METHODS FOR CREATING CORNER COLORED WANG TILES

By

MICHAEL JOSEPH PERSONS

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Engineering and Computer Science

May 2010

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of MICHAEL JOSEPH PERSONS find it satisfactory and recommend that it be accepted.

Wayne O. Cochran, Ph.D., Chair

Scott A. Wallace, Ph.D.

Orest J. Pilskalns, Ph.D.

METHODS FOR CREATING CORNER COLORED WANG TILES

Abstract

by Michael Joseph Persons, M.S.
Washington State University
May 2010

Chair: Wayne O. Cochran

Corner and edge colored *Wang tiles* are used in computer graphics to synthesize large seamless and non-periodic textures while reducing the storage requirements that expansive textures require. Wang tiles have matching constraints that dictate which tiles may be placed next to each other to ensure a seamless transition between adjacent tiles; while using a finite set of tiles for use in tiling limits the storage required to generate texture. The most popular technique in the current literature for constructing Wang tiles is to combine samples from an image and perform a *graph cut* between four abutted samples and an overlay sample where the graph cut should find the cut that selects pixels that creates the least objectionable tile. This thesis introduces new methods for generating corner colored Wang tiles and discusses the important characteristics in generating them. *Synthesized Wang Tiles* (SWT) use a controlled texture synthesis algorithm for tile construction. *Bilinear Blended Wang Tiles* (BLWTiles) use blending for tile construction and *Gauss Laplace Pyramid Blended Wang Tiles* (GLWTiles) use blending over different levels of Laplacian pyramids for tile construction. BLWTiles and GLWTiles are evaluated procedurally and can be mapped to GPU evaluation, while the SWTile method is intended to precompute a set of Wang tiles. The BLWTile, GLWTile, and SWTile techniques can supplement existing methods for creating Wang tiles. The characteristics that influence the quality of corner colored Wang tiles can serve to improve new and existing tile creation techniques.

Table of Contents

Abstract **iii**

List of Tables **vii**

List of Figures **viii**

1 Introduction **1**

1.1 Problem Domain 1

1.2 Problem Statement 1

1.3 Existing Solutions 2

1.4 Thesis Contributions 6

2 Background **8**

2.1 Texture 8

2.2 Image Quilting 10

2.3 Wang Tiles 11

2.3.1 Cohen's Graph Cut Wang Tiles 13

2.3.2 ω-Tiles and "corner" tiles 16

2.4 Texture Synthesis 17

2.4.1 Kwatra's Texture Optimization 17

2.5 Gauss Laplace Pyramid Blending 19

3 BLWTiles **22**

3.1 Hypothesis 22

3.2 Initial Approach 23

3.2.1 Colors 23

3.2.2 Tile Construction 25

3.2.3 Tiling The Plane 28

3.3 Methods Investigated 30

3.3.1 Interpolation 30

3.4 Results 31

3.5 Analysis 40

3.6 Conclusion 40

4 GLWTiles **42**

4.1 Hypothesis 42

4.2 Initial Approach 43

4.3 Methods Investigated 49

4.3.1 Increasing Blending Region Size 50

4.3.2 Per Pixel Reconstruction for Gauss Laplace Pyramid 51

4.4 Results 53

4.5 Analysis 60

4.6 Conclusion 60

5 SWTiles **61**

5.1 Hypothesis 61

5.2 Initial Approach 62

5.2.1 Constructing an Example Tile $t0110$ . . . 63
5.3 Methods Investigated . . . 69
5.3.1 Honoring Immutables . . . 69
5.3.2 Partially Immutable Pixels . . . 71
5.3.3 Synthesizing Wang Tiles with Padding . . . 74
5.3.4 Introducing Diversity: Random Middles . . . 74
5.3.5 Revisiting Color Selection . . . 75
5.3.6 Regular Textures . . . 76
5.4 Results . . . 79
5.5 Analysis . . . 96
5.6 Conclusion . . . 96

6 Key Aspects for Corner Colored Wang Tiles **98**
6.1 Feature Size . . . 98
6.2 Alignment . . . 99
6.3 Diversity . . . 99
6.4 Abnormal Features . . . 100
6.5 Exemplar Quality . . . 100

7 Conclusion **102**
7.1 Future Work . . . 103

Bibliography **104**

List of Tables

4.1 GLWTile blending formulas: 2 pixel overlap 49

4.2 GLWTile blending formulas: more than 2 pixel overlap 51

5.1 Varying immutable percentage over neighborhood size 72

5.2 Varying immutable percentage over resolution and neighborhood size 73

List of Figures

1.1 Sample edge colored Wang tiles . . . 3

1.2 Valid and Invalid tilings: edge colored tiles . . . 4

1.3 Sample edge colored Wang tiles . . . 4

1.4 Valid and Invalid tilings: corner colored tiles . . . 5

2.1 Spectrum of Texture Classification: regular to stochastic . . . 9

2.2 Globally varying textures . . . 9

2.3 Graph cut between two overlapping image samples . . . 10

2.4 Colored Edges and Colored Corners . . . 12

2.5 Lagae's toroidal packing of corner colored tiles . . . 14

2.6 Cohen's Wang tile construction . . . 15

2.7 Texture tiled with Cohen's Wang tiles . . . 15

2.8 ω-Tile and "corner" tile construction . . . 16

2.9 Example of Gauss Laplace pyramid blending . . . 21

3.1 Creating color samples from a source image . . . 24

3.2 Checking the range of the color columns. . . . 25

3.3 Numbering corners of a BLWTile . . . 25

3.4 Applying a color sample's quadrant to a corner . . . 26

3.5 BLWTile construction: overlapped colors . . . 27

3.6 BLWTile u, v coordinate systems . . . 28
3.7 Graph of interpolation functions . . . 30
3.8 Closeups of interpolation functions . . . 31
3.9 Bilinear “Glow” . . . 32
3.10 “MonaCanvas” source . . . 32
3.11 “MonaCanvas” BLWTile results . . . 33
3.12 “MonaCanvas2” source and BLWTile results . . . 34
3.13 “Slate” source and BLWTile results . . . 35
3.14 “Shell Stone Tile” source . . . 36
3.15 “Shell Stone Tile” BLWTile results . . . 37
3.16 Source and BLWTile results 1 . . . 38
3.17 Source and BLWTile results 2 . . . 39
3.18 Source and BLWTile result: failure . . . 40

4.1 Blending Regions . . . 44
4.2 Gauss Laplace pyramids - Leafs . . . 46
4.3 Gauss Laplace pyramid - Stones . . . 47
4.4 Fixed size Gauss Laplace blending region . . . 48
4.5 Numbering Gauss Laplace blending regions . . . 48
4.6 Gauss Laplace pyramid pixels required at level N . . . 52
4.7 “MonaCanvas” GLWTile results: 8×8 grid . . . 53
4.8 “MonaCanvas” GLWTile results: 16×16 grid . . . 54
4.9 “MonaCanvas2” GLWTile results: 4×4 grid . . . 54
4.10 “MonaCanvas2” GLWTile results: 8×8 grid . . . 55
4.11 “MonaCanvas2” GLWTile results . . . 55
4.12 “Slate” GLWTile results: 8×8 grid . . . 56

4.13 “Slate” GLWTile results: 16×16 grid 56

4.14 “Shell Stone Tile” GLWTile results: 8×8 grid 57

4.15 “Shell Stone Tile” GLWTile results: 16×16 grid 57

4.16 Source and GLWTile results 1 58

4.17 Source and GLWTile results 2 59

5.1 SWTiles - Selecting two colors 64

5.2 Constructing horizontal edge $h00$ 65

5.3 Constructing horizontal edge $h11$ 65

5.4 Constructing vertical edge $v01$ 66

5.5 Constructing a SWTile 66

5.6 Cutting out a SWTile 67

5.7 Two color horizontal edge construction 68

5.8 Two color vertical edge construction 68

5.9 Edge and tile immutable regions 71

5.10 Sampling colors for diversity 75

5.11 Adjusting tile size for alignment 76

5.12 Initialization with aligned samples 77

5.13 Alignment for Sampling Colors 77

5.14 Adjusting random samples for alignment 78

5.15 “escher2” SWTile results 80

5.16 “escher2” comparison of SWTile and Kwatra’s results 81

5.17 “bricks2” SWTile results 82

5.18 “bricks2” comparison of SWTile and Kwatra’s results 83

5.19 “strawberries2” SWTile results 84

5.20 “strawberries2” comparison of SWTile and Kwatra’s results 85

5.21 “olives” SWTile results 86
5.22 “olives” comparison of SWTile and Kwatra’s results 87
5.23 “stone” SWTile results 88
5.24 “stone” comparison of SWTile and Kwatra’s results 89
5.25 “tomatoes” SWTile results 90
5.26 “tomatoes” comparison of SWTile and Kwatra’s results 91
5.27 “people” SWTile results 92
5.28 “people” comparison of SWTile and Kwatra’s results 93
5.29 “choc_scale” SWTile results 94
5.30 “choc_scale” comparison of SWTile and Kwatra’s results 95

Chapter 1

Introduction

1.1 Problem Domain

In computer graphics, 2D "textures" are images that are mapped onto a 3D object's surface which enhances the object's realism. The texture typically provides fine color detail involving repetition of similar elements or intricate patterns. A large number of textures are used in most video games as well as computer animated films. Without textures, computer generated surfaces appear bland even when shading from synthesized light sources is applied.

1.2 Problem Statement

The ability of texture to enhance a video game or film has increased the end user's expectation for nearly every scene to incorporate numerous quality textures. Yet, given the demand for quality texture, the ability of content creators to generate textures for large environments (such as a virtual world) while keeping the textures realistic and economic memory-wise has proven difficult. Even for smaller environments such as a scene in a coffee shop, there exist surfaces or objects that either require large textures (such as a tile or stone floor or a brick wall) or require many seemingly

unique yet similar textures (such as a tray of cookies or muffins).

Even if a content creator is able to manually create the quantity (and quality) of texture required, the storage of the texture then becomes demanding. Storage may be relatively “cheap”, yet having to store large amounts of texture implies that the texture must be retrieved for processing. Although current rendering systems have access to increasingly large amounts of RAM, both video cards and rendering systems have RAM limits which leads to an increase in the processing time required due to swapping textures in and out of memory.

1.3 Existing Solutions

Current methods commonly used to address the problems of texture creation and storage include edge and corner colored Wang tiles, procedural generation (including Perlin noise), and texture synthesis.

When using Wang tiles to create texture, a larger texture plane is created with a finite set of tiles containing texture by mapping the tiles onto the plane. Each tile is used multiple times (for an infinite plane, an infinite number of times) yet the generated texture is non-periodic. The periodicity is eliminated by either using a stochastic tiling method where the tiles to be placed are determined by random hashing or by using a Wang tile set that is aperiodic (meaning the tile set will never emit a periodic tiling). Wang tiles reduce storage by only requiring a finite set of tiles to be stored. The Wang tile set can contain 256 tiles (or more) but is often reduced to limit the amount of texture tiles stored[4]. As the infinite plane can be sampled anywhere without precomputing the entire plane, Wang tiles also help increase the diversity of objects requiring similar yet “unique” textures[19].

Wang tiles are square tiles with a constraint that dictates which tiles are allowed to be placed next to each other. Each edge of a tile is given a label termed a “color”. The constraint for tiling only allows tiles having edges with the same color (*i.e.,* label) to be placed next to each other. A

well designed Wang tile set will have the texture features match seamlessly across the edges with the same color. Figure 1.2a illustrates a 2×2 valid tiling of Wang tiles with matching features at the edges where the individual tiles are shown in Figure 1.1. An example of an invalid tiling is shown in Figure 1.2b where the bottom two tiles do not have matching colors (and therefore the features at the edges do not match).

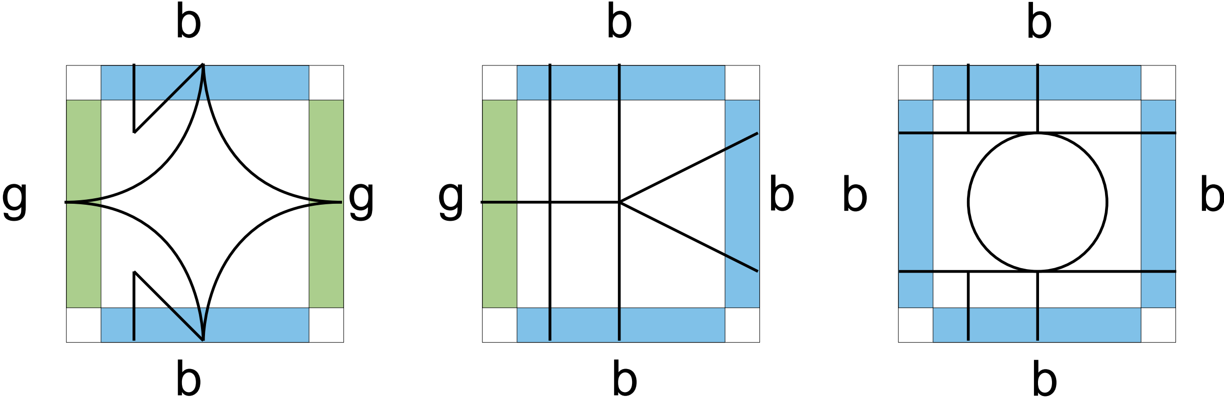
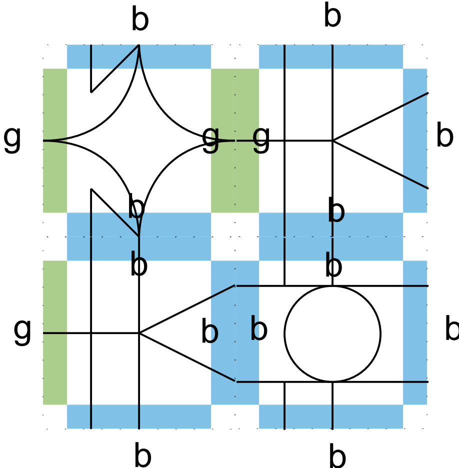


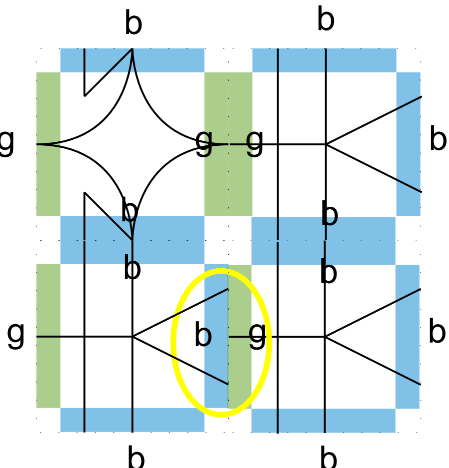
Figure 1.1: Three tiles from a set of Wang tiles. The edges are colored with letters 'g' and 'b'. Notice that the top edges have features that match the bottom edge with the same colors and the left edges have features that match the right edge with the same colors.

By definition Wang tiles have colored edges. A concept similar to edge colored Wang tiles is to have tiles with colored corners. The terminology used to describe *corner colored Wang tiles* has not yet been solidified (e.g. ω-Tiles versus "corner" tiles). Although corner colored tiles are not strictly a "Wang" tile, hereafter, both corner colored tiles and edge colored Wang tiles will be referred to as "Wang" tiles. Figure 1.4a illustrates a 2×2 valid tiling of corner colored Wang tiles with matching features at the corners (and therefore also the edges) where the individual tiles are shown in Figure 1.3. An example of an invalid tiling is shown in Figure 1.4b where the top right and bottom left tiles have corner colors that do not match the bottom right tile.

Creating edge colored or corner colored Wang tiles manually is a difficult process and at least partially automated techniques are preferred for all but the simplest textures. Common automated



(a) Valid tiling of edge colored Wang tiles.



(b) Invalid tiling of edge colored Wang tiles.

Figure 1.2: Valid and Invalid tilings using edge colored Wang tiles.

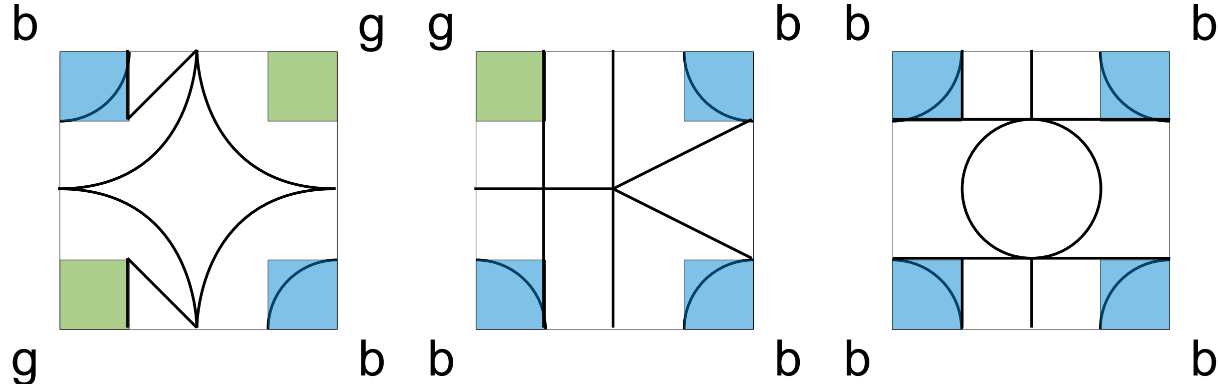
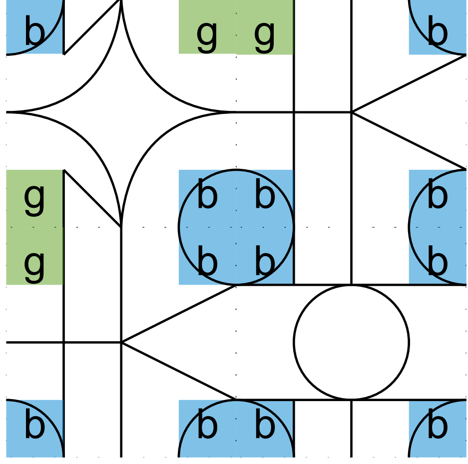
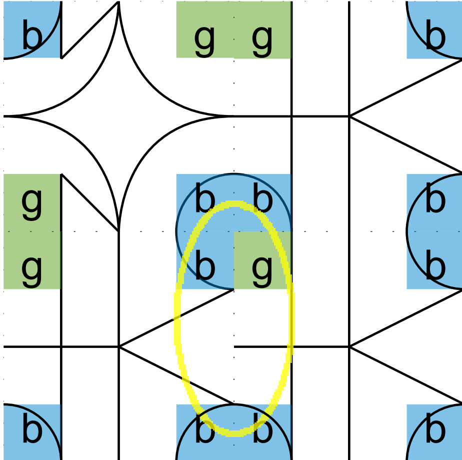


Figure 1.3: Three tiles from a set of corner colored Wang tiles. The corners are colored with letters 'g' and 'b'.



(a) Valid tiling of corner colored Wang tiles.



(b) Invalid tiling of corner colored Wang tiles.

Figure 1.4: Valid and Invalid tilings using corner colored Wang tiles.

methods such as Cohen's Wang tiles[4], ω-Tiles[34], and Strict Wang Tiles[35] exist to form edge colored or corner colored Wang tile sets (all of which use *image quilting* and *graph cuts* as discussed in Chapter 2.2). Each method has types of textures which when used as an exemplar produce tiles with undesirable artifacts (such as cutting through features of a texture or having unnatural light to dark transitions) or fail to provide a reasonable level of diversity in the tile set.

Procedural methods generate texture by evaluating a function (or procedure). These can produce high quality textures yet they typically require a new function to be written for each type of texture to be generated. The functions often use random noise (in particular Perlin noise) to provide for the ability to generate more realistic textures as well as to generate variations of the texture the function is capable of producing. Storage requirements are typically minimal, however it is not easy to generate new procedures for textures.

Texture synthesis methods take an example image called an exemplar and use it to guide the

generation of a new texture that is meant to be similar in character to the exemplar. Some methods simply paste irregular shaped samples into a new image in a random fashion[25] and others have used simple blending combined with search for the overlapping regions of samples[20]. However when overlaying patches, the most common method has been to use dynamic programming or graph cuts to "quilt" (or stitch) together overlapped samples of the exemplar into a new texture image[7][18]. Others have used the concept of a neighborhood and change pixels in a new texture until a best (or good) fit is found for all neighborhoods[8]. Neighborhood based texture synthesis techniques have typically been used to create a larger texture from a small example image.

1.4 Thesis Contributions

This research introduces new solutions for solving the creation and storage requirements for textures. Each method is based on the concept of creating corner colored Wang tile sets and each method uses an exemplar (*i.e.,* example image) as a basis to generate the tile set. Strictly speaking, the new tiles introduced are not Wang tiles as Wang tiles must have colored edges whereas the tiles introduced here have colored corners. In order to convey the similarity in the usage of Wang tiles and corner colored tiles, Wang tiles will be considered to include corner colored tiles and, where necessary, the distinction between whether the tile is a proper Wang tile or a corner colored tile will be made.

The *Synthesized Wang tile* (SWTile) method uses a modified version of Kwatra's controlled texture synthesis algorithm to construct a set of corner colored Wang tiles[17]. The *Bilinear Blended Wang tile* (BLWTile) method uses bilinear blending to construct a set of corner colored Wang tiles. The *Gauss Laplace Blended Wang tile* (GLWTile) method is a variation of the BLWTile method but instead of performing a single blend, the corner colors being blended are decomposed into a set of difference images where each difference image is blended with a different blending size and then reconstructed to form a GLWTile.

Both the BLWTile and the GLWTile methods are intended to allow for dynamic (*i.e.,* per pixel) evaluation without having to precompute a complete corner tile set. The blending methods reduce texture storage requirements by using an exemplar image and allowing dynamic evaluation of any pixel of a tiled plane. The SWTile method is intended to reduce texture storage by precomputing a set of corner colored Wang tiles based on an exemplar.

This research additionally provides key points that should be considered when designing new methods or enhancing current methods for creating corner colored Wang tiles (such as the importance of corner selection and the role of feature sizes). These key points were gathered by implementing and examining existing methods for creating Wang tile sets and through the design process for the new methods (BLWTiles, GLWTiles, and SWTiles) introduced here.

Chapter 2

Background

In order to provide a more complete understanding of the problem domain and current solutions this Chapter provides additional coverage on textures and Wang tiles (including a discussion of graph cuts). The concept of corner colored Wang tiles plays a large part in the new techniques, however Synthesized Wang Tiles include a modified version of controlled texture synthesis while Gauss Laplace Pyramid Blended Wang Tiles include a variant of Gauss Laplace Pyramid blending. As such, both texture synthesis and Gauss Laplace Pyramid blending are also covered here in more detail.

2.1 Texture

Textures are used for a wide variety of objects in computer graphics ranging from surfaces such as walls, floors, vehicles, water and even for representing things that do not have an obvious surface such as clouds and flames. With such a wide variety of application, it is useful to provide classifications for textures. Although texture images may appear quite different, they may still be placed into the same classification or group based on the underlying characteristics of the textures.

Textures are often classified as ranging from regular to stochastic. An example image contain-

ing textures covering this spectrum was created by James Hays and is shown in Figure 2.1.

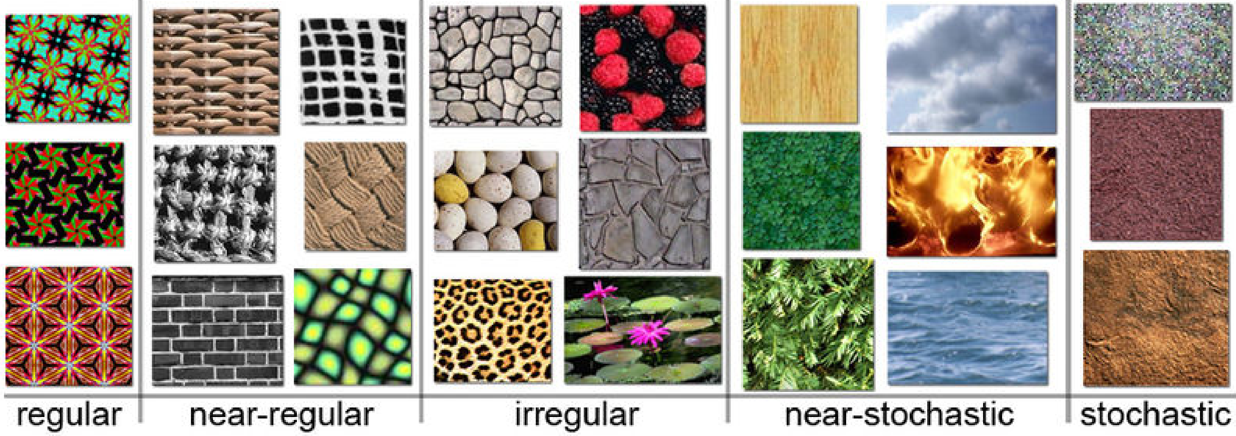


Figure 2.1: Spectrum of Texture Classification: regular to stochastic[21][12].

Another characteristic of textures is how the texture being model is intended to vary across the model's surface. For example, some textures are globally uniform (*i.e.,* stationary) in the sense that when looking at any area of a texture it appears similar to any other area. Other textures have a characteristic of globally varying. Globally varying textures have the property that not all areas of the texture will appear similar to all other areas, yet the texture still appears to be a single texture. Examples of globally varying textures from Li-Yi Wei's research are shown in Figure 2.2.

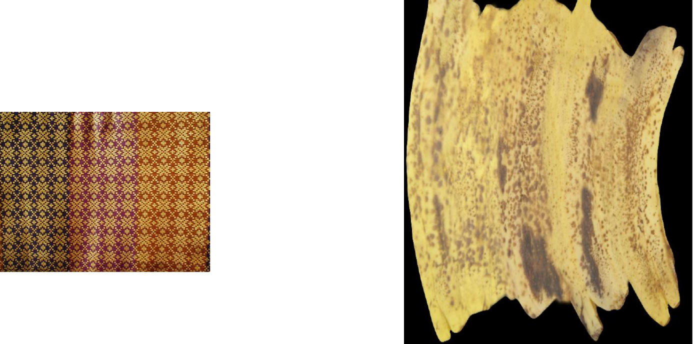


Figure 2.2: Examples of globally varying textures[32]. Left: Fabric. Right: A banana peel.

2.2 Image Quilting

Image Quilting is a technique for generating texture where a new texture is synthesized by stitching together samples from an example texture in such a way as to convince humans that the new texture is the same texture as the example[7]. The stitching of the overlapped samples is typically performed by treating the overlapped regions as a graph and performing a cut. The graph is constructed where the pixels are the nodes in the graph and the cost to visit a node is computed by a difference metric between the two samples at the pixel (typically the difference metric is the difference between the pixel color values of the two overlapping samples at the node with the difference computed by taking the Euclidean distance between the color values of the pixels). The cut will separate the overlapping region into two regions: one region will be populated with pixels from one sample and the other region will be populated with pixels from the other sample. The cut to select is the cut that minimizes the difference value generated by the metric used.

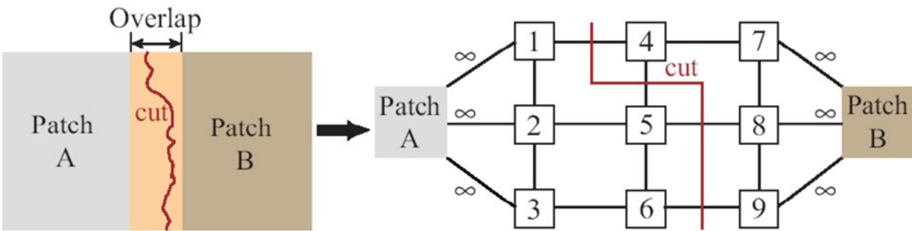


Figure 2.3: Left: Two overlapping samples and a cut where the final pixels in the overlapping region left of the cut will come from patch A and those to the right of the cut will come from patch B. Right: The sampe patches represented as a graph with a cut separating the two patches[18].

