2004-11-16

Real-time Image Enhancement Using Texture Synthesis

Matthew J. Sorensen
Brigham Young University - Provo

Real-time Image Enhancement using Texture Synthesis

by

Matthew Sorensen

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
Brigham Young University
November 2004

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Matthew Sorensen

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

_____	_____
Date	Parris K. Egbert, Chair
_____	_____
Date	Bryan S. Morse
_____	_____
Date	Kevin Seppi

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Matthew Sorensen in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

_____ _____

Date

Parris K. Egbert
Chair, Graduate Committee

Accepted for the Department

_____

David Embley
Graduate Coordinator

Accepted for the College

_____

G. Rex Bryce
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

Real-time Image Enhancement using Texture Synthesis

Matthew Sorensen

Department of Computer Science

Master of Science

This thesis presents an approach to real-time image enhancement using texture synthesis. Traditional image enhancement techniques are typically time consuming, lack realistic detail, or do not scale well for large magnification factors.

Real-time Enhancement using Texture Synthesis (RETS) combines interpolation, classification, and patch-based texture synthesis to enhance low-resolution imagery, particularly aerial imagery. RETS uses as input a low-resolution source image and several high-resolution sample textures. The output of RETS is a high-resolution image with the structure of the source image, but with detail consistent with the high-resolution sample textures. We show that RETS can enhance large amounts of imagery in real-time. Our implementation can produce over twenty-five million pixels per second on an average PC.

ACKNOWLEDGMENTS

I would like to thank Dr. Egbert and the committee members for the time spent reviewing and assisting me in writing this thesis. I would also like to thank my wife, Laurie, for her support.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Statement of the Problem

During the last 30 years, researchers have developed proven techniques for enlarging images. The primary challenge was that simply stretching the digital image created a blocky result, because a digital image contains only a small, finite sampling of the continuous function it represents. Figure 1.1 shows blocky artifacts caused by repeated magnification. Researchers were able to reduce the blocky appearance (often referred to as aliasing) by using a smooth interpolation on the image. Interpolation produces good results when the magnification factor is relatively small. However, images magnified by a sizeable factor still do not appear realistic, because interpolation enlarges the image without adding

