Computer Communications Society, 2005, pp. 219–226.
- [29] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu, *Spatial tessellations: concepts and applications of Voronoi diagrams*. John Wiley & Sons, 2009, vol. 501.
- [30] K. Perlin, “An image synthesizer,” *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.
- [31] ———, “Improving noise,” in *ACM transactions on graphics (TOG)*, vol. 21, no. 3. ACM, 2002, pp. 681–682.
- [32] J. Portilla and E. P. Simoncelli, “A parametric texture model based on joint statistics of complex wavelet coefficients,” *Int. J. Comput. Vision*, vol. 40, no. 1, pp. 49–70, Oct. 2000.
- [33] E. Praun, A. Finkelstein, and H. Hoppe, “Lapped textures,” in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 2000, pp. 465–470.
- [34] Y. Qi, X.-K. Shen, M.-Y. Duan, and H.-L. Cheng, “A method of rendering clouds with perlin noise,” *Acta Simulata Systematica Sinica*, vol. 9, 2002.
- [35] A. Radford, L. Metz, and S. Chintala, “Unsupervised representation learning with deep convolutional generative adversarial networks,” *CoRR*, vol. abs/1511.06434, 2016.
- [36] M. Sabha, P. Peers, and P. Dutré, “Texture synthesis using exact neighborhood matching,” in *Computer Graphics Forum*, vol. 26, no. 2. Wiley Online Library, 2007, pp. 131–142.
- [37] E. P. Simoncelli and J. Portilla, “Texture characterization via joint statistics of wavelet coefficient magnitudes,” in *Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*, vol. 1. IEEE, 1998, pp. 62–66.
- [38] K. Suffern, *Ray Tracing from the Ground up*. AK Peters/CRC Press, 2016.
- [39] G. Tartavel, Y. Gousseau, and G. Peyré, “Variational texture synthesis with sparsity and spectrum constraints,” *Journal of Mathematical Imaging and Vision*, vol. 52, no. 1, pp. 124–144, 2015.
- [40] X. Tong, J. Zhang, L. Liu, X. Wang, B. Guo, and H.-Y. Shum, “Synthesis of bidirectional texture functions on arbitrary surfaces,” in *ACM Transactions on Graphics (ToG)*, vol. 21, no. 3. ACM, 2002, pp. 665–672.
- [41] G. Turk, “Generating textures on arbitrary surfaces using reaction-diffusion,” in *ACM SIGGRAPH Computer Graphics*, vol. 25, no. 4. ACM, 1991, pp. 289–298.
- [42] D. Ulyanov, V. Lebedev, A. Vedaldi, and V. Lempitsky, “Texture networks: Feed-forward synthesis of textures and stylized images,” in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML’16. JMLR.org, 2016, pp. 1349–1357. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3045390.3045533>

- [43] D. Ulyanov, A. Vedaldi, and V. Lempitsky, “Improved texture networks: Maximizing quality and diversity in feed-forward stylization and texture synthesis,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6924–6932.
- [44] J. J. Van Wijk, “Spot noise texture synthesis for data visualization,” *ACM Siggraph Computer Graphics*, vol. 25, no. 4, pp. 309–318, 1991.
- [45] L.-Y. Wei and M. Levoy, “Fast texture synthesis using tree-structured vector quantization,” in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 2000, pp. 479–488.
- [46] Wikipedia, “Procedural texture — Wikipedia, the free encyclopedia,” <http://en.wikipedia.org/w/index.php?title=Procedural%20texture&oldid=885897496>, 2019, [Online; accessed 17-March-2019].
- [47] A. Witkin and M. Kass, “Reaction-diffusion textures,” *ACM Siggraph Computer Graphics*, vol. 25, no. 4, pp. 299–308, 1991.
- [48] S. Worley, “A cellular texture basis function,” in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, 1996, pp. 291–294.
- [49] Q. Wu and Y. Yu, “Feature matching and deformation for texture synthesis,” *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 364–367, 2004.
- [50] L. Xu and D. Mould, “Procedural tree modeling with guiding vectors,” in *Computer Graphics Forum*, vol. 34, no. 7. Wiley Online Library, 2015, pp. 47–56.
- [51] Y. Zhou, Z. Zhu, X. Bai, D. Lischinski, D. Cohen-Or, and H. Huang, “Non-stationary texture synthesis by adversarial expansion,” *ACM Transactions on Graphics (Proc. SIGGRAPH)*, vol. 37, no. 4, pp. 49:1–49:13, 2018.

Appendix A

Image Gallery

This appendix provides some other interesting texture output.