Implementations for solving the graph cut problem in regards to overlapping textures have used dynamic programming to find the “best” cut[7] as well as a more robust (yet slower) graph cut approach[18] as shown in Figure 2.3. When cast as a graph problem, the problem is to find the minimum cost cut of the graph that separates one overlapping sample from the other. To restrict

the cut to the overlapping region of the samples, the cost of a cut outside of the overlapping region is increased to infinity.

2.3 Wang Tiles

Each of the new methods discussed in this paper utilize the corner colored Wang tile concept in an attempt to create seamless textures. Corner colored Wang tiles are a variant of Wang tiles where instead of using colored edges as a tiling constraint, corner colored Wang tiles use colored corners as the tiling constraint. A majority of the concepts relating to Wang tiles with colored edges may also be applied to corner colored Wang tiles.

Wang tiles are a collection of tiles that can be repeatedly placed to non-periodically tile a plane. Ideally this set of tiles would tile the plane aperiodically while at the same time represent the domain being modeled (such as a texture, although other domains for Wang tiles exist such as tiles containing object distributions). An aperiodic set of Wang tiles would never allow for a periodic tiling, whereas a non-periodic tiling is a tiling that is not periodic, however the tile set used for a non-periodic tiling might also be used to generate a periodic tiling. Wang tiles have constraints that dictate which tiles are allowed to be placed next to each other. This constraint is called a color. Wang tiles were named after Hao Wang who believed that any set of tiles that could tile a plane must also be capable of producing a periodic tiling of the plane[28][29]. This was later shown to be incorrect when Culík presented a set of 13 tiles that were aperiodic[5].

The ideal Wang tile set is classified as a "small" set of tiles that:[35]

- are representative of the domain (*i.e.,* it has as much diversity as the sample)
- are able to produce non-periodic tilings (*i.e.,* it can tile a plane without repeating a pattern)
- have no visible seams when tiled (*i.e.,* individual tiles are not visually discernible)

Active research is being performed in each of the above areas[33].

There are two common methods for coloring a tile: coloring edges and coloring corners[19]. (See Figure 2.4). With colored edges, only tiles with the same colored edge may be placed next to each other.

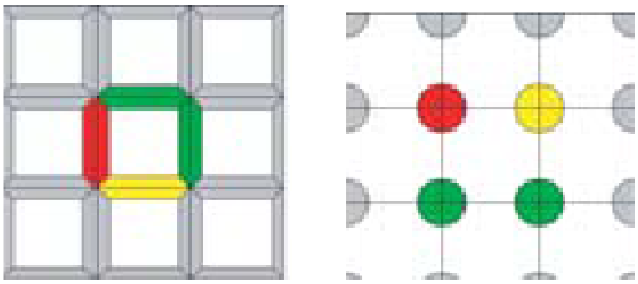


Figure 2.4: Colored Edges (left) and Colored Corners (right)[19].

With colored corner Wang tiles, only tiles whose corners have the same colors may be placed next to each other. Regardless of whether using colored corners or colored edges, the size of the tile set is the same: $tiles = colors^4$. In other words, when using 4 unique edge colors to construct an edge based tile set, 256 tiles may be constructed (although this has traditionally been reduced to a smaller subset, when generating textures on a GPU, the full tile set is used). And when using 4 unique corner colors to construct a corner based tile set, 256 tiles may also be constructed. In both cases, regardless of the number of colors available in the tile set, when constructing a tile, 4 colored edges or 4 colored corners are used.

Using colored edges as a constraint, the horizontal and vertical neighbor tiles will have matching edges. However using an edge constraint for a tile will not ensure that the diagonal neighbors match. Cohen recognized this and presented an additional encoding of colored edges to map to colored corners, but this resulted in an 8 fold increase in the number of tiles required (for the case of 2 colors, this encoding resulted in 128 tiles[4]). Corner colored Wang tiles not only ensure that the vertical and horizontal neighbors of a tile match, but also that the diagonal neighbors match. Other benefits of using native corner colored Wang tiles over edge colored Wang tiles include more efficient tiling algorithms and a smaller texture memory size (one half) when compared to Wang

tiles[19]. "Native" corner colored Wang tiles were initially investigated by Tuen-Young Ng, et al while devising what they called ω-Tile sets (omega-Tile)[34]. As well as by Ares Lagae, et al's research into construction and use of what they termed "corner" tiles.

Using tiles as a texture map requires either loading a texture for each tile or packing all tiles into a single texture. Typically, textures are packed to save space, but more importantly the textures are packed in a manner that provides consistency when *texels* (texture pixels) are fetched (this is important due to the filtering that happens when fetching texels on a GPU). Lagae presented a packing method for corner colored Wang tiles that saves space and improves the quality of textures generated from a packed tile texture by ensuring that the packing itself is a valid tiling of the corner tiles (and toroidal). The packing presented by Lagae (and shown in Figure 2.5) was for 1,2,3, and 4 color tiles and was computed (not derived) taking almost a year of computation time using 400 2.5 GHz CPUs[19].

2.3.1 Cohen's Graph Cut Wang Tiles

The method introduced by Cohen for generating Wang tiles and applying them for use in textures popularized the concept of Wang tiles for use in computer graphics. Cohen's method involves selecting diamond shaped samples to be used as edge colors from an exemplar. The diamond shapes are butted next to each with a slight overlap. A square is then cut out of the center of the overlapped diamonds after a graph cut is made through each overlapping region. The cuts are made so as to minimize the appearance of a seam. An example of the construction technique is shown in Figure 2.6. Samples of textures created by using Wang tiles as presented by Cohen are shown in Figure 2.7.

Cohen's method sometimes results in visible artifacts which can be reduced by increasing the number of tiles in the tile set and by minimizing the quilting errors (finding the minimum cut by iterating over sets of samples). Additional work such as "Strict Wang Tiles" investigated enhancing

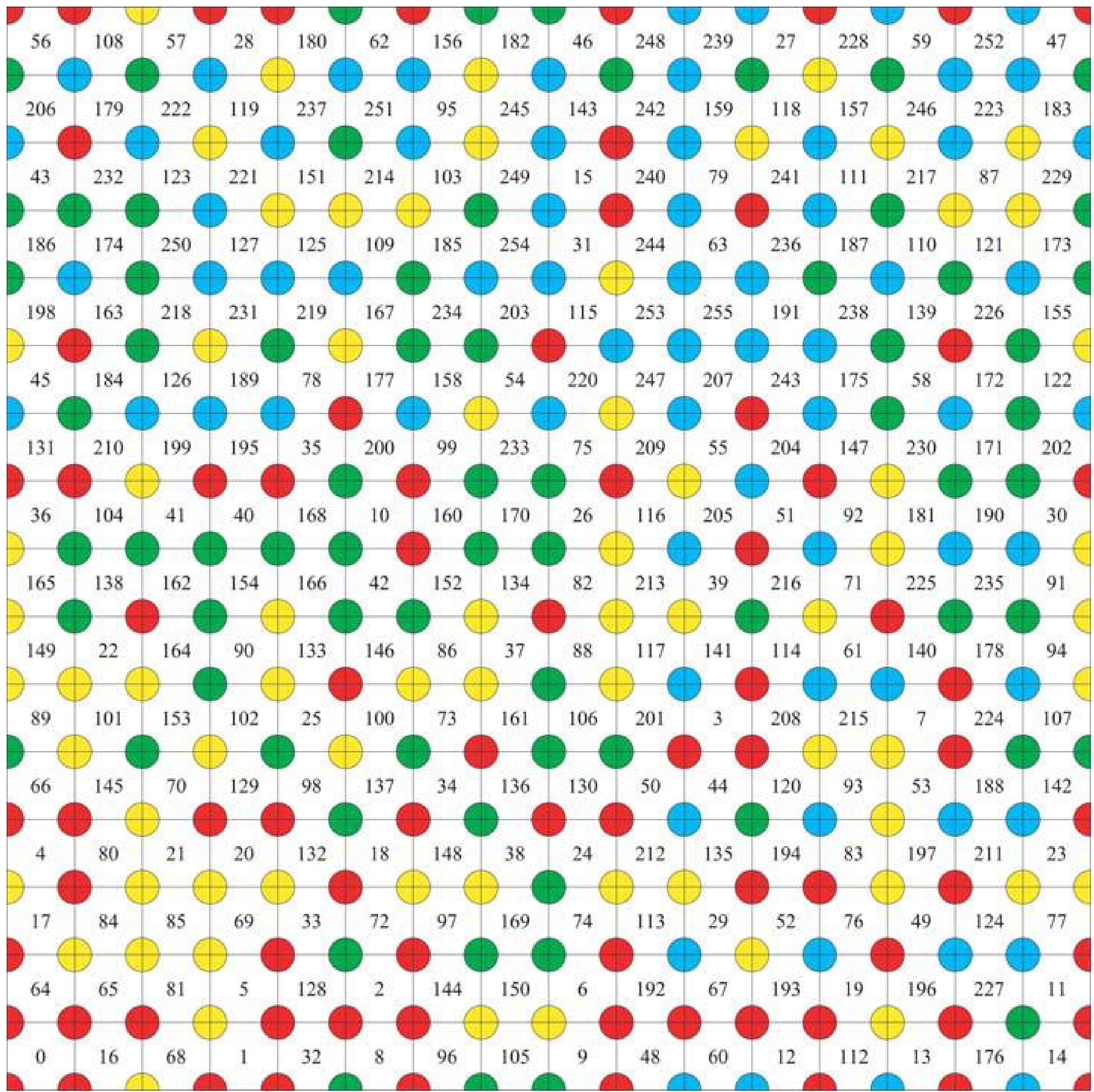


Figure 2.5: Lagae's recursive texture packing for 1,2,3 and 4 color "corner" tiles contained in a single texture. The packing for each count of colors is toroidal[19].

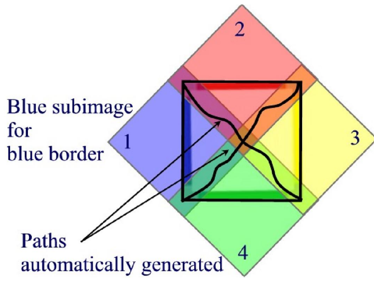


Figure 2.6: Cohen's technique for constructing a Wang tile[4].

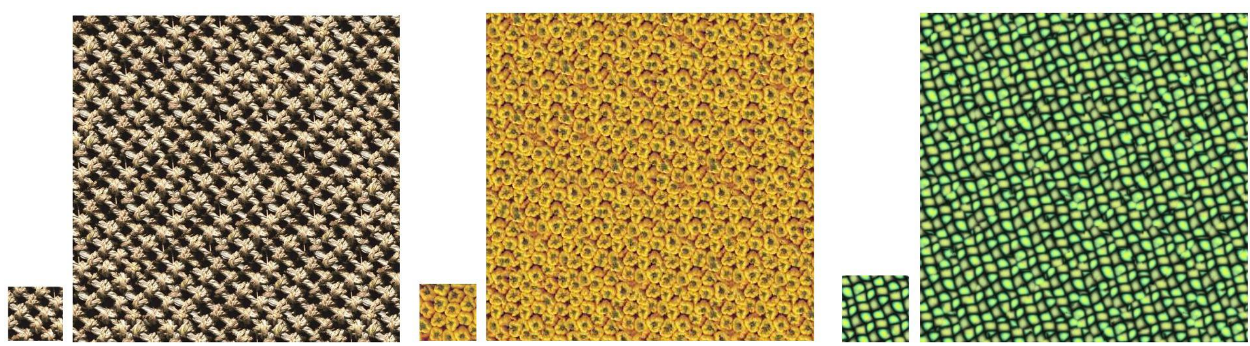
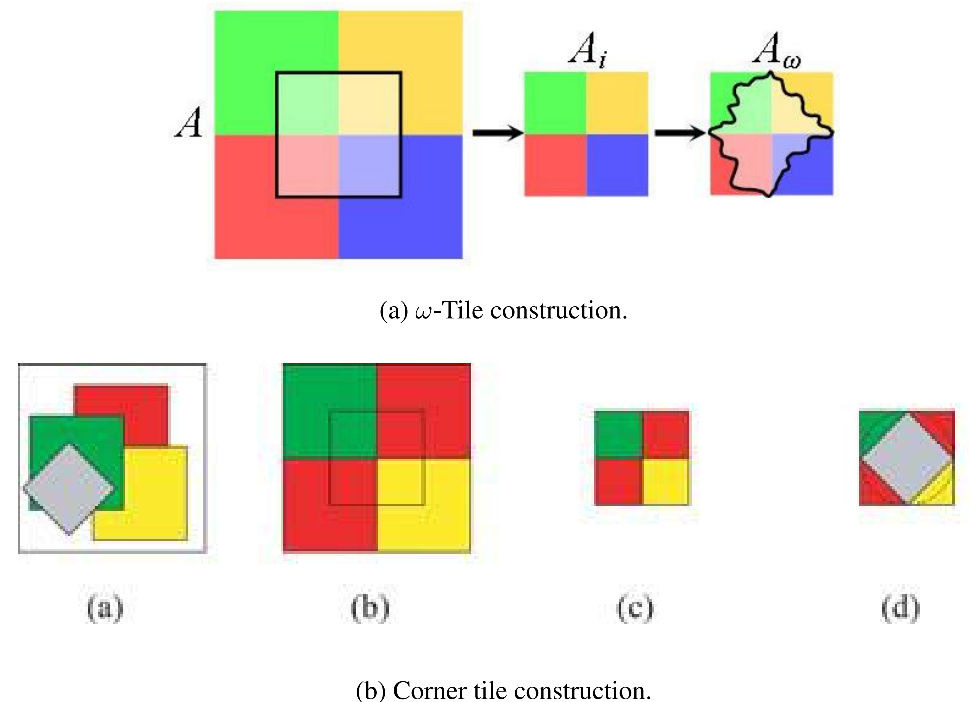


Figure 2.7: Sample tilings using Cohen's method. Shown are the exemplar used and a tiling of each using 18 Wang tiles[4].

Cohen's technique of creating Wang tiles using diagonal samples and graph cuts[35].

2.3.2 ω-Tiles and "corner" tiles

Both ω-Tiles and corner tiles use corner colored tiles and graph cuts to perform texture tiling. Both methods abut 4 square color samples and overlay the seams with a random sample from the exemplar. The tiles are then created by performing a cut between the overlay and the colored corners. ω-Tile and corner tile construction are shown in Figure 2.8.



(a) ω-Tile construction.

(b) Corner tile construction.

Figure 2.8: Construction of a ω-Tile[34] and a "corner"[19] tile as depicted in respective author's articles.

The only difference between ω-Tiles and corner tiles, according to Lagae, is that ω-Tiles only constructed a limited number of tiles while corner tiles constructed the complete tile set for four colors.

2.4 Texture Synthesis

A variety of texture synthesis techniques for synthesizing texture have previously been investigated such as pixel based, patch based and texture optimization. Pixel based methods synthesize a texture one pixel at a time by comparing the neighborhood (*i.e.,* a region) surrounding the pixel to be synthesized with neighborhoods from the exemplar to determine the best value for the synthesized pixel. Patch based techniques copy patches from the exemplar to the output and typically perform a graph cut for overlapping patches (as with pixel based techniques, patch selection is often guided by neighborhood searches). Texture optimization is an iterative technique that synthesizes a texture by searching for neighborhood matches for each pixel while at the same time attempting to minimize the mismatches in exemplar and output neighborhoods for the entire texture[33].

The reason that many of the texture synthesis techniques use a concept of a neighborhood is due to the fact that many of the techniques are relaxations of a more rigorous model known as *Markov Random Fields* (MRF). Each pixel is statistically related to the pixels that are spatially close to it regardless of what part of the texture is being examined. Therefore, to synthesize a texture that is representative of an exemplar, one only need to ensure that all pixels in the synthesized texture have neighborhoods that are similar to some exemplar pixel's neighborhood.

2.4.1 Kwatra's Texture Optimization

The Synthesized Wang Tile technique is a modification of the texture optimization method presented by Vivek Kwatra, et al. In Kwatra's terminology the synthesized image is called the X image and the input exemplar is the Z image. Kwatra's algorithm cast the neighborhood matching and pixel synthesis problem as a quadratic energy minimization problem. Both the Z and X image are split into evenly spaced (yet overlapping) neighborhoods. The spacing of neighborhoods in the Z image may be different than the spacing in the X image.

The energy to be minimized is determined as the sum of the Euclidean distance of pixel colors

between each X pixel and its corresponding Z pixel (where correspondence is determined as the pixel within the best fitting Z neighborhood). The energy function is given in Equation 2.1 where $x; \{z_p\}$ is the set of all X to Z neighborhood matches for each pixel in X, $p \epsilon X^{\dagger}$ are all pixels contained in a neighborhood of X after adjusting for the desired X neighborhood spacing, x_p is a vector of all pixels in the X neighborhood for pixel p, and z_p is a vector of all pixels in the corresponding Z neighborhood for pixel p.

$$E_t(x; \{z_p\}) = \sum_{p \epsilon X^{\dagger}} \| x_p - z_p \|^2 \tag{2.1}$$

To minimize the energy, Kwatra uses a modified *Expectation-Maximization* (EM) algorithm where both the variables (the synthesized texture) and the parameters (the matched neighborhoods) are unknown. To solve this problem an iterative approach is taken, repeatedly alternating between an Expectation step and a Maximization step until the energy is zero or some predefined limit is reached. The Expectation step entails minimizing the energy by averaging all of the corresponding Z pixel values for each X neighborhood that overlaps the X pixel. The Z pixel value used is the Z pixel with the same offset relative to its neighborhood as the X pixel has to the overlapping X neighborhood.

In order to better capture large scale features, Kwatra performs the energy minimization steps at multiple neighborhood sizes (typically 32×32, 16×16, and 8×8). The larger neighborhood sizes attempt to reproduce the larger scaled features while each smaller neighborhood size serves to refine the previous result of the larger neighborhood and more accurately reproduce the smaller scaled features. In addition to performing the energy minimization at multiple neighborhood sizes, the whole process is also repeated at multiple X image sizes (starting at a reduced scale and using the previous result as the initialization of X for the larger sized image). Performing the energy minimization at the smaller X image size helps to capture consistency information for the entire texture in addition to providing a better initialization for the small scale (fine detail) synthesis.

Kwatra also introduced a method to control the synthesis by including the control parameters in the energy function as shown in Equation 2.2. This energy function is composed of a weighted sum of the energy describing how closely the X image matches the Z image, as given in Equation 2.1, and the energy describing the match of X to the control, where λ is a relative weighting coefficient, u is the control, and E_c is the sum of squared distances between the control and synthesized pixel values.

$$E(x) = E_t(x; \{z_p\}) + \lambda E_c(x; u) \tag{2.2}$$

The modified energy function is minimized at each iteration by taking a weighted average of the control pixels and that of the synthesized pixels. Kwatra investigated other characteristics for use in the energy function including the use of gradients, iteratively re-weighted least squares, flow guided synthesis, and a per pixel weight based on a Gaussian fall-off depending on the pixels relative location in the neighborhood. A full description of Kwatra's methods are given in "Texture Optimization for Example-based Synthesis"[17]. A description of how Kwatra's controlled texture synthesis algorithm is modified for use in creating Synthesized Wang Tiles is given in Chapter 5.

2.5 Gauss Laplace Pyramid Blending

The Gauss Laplace Pyramid Blended Tile technique introduced in this thesis incorporates the Gauss Laplace Pyramid blending by Burt and Adelson for use in creating image mosaics[3]. Gauss Laplace Pyramid blending involves creating a set of low-pass images and computing the differences (or errors) between the low-pass images (resulting in a band-pass image), blending each difference image together, and then reconstructing the blended result. The technique has the effect of blending two images together where the different frequencies in the images are blended at different blend widths. The Gauss Laplace Pyramid blending technique is also termed multi-band

blending as multiple bands of the image frequencies are blended separately.

The low-pass generated images are created by using a Gaussian filter and sub-sampling. As each consecutive low-pass image generated is a sub-sample of the previous, the term Gaussian pyramid is used as the lowest level (the first unfiltered image) is the largest image and the higher levels of the pyramid are progressively smaller (and more blurry) images. The difference images created by subtracting 2 levels of the Gaussian pyramid are termed the Laplacian pyramid. The highest level of the Laplacian pyramid is set to the highest level of the Gaussian image as there is no higher Gaussian to create a difference (therefore the difference or error is simply the highest level Gaussian image).

The reconstruction process of the blended Laplacian pyramid is to sum the levels of the Laplacian pyramid. Letting G_x represent a level in the Gaussian pyramid and L_x represent a level in the Laplacian pyramid, and R_x represents a level in the reconstruction where 0 is the lowest level, Equation 2.3 governs the computation of the Laplacian levels and Equation 2.4 is used in the computation for reconstructing an image from the Laplacian pyramid. In the case where no blending is performed or the overlapped images are identical, the reconstruction process results in the identical image (*i.e.,* $R_x = G_x$).

$$L_x \leftarrow G_x - G_{x+1} \tag{2.3}$$

$$R_x \leftarrow L_x + R_{x+1} \tag{2.4}$$

Since different levels of the pyramids have different sized images, when performing subtraction or addition of images from different levels, an expand or reduce operation is performed as needed. The reduce operator scales the image down while applying a Gaussian filter while the expand operator scales an image up and uses Gaussian weights to act as the reverse of reduce.

An arbitrary boundary can also be used to control the blending as opposed to using a simple

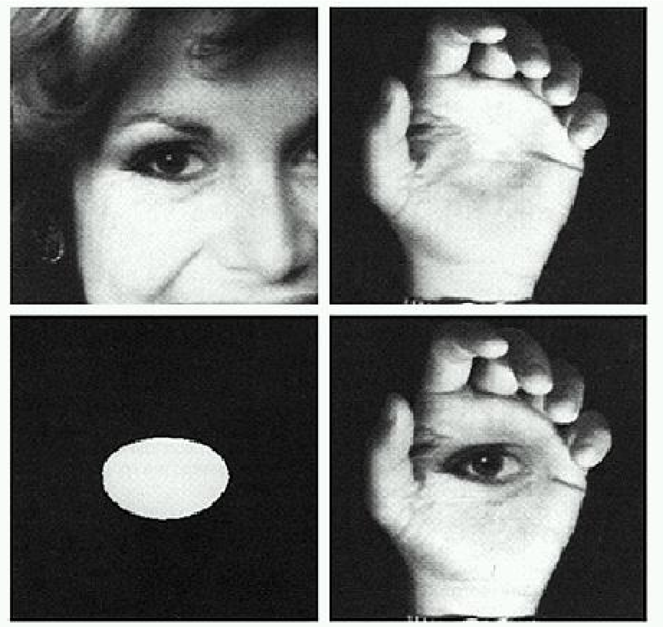


Figure 2.9: Result of blending using Gauss Laplace pyramids with a blending boundary[3].

overlap and is illustrated in Figure 2.9.

A simplification of multi-band blending uses only two bands for blending (the high frequency and the low frequency)[1]. An alternative to using pyramids is to use gradient domain blending where the steps involve differentiation, blending, and reintegration. Gradient domain blending requires that a solution be found similar to solving a Poisson equation[9].

For more complete description of Gauss Laplace Pyramid blending, refer to “A Multiresolution Spline with Application to Image Mosaics”[3] and for the complete set of formulas also refer to “The Laplacian Pyramid as a Compact Image Code”[2].

Chapter 3

BLWTiles

The most common method for creating Wang tile sets in current literature as described in Chapter 2.3 is by use of graph cuts. However, some popular gaming engines incorporate blending as a method of generating texture for use in games with a large world environment[10]. This method typically involves manually adjusting parameters used to control the blending and uses blended overlays in attempts to hide seams in the tiling.

The concept of Bilinear Blended Wang Tiles (BLWTiles) is that of using an exemplar and bilinear interpolation to generate Wang tiles procedurally. The BLWTiles should meet the definition of Wang tiles in regards to tiling as well as with regards to the quality traits deemed desirable in a Wang tile set (*i.e.,* minimizing or reducing seams, preserving the diversity of the exemplar, etc).

3.1 Hypothesis

It is expected that a Wang tile set of reasonable quality (yet slightly blurry) can be created from an exemplar such that a large count of colors may be used (yielding a diverse tile set) while enabling procedural evaluation (*i.e.,* per pixel evaluation) for tiling a surface. The blurring is expected as a result of using blending and bilinear interpolation. As tiles would be generated procedurally from

an exemplar, there is no need to store the complete set of tiles, allowing the tile set to use a virtually unbounded number of colors.

3.2 Initial Approach

BLWTiles use colored corners as opposed to colored edges. The use of colored corners and blending between the corners enables the creation of a Wang tile that reduces the appearance of seams between adjacent tiles. Diversity is obtained by using most (if not all) of the exemplar area to create colors for use in tile construction. It quickly became apparent that as linear interpolation only guaranteed C^0 continuity, non-linear (*e.g.* polynomial) interpolation was required to guarantee C^1 and even C^2 continuity as human vision is well adapted to identify C^0 and C^1 discontinuities.

The steps illustrating the construction of a BLWTile are:

1. Determine colors to be available for use in tile construction.
2. Assign colors for each corner of a tile.
3. For each pixel in the tile, compute a u and v value for use in a bilinear blend operation of the four colors.
4. Compute the pixel's value by using u,v, and the four color samples assigned to each corner in the bilinear blend function.

3.2.1 Colors

The color samples available for the BLWTiles are taken from an exemplar and are determined by dividing the exemplar into a grid of size $colorWidth/2 \times colorHeight/2$. In this way, the content of each color partially overlaps with the content of other colors (see Figure 3.1). The size of the color and the size of the exemplar dictate how many colors will be present for use in constructing

the BLWTiles. The number of colors in the exemplar available for use in BLWTile construction is given in Equation 3.1

$$colorsAvail = colorSampleColumns \times colorSampleRows \tag{3.1}$$

As each color overlaps by half the color width the number of color sample columns is as shown in Equation 3.2.

$$colorSampleColumns = exemplarWidth \div (colorWidth \div 2) \tag{3.2}$$

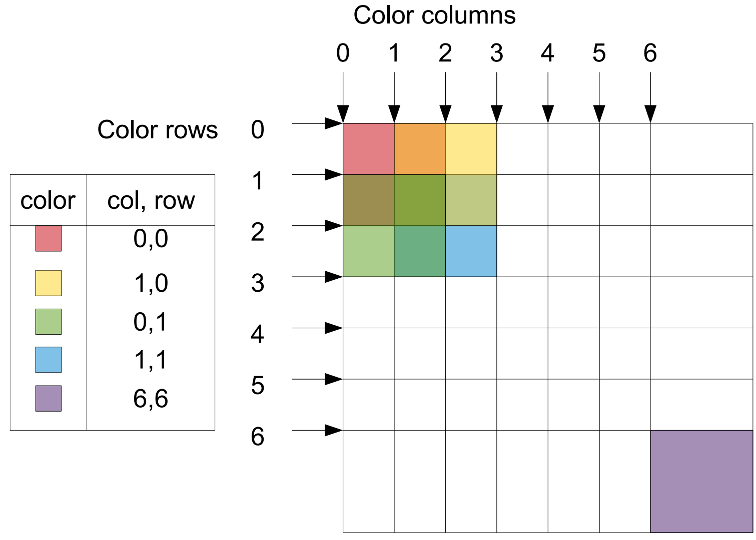


Figure 3.1: Creating color samples from a source image.

A test must be performed to ensure that the last color sample in the column does not extend beyond the exemplar as given in Figure 3.2.

The same method can be used to determine the number of available color sample rows in the exemplar.