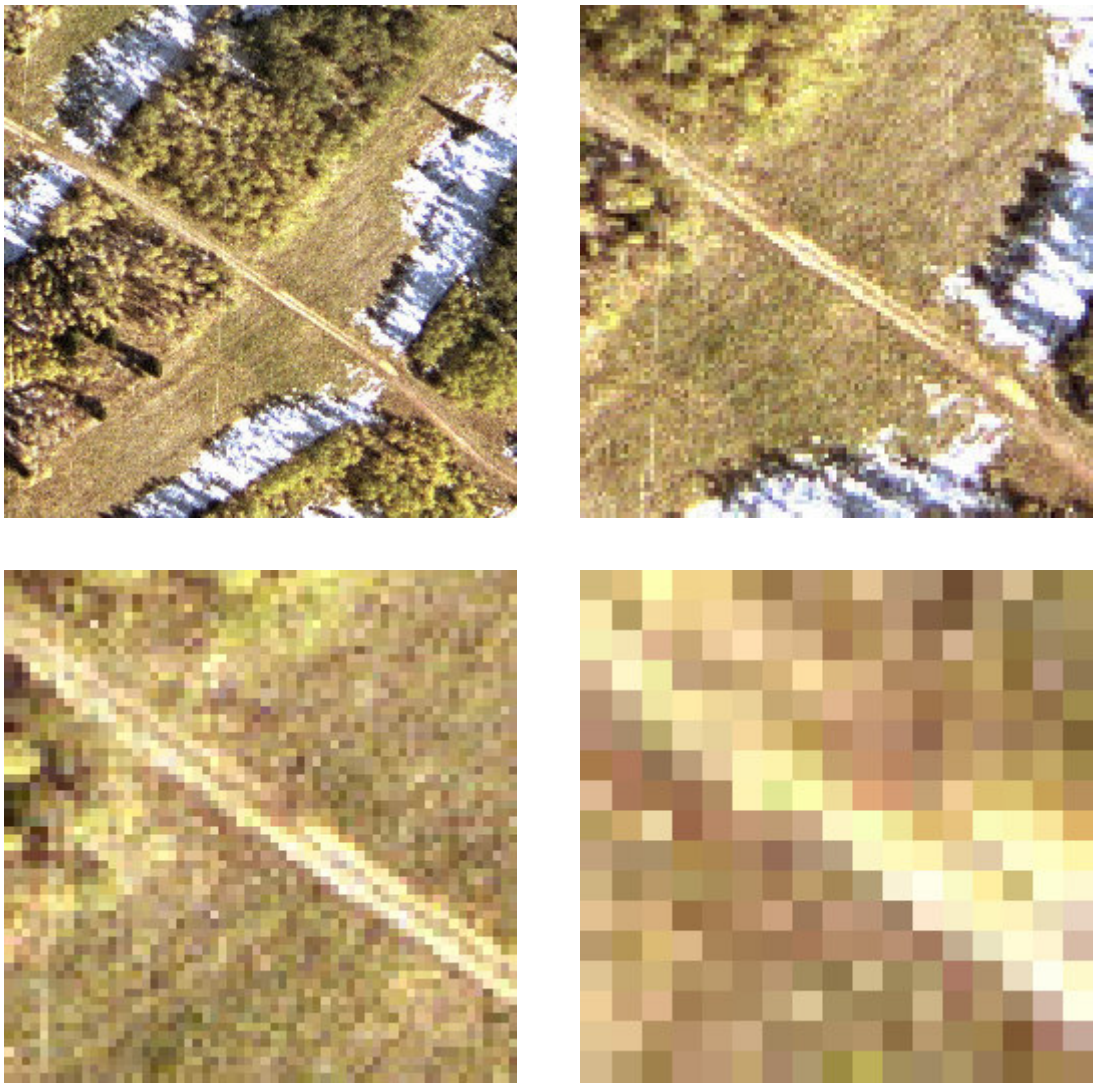


Figure 1.1: Aliasing Artifacts. Here we see the effects caused by large magnifications of aerial imagery. The top left image is a portion of the original aerial photograph. The top right, bottom left, and bottom right images are progressively magnified versions of the original. The magnification factors are 2x, 4x, and 16x. The bottom right image lacks detail and has very visible aliasing artifacts.

(a) Image stretched (b) Bicubic Interpolation (c) Detail added

Figure 1.2: Enlargement Comparison. Comparison of three enlargements of a 15x11 pixel image. All three images have been scaled 32 times the original size. (a) Shows the image being stretched; this is called Nearest Neighbor Interpolation. (b) Shows bicubic interpolation smoothing the image. (c) Shows results using Image Enhancement by Texture Synthesis to add detail.

appropriate detail. Figure 1.2 compares the results of bicubic interpolation and an enhanced image.

1.2 Practical Example of Problem

The graphics research lab at Brigham Young University has previously developed an application named EView that uses aerial imagery and elevation data to generate virtual environments. As can be seen in Figure 1.3, the results are visually appealing when seen from a viewpoint far from the surface of the terrain. However, when the viewpoint is near the surface of the terrain (see Figure 1.4), EView must magnify the low-resolution aerial imagery (e.g., one pixel per square meter) by a large factor.

Merely stretching the images produces badly aliased terrain. Using an interpolation algorithm reduces the blocky appearance but does not add details such as rock, grass, bushes, and dirt that we expect of virtual environments. There is a need for an image enhancement algorithm that synthesizes a higher-resolution image and automatically inserts visually appropriate detail.

1.3 Thesis Statement

For certain classes of images (e.g., aerial photographs), real-time image enhancement using texture synthesis (RETS) can produce visually realistic results in real-time. This process receives as input a low-resolution source image and high-resolution sample textures. From these inputs, the process automatically produces a high-resolution version of the source image with local characteristics matching the high-resolution sample textures.

RETS includes several new advancements over previous techniques. First, we combine patch-based texture synthesis with image interpolation. Second, our algorithm enlarges images inserting detail locally similar to supplied sample images in *real-time*. Third, RETS allows multiple texture classes to appear in a single synthesized image.