As an example, if the exemplar is of size 256×256 and the color size is 64×64, then the number

color sample columns $\leftarrow$ *exemplarWidth / (colorWidth / 2)*
last color start X $\leftarrow$ *color sample columns*$\times$*(colorWidth / 2)*
if (last color start X + colorWidth $>$ *exemplarWidth)*
    color sample columns $\leftarrow$ *color sample columns-1*

Figure 3.2: Checking the range of the color columns.

of color columns is 7 and the number of color rows is 7. This yields 49 total colors available for use in generating the BLWTiles. As all color combinations are available for generating BLWTiles in the tile set, the number of Wang tiles in the set is given by $colors^{(corners)}$. Therefore, in this example, the number of BLWTiles in the set is 49^4 (*i.e.,* more than 5 million tiles). Given the large set of tiles, it is unrealistic that the complete set of tiles would be stored in memory. This necessitates the evaluation of tiles (or pixels within a tile) to be performed procedurally (ideally as a GPU fragment shader).

3.2.2 Tile Construction

The method to construct BLWTiles relies on the use of colored corners and blending to ensure that the BLWTiles are tileable while reducing the seams between tiles. The convention used to identify the BLWTile corners is shown in Figure 3.3.

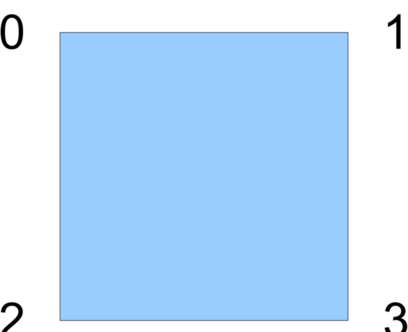


Figure 3.3: Numbering corners of a BLWTile

During construction of a tile, each numbered corner is assigned a color. Each color (*i.e.,* label)

is associated with a sample from the exemplar. The sample associated with the color is also often called a color where the context it is used within dictates if "color" is being used as the label or as the image sample. The color is split into 4 quadrants. A corner color's contribution to the tile is the content of one of the 4 quadrants of the color, based on which corner the color is applied to as shown in Figure 3.4.

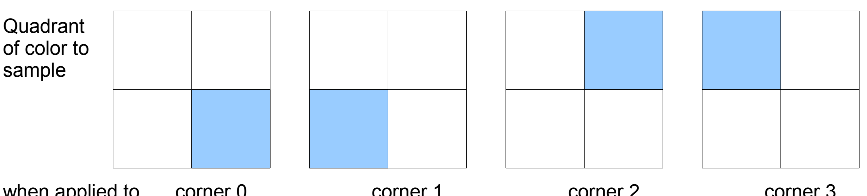


Figure 3.4: Quadrants of a color sample to apply when coloring a corner.

Figure 3.5 illustrates how each of the four colors (one for each corner) are overlapped during construction of a BLWTile.

Each quadrant of a color sample completely overlaps the interior of the BLWTile. In order to ensure that the BLWTile has no seams, bilinear interpolation is performed over the colors applied at the four corners. This ensures that the pixel at each corner is the pixel from the color for that corner while simultaneously blending between the edges and throughout the middle of the tile.

To evaluate a pixel value within the BLWTile, a bilinear blending function is used that interpolates from left to right as well as from top to bottom of the tile. The blending function is given in equation 3.3.

$$\begin{aligned} BLWT_{0123}(u,v) &= (1-v)*[(1-u)*C_0\,(u,v)+u*C_1\,(u-1,v)] \\ &+ v*[(1-u)*C_2\,(u,v-1)+u*(C_3\,(u-1,v-1))] \end{aligned} \tag{3.3}$$

where:

- $BLWT_{0123}$ is the tile being generated where each number indicates the numbered corners;

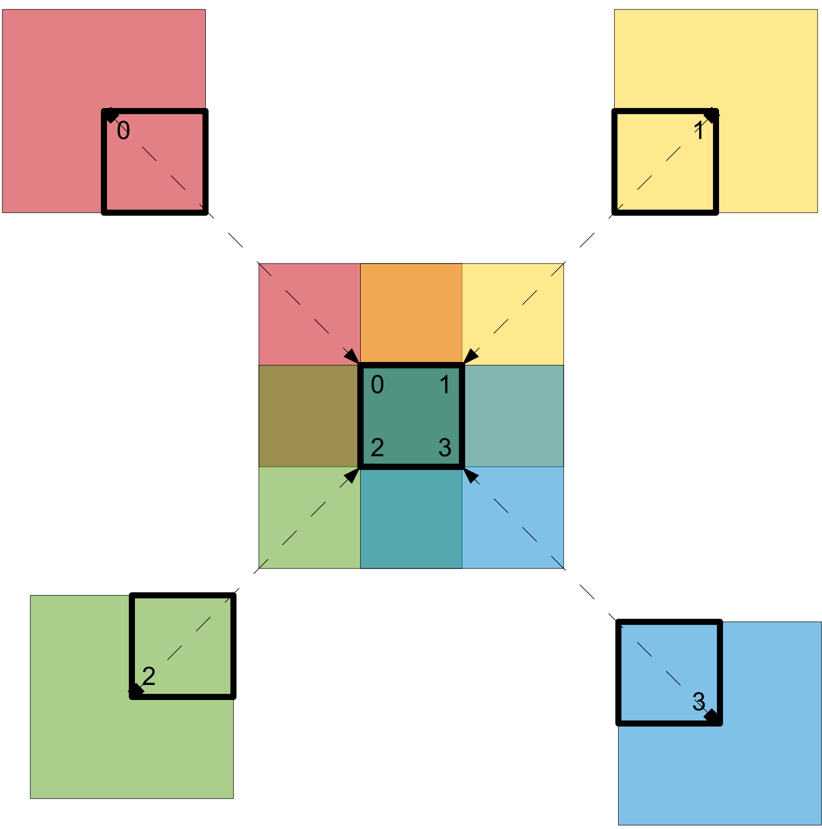
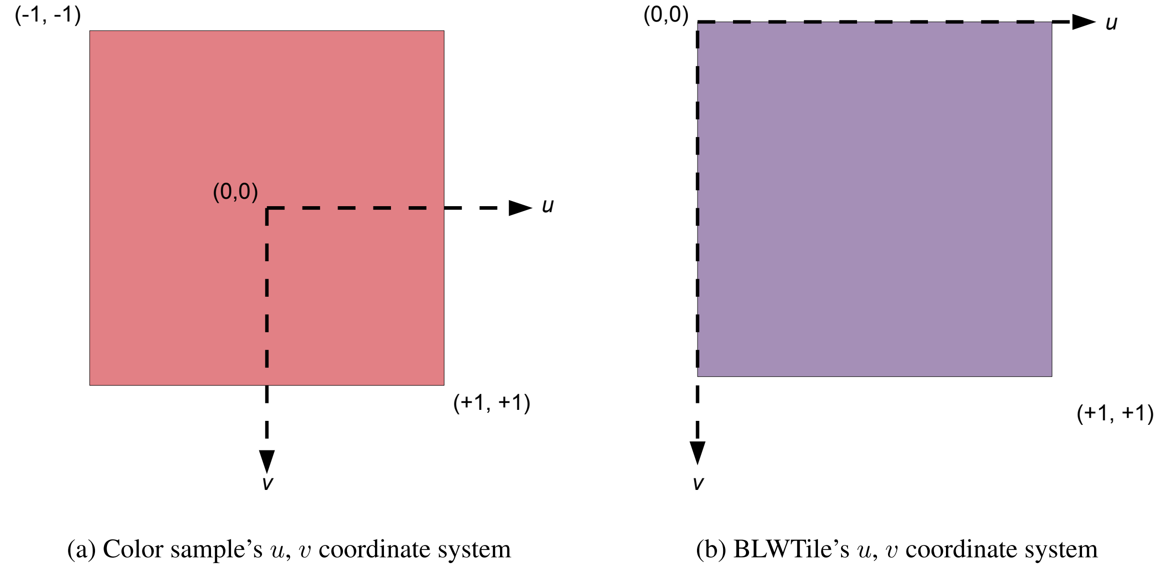


Figure 3.5: BLWTile construction: overlapped colors

- C_0, C_1, C_2, and C_3 are the color samples associated with corner 0, 1, 2 and 3 (respectively);
- the origin of u, v with respect to a color sample is the middle of the sample with positive v facing downward (see Figure 3.6a);
- and the origin for u, v with respect to $BLWT_{0123}$ is the top left of the tile with positive v facing downward (see Figure 3.6b).

The initial interpolation for u and v from 0 to 1 is a linear interpolation. As such, once the tile's colors are determined, Equation 3.3 makes it possible to directly evaluate the pixel for a BLWTile at any u, v within the tile.



(a) Color sample's u, v coordinate system

(b) BLWTile's u, v coordinate system

Figure 3.6: u, v coordinate systems for a color sample (left) and for a BLWTile (right).

3.2.3 Tiling The Plane

In order to avoid precomputing a tiling of the plane using all of the possible BLWTiles, a random hash based method[30] may be used to identify what colors to use for a tile containing a point x,y. The hash based method computes the BLWTile row and col (j, i, where i and j are integers within an infinite integer lattice) that would contain point x, y if the plane was infinitely tiled with tiles of size $colorwidth/2$ x $colorheight/2$. Once the tile row and column are computed, the colors for the tile's corners can be computed using the row and column for the tile as seeds to a random function as shown in equation 3.4.

$$\begin{aligned} seedRandom(i) & \\ randJSeed &= j + random() \\ seedRandom(randJSeed) & \\ color &= random() \mod maxColors \end{aligned} \tag{3.4}$$

Since all possible color combinations are allowed in creating a tile and the colors for each corner are chosen randomly, the tile set is guaranteed to tile the plane. The four corner colors are selected for a point x, y by computing the j, i row and col for each corner surrounding the point x, y. The computations for i and j for use in method 3.4 are given below:[30]

$$\begin{aligned} C_0 i &= hash(\lfloor x \rfloor) \\ C_0 j &= hash(\lfloor y \rfloor) \end{aligned} \tag{3.5}$$

$$\begin{aligned} C_1 i &= hash(\lfloor x + 1 \rfloor) \\ C_1 j &= hash(\lfloor y \rfloor) \end{aligned} \tag{3.6}$$

$$\begin{aligned} C_2 i &= hash(\lfloor x \rfloor) \\ C_2 j &= hash(\lfloor y + 1 \rfloor) \end{aligned} \tag{3.7}$$

$$\begin{aligned} C_3 i &= hash(\lfloor x + 1 \rfloor) \\ C_3 j &= hash(\lfloor y + 1 \rfloor) \end{aligned} \tag{3.8}$$

3.3 Methods Investigated

The approach documented in Section 3.2 served as the core methods for evaluating the technique for generating BLWTiles. The main alternative investigated for generating tilings of BLWTiles was to use non-linear interpolation for the u, v coordinates.

3.3.1 Interpolation

The equation for blending the four corner colors contributing to a tile used a linear interpolation for u and v. Besides the linear interpolation, the other interpolation methods investigated are termed t3 and perlin (t3 is the formula used in the original Perlin noise generation[23] and perlin is the formula used in Perlin's improved noise generation[24]). Perlin's original formula is a cubic polynomial whereas the improved formula is a 5th order polynomial. Figure 3.7 shows all three functions graphed in a transition from t=0 to t=1.

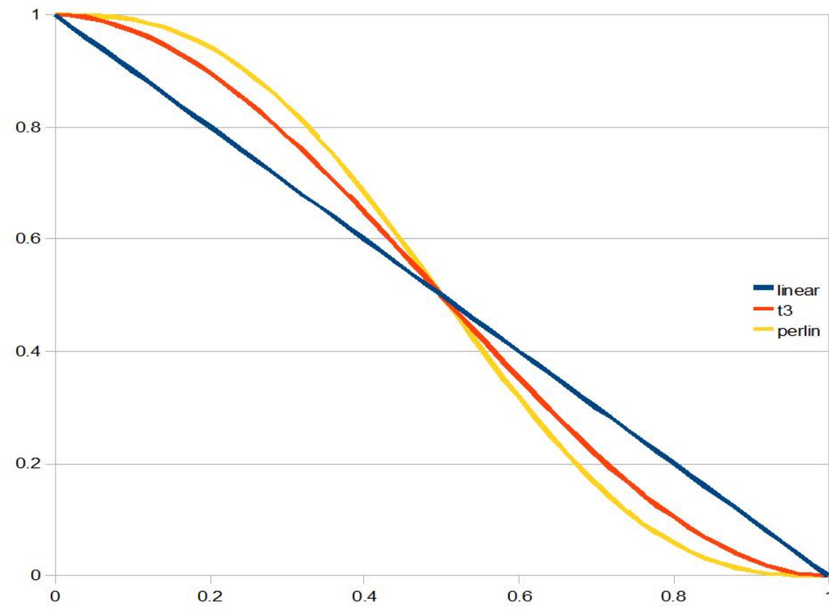
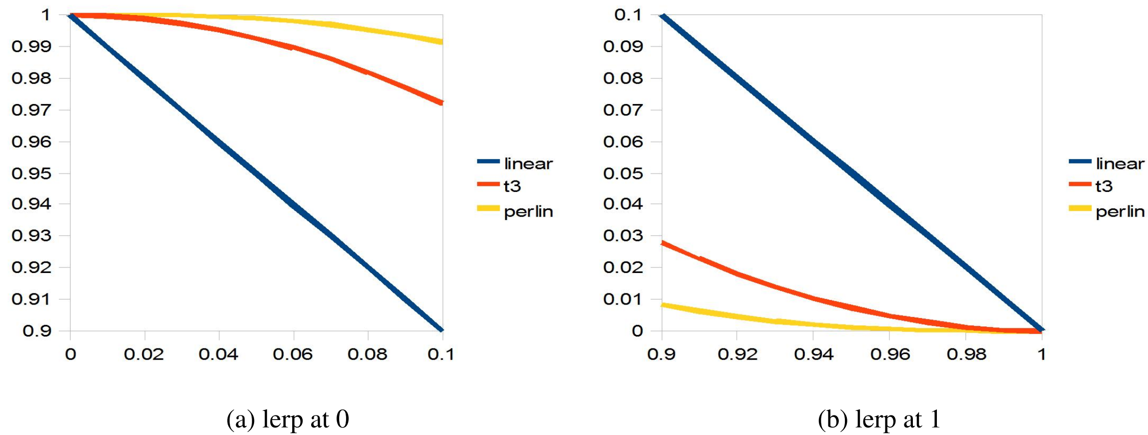


Figure 3.7: Interpolating 0 to 1 using 3 different functions.



(a) lerp at 0

(b) lerp at 1

Figure 3.8: Closeups illustrating the smoothness at 0 and 1.

The formulas for each of the methods are given below.

$$linear = u(t) = t \tag{3.9}$$

$$t3 = u(t) = -2t^3 + 3t^2 \tag{3.10}$$

$$perlin = u(t) = 6t^5 - 15t^4 + 10t^3 \tag{3.11}$$

The investigations into the non-linear interpolation methods were due to observations of the prototypical bilinear ridge or "glow" effect as seen in figure 3.9. Even though the other interpolations (t3 and perlin) are polynomial, it is common for the term bilinear interpolation to be applied.

3.4 Results

For each generated result, the source exemplar shown and the source size is provided in the caption. For each generated result, the result size and the size of the BLWTile is also given. The grid of tiles specified in the captions for the generated results is the number of BLWTiles contained in the

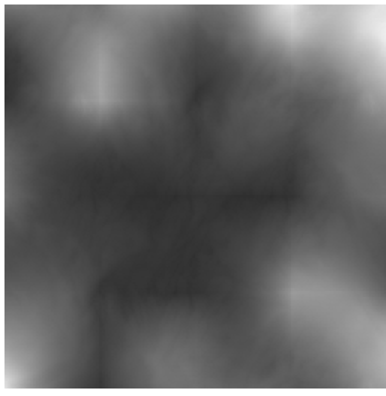


Figure 3.9: Bilinear “Glow”

result. The number of colors given for each output is the possible number of colors available in the exemplar based on the parameters used, not the number of colors contained in the output. A gridding effect is most obvious for the large exemplars used where the exemplar contains a globally varying intensity (*e.g.* a light to dark transition of the entire exemplar).

(a) Source of size 888×608[6].

Figure 3.10: Source for “MonaCanvas”.

(a) BLWTile result of size 512×512. A 4×4 grid of tiles. Each tile of size 128×128. 15 colors available.

(b) BLWTile result of size 512×512. An 8×8 grid of tiles. Each tile of size 64×64. 96 colors available.

(c) BLWTile result of size 512×512. A 16×16 grid of tiles. Each tile of size 32×32. 468 colors available.

Figure 3.11: BLWTile results for "MonaCanvas".

(a) Source of size 512×256[6].

(b) BLWTile result of size 512×512. A 4x4 grid of tiles. Each tile of size 128×128. 3 colors available.

(c) BLWTile result of size 512×512. An 8×8 grid of tiles. Each tile of size 64×64. 21 colors available.

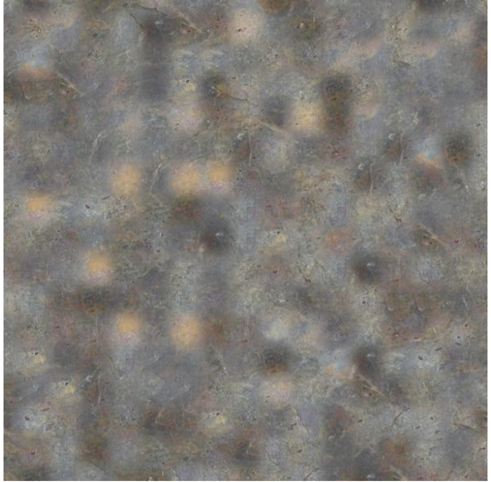
(d) BLWTile result of size 512×512. A 16×16 grid of tiles. Each tile of size 32×32. 105 colors available.

Figure 3.12: Source and BLWTile results for “MonaCanvas2”.

(a) Source of size 385×385[27].

(b) BLWTile result of size 512×512. A 4x4 grid of tiles. Each tile of size 128×128. 4 colors available.

(c) BLWTile result of size 512×512. An 8×8 grid of tiles. Each tile of size 64×64. 25 colors available.



(d) BLWTile result of size 512×512. A 16×16 grid of tiles. Each tile of size 32×32. 121 colors available.

Figure 3.13: Source and BLWTile results for “Slate”.

(a) Source of size 712×710[14].

Figure 3.14: Source for “Shell Stone Tile”.

(a) BLWTile result of size 512×512. A 4x4 grid of tiles. Each tile of size 128×128. 16 colors available.

(b) BLWTile result of size 512×512. An 8×8 grid of tiles. Each tile of size 64×64. 100 colors available.

(c) BLWTile result of size 512×512. A 16×16 grid of tiles. Each tile of size 32×32. 441 colors available.

Figure 3.15: BLWTile results for "Shell Stone Tile".

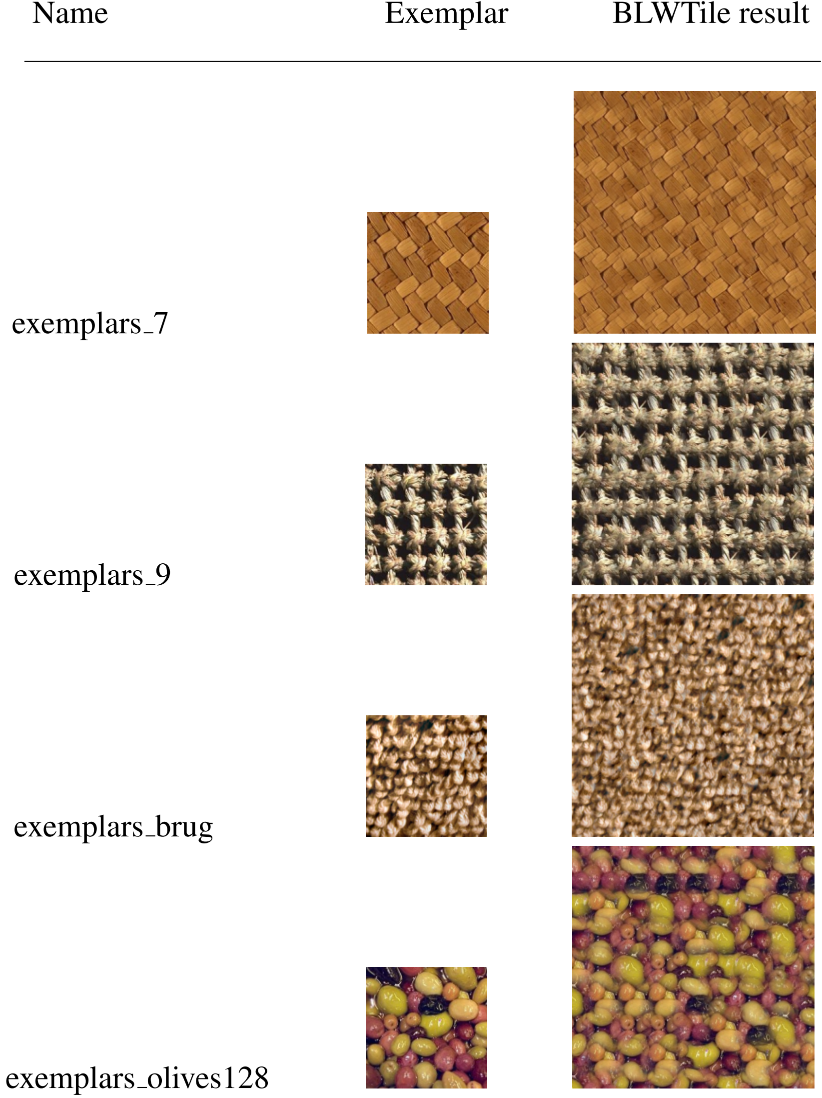


Figure 3.16: Source and BLWTile results. Source is 128×128[13]. BLWTile result is 256×256 using an 8×8 grid of tiles sized 32×32. 9 colors are available. Exception is exemplars_brug which uses a 16×16 grid of tiles sized 16×16 with 49 colors available.

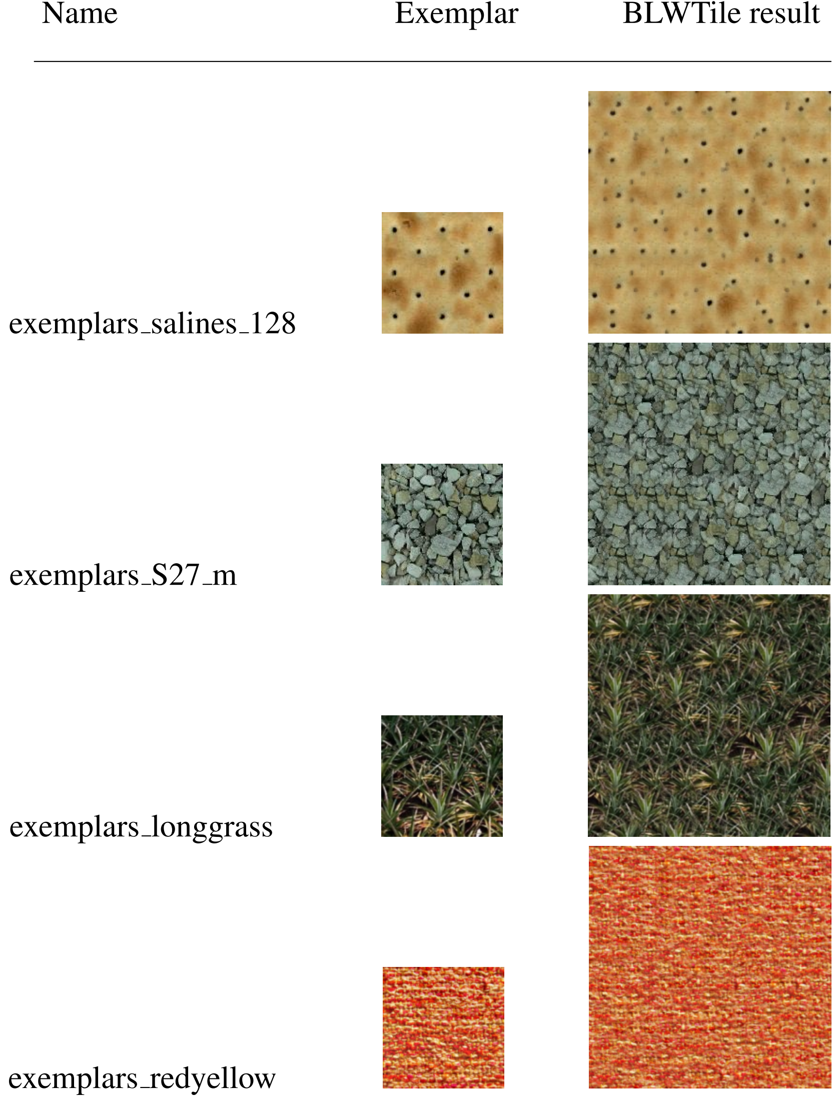


Figure 3.17: Source and BLWTile results. Source is 128×128[13]. BLWTile result is 256×256 using an 8×8 grid of tiles sized 32×32. 9 colors are available. Exception is exemplars_redyellow which uses a 16×16 grid of tiles sized 16×16 with 49 colors available.

3.5 Analysis

Using the smooth interpolation methods (such as Perlin's cubic and 5th order polynomials) did improve the quality of the BLWTiles by reducing or eliminating the bilinear "glow" effect. However, some tiles such as exemplars_S27_m (shown in Figure 3.17) and exemplars_9 (shown in Figure 3.16) were blurry enough to consider the result less than ideal. Using multiple colors in the BLWTile creation resulted in tiles that had a wide range of diversity reducing or eliminating the visibility of the underlying tiling grid. However many of the large exemplars contained globally varying features that resulted in a noticable gridding effect in the generated output as an excessively blurred line or as dramatic changes in color.

Many of the stochastic textures produced good results as seen in exemplars_redyellow (Figure 3.17). The effectiveness of the BLWTile method over a larger spectrum of textures was unexpected. However, the BLWTile method had significant failure cases such as the blurred images and more severe ones relating to some near regular textures as shown in Figure 3.18.

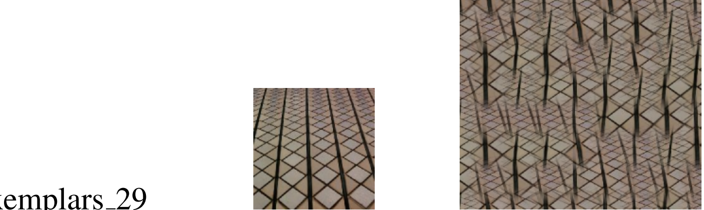


Figure 3.18: Source and BLWTile result. Source is 128×128[13]. BLWTile result is 256×256 using an 8×8 grid of tiles sized 32×32. 9 colors are available.

3.6 Conclusion

The BLWTile method for creating textures is suitable for a variety of textures where high-fidelity is not necessary. The result is typically slightly blurry but there exist significant failure cases where

the characteristics of the exemplar are not captured. For the cases where the BLWTile method does work however, the benefit is a highly diverse tile set (due to the sheer number of colors available for producing tiles) that is created with an economic storage of texture.

Possible future work includes creating a GPU shader program to implement the BLWTile method for tiling. This shader could use hashing and a stochastic tiling algorithm to determine which corner colored Wang tile should be generated. After the tile is determined, 4 texture fetches would need to be performed and the result blended to evaluate to a pixel color. In addition, it would be useful to create a tool to enable real time visualization of modifying the color sizes used (which could be accelerated by using the GPU rendering mechanism).

Chapter 4

GLWTiles

Gauss Laplace Pyramid blending has been used to stitch together panoramas where the individual images being stitched together are expected to match along a seam[1]. Work by Burt and Adelson illustrated the importance of feature size in relation to blending size as well as introduced a technique of blending different frequencies using different blending widths[3]. The *Gauss Laplace Pyramid Blending* method for creating Wang tiles incorporates this concept of blending different frequencies (*i.e.,* features of varying sizes) over blend regions of varying sizes. The tile construction method for Gauss Laplace Pyramid Blended Wang tiles (GLWTiles) is based on the methods investigated in creating Bilinear Blended Wang tiles (BLWTiles).

4.1 Hypothesis

The expectation is that constructing Wang tiles by blending the color samples at each corner of a tile over regions of varying sizes, excessive blurring as found in the BLWTile technique should be reduced. At the same time, the seams between tiles should remain minimized or eliminated and the diversity of the Wang tile set with respect to the exemplar should be maintained.

4.2 Initial Approach

As with BLWTiles, GLWTiles are based on constructing corner colored Wang tiles. The construction process for GLWTiles uses the same method for determining which Wang tile an x, y point lies within (*i.e.,* the same tiling method). GLWTile construction also uses the same method as BLWTiles for determining what color samples are available (*i.e.,* the number of color samples, and therefore color labels, available are determined by dicing the exemplar up into a grid).

As the blending for GLWTiles is intended to blend different frequencies over regions of varying sizes, performing a bilinear blend of the combination of four colors over the entire tile would restrict the blending region to a single size. Instead, the four corner colors are overlaid and blending is performed over various regions of the tile such that the size of the region being blended can be modified for each frequency while still maintaining the generation of seamless tiles.

As seen in Figure 4.1, the tile is decomposed into five blending regions (the colored regions), and four non-blending regions (the white regions) located at each corner of the tile. Four of the blending regions are located at the edges of the tile and perform blending over the two corners that connect the region at each edge of the tile. The blending region located at the center of the tile joins all four colors by blending over the four corner colors. This decomposition of the blending regions enables the width and height of the blending region to be varied for each frequency being blended. At the same time, the quality of the seams between tiles is preserved as the blending always occurs over the corners composing an edge.

Burt and Adelson's multiresolution spline method[3] is modified so that it may be applied to the creation of a GLWTile as described in the following steps:

1. Compute Laplacian pyramid for C_0, C_1, C_2, and C_3.

2. Construct an output Laplacian pyramid to store result of blending the C_0, C_1, C_2, and C_3 pyramids.

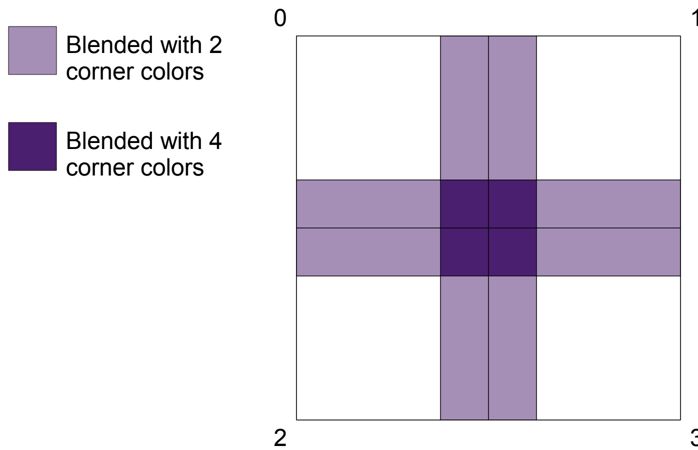


Figure 4.1: Blending Regions

3. For each level of the Laplacian pyramid, blend the four corner colors according to the rules for each bending region and assign the result to the output Laplacian pyramid.

4. The GLWTile is then reconstructed from the output Laplacian pyramid by performing the expand and sum operations as specified by Burt and Adelson[3].

The “Gauss” portion of GLWTiles comes from the use of generating Laplacian images as a difference between Gaussian pyramid levels. Note that as the GLWTile method is intended to be evaluated per pixel, the Laplacian pyramid of the exemplar may be precomputed and stored. Sample Gauss and Laplace images of a “leaf” exemplar and a “stone” exemplar are shown in Figures 4.2 and 4.3. One method for representing the Laplacian pyramid with imagery is to account for the negative difference values present in the Laplacian pyramid by taking the absolute value of each Laplacian pixel (which is a difference of two pixels from different levels of the Gaussian pyramid). An image of a Laplacian pyramid level created by using the absolute value of the differences will display the smaller differences as close to black and larger differences as being saturated. Another method is to add the maximum tone value to each pixel and then divide by 2.

Both methods are shown in the figures.

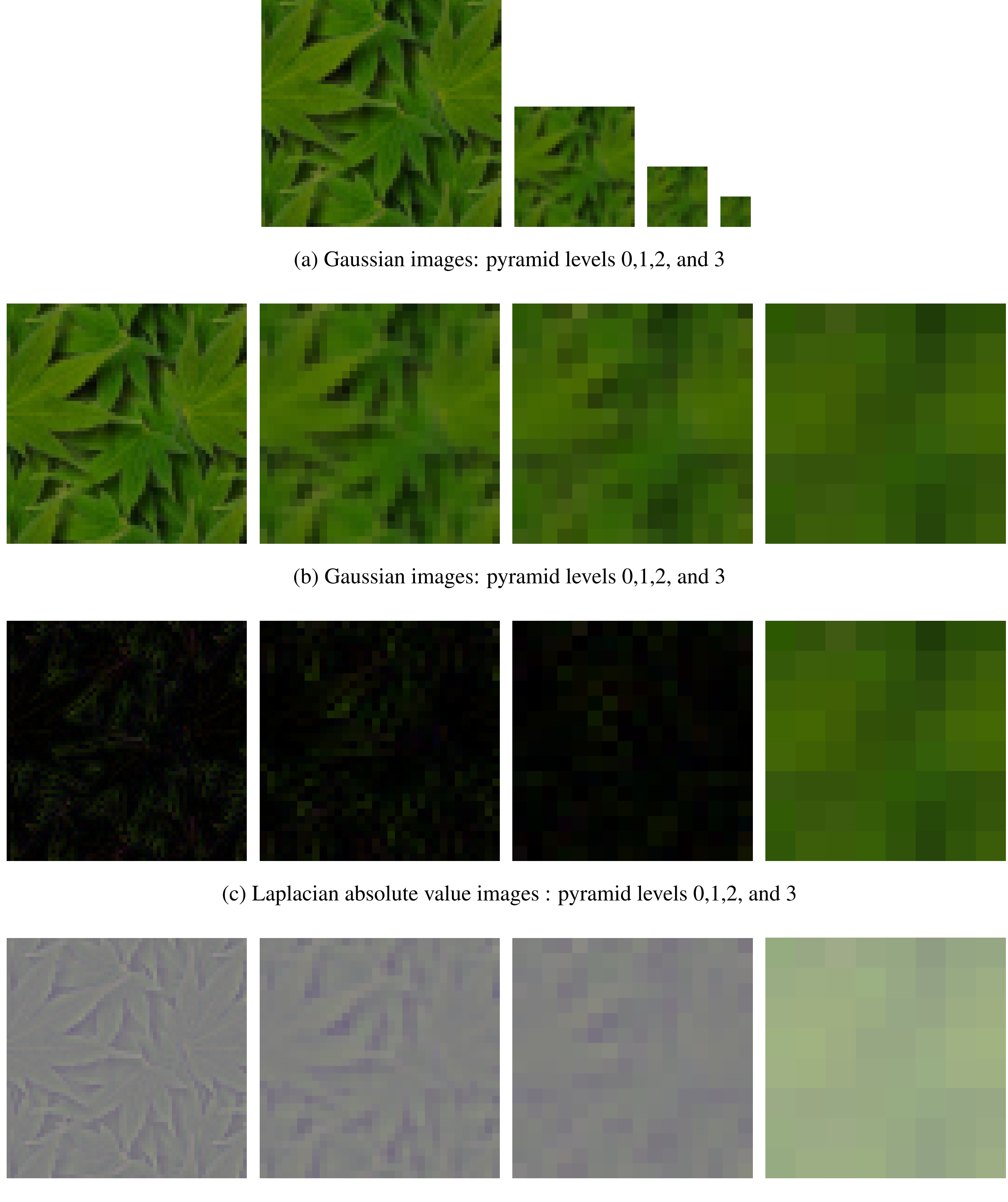
Recall that the highest level of the Laplacian pyramid is actually the same as the highest level of the Gaussian pyramid, therefore in the figures the highest level Laplacian images should appear the same as the highest level Gaussian image (of course the tone shifted method will result in the appearance of being different whereas the values are actually the same). In addition, for illustration purposes, the images of the higher levels of the pyramids may be shown at the same size as the initial image (i.e scaled up) when they are in fact subsampled and of smaller size.

Varying the size of the blending region for each frequency is realized by the subsampling mechanism of the pyramid. Each level of the Laplacian pyramid is a difference between two levels of the Gaussian pyramid expressing the error (or difference) between each Gaussian level. In both cases, each level of the pyramid is a smaller scale (and blurred) image of the previous level. The result of scaling and blurring at each pyramid level has the effect that an operation performed on a higher level of the Laplace pyramid operates on more pixels than in a lower level (due to the reconstruction process). In other words, blending over a fixed region size at each level of the Laplacian pyramid will operate on each frequency of the image over a different spatial region.

As the blending operations are applied from the highest level of the Laplacian pyramid to the lowest level of the Laplacian pyramid, the size of the region in terms of the reconstructed image will proceed from a large size to a small size. This directly corresponds to operating on a large region size for low frequencies (represented at the higher levels of the Laplacian pyramid) and operating on a small region size for high frequencies (represented at the lower levels of the Laplacian pyramid). The blending regions for each level of an example 3 level pyramid are shown in Figure 4.4.

To discuss the blending formulas applied at each region, Figure 4.5 labels each region with a number.

Assuming that for each level of the pyramid an overlap of 2 pixels is used as the blending region size, the blending for each pixel in the pyramid is simply a weighted sum of the colors



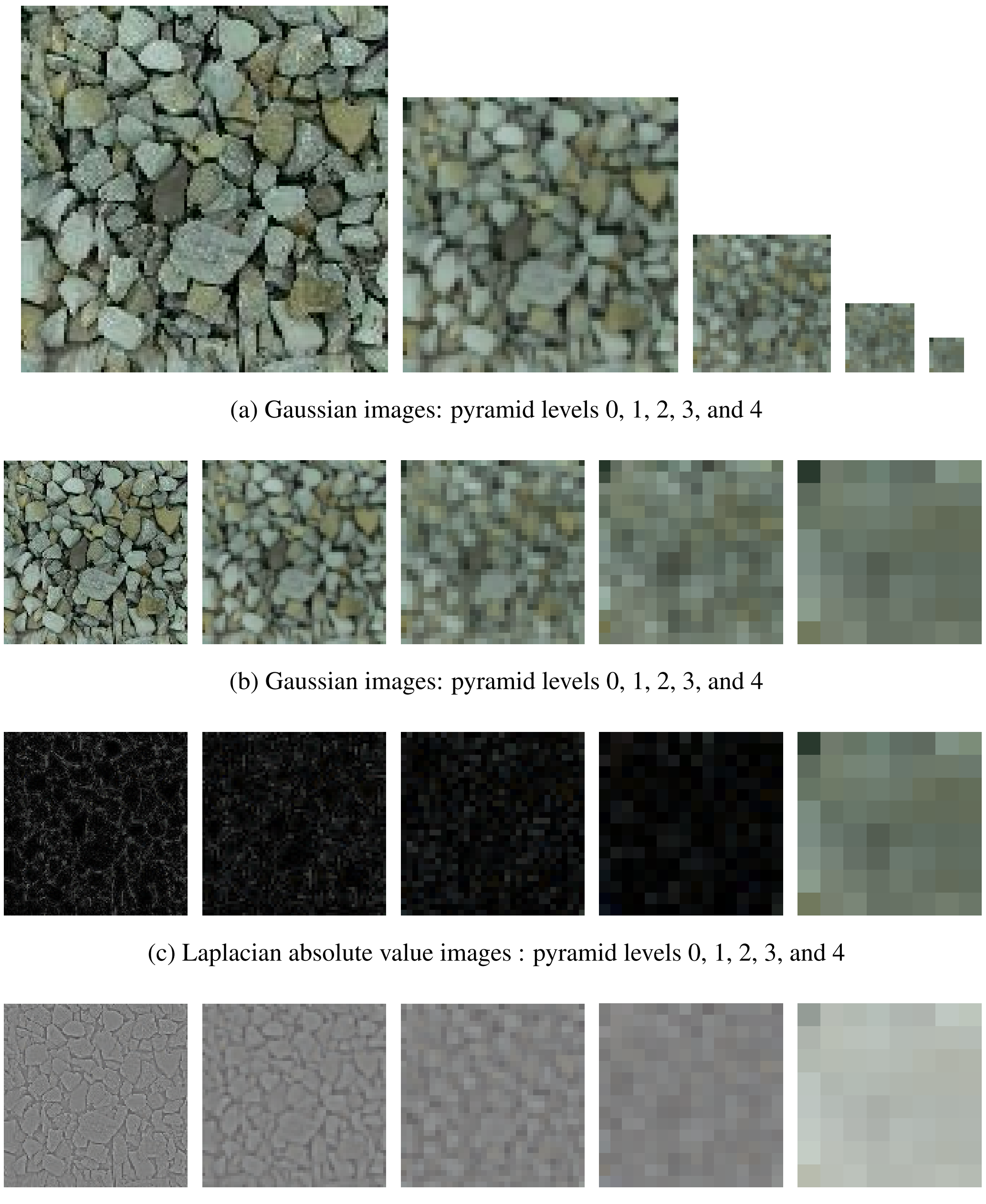
(a) Gaussian images: pyramid levels 0,1,2, and 3

(b) Gaussian images: pyramid levels 0,1,2, and 3

(c) Laplacian absolute value images : pyramid levels 0,1,2, and 3

(d) Laplacian tone shifted images: pyramid levels 0,1,2, and 3

Figure 4.2: Visualization of Gauss Laplace pyramids - Leafs.



(a) Gaussian images: pyramid levels 0, 1, 2, 3, and 4

(b) Gaussian images: pyramid levels 0, 1, 2, 3, and 4

(c) Laplacian absolute value images : pyramid levels 0, 1, 2, 3, and 4

(d) Laplacian tone shifted images: pyramid levels 0, 1, 2, 3, and 4

Figure 4.3: Visualization of Gauss Laplace pyramid - Stones.

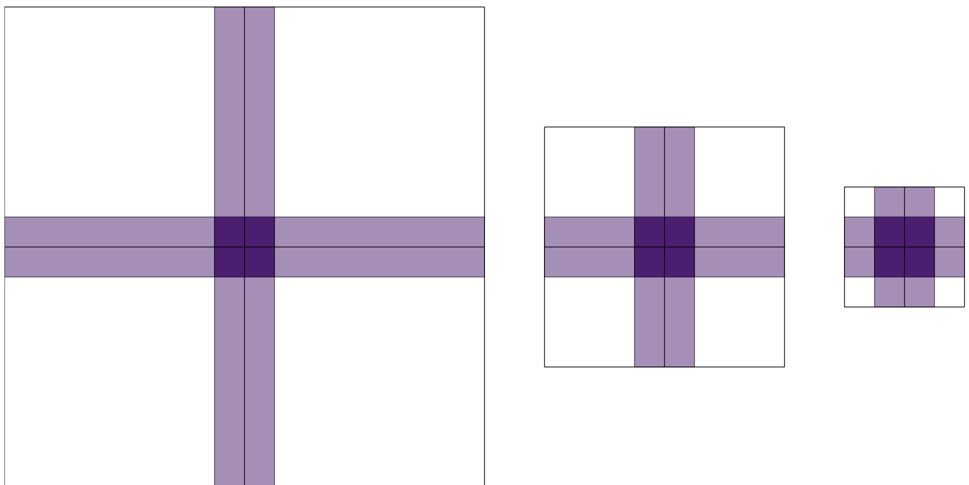


Figure 4.4: Fixed size Gauss Laplace blending region at each level of a pyramid. From left to right: Level 0, Level 1, Level 2

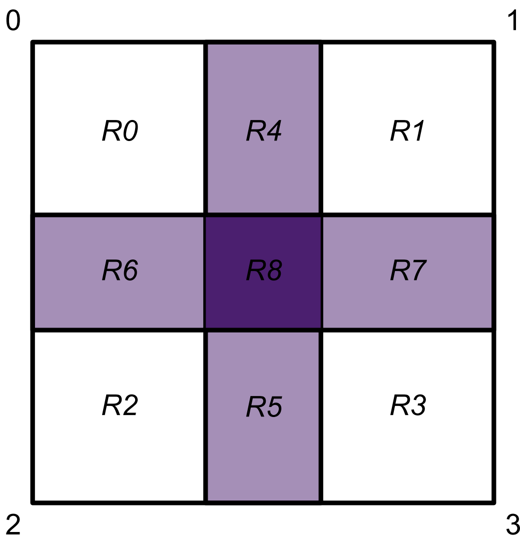


Figure 4.5: Convention used to number the various blending regions.

contributing to the blending region. Table 4.1 provides the formulas for each pixel on either side of the center of the blending region, where $weight$ is a value between 0 and 1 inclusive, and left/right and top/bottom refer to the pixel on either side of the middle of the blending region. Regions $R0$, $R1$, $R2$, $R3$ are not listed as no blending is performed and instead the values for the pyramid are taken directly from that of the corresponding colored corner. The table uses the same notation for expressing the evaluation of a colored corner's pixel as that discussed for the BLWTile technique (e.g. given a u and v location within a tile, and a corner color C_0, the pixel is specified as $C_0(u', v')$ where u' and v' are determined by which corner the color is being applied to).

Blend Region	Colors Blended	Formula
$R4$	C_0, C_1	left pixel: $weight * C_0(u, v)) + (1 - weight) * C_1(u - 1, v)$ right pixel: $(1 - weight) * C_0(u, v)) + weight * C_1(u - 1, v)$
$R5$	C_2, C_3	left pixel: $weight * C_2(u, v - 1)) + (1 - weight) * C_3(u - 1, v - 1)$ right pixel: $(1 - weight) * C_2(u, v - 1)) + weight * C_3(u - 1, v - 1)$
$R6$	C_0, C_2	top pixel: $weight * C_0(u, v)) + (1 - weight) * C_2(u, v - 1)$ bottom pixel: $(1 - weight) * C_0(u, v)) + weight * C_2(u, v - 1)$
$R7$	C_1, C_3	top pixel: $weight * C_1(u - 1, v)) + (1 - weight) * C_3(u - 1, v - 1)$ bottom pixel: $(1 - weight) * C_1(u - 1, v)) + weight * C_3(u - 1, v - 1)$
$R8$	C_0, C_1, C_2, C_3	all pixels: $0.25 * [C_0(u, v) + C_1(u - 1, v) + C_2(u - 1, v) + C_3(u - 1, v - 1)]$

Table 4.1: GLWTile blending region formulas using overlap of 2 pixels.

If $weight$ is equal to 0.5, then the blending results in a simple average of each color participating in the blend region. Blend region $R8$ is always a simple average of all four corner colors.

4.3 Methods Investigated

In addition to the methods presented in section 4.2, other variations were investigated including:

- Increasing the blending overlap region to sizes greater than 2 pixels (described in Chap-

ter 4.3.1).

- Per pixel evaluation for a Gauss Laplace Pyramid (described in Chapter 4.3.2).

4.3.1 Increasing Blending Region Size

It may be desirable for the blend regions to be blended over more pixels in a smooth manner as opposed to overlapping two pixels and performing a blend with a simple weighted sum. Extending the blending region size reduces discontinuities where the blend regions meet non-blended regions. The bilinear blending formulas used in BLWTile construction can also be used in the blending step for each level of the GLWTile re-construction providing for a more smooth junction between blended and non-blended regions.

When increasing the size of the blending region such that the blending region has an overlap of more than 2 pixels, the formulas in Table 4.2 may be used for each blending region. The formulas used for the increased overlap case perform either a bilinear or linear interpolation from 0 to 1 depending on the blending region. Each blending region has its bu, bv coordinate system centered at the top left corner of each blending region. (The top left corner of each blending region has $bu = 0$ and $bv = 0$). For each blending region, the origin for bu and bv has the positive direction of bu to the right and the positive direction of bv facing downward.

A linear blend is used for blend regions $R4$ and $R5$ where bu varies from 0 to 1 over the blending region width. When performing a linear blend for blend regions $R6$ and $R7$, bv varies from 0 to 1 over the blending region height. Combining the two, a bilinear blend is used for blend region $R8$ where bu and bv vary from 0 to 1 over the blending region's rectangle. Although the formulas used in the increased region size case can be used for an overlap of 2 or 3 pixels, the result of performing bilinear blending with pixel overlaps of less than 4 pixels is a blend over a smaller region than if the formulas in Table 4.1 were used.

In Table 4.2, bu and bv are the coordinates relative to the top left of each blending region while

Blend Region	Colors blended	Formula
$R4$	C_0, C_1	$(1 - bu) * C_0(u, v)) + bu * C_1(u - 1, v)$
$R5$	C_2, C_3	$(1 - bu) * C_2(u, v - 1)) + bu * C_3(u - 1, v - 1)$
$R6$	C_0, C_2	$(1 - bv) * C_0(u, v)) + bv * C_2(u, v - 1)$
$R7$	C_1, C_3	$(1 - bv) * C_1(u - 1, v)) + bv * C_3(u - 1, v - 1)$
$R8$	C_0, C_1, C_2, C_3	$(1 - bv) * [(1 - bu) * C_0(u, v) + bu * C_1(u - 1, v)]$ $+bv * [(1 - bu) * C_2(u, v - 1) + bu * C_3(u - 1, v - 1)]$

Table 4.2: GLWTile blending region formulas for overlaps greater than 2 pixels.

u and v are the coordinates relative to the top left of the tile being generated. The method used to interpolate bu and bv from 0 to 1 can also use the 3 methods for interpolating as described in constructing BLWTiles (linear, t3, and perlin).

4.3.2 Per Pixel Reconstruction for Gauss Laplace Pyramid

Although computation of the GLWTiles can be performed on a per tile basis, being able to evaluate a GLWTile on a per pixel basis provides for greater flexibility (such as mapping the evaluation to a GPU fragment shader program). In order to evaluate the reconstructed pixel value within a GLWTile, each corner color at each level of the Laplacian pyramid must be referenced (due to the blending and the reconstruction process).

The number of pixels in one level of the Laplacian pyramid that contribute to the next lower level of the reconstructed pyramid is limited. The reconstruction process for a pixel at level N only requires up to 1 pixel on either side of the same pixel at level N+1 (due to subsampling and the size of the Gaussian kernel used for the reduce operation). Therefore, at most 3 pixels in one dimension (*i.e.,* 9 pixels in 2 dimensions) contribute to the reconstruction process of a pixel when moving from level $N + 1$ to level N.

The number of pixels that contribute to the reconstruction of a single pixel at level 0 varies based on whether the pixel is even or odd (3 pixels for even and 2 pixels for odd). Using Laplacian

pyramids bounds the number of pixels required to reconstruct a pixel at any level of the pyramid to 4 in one dimension (*i.e.,* a 4×4 sample in 2 dimensions). An algorithm for determining which level N pixels are required for reconstruction given level $N-1$ is shown in Figure 4.6 where $LeftX$ and $RightX$ refer to the "left" and "right" most pixels that are required by level $N-1$ to reconstruct a pixel at level 0. Although the algorithm is provided for the x dimension, the same algorithm applies for the y dimension.

$$
\begin{aligned}
&if\ prevLeftX\ is\ odd \\
&\qquad curLeftX \leftarrow (prevLeftX - 1) \div 2 \\
&else \\
&\qquad curLeftX \leftarrow (prevLeftX - 2) \div 2 \\
&if\ prevRightX\ is\ odd \\
&\qquad curRightX \leftarrow (prevRightX + 1) \div 2 \\
&else \\
&\qquad curRightX \leftarrow (prevRightX + 2) \div 2 \\
&xPixelsRequired \leftarrow (curRightX - curLeftX) + 1
\end{aligned}
$$

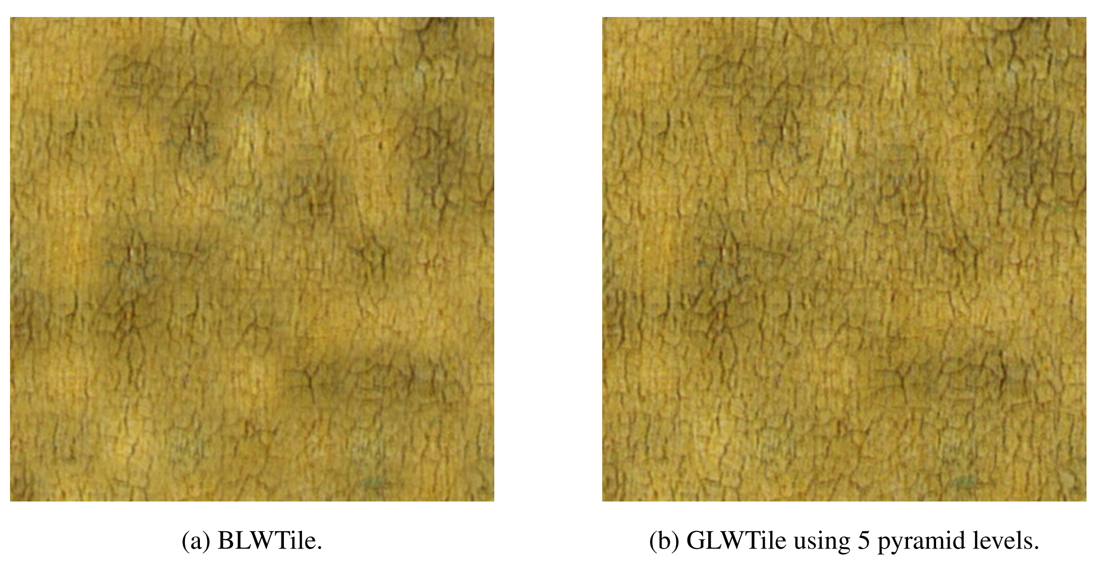
Figure 4.6: Determining pyramid pixels required for reconstruction at level N (in the x dimension)

To reconstruct a pixel in a GLWTile on a per pixel basis, blending is performed at each level according to the blend rules for the region, but only for the pixels at each level that contribute to the level 0 reconstructed pixel. This can be accomplished by reserving a 4×4 sample region for each level of the pyramid, copying the appropriate pixels from each corner color's Laplacian pyramid, and then performing a reconstruction using the 4×4 pyramids. Each level of the 4×4 pyramid samples maintain a mapping to the absolute location of the pixels it stores along with its width and height.

The results that follow in Chapter 4.4 were generated using this per pixel approach.

4.4 Results

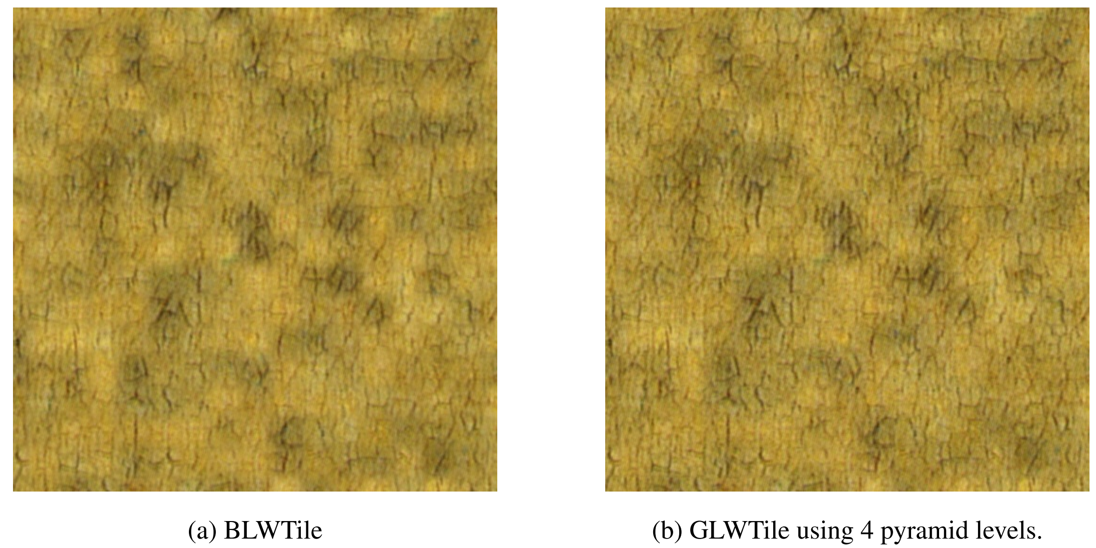
Some of the results of using the GLWTile method were not very different from those generated by BLWTiles. Those results for the large exemplars that are not very different are not shown here. The GLWTile results are shown next to the results of the BLWTile method to illustrate the differences.



(a) BLWTile.

(b) GLWTile using 5 pyramid levels.

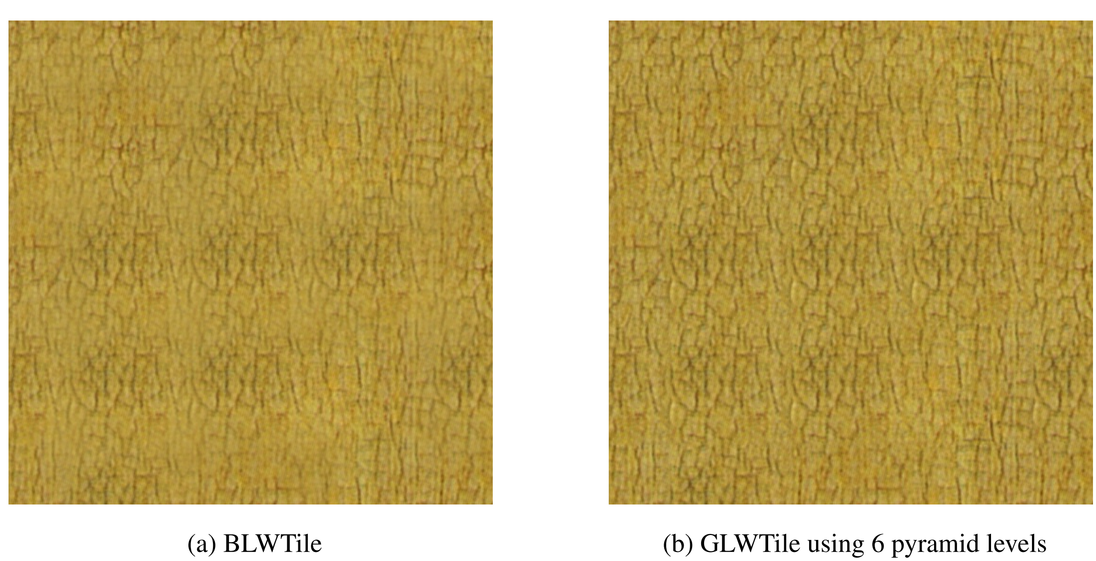
Figure 4.7: GLWTile results for "MonaCanvas". Result of size 512×512. An 8×8 grid of tiles. Each tile of size 64×64. 96 colors available.



(a) BLWTile

(b) GLWTile using 4 pyramid levels.

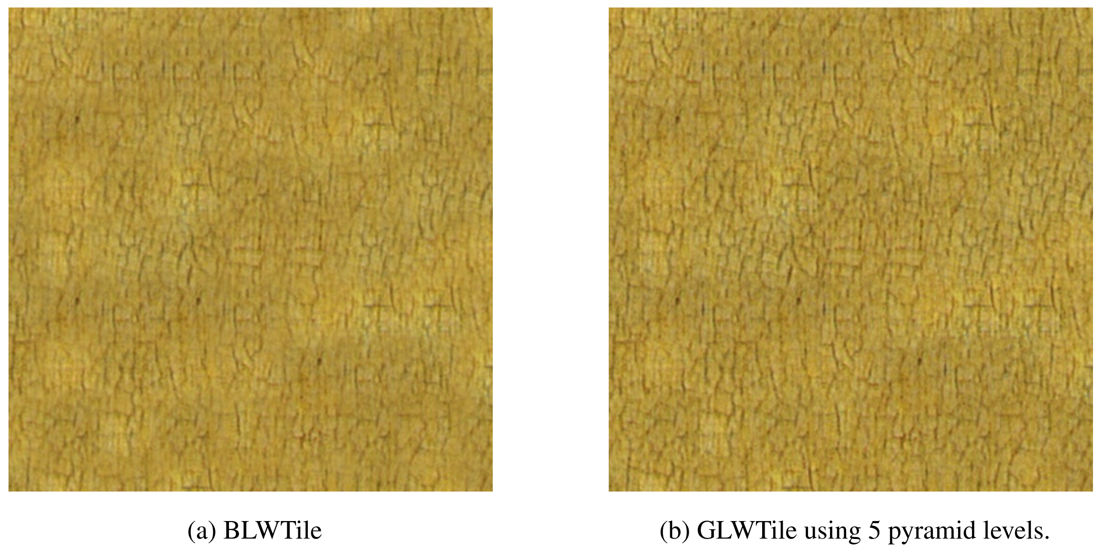
Figure 4.8: GLWTile results for “MonaCanvas”. Result of size 512×512. A 16×16 grid of tiles. Each tile of size 32×32. 468 colors available.



(a) BLWTile

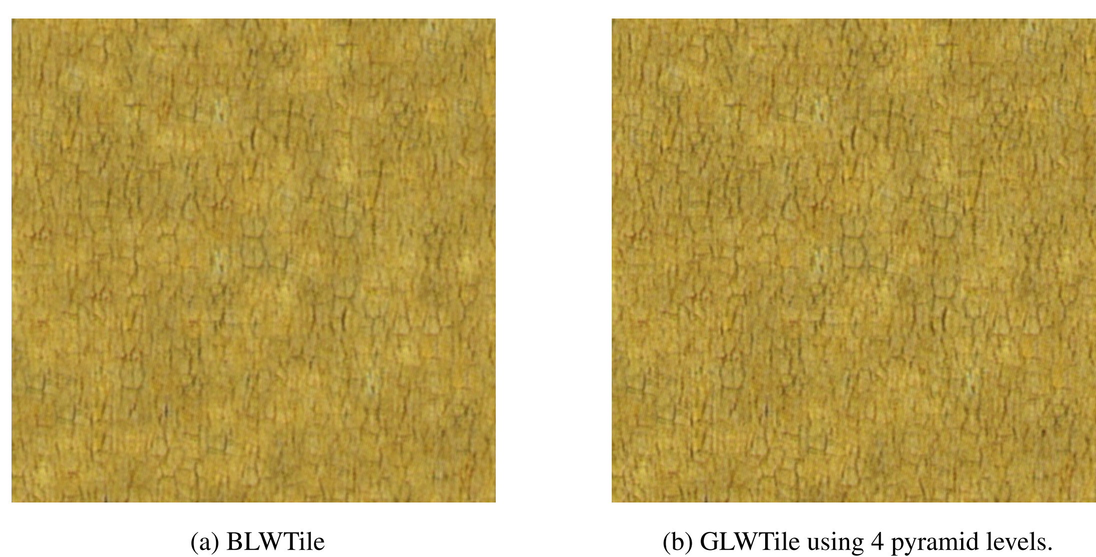
(b) GLWTile using 6 pyramid levels

Figure 4.9: GLWTile results for “MonaCanvas2”. Result of size 512×512. A 4×4 grid of tiles. Each tile of size 128×128. 3 colors available.



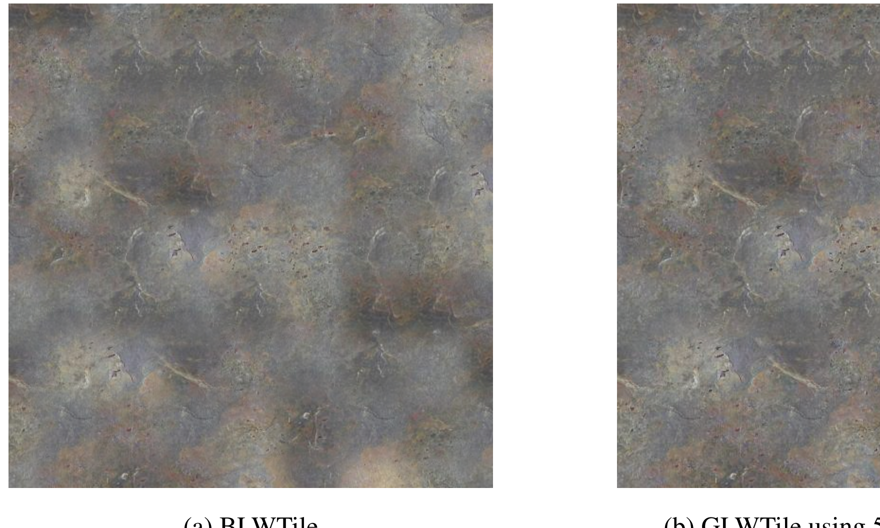
(a) BLWTile (b) GLWTile using 5 pyramid levels.

Figure 4.10: GLWTile results for “MonaCanvas2”. Result of size 512×512. An 8×8 grid of tiles. Each tile of size 64×64. 21 colors available.



(a) BLWTile (b) GLWTile using 4 pyramid levels.

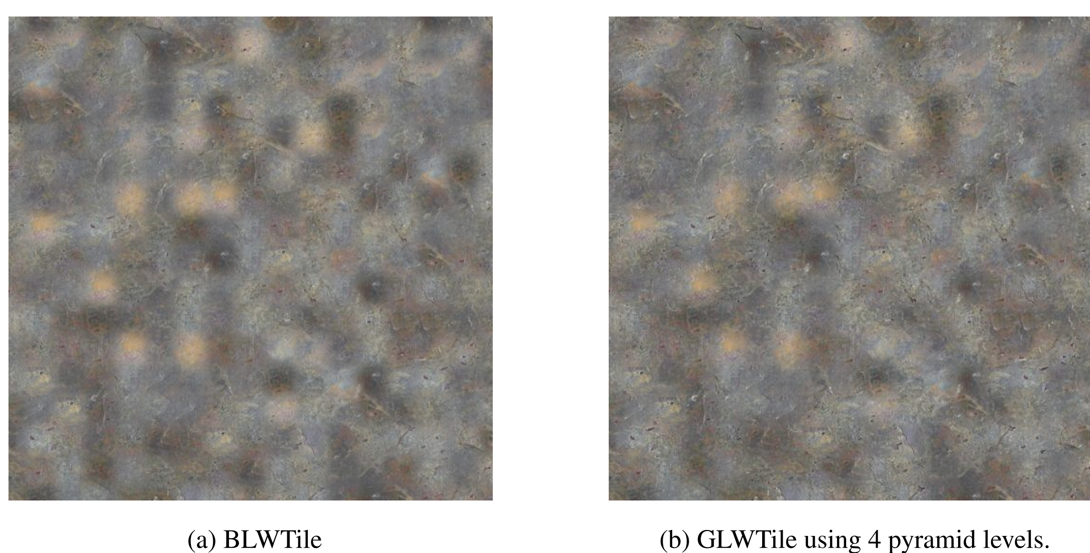
Figure 4.11: GLWTile results for “MonaCanvas2”. Result of size 512×512. A 16×16 grid of tiles. Each tile of size 32×32. 105 colors available.



(a) BLWTile

(b) GLWTile using 5 pyramid levels.

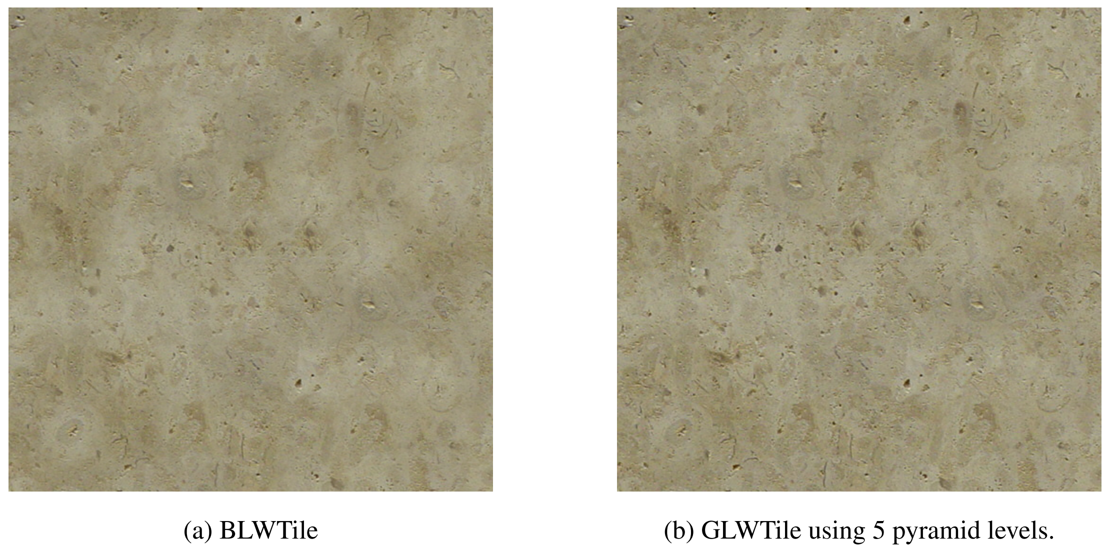
Figure 4.12: GLWTile results for “Slate”. Result of size 512×512. An 8×8 grid of tiles. Each tile of size 64×64. 25 colors available.



(a) BLWTile

(b) GLWTile using 4 pyramid levels.

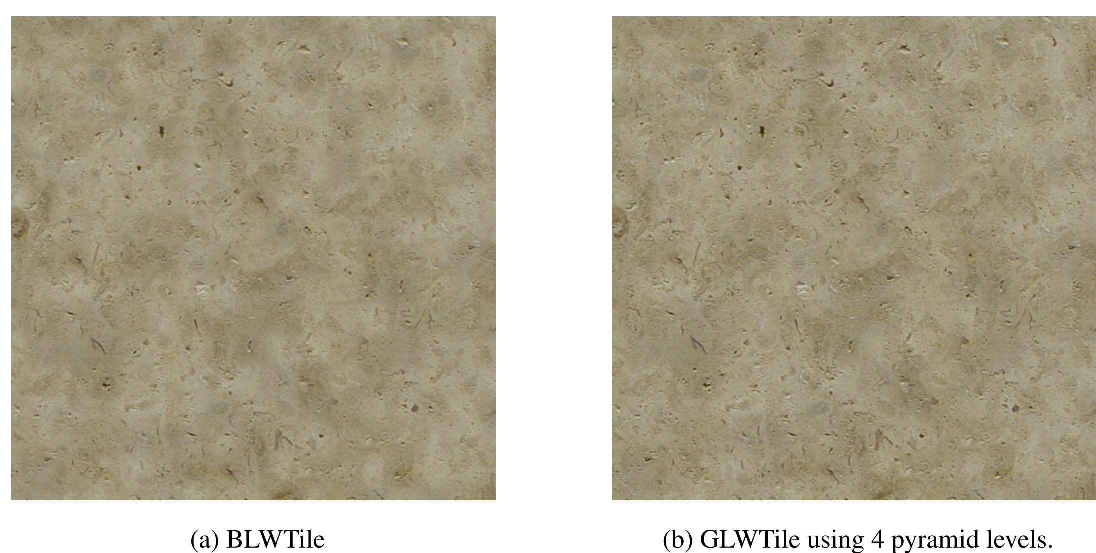
Figure 4.13: GLWTile results for “Slate”. Result of size 512×512. A 16×16 grid of tiles. Each tile of size 32×32. 121 colors available.



(a) BLWTile

(b) GLWTile using 5 pyramid levels.

Figure 4.14: GLWTile results for “Shell Stone Tile”. Result of size 512×512. An 8×8 grid of tiles. Each tile of size 64×64. 100 colors available.



(a) BLWTile

(b) GLWTile using 4 pyramid levels.

Figure 4.15: GLWTile results for “Shell Stone Tile”. Result of size 512×512. A 16×16 grid of tiles. Each tile of size 32×32. 441 colors available.

Name	Exemplar	BLWTile result	GLWTile result	Levels
exemplars_7				3
exemplars_9				4
exemplars_brug				2
exemplars_olives128				3

Figure 4.16: Source and GLWTile results. Source is 128×128. Result is 256×256 using an 8×8 grid of tiles sized 32×32. 9 colors are available. Exception is exemplars_brug which uses a 16×16 grid of tiles sized 16×16 with 49 colors available.

Name	Exemplar	BLWTile result	GLWTile result	Levels
exemplars_salines_128				3
exemplars_S27_m				3
exemplars_longgrass				3
exemplars_redyellow				3

Figure 4.17: Source and GLWTile results. Source is 128×128. Result is 256×256 using an 8×8 grid of tiles sized 32×32. 9 colors are available. Exception is exemplars_redyellow which uses a 16×16 grid of tiles sized 16×16 with 49 colors available.

4.5 Analysis

The GLWTile method did reduce the blurring effect when compared to the BLWTile method as most noticeable for exemplars_9 in Figure 4.16 and for exemplars_S27_m in Figure 4.17. As the GLWTile method uses the same underlying grid of colors as the BLWTile method, it has similar failure cases. For many of the larger exemplars, reducing the blurring helped reduce or eliminate a gridding effect.

The number of pyramid levels able to be created is based on the size of the exemplar. Most of the 128×128 exemplars showed best results using 3 pyramid levels, however some textures showed better results using 2 or 4 pyramid levels.

4.6 Conclusion

The GLWTile method helps to produce textures that are less blurry than the equivalent BLWTile method. In many cases the difference between the GLWTile method and the BLWTile method is minor. In those cases it would be best to use the BLWTile method as the BLWTile method only needs to process at most 4 pixels to generate a pixel in the output while the GLWTile method must process up to a 4×4 set of pixels for each level of the pyramid to generate an output pixel.

Future work includes that specified for BLWTiles. In addition, for both GLWTiles and BLWTiles methods, it may be possible to modify the color selection to enable selecting four colors in which case a set of Wang tiles could be precomputed rather than computing the pixels procedurally.

Chapter 5

SWTiles

Texture Synthesis techniques have been used to synthesize textures from an exemplar as discussed in Chapter 1.3 and Chapter 2.4. The Synthesized Wang Tile (SWTile) technique introduced here modifies a texture synthesis algorithm devised by Kwatra et al to generate a set of Wang tiles[17].

Synthesizing Wang tiles is made possible by using colored corners as opposed to colored edges. The following subsections provide information on the methods used, the results, and an analysis.

5.1 Hypothesis

The general hypothesis is that it should be possible to generate a set of Wang tiles by using a synthesis technique that incorporates a concept of "immutable" or unchangeable regions during synthesis to ensure that corners and edges of tiles match. Artifacts in the base synthesis technique are expected to be present in the generated tiles. If formulated correctly, the transition from tile to tile should not be discernible.

5.2 Initial Approach

The approach taken to create a tile by synthesis is to build the tile incrementally. First, the color corners are sampled from the exemplar. Each step after this contains a synthesis step where some regions of the area are to be synthesized, while other regions are considered unchangeable or immutable. The synthesis algorithm must be modified in order to integrate the process of maintaining the immutable region. The color corners are used during the synthesis of edges where the ends of the edge are comprised of the immutable colors and the region between the ends is synthesized. The edges are then used to construct a border of a tile. The border of the tile is kept constant while the inside of the tile is synthesized.

The basic algorithm is as follows:

1. *Select corner colors from the exemplar*

2. *For each combination of 2 colors*

   - construct a horizontal edge with the 2 colors at opposing ends

   - synthesize the horizontal edge while preserving the immutable color regions

   - construct a vertical edge with the 2 colors at opposing ends

   - synthesize the vertical edge while preserving the immutable color regions

3. *For each valid combination of four synthesized edges (two vertical and two horizontal)*

   - construct a tile with the four synthesized edges on the border

   - synthesize the tile while preserving the immutable edge regions

After following these steps all of the possible Wang tiles will be synthesized.

5.2.1 Constructing an Example Tile $t0110$

In describing the construction of the tile $t0110$ the following naming convention is used for naming colors, edges, and tiles. The colors selected are given names that are a number prefixed by 'c'. In this example two colors will be used to define the Wang tile set, therefore the colors in the set will be $c0$ and $c1$.

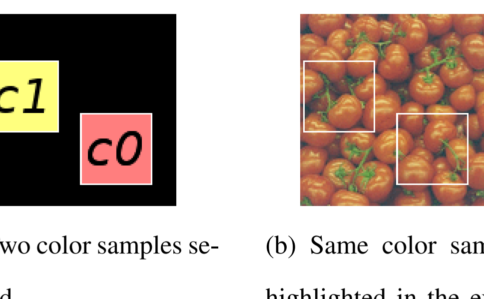
The edge name is given by listing the two numbers of the corner colors used to construct the edge prefixed by either an 'h' for a horizontal edge or a 'v' for a vertical edge. For a horizontal edge the edge name is formed by listing the number of the color on the left side of the edge followed by the number of the color on the right side of the edge. If a horizontal edge has $c0$ on the left and $c1$ on the right, the edge name is $h01$.

For a vertical edge, the name is formed by listing the number of the color on the bottom side of the edge followed by the number of the color on the top side of the edge. For example, if the bottom color is $c0$ and the top color is $c1$, the edge is named $v01$.

The tile is given a name by listing the numbers of the colors used in clockwise order starting from the bottom left corner of the tile and prefixed by 't'.

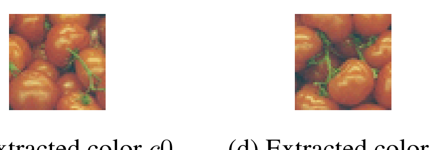
As a prerequisite for constructing the tile $t0110$, two colors must be selected. The selection of colors will occur only once (*i.e.,* the same colors are used in different combinations to form the entire Wang tile set). Figure 5.1 shows an example of selecting two colors to be used in the construction of the Wang tile set.

Once the corner colors are selected, the next step in constructing the tile $t0110$ is to construct and synthesize the edges. During synthesis of the edges there is no distinction between synthesizing a left vertical edge versus synthesizing a right vertical edge. Similarly, there is no distinction between synthesizing a top horizontal edge versus a bottom horizontal edge. In either case (left-/right or top/bottom) the synthesis process is the same. In the case of tile $t0110$, three distinct edges must be used:



(a) Two color samples selected.

(b) Same color samples highlighted in the exemplar.



(c) Extracted color $c0$.

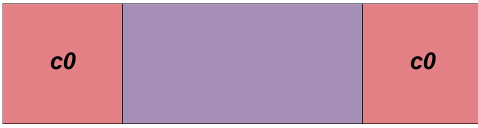
(d) Extracted color $c1$.

Figure 5.1: SWTiles - Selecting two colors.

1. Horizontal edge $h00$ for the bottom edge
2. Horizontal edge $h11$ for the top edge
3. And vertical edge $v01$ for the left and right edges

Figure 5.2 shows the construction of the horizontal edge $h00$, Figure 5.3 shows the construction of horizontal edge $h11$, and Figure 5.4 shows the construction of the vertical edge $v01$. Each edge is only constructed and synthesized once. After the edge is synthesized it is saved and used for other tiles that require the same edge.

Once the edges are constructed and synthesized, the tile can be constructed by placing the edges $h00$, $h11$, and $v01$ as in Figure 5.5. The inside of the tile can then be synthesized. After the tile is synthesized, the content for the tile $t0110$ to be included in the Wang tile set must be cut out from the synthesized tile as shown in Figure 5.6. Cutting the tiles out by splitting down the edges ensures that tiles that are allowed to be adjacent tiles have no seams. Other tiles can then be constructed by following the same steps.

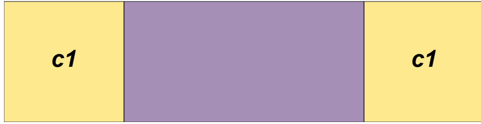


(a) Representation of edge to construct.

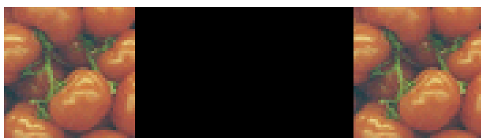
(b) Edge with colors placed at corners.

(c) Edge after synthesis.

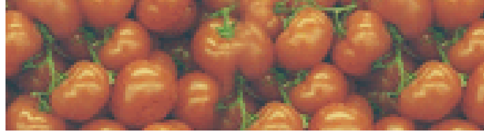
Figure 5.2: Constructing horizontal edge $h00$.



(a) Representation of edge to construct.

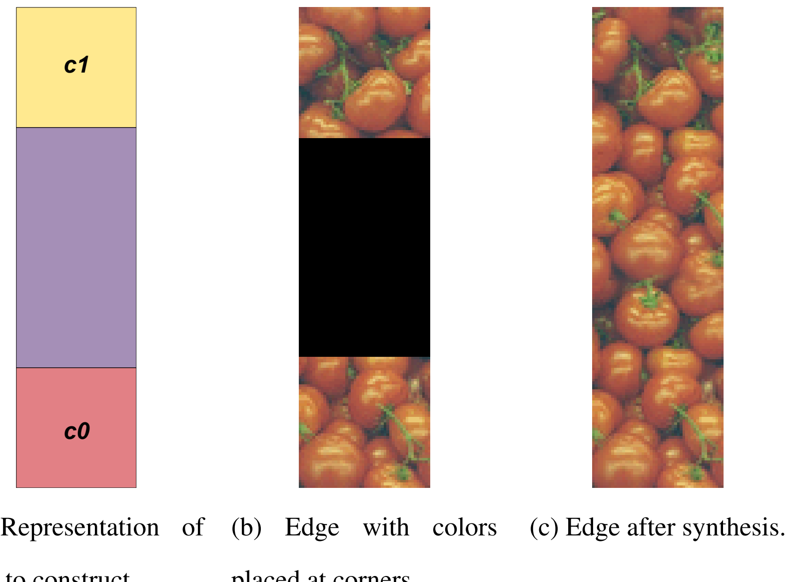


(b) Edge with colors placed at corners.



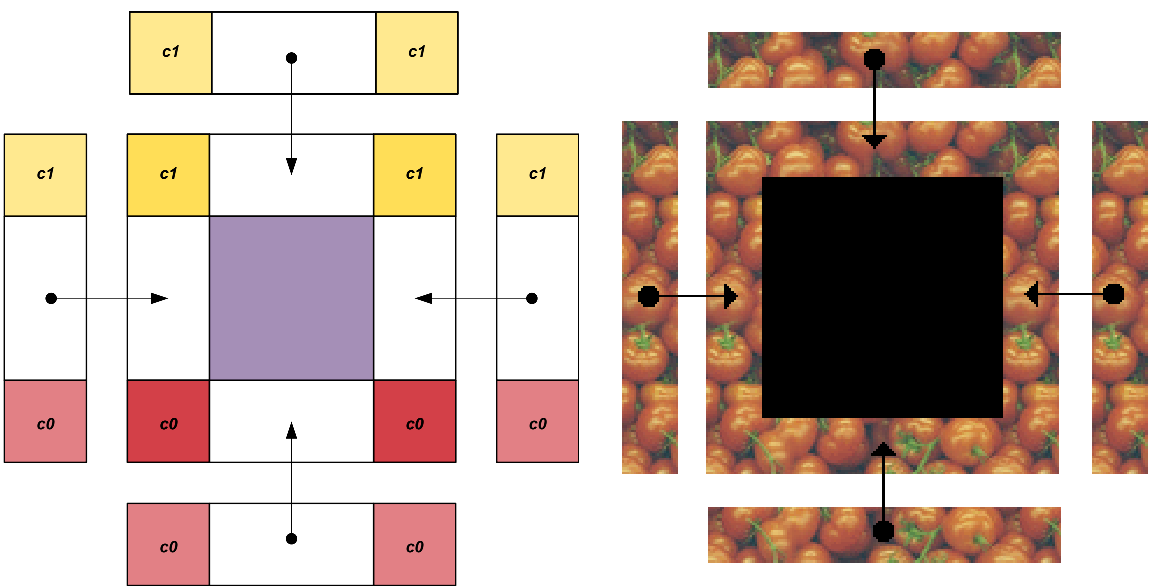
(c) Edge after synthesis.

Figure 5.3: Constructing horizontal edge $h11$.



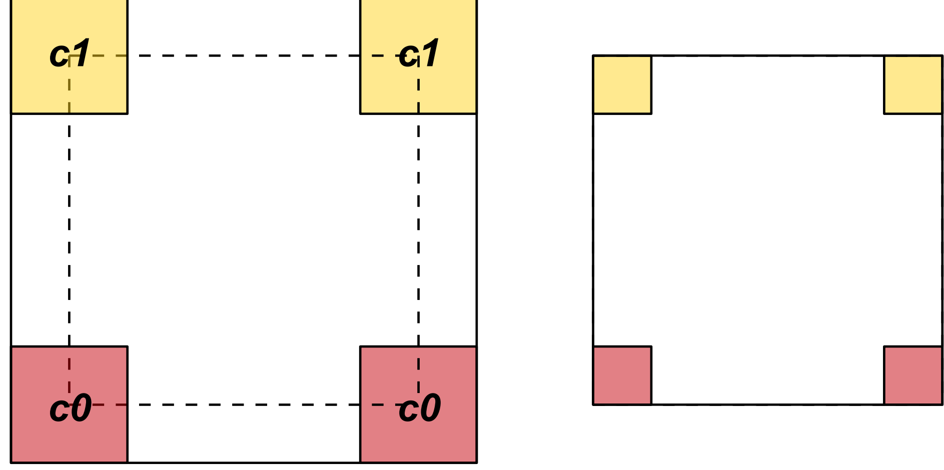
(a) Representation of edge to construct.
(b) Edge with colors placed at corners.
(c) Edge after synthesis.

Figure 5.4: Constructing vertical edge $v01$.

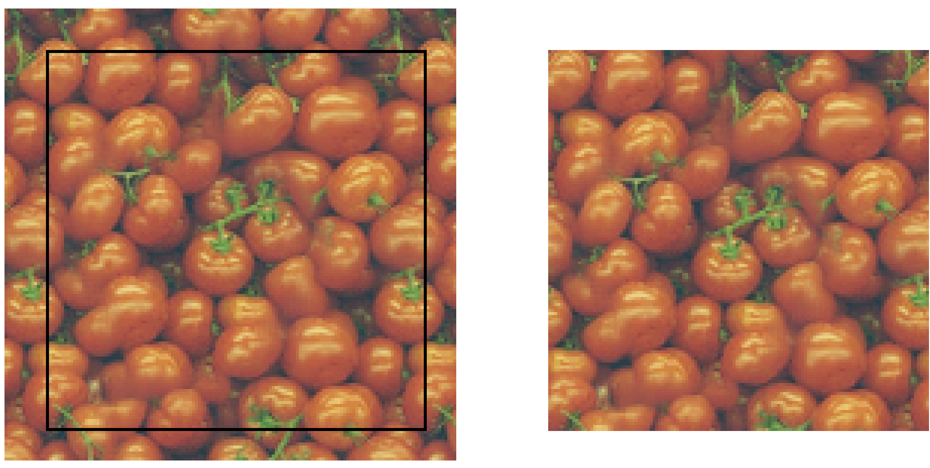


(a) Representation of tile construction using edges.
(b) Tile construction using synthesized edges.

Figure 5.5: Constructing a SWTile for synthesis.



(a) Representation of cutting out a SWTile.



(b) Cutting out a SWTile.

Figure 5.6: Cutting the SWTile out of the synthesized tile. Synthesized tile on the left and SWTile on the right.

Although edges could be synthesized as needed and then cached, the approach selected was to select all the colors first, then synthesize all the edges, followed by synthesizing all of the tiles. Figure 5.7 shows the construction of all of the horizontal edges and Figure 5.8 shows the construction of all of the vertical edges given a Wang tile set with two colors ($c0$ and $c1$).

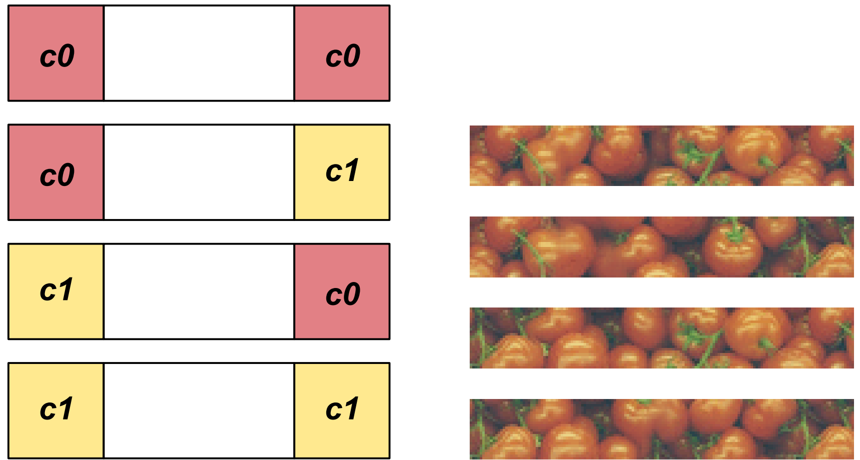


Figure 5.7: Horizontal edge construction for two colors.

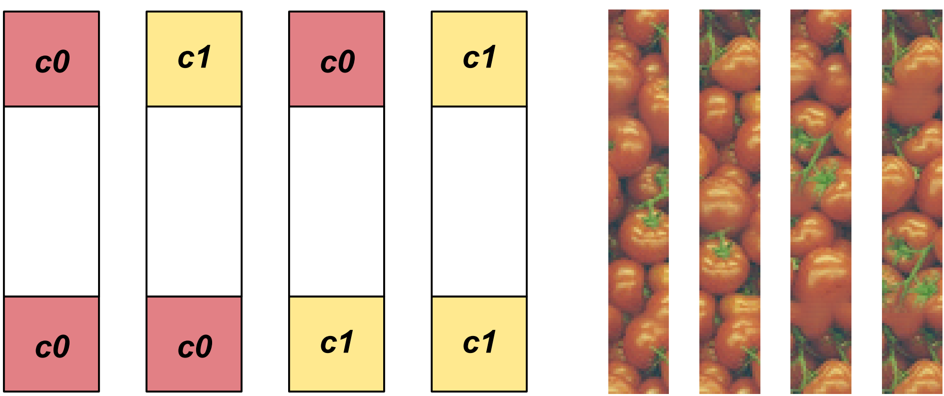


Figure 5.8: Vertical edge construction for two colors.

As shown in Figure 5.6, the method of constructing a tile ensures that valid adjacent tiles will match due to the use of corners. It is important to realize that only one quarter of a corner color and one half of the synthesized edge will be used in a single tile. The area where the corners for

four Wang tiles meet is covered by a single corner color where one quarter of the corner color is contained in each tile. Similarly, the area where the edges for two Wang tiles meet is covered by a single synthesized edge where half of the edge is contained in each Wang tile.

5.3 Methods Investigated

Although the steps provided in the previous section detail the basic approach selected at the start of the investigation, no mention was made as to how the synthesis technique was modified to preserve the immutable nature of the color corners during edge synthesis and the immutable nature of the edges during tile synthesis. This section details the modification to the synthesis algorithm to honor the immutable nature of the desired regions being synthesized. In addition, this section discusses the investigation of variations made to the basic Wang tile synthesis algorithm to increase the quality of the Wang tiles created in regards to minimizing synthesis artifacts as well as increasing diversity of the tiles created.

5.3.1 Honoring Immutables

Kwatra's algorithm used for texture synthesis is comprised of two main steps, a best fit search step and a minimization step. The exemplar (or source image) is called the Z image and the image being synthesized is called the X image. The algorithm creates neighborhoods in both the Z image and the X image. The neighborhoods are created on a regular grid and are allowed to overlap. The spacing of the neighborhoods for the Z image may be different than the spacing between neighborhoods in the X image.

The terminology used in Kwatra's algorithm for describing the two steps of the the synthesis process are Expectation, or E-step (minimizing the texture energy), and Maximization, or M-step (finding set of closest input neighborhoods). In this paper, those terms will be references as the minimization step and the best fit search step as it should help avoid confusion between whether

an M-step is a match step, maximization step, or a minimization step, etc.

At various iterations of the algorithm the neighborhood size is changed. Typically three neighborhood sizes will be used: 32×32, 16×16, and 8×8 (listed in the order applied). In addition the synthesis may be repeated at various image sizes (resolutions) where larger X images use the previous (and smaller) X image to initialize X. During the best fit search step, each X neighborhood is compared to each Z neighborhood to find the closest match (*i.e.,* best fit). The energy of the system is the sum of the differences for all X neighborhood-Z neighborhood pairs. During the energy minimization step, the energy minimization may be performed for each pixel in X by taking all of the X neighborhoods that overlap a pixel and averaging the pixel's color values as dictated by each X neighborhood. The pixel color dictated by the X neighborhood is the color of the pixel of the corresponding best fit Z neighborhood at the same relative offset that the pixel is at within the X neighborhood.

Kwatra included a "controlled" texture synthesis variant that would enable pixels in the X image to remain unchanged. Applying this to a method for constructing a tile or edge, the control pixels are the immutable pixels. When synthesizing a texture containing immutables (an edge or a tile) the immutable regions of the edge or tile being synthesized must remain unchanged in the final synthesized output. The minimization step of the algorithm is the only step of the synthesis algorithm where changes are made to the texture being synthesized (the X pixels). Therefore, during the minimization step, when processing a pixel which is contained in an immutable region, the immutable pixel is left unchanged and the minimization continues with the non-immutable pixels. Figure 5.9 illustrates the immutable regions marked as white and the mutable regions marked as black for a horizontal edge, a vertical edge, and a tile. Due to artifacts that result from having all pixels be either completely immutable or completely mutable, a variation of this approach was used and is discussed in section 5.3.2.

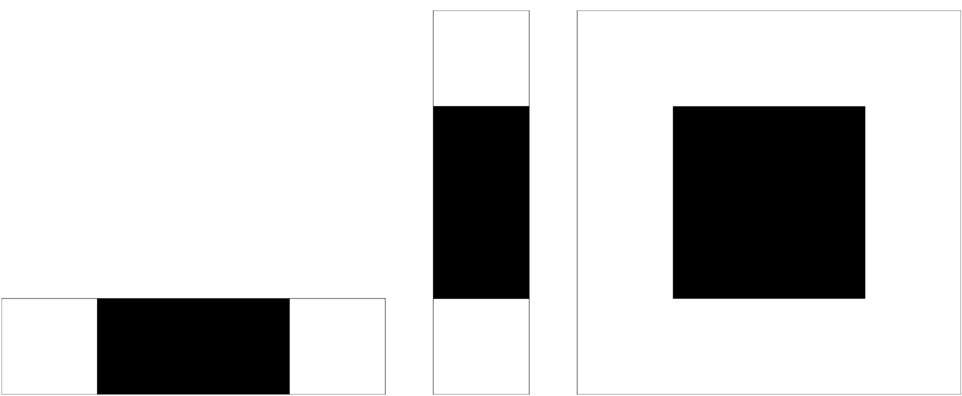


Figure 5.9: Immutable regions (in white) for horizontal edge, vertical edge, and tile synthesis.

5.3.2 Partially Immutable Pixels

Synthesizing edges and tiles only using pixels that are fully immutable or fully mutable resulted in textures with sharp lines between the regions that were mutable and those regions that were immutable. The immutable region only extends its influence on the synthesis algorithm beyond the immutable region to a minor extent and often fails to significantly influence the synthesis, resulting in noticeable seam artifacts between immutable and mutable regions.

In order to increase the quality of the SWTiles, the influence of the immutable pixels were extended beyond the immutable region yielding a better synthesis result. To achieve the effect of extending the influence of the immutable pixels beyond the immutable region, the initial immutable region chosen during tile (or edge) construction is reduced and padded with a border of pixels that are allowed to be partially immutable. The partially immutable pixels are given an initial percentage of immutability. As the synthesis progresses through the different resolution levels and the different neighborhood sizes, the partially immutable pixels become increasingly less immutable (*i.e.,* increasingly more mutable).

The method to apply partially immutable pixels during synthesis is to modify the minimization step. During the minimization step for completely immutable pixels, the pixels are left unchanged.

For completely mutable pixels, the pixels average the contribution of each of the X neighborhoods that overlap the pixel. Determining the pixel value for a partially immutable pixel during the minimization step uses a weighted sum of the partially immutable pixel's value and the values that would have been assigned to the pixel if it were mutable. The weight assigned to the partially immutable pixel is the percentage of immutability. The weight assigned to the averaged X neighborhood contributions is one minus the percentage of immutability. The contribution of the partially immutable pixel always derives from the source immutable region and not from the previously synthesized X image (*i.e.,* it always samples the initial immutable region that was extracted from the Z image).

An example of partially immutable percentages that could be used if the synthesis is configured to use 1 resolution level and 3 neighborhood sizes (32×32, 16×16, and 8×8), is given in Table 5.1.

	neighborhood size 32×32	neighborhood size 16×16	neighborhood size 8×8
Percent Immutable	80%	20%	0%

Table 5.1: Example percentages of immutability for partially immutable pixels for varying neighborhood sizes.

Notice that the first neighborhood size used in synthesis (32×32) has the largest percentage of immutability and that the percentage of immutability decreases with each next neighborhood size used in the synthesis.

If the synthesis is configured to use 3 resolution levels (images sizes of 128×128, 64×64, and 32×32) with 3 neighborhood sizes (32×32, 16×16, 8×8), then the partially immutable pixels could use the immutable percentages as in Table 5.2.

Notice that the smallest image size (the first row in the Table 5.2) has the highest percentage of immutability since it is used first during the synthesis process. As the X image size increases, the percentage of immutability decreases.

Resolution (X Image Size)	neighborhood size 32×32	neighborhood size 16×16	neighborhood size 8×8
32×32	100%	90%	80%
64×64	40%	30%	20%
128×128	10%	5%	0%

Table 5.2: Example percentages of immutability for partially immutable pixels given a resolution level and neighborhood size.

Allowing the partially immutable region to start at a high percentage of immutability and then decrease its contribution to 0 percent immutable enables the synthesis algorithm to first find matching Z neighborhoods that maintain consistency with the immutable region and the newly synthesized region while favoring neighborhoods that more closely match the values of the partially immutable pixels. As the partially immutable pixel's percentage of immutability decreases, the synthesis algorithm is able to replace the partially immutable pixels that provide for poor neighborhood matches with new pixels that provide for a better match of neighborhoods, reducing the seam between the immutable and mutable regions. In this manner the constraints for creating Wang tiles for having matching corners and edges are preserved while enabling the synthesis technique to synthesize a Wang tile with pixels that are consistent with the Z Image.

Kwatra's synthesis algorithm uses multiple iterations at each resolution level and neighborhood size combination. Keeping the percentage of immutability constant for partially immutable pixels for each iteration can prove problematic for certain textures or when using only one or two resolution levels. Therefore, the percentage of immutability is also modified based on how many iterations have been made for the current resolution level and neighborhood size. Combining the percentage of immutability decrease for each resolution and neighborhood size with the the decrease in immutability for each iteration provides for a progressively smaller influence of the partially immutable pixels. This enables the synthesis algorithm to initially favor neighborhoods that match the immutable region (even at the expense of additional energy from exemplar

mismatches) and then as the percentage of immutability decreases, the algorithm is able to start slowly replacing the partially immutable pixels with pixels that provide consistency between both the completely immutable pixels and the synthesized pixels.

5.3.3 Synthesizing Wang Tiles with Padding

When edge and tile synthesis are performed with the exact size of the final edge or tile, artifacts are often introduced. These artifacts are due to not having enough neighborhoods and not being able to apply various sizes of neighborhoods while the image is being synthesized. To enable neighborhoods of multiple sizes to be applied during synthesis and to increase the number of neighborhoods considered during synthesis, a padded border is introduced on the image being synthesized.

During synthesis, padding is added around all edges of the image being synthesized. For synthesized edges, the padding ensures the edge has a higher chance of matching exemplar neighborhoods when placed within a larger texture. And for synthesized tiles, the padding increases the chance of finding good interior matches that are also consistent with the exemplar when transitioning across a tile's boundary. Before the image is available for use in either tile synthesis (for edges) or for use as a SWTile (for tiles) the image is clipped to remove the padded region from the final synthesized image.

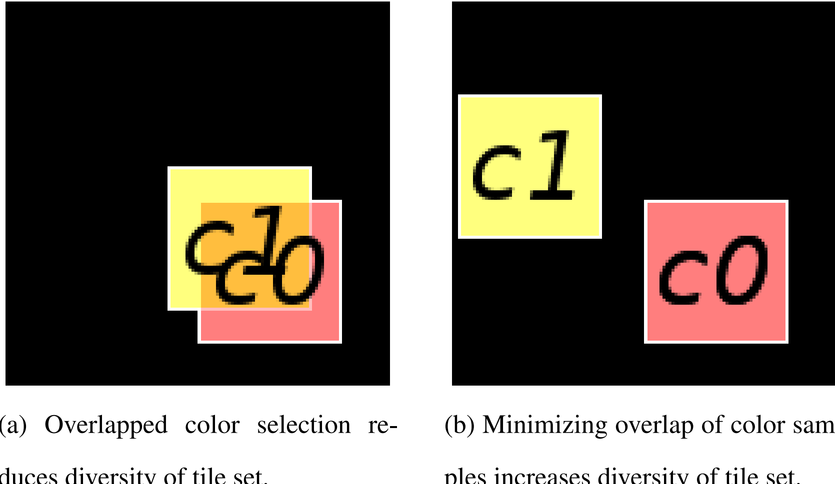
5.3.4 Introducing Diversity: Random Middles

Up to this point, diversity was controlled strictly by the corner colors used and Kwatra's synthesis algorithm. Some SWTiles contain a high degree of diversity without adding any other controls. However, for other textures large portions of one SWTile may completely match (or mostly match with slight offsets) another SWTile or even multiple SWTiles. For these textures, adding samples randomly selected from the Z image to be placed in the middle of the synthesized edges and

the middle of the synthesized tiles helps increase the diversity of the SWTile set. The randomly selected middles are treated the same as the other immutable regions (i.e they use the same rules for determining the partially immutable padding and for determining the percentage of immutability).

5.3.5 Revisiting Color Selection

Using a random process for selecting corner colors for use in creating SWTiles can sometimes result in selecting patches from the exemplar that are largely overlapping. This has the effect of reducing the variety of the SWTile set. To alleviate this problem, colors can be selected in a fashion that attempts to minimize the overlap of the color samples. Figure 5.10 shows a scenario where the randomly selected colors had a large region of overlap compared to the result of applying a color selection technique that minimizes overlap. In essence, one aspect of selecting colors is a sampling problem and well known sampling methods such as n-rooks, jittered, or Poisson-disk could be applied for selecting colors[26].



(a) Overlapped color selection reduces diversity of tile set.

(b) Minimizing overlap of color samples increases diversity of tile set.

Figure 5.10: Sampling colors for diversity.

For regular textures, a random selection or randomized minimal overlap selection approach may (and often does) result in colors that force the generated texture to have obvious defects such

as misaligned lines for a brick texture. To ensure that the selection of colors does not introduce defects into the synthesized tiles, all color samples must preserve the alignment observed in the exemplar when placed in any of the four corners and in combination with the other colors.

5.3.6 Regular Textures

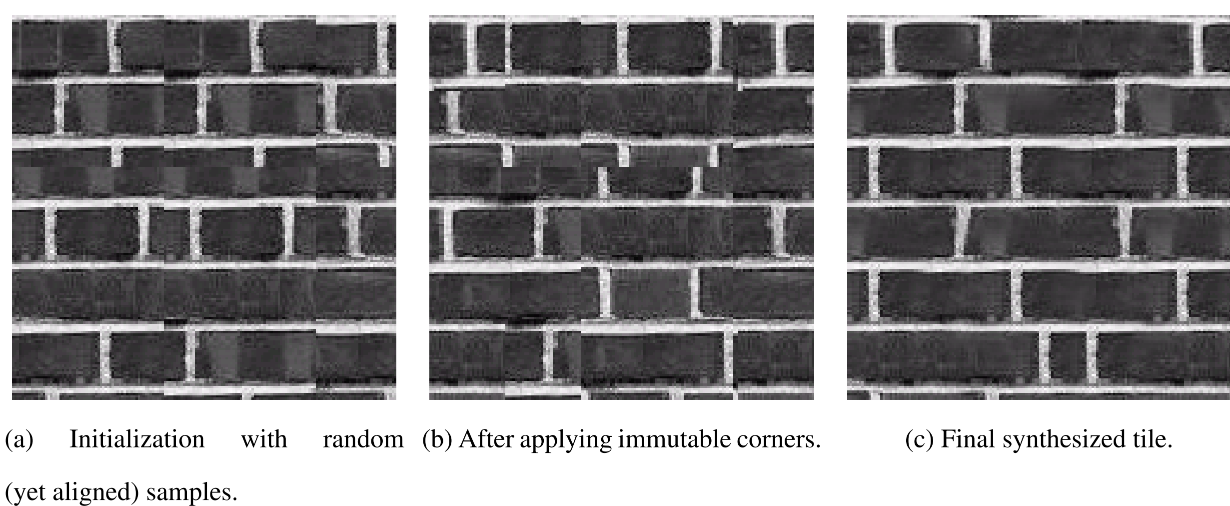
As discussed in Section 5.3.5, regular textures are problematic due to the strict alignment requirements. Requiring that the corner samples meet alignment requirements improves the SWTile quality, however in some cases even after aligning the corner samples and the random middles the synthesized tiles still have misalignment. Two techniques were used to further improve the quality of the synthesized tiles for regular textures. One involved ensuring the spacing between colors honored the exemplar alignment and the other involved providing an initialization of the X image with aligned samples.

Although color selection was previously modified to honor the alignment of the exemplar in terms of allowing any color to be placed in any corner and still have vertical or horizontal alignment conditions hold, nothing was done to ensure that the spacing between the corners also honored the alignment criteria. For regular textures the size of the output edge or tile must be modified so that the spacing between the corners maintains the alignment those corners have in the exemplar. An algorithm for doing this (shown in Figure 5.11) is to modify the dimension of the tile such that the corners are placed at an interval of the alignment size.

```
if alignment is 0
        outSize ← size
else
        gridLines ← floor(size ÷ alignment)
        outSize ← gridLines × alignment
```

Figure 5.11: Adjusting tile size to ensure corner spacing is aligned for a regular texture (division and multiplication are truncated to integers).

In order to increase the quality of the synthesized tiles, instead of randomizing the X image, the X image is initialized with random aligned samples. Figure 5.12 shows the high quality of the initial X image using this technique, along with the X image after applying the immutables for the tile, and the final synthesized tile.



(a) Initialization with random (yet aligned) samples. (b) After applying immutable corners. (c) Final synthesized tile.

Figure 5.12: Initialization of X image with aligned samples from the Z image.

if align is 0 or max is 0
    newLoc $\leftarrow$ *srcLoc*
else
    chunks $\leftarrow$ *(srcLoc+align÷2)÷align*
    newLoc $\leftarrow$ *chunks*$\times$*align*
    while (newLoc + dim >= max)
        newLoc $\leftarrow$ *newLoc*$-$ *align*

Figure 5.13: Algorithn for sampling aligned colors.

for each sample to place in X image
  compute alignment offset for the samples location in the X image
  obtain a random location for sampling the exemplar
  align the random location to the closest aligned location
  offset the aligned random location by the offset previously computed

Figure 5.14: Algorithm for adjusting random samples to place in an initial X image for alignment.

5.4 Results

The general format for displaying results is to display the exemplar with the generated SWTile set and then display example tilings along with the result of a synthesized texture from Kwatra. The exemplar and Kwatra's results are taken from his website[16]. The SWTiles presented here were generated using two colors yielding 16 tiles in the set.

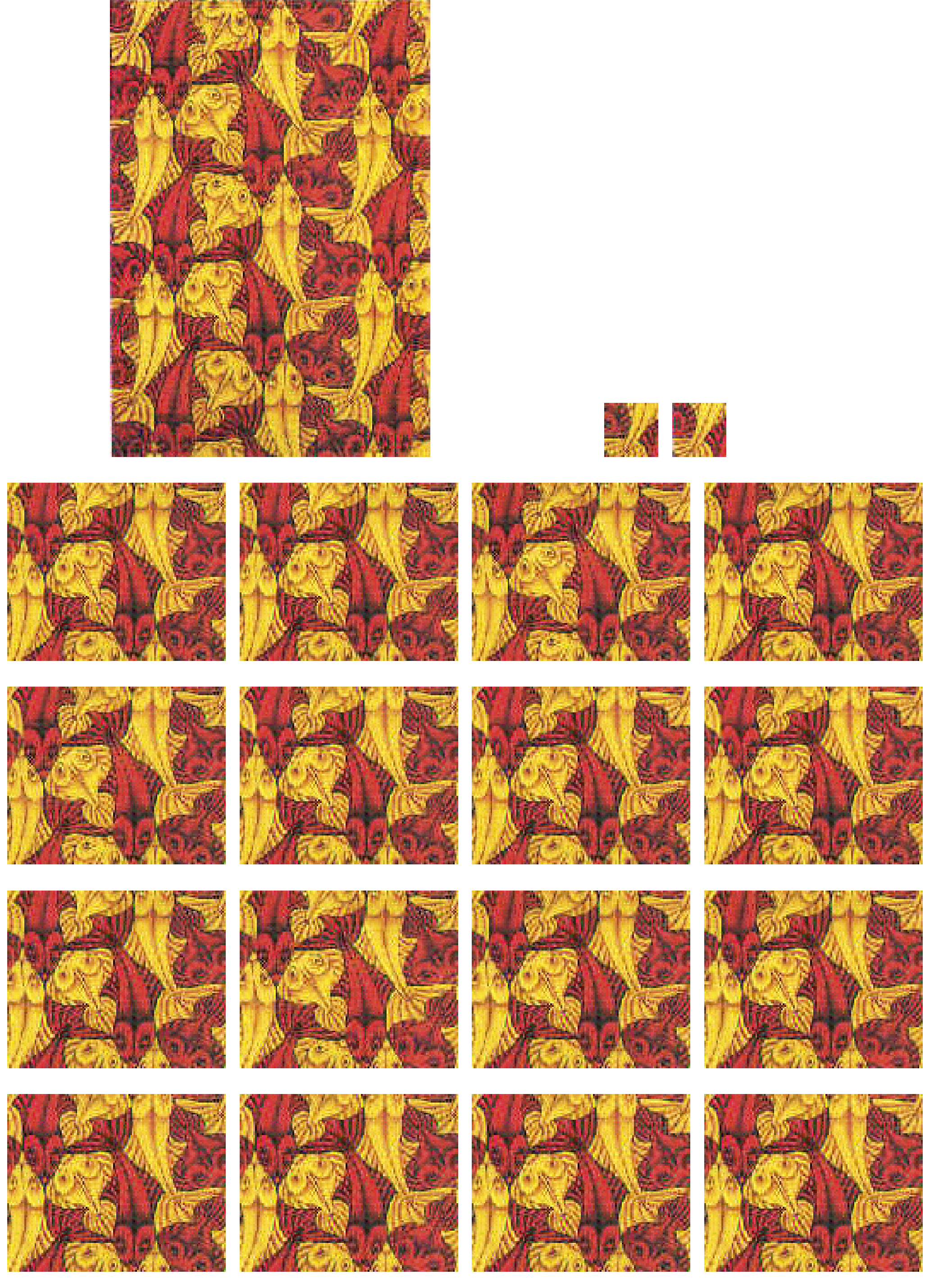


Figure 5.15: Top row: Exemplar of size 188×269 (left) and 2 selected colors of size 32×32. Remaining rows are SWTiles of size 128×104 for "escher2".

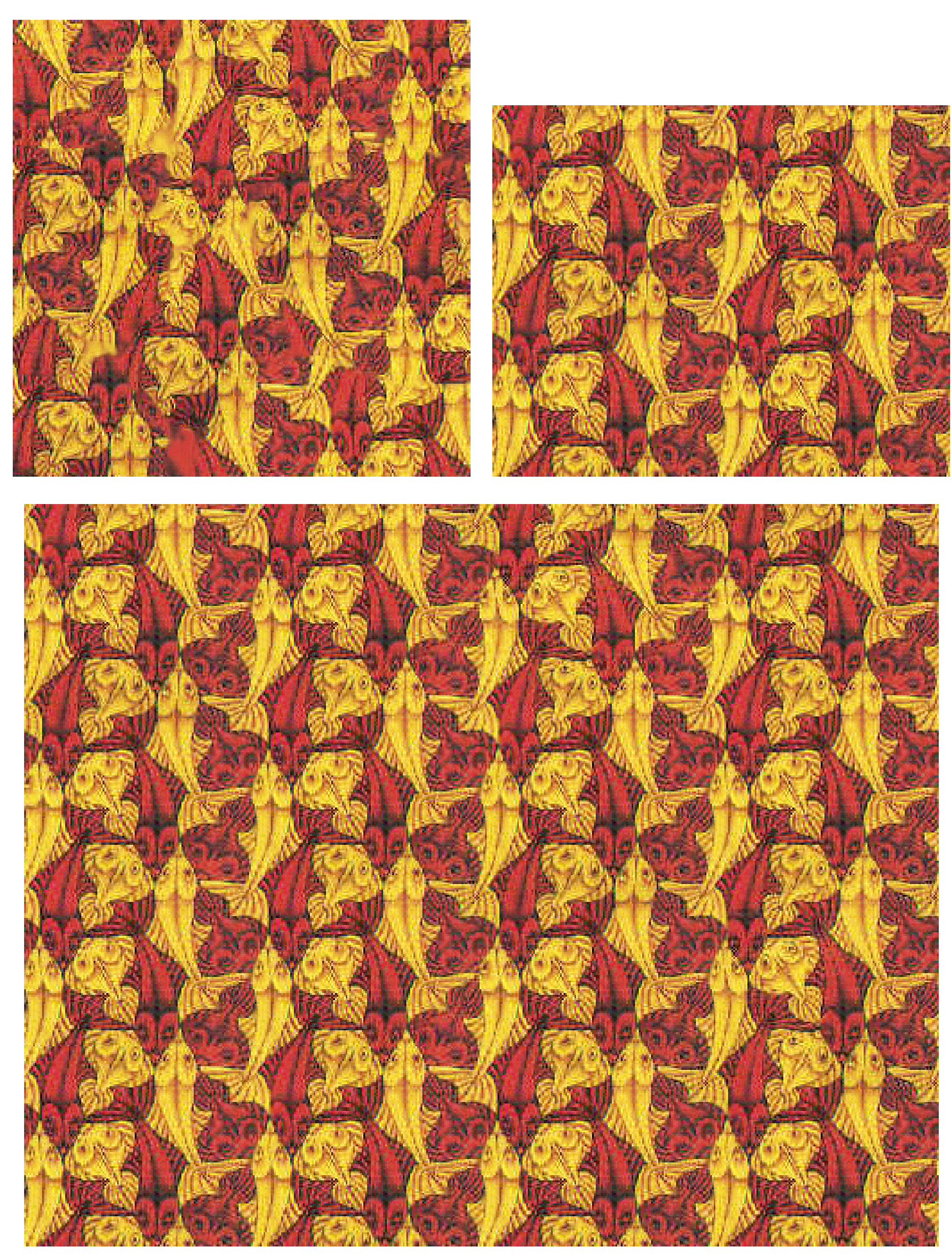


Figure 5.16: Kwatra's synthesized output (top left), a 2×2 grid of SWTiles (top right), and a 4×4 tiling of SWTiles for "escher2".

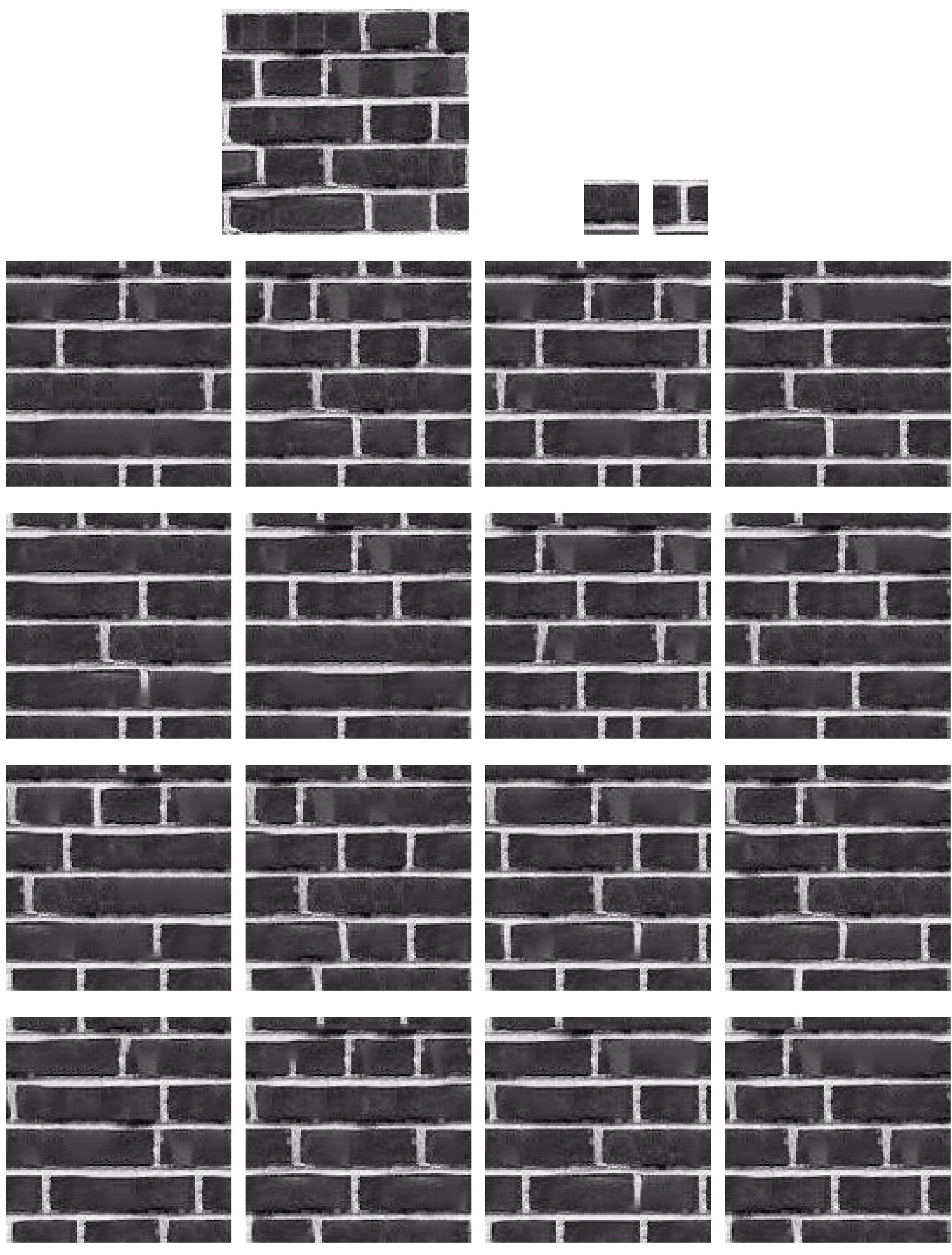


Figure 5.17: Top row: Exemplar of size 142×130 (left) and 2 selected colors of size 32×32. Remaining rows are SWTiles of size 130×130 for "bricks2".

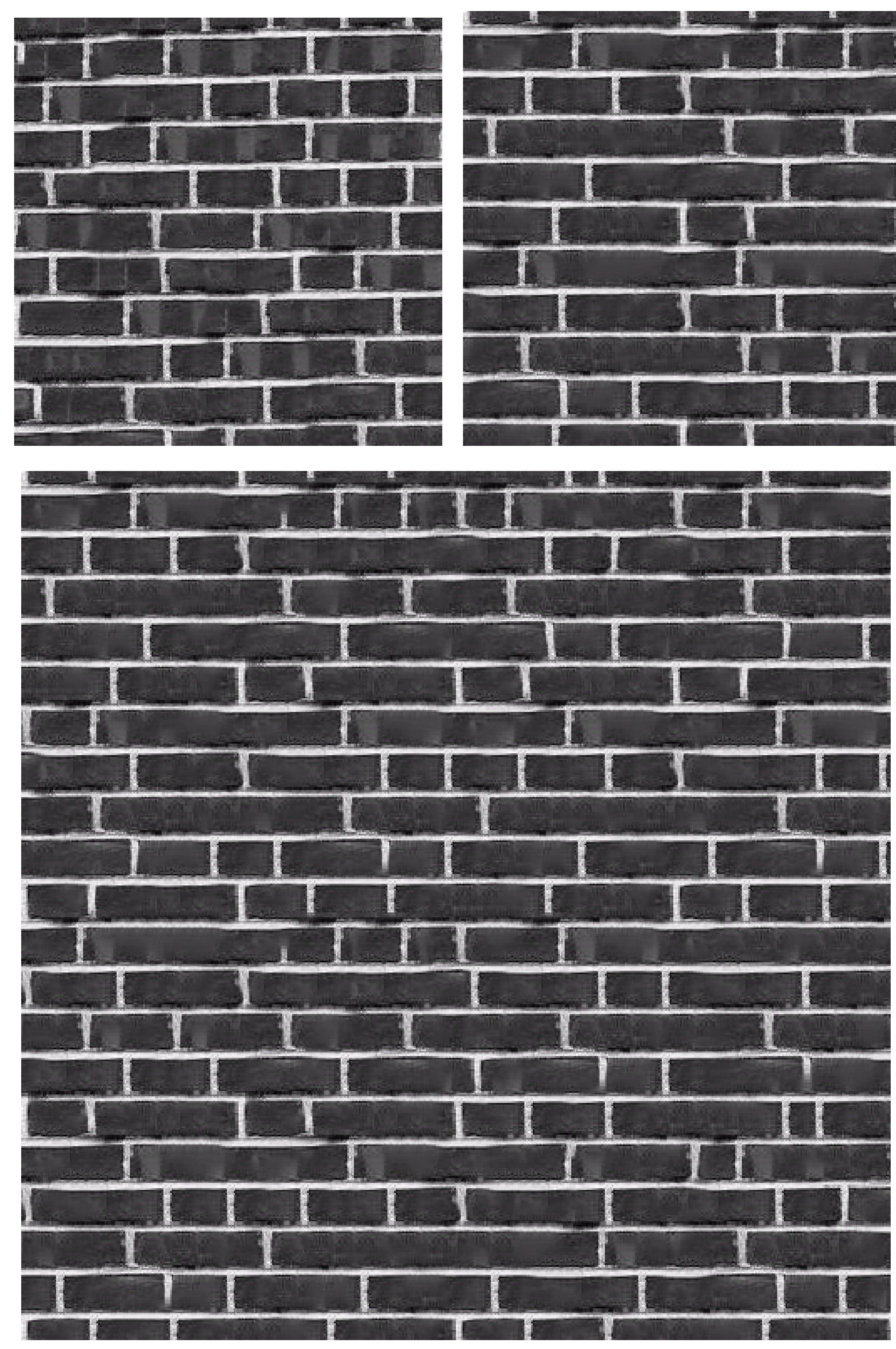


Figure 5.18: Kwatra's synthesized output (top left), a 2×2 grid of SWTiles (top right), and a 4×4 tiling of SWTiles for "bricks2".

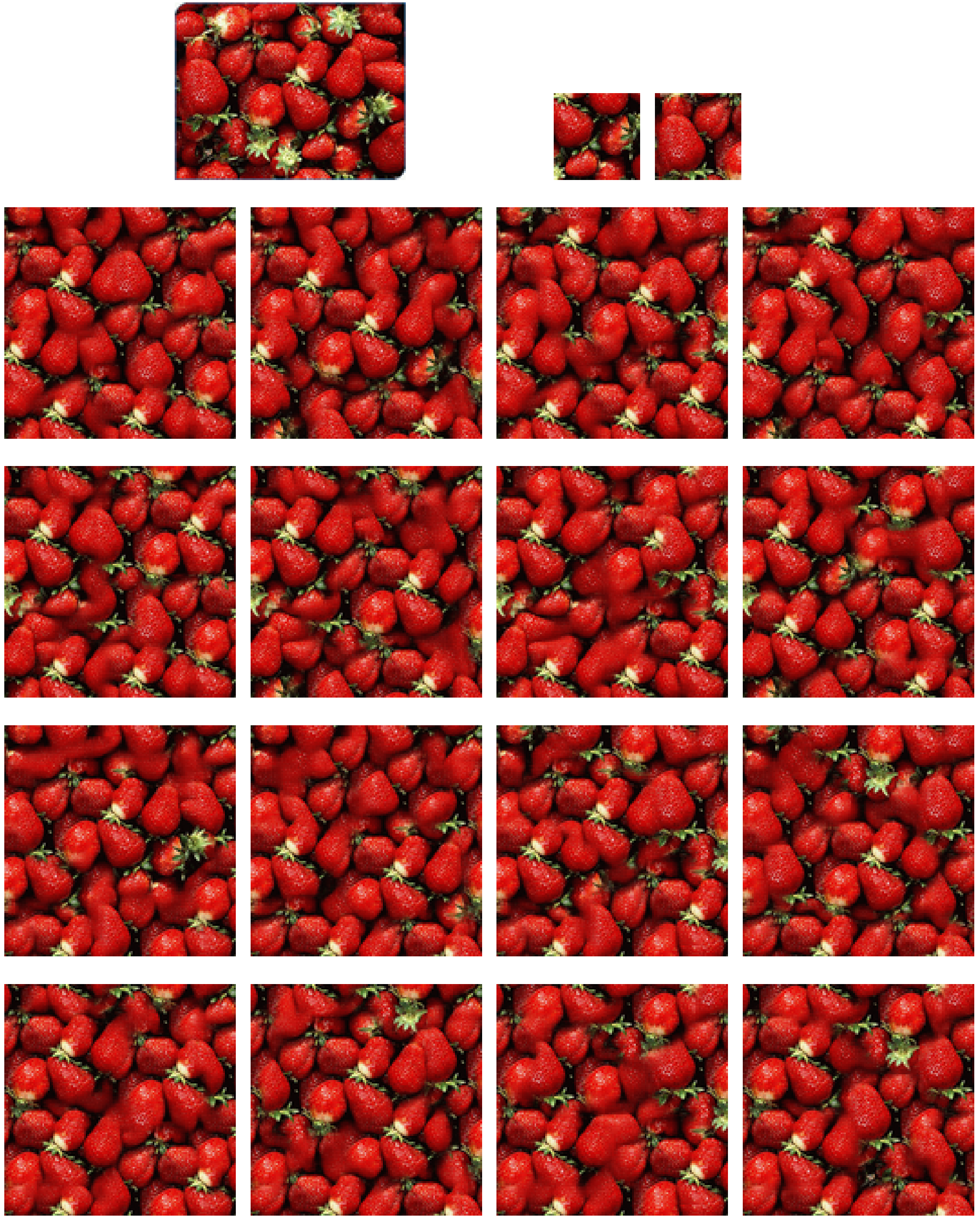


Figure 5.19: Top row: Exemplar of size 128×98 (left) and 2 selected colors of size 48×48. Remaining rows are SWTiles of size 128×128 for "strawberries2".

Figure 5.20: Kwatra's synthesized output (top left), a 2×2 grid of SWTiles (top right), and a 4×4 tiling of SWTiles for "strawberries2".

Figure 5.21: Top row: Exemplar of size 128×128 (left) and 2 selected colors of size 48×48. Remaining rows are SWTiles of size 128×128 for "olives".

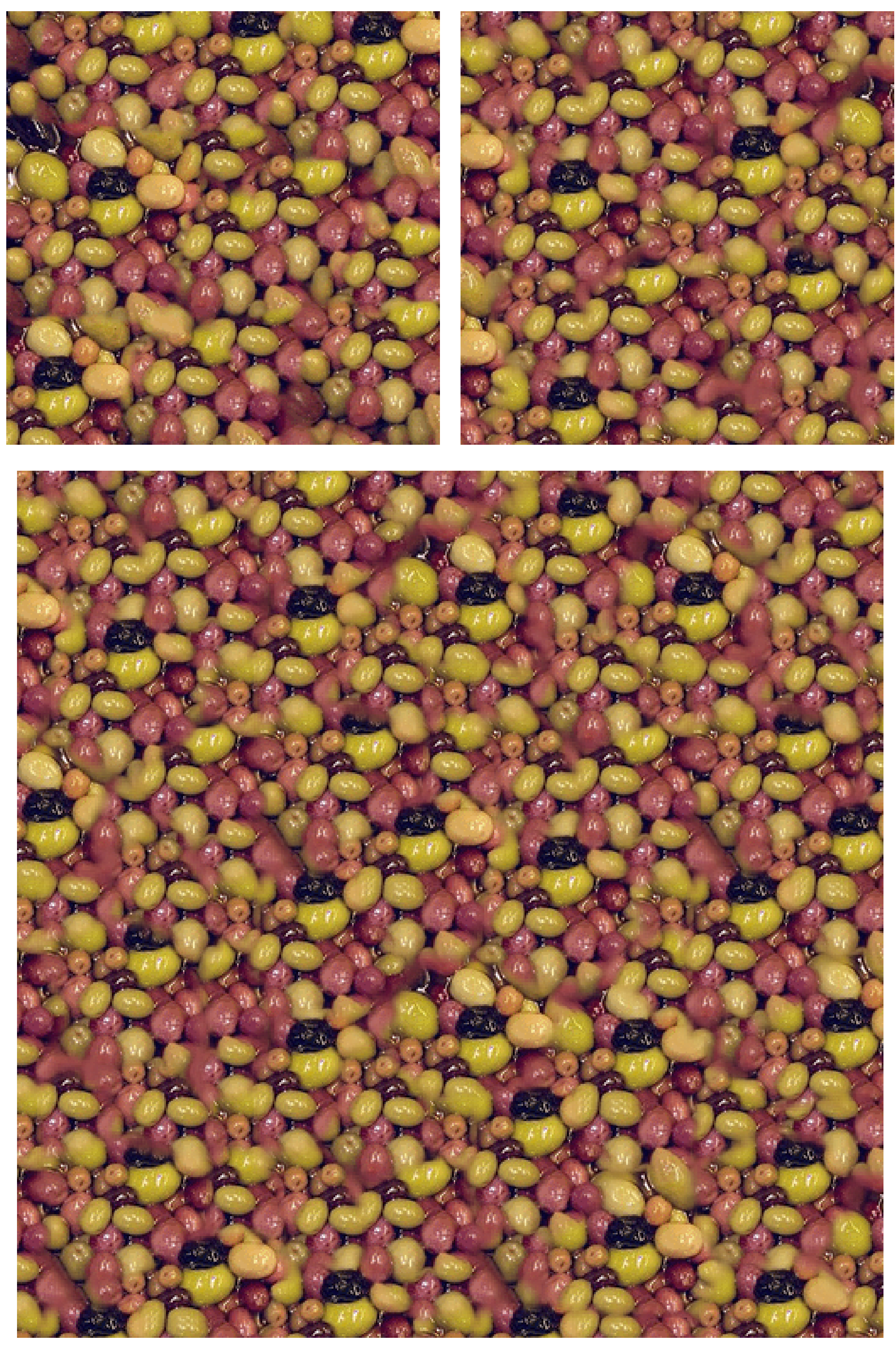


Figure 5.22: Kwatra's synthesized output (top left), a 2×2 grid of SWTiles (top right), and a 4×4 tiling of SWTiles for "olives".

Figure 5.23: Top row: Exemplar of size 128×128 (left) and 2 selected colors of size 48×48. Remaining rows are SWTiles of size 128×128 for "stone".

Figure 5.24: Kwatra's synthesized output (top left), a 2×2 grid of SWTiles (top right), and a 4×4 tiling of SWTiles for "stone".

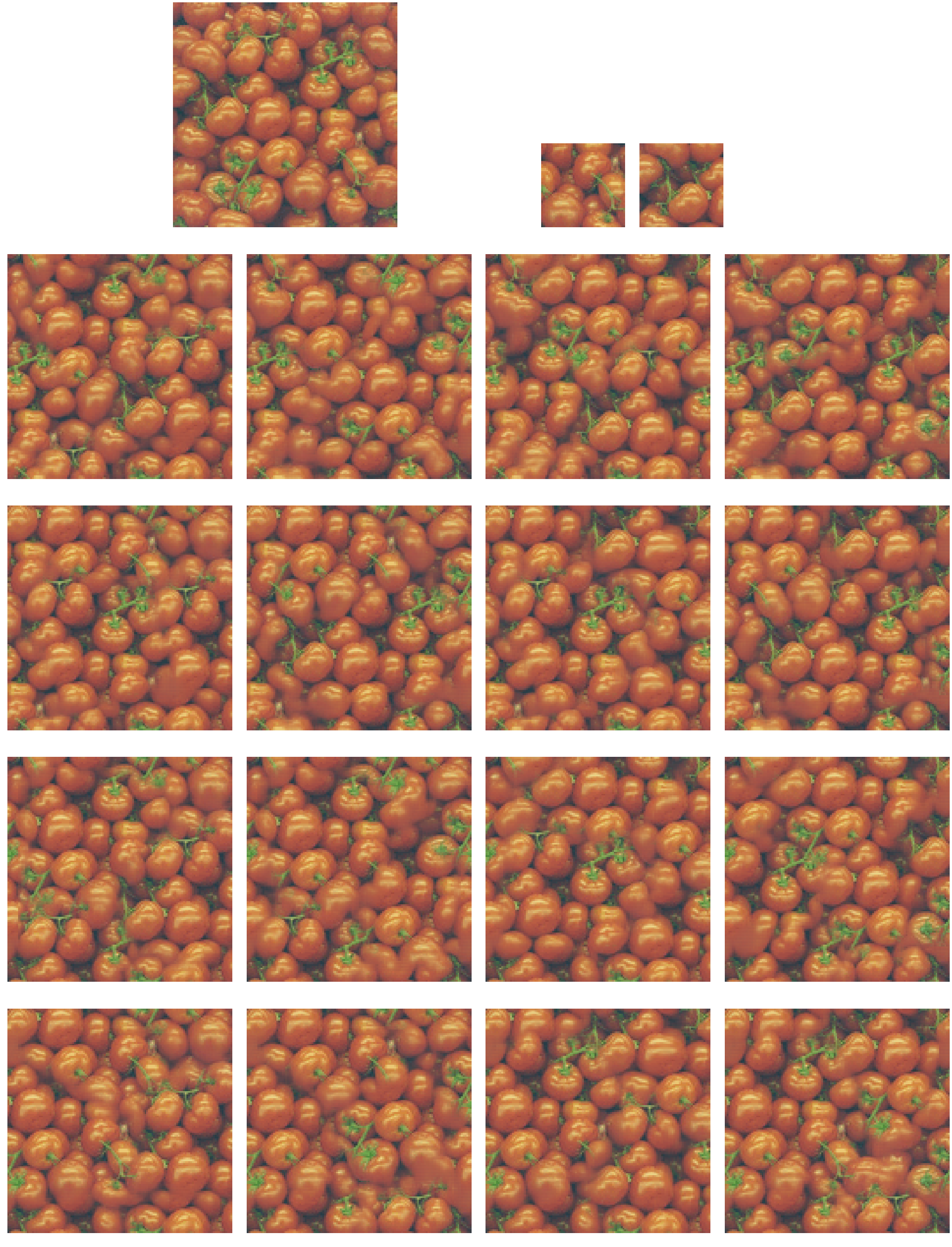


Figure 5.25: Top row: Exemplar of size 128×128 (left) and 2 selected colors of size 48×48. Remaining rows are SWTiles of size 128×128 for "tomatoes".

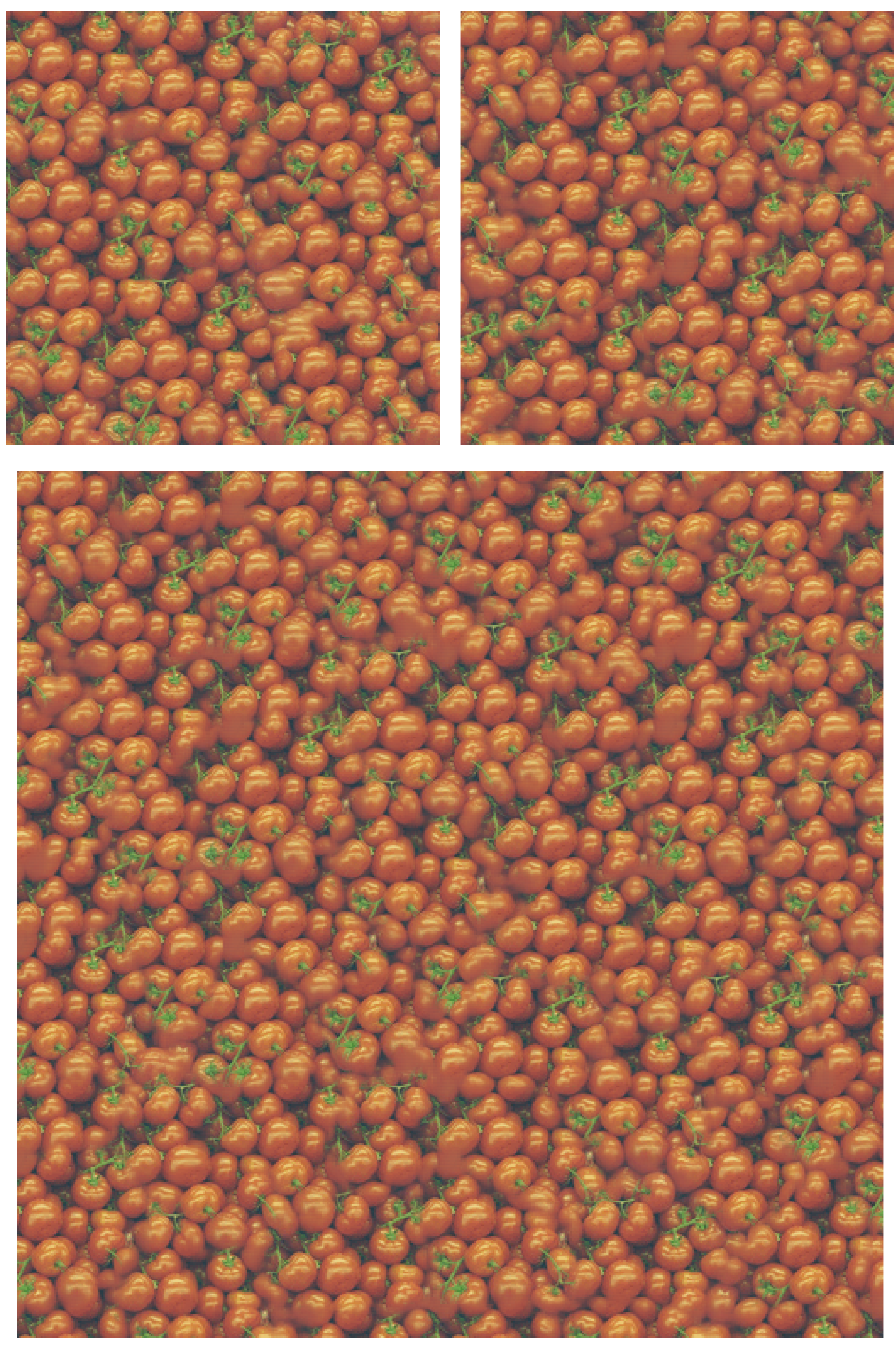


Figure 5.26: Kwatra's synthesized output (top left), a 2×2 grid of SWTiles (top right), and a 4×4 tiling of SWTiles for "tomatoes".

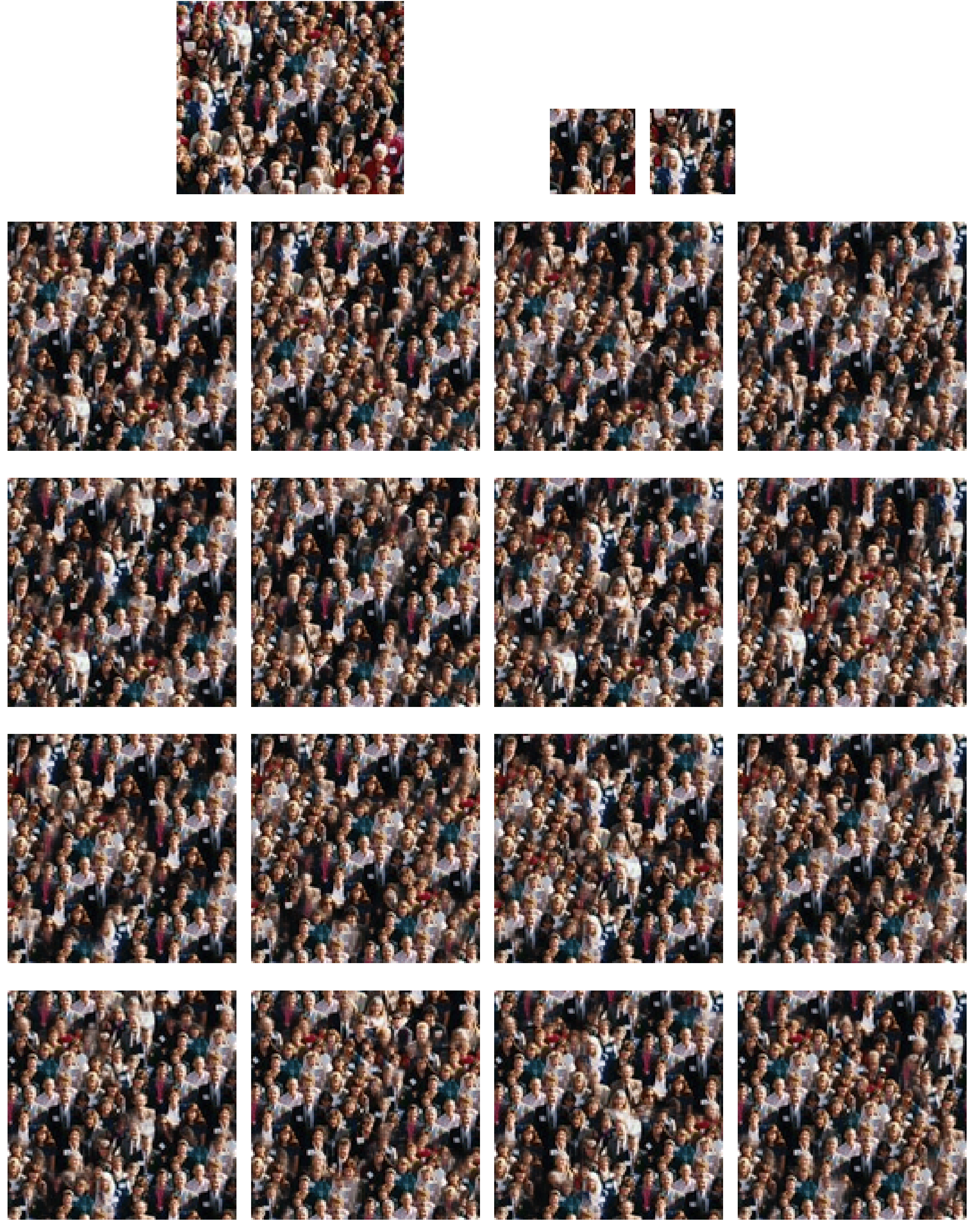


Figure 5.27: Top row: Exemplar of size 127×108 (left) and 2 selected colors of size 48×48. Remaining rows are SWTiles of size 128×128 for “people”.

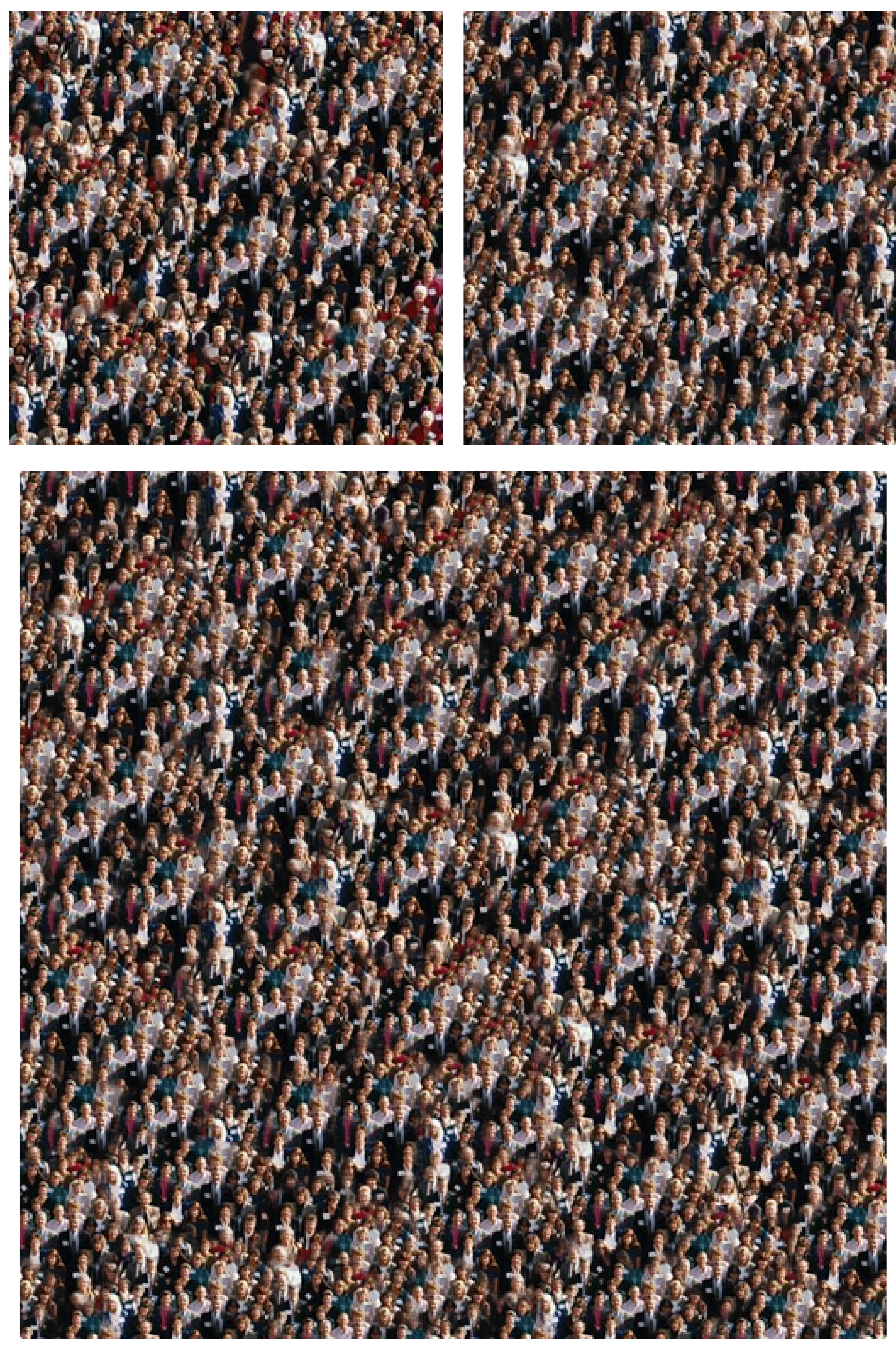


Figure 5.28: Kwatra's synthesized output (top left), a 2×2 grid of SWTiles (top right), and a 4×4 tiling of SWTiles for "people".

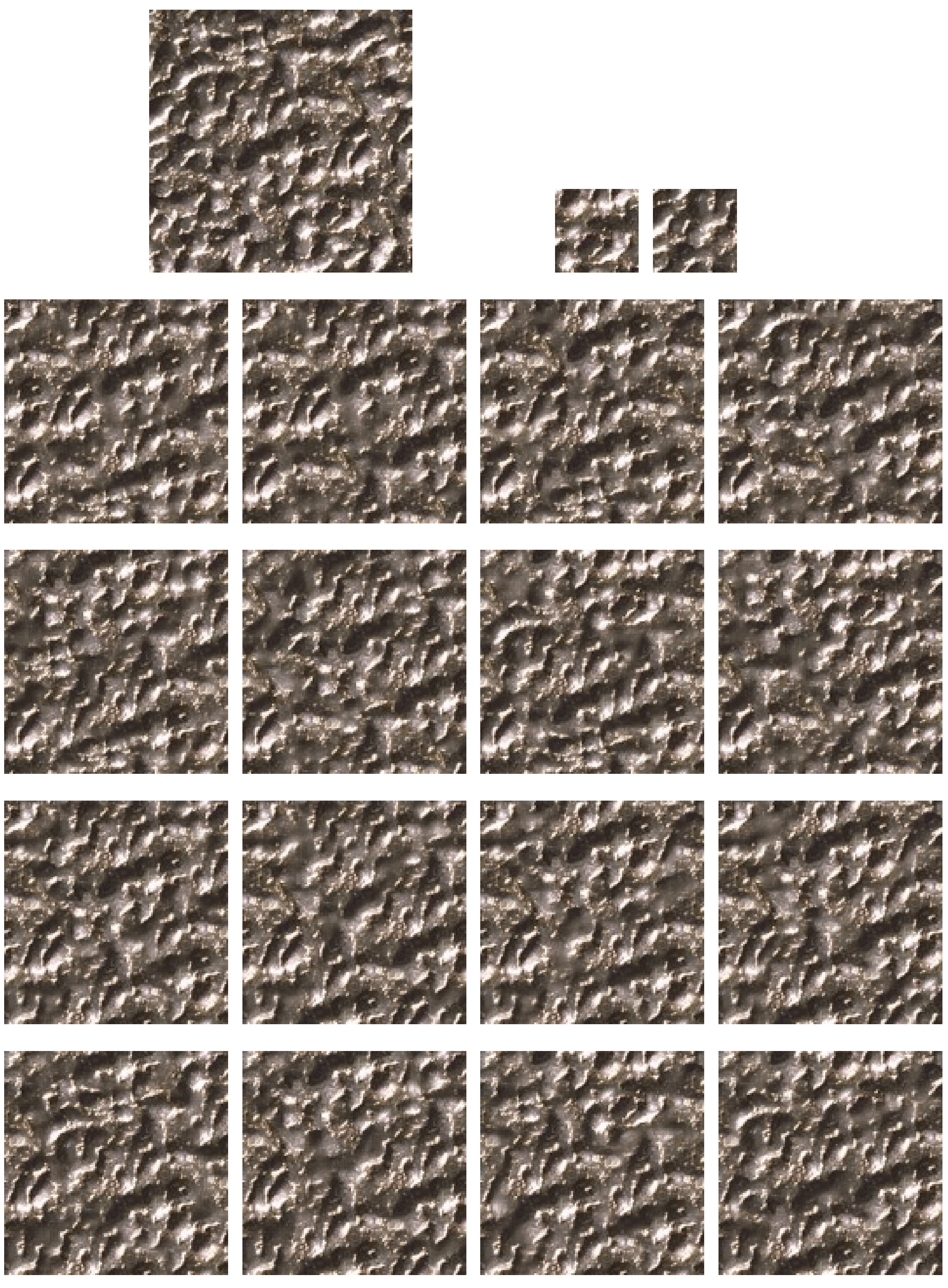


Figure 5.29: Top row: Exemplar of size 150×150 (left) and 2 selected colors of size 48×48. Remaining rows are SWTiles of size 128×128 for "choc_scale".

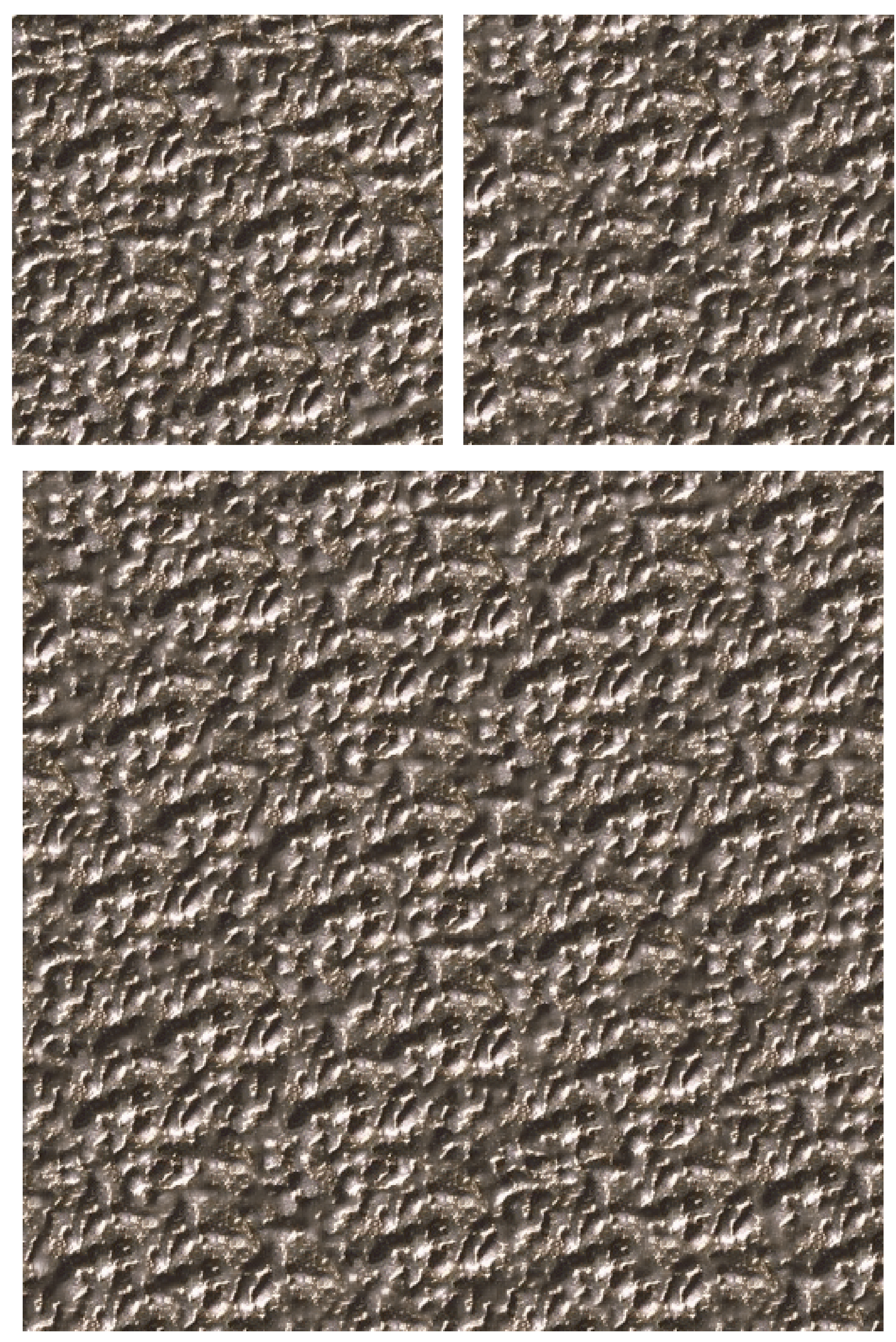


Figure 5.30: Kwatra's synthesized output (top left), a 2×2 grid of SWTiles (top right), and a 4×4 tiling of SWTiles for "choc_scale".

5.5 Analysis

The SWTile method successfully creates a set of Wang tiles using 2 colors, however 3 or 4 colors would increase diversity. The diversity for the SWTile sets created is rather large considering that only two colors are used for most of the tile sets.

For regular textures, the selected color samples often look similar (as seen in Figure 5.15). This is a result of the alignment requirements, yet when comparing the pixel values they are actually different and are taken from different regions of the exemplar. Although it may seem that a single self tiling tile would work just fine for a strict regular texture like the "escher" textures, the SWTiles introduce diversity such as varying depth of tones over similar looking regions.

For the near regular textures, such as the "bricks2" in Figure 5.17, it may be desirable to manually edit the regions of the tile after automatic creation to close some of the vertical lines that do not quite go from one row of bricks to the other or to remove the undesirable "long" bricks seen in Figure 5.18.

Some of the irregular textures such as the "olives" in Figure 5.21 have more blurring than is desirable. Using different parameters such as using Kwatra's "robust" optimization (which uses Gaussian weights centered over a neighborhood) can alleviate some of the blurring as can adjusting the color size.

The near-stochastic "choc_scale" texture (Figure 5.29) is a good example of only needing two colors and still generating quality tilings.

5.6 Conclusion

The SWTile method has illustrated the feasibility of modifying synthesis techniques to generate corner colored Wang tiles as opposed to requiring dynamic synthesis.

For regular textures, selecting colors such that they honor the alignment of the exemplar and

spacing the corners to also honor the alignment of the exemplar greatly increase the quality of the generated SWTiles. Adding a region of partially immutable pixels that gradually become completely mutable increases the likelihood that the synthesis technique finds good candidates for the mutable regions being synthesized.

An area of interest for future work would be to modify other synthesis methods (in particular those that avoid blurring[11]) to generate SWTiles where instead of averaging the partially immutable pixels with the synthesized pixels, a dithering approach could be used. In addition, the SWTile method could be enhanced to use a Gaussian falloff as opposed to linear falloff for transitioning from partially immutable to immutable.

Chapter 6

Key Aspects for Corner Colored Wang Tiles

During the research and analysis phases of this thesis, some key aspects were gathered and observed while performing tests. For example, many publications mention the importance of feature size in regards to various methods for generating texture. Also, while developing the new tile creation techniques, the importance of alignment for regular and often near-regular textures became painfully obvious. A summary of these key aspects are provided here.

6.1 Feature Size

The relationship between feature size and quality Wang tile creation applies not only to blending as mentioned by Burt and Adelson, but for neighborhood matching and graph cut techniques as well. For blending, if the overlap region is too small, sharp edges in features may appear, while if the blending region is too large, features may be ghosted (duplicated but in lighter tones). For synthesis, if the neighborhood sizes are not large enough to capture the largest features, the largest features often are unable to be recreated in the synthesized texture. Whereas, if the smallest neighborhood size used is much greater than the smallest feature of the texture, than those small features are often unable to be synthesized. For graph cut techniques, if the overlapped region being cut

is not larger than the largest feature, then the cutting path will be forced to cut through a feature rather than cutting around it.

As Burt and Adelson point out, a good blending overlap size is one that contains the largest feature but not more than twice the size of the smallest feature. For cutting regions, the overlaps should be at least as large as the largest feature to enable a path to be found that cuts around the features if necessary. And for neighborhoods, the sizes should be large enough to capture one to two times the size of the large features and progress down smaller in order to capture the smallest features.

6.2 Alignment

Processing regular and near-regular textures with unaligned corners will typically result in artifacts of either sharp discontinuities in the regular pattern or a deformation of the regular or near-regular pattern. Ensuring that the colored corners selected honor the pattern of the exemplar when placed in any corner, and in combination with all of the other colors, increases the chance that a representative Wang tile is created.

Aligning the corners alone may not be enough to honor the alignment patterns of the exemplar due to the spacing of the corners. When placing corners, they must also be spaced such that the spacing preserves the alignment found in the exemplar. These alignment requirements apply not only to the selected corner colors, but also to any random exemplar sample patches that are to be applied to the Wang tile.

6.3 Diversity

Capturing the diversity of the exemplar is a trait desired in Wang tiles. However even more important is for the set of Wang tiles to have enough diversity to prevent a human from easily recognizing

that some of the same tiles are repeatedly used. If the tiles in the set contain many of the same features (limited diversity) a repeated pattern will be easily detected. Some researchers have tried to increase diversity by using random centers for a tile, however the cutting constraints for ensuring tilability restrict the increase of diversity generated by the random centers[34]. The SWTile method introduces diversity with random centers for tiles and edges, but also by using a synthesis method that can generate pixels not found in the exemplar while maintaining the characteristics of the exemplar. The BLWTile and GLWTile methods introduce diversity by using large numbers of colors. The more colors available, the more diverse the tiled texture should appear. When selecting colors, attention should be given to how the colors overlap, as overlapping colors provide less diversity than non-overlapping colors.

6.4 Abnormal Features

If an abnormal or uncommon feature is selected as a corner color, the uncommon feature will become more common in the textures generated by the Wang tiles. In addition, the feature will be repeated on the grid of the tiling. Combining the gridding of the tiling with a relative increase in occurrence will often result in a noticeable gridding artifacts with the uncommon feature. To minimize this, the abnormal or uncommon features should be avoided for use as corner colors, yet still available for use in the generated tiles either by random centers, synthesis, or some other method.

6.5 Exemplar Quality

The quality of an exemplar directly influences the quality of the created Wang tiles. Any undesirable features or artifacts present in the exemplar may manifest in undesirable characteristics in the created Wang tiles based on which method is used for creating the tiles. Some examples of things

to avoid are lossy compressed textures (*e.g.* jpeg) and out of focus or blurred textures. In addition many current techniques have difficulty accurately re-creating a texture with uneven lighting. Uneven lighting may be viewed as an example of a globally varying texture, but it would likely be better to extract the lighting from the texture so that the texture can be used with various lighting models.

Chapter 7

Conclusion

Generating quality textures has become a common requirement for use in video games, films, and other computer graphic applications. Existing methods include manual creation, corner and edge colored Wang tiles, synthesis, and procedural methods. Current techniques for creating corner and edge Wang tiles use graph cuts.

New methods for generating texture based on using corner colored Wang tiles have shown to be successful for a variety of textures. The SWTile method is intended to be used offline to precompute a set of corner colored Wang tiles. The BLWTile and GLWTile methods are intended to be evaluated dynamically as needed.

When using techniques to create a set of corner colored Wang tiles, or when designing new techniques, certain characteristics should be observed when selecting an exemplar and when selecting corners. In particular, the size required by the method to create corner colored Wang tiles should be matched to the size of the features in the exemplar.

7.1 Future Work

Potential future work includes modifying the approach taken by Han, et al. to synthesize corner colored Wang tiles in the same approach that Kwatra's synthesis algorithm was modified[11]. The SWTile method can also be modified to incorporate the acceleration techniques for the synthesis steps.

Developing an editor to enable corner selection, synthesized edge modifications, and tile modifications could provide for better corner colored Wang tiles than using either a completely automated or completely manual approach.

In a similar vein, the BLWTile and GLWTile methods would also benefit from a preview tool that would enable viewing the effect of changing the color sizes as well as manually selecting colors. It may also be useful to allow the BLWTile and GLWTile methods to precompute a set of Wang tiles as opposed to computing the tiles dynamically.

Bibliography

[1] M. Browne and D.G. Lowe. Recognising panoramas. In *Computer Vision*, 2003.

[2] Peter J. Burt and Edward H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, COM-31,4:532–540, 1983.

[3] Peter J. Burt and Edward H. Adelson. A multiresolution spline with application to image mosaics. *ACM Transcations on Graphics*, 2(4), October 1983.

[4] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang Tiles for image and texture generation. *ACM Transactions on Graphics*, 22(3):287–294, July 2003.

[5] Karel Culík. An aperiodic set of 13 wang tiles. *Discrete Mathematics*, 1996.

[6] Leonardo da Vinci. Mona lisa. http://upload.wikimedia.org/wikipedia/commons/6/6a/Mona_Lisa.jpg.

[7] A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. In *SIGGraph-01*, 2001.

[8] A. A. Efros and T. K. Leung. Texture synthesis by non-parametric sampling. In *ICCV*, pages 1033–1038, 1999.

[9] Alexei Efros. Image compositing and blending. http://graphics.cs.cmu.edu/courses/15-463/2007_fall/Lectures/blending.pdf, 2007.

[10] David Green. Terrain advanced textures, 2009. http://udn.epicgames.com/Three/ TerrainAdvancedTextures.html.

[11] Jianwei Han, Kun Zhou, Li-Yi Wei, Minmin Gong, Hujun Bao, Xinming Zhang, and Baining Guo. Fast example-based surface texture synthesis via discrete optimization. *The Visual Computer*, 22(9-11):918–925, 2006.

[12] James Hayes. File:texture spectrum.jpg - wikipedia, the free encyclopedia, April 2006.

[13] Hugues Hoppe. Parallel controllable texture synthesis. http://research.microsoft.com/en-us/um/redmond/projects/ParaTexSyn/.

[14] VSAJ INC. Shellstone tiles. http://vegas-tile.com/SHELLSTONE_TILES.html.

[15] Johannes Kopf, Daniel Cohen-Or, Oliver Deussen, and Dani Lischinski. Recursive Wang tiles for real-time blue noise. *ACM Transactions on Graphics*, 25(3):509–518, July 2006.

[16] Vivek Kwatra. Texture optimization for example-based synthesis. http://www.cc.gatech.edu/cpl/projects/textureoptimization/image_results.html.

[17] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *ACM Transactions on Graphics*, 24(3):795–802, July 2005.

[18] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics*, 22(3):277–286, July 2003.

[19] Ares Lagae and Philip Dutré. An alternative for Wang tiles: colored edges versus colored corners. *ACM Transactions on Graphics*, 25(4):1442–1459, October 2006.

[20] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis using patch-based sampling. *ACM Transaction of Graphics*, 2001.

[21] Yanxi Liu, Wen-Chieh Lin, and James Hays. Near-regular texture analysis and manipulation. *ACM Transactions on Graphics*, 23(3):368–376, August 2004.

[22] Massachusetts Institute of Technology. Index of /vimod/imagery/visiontexture/images. http://vismod.media.mit.edu/vismod/imagery/VisionTexture/Images/.

[23] Ken Perlin. An image synthesizer. *Computer Graphics*, 19(3):287–296, July 1985.

[24] Ken Perlin. Improving noise. *ACM Transactions on Graphics*, 21(3):681–682, July 2002.

[25] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In Sheila Hoffmeyer, editor, *Proceedings of the Computer Graphics Conference 2000 (SIGGRAPH-00)*, pages 465–470, New York, July 2000. ACMPress.

[26] Peter Shirely and R. Keith Morley. *Realistic Ray Tracing*. AK Peters Limited, second edition, 2003.

[27] Touchdown Tile. Slate tile - touchdown tile slate tiles selection. http://www.touchdowntile.com/slate-tile.html.

[28] Hao Wang. Proving theorems by pattern recognition II. *Bell Systems Technical Journal*, 1961.

[29] Hao Wang. Games, logic, and computers. *Scientific American*, pages 98–106, November 1965.

[30] Li-Yi Wei. Tile-based texture mapping on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 55–64. Eurographics Association, August 2004.

[31] Li-Yi Wei. *GPU Gems 2: Programming Techniques for High-Perofrmance Graphics and General-Purpose Computation*, chapter Tile-Based Texture Mapping. Addison-Wesley Professional, 2005.

[32] Li-Yi Wei, Jianwei Han, Kun Zhou, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Inverse texture synthesis. *ACM Transactions on Graphics*, 27(3), 2008.

[33] Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the art in example-based texture synthesis. *Eurographics*, 2009.

[34] Tuen young Ng, Conghua Wen, Tiow seng Tan, Xinyu Zhang, and Young J. Kim. Generating an ω-tile set for texture synthesis. In *Proceedings of Computer Graphics International 2005*, pages 177–184, 2005.

[35] Xinyu Zhang and Young J. Kim. Efficient texture synthesis using strict wang tiles. *Graphical Models*, 70(3):43–56, 2008.