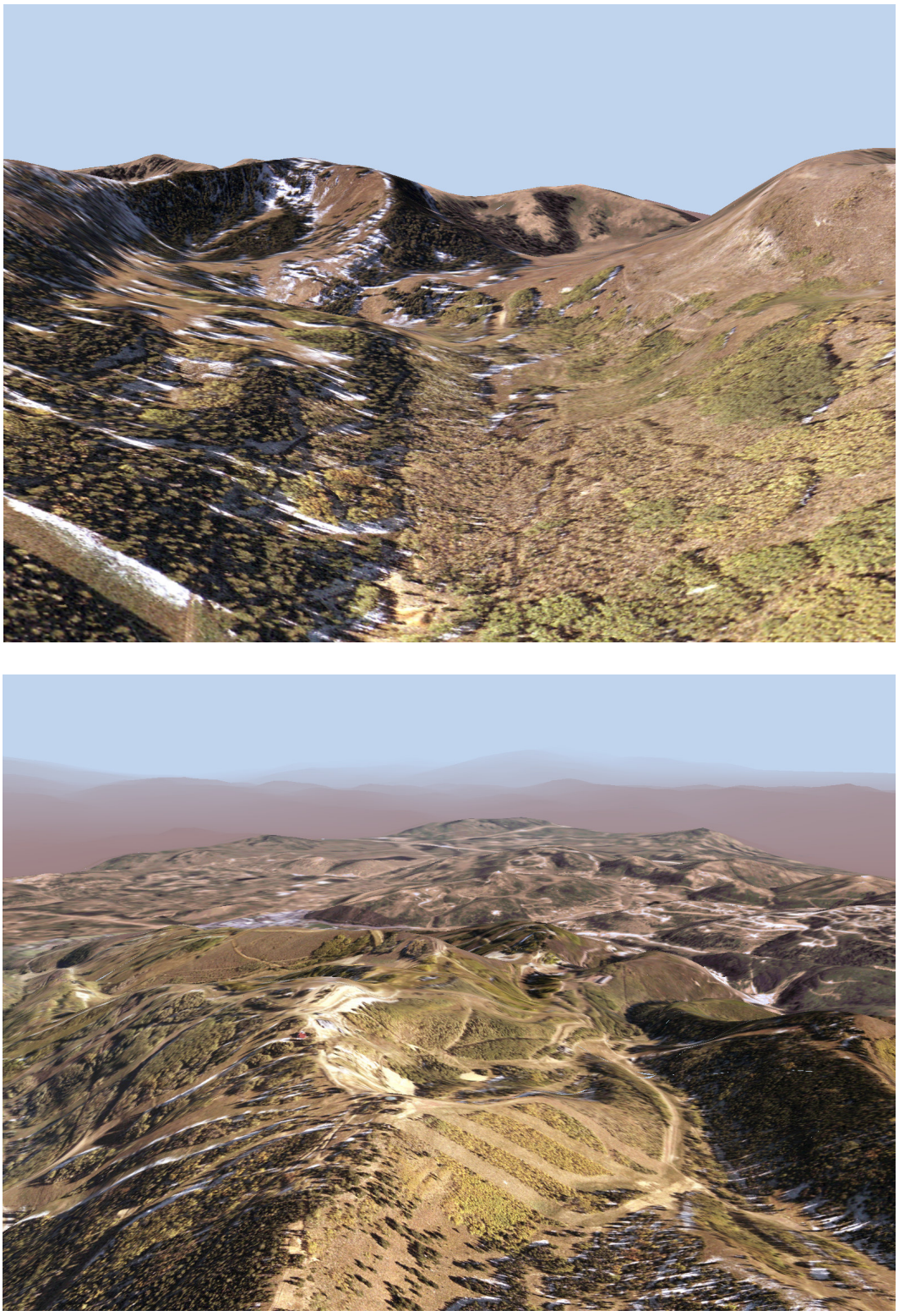


Figure 1.3: EView Success. Images produced by EView when the viewpoint is far from the surface. The images are very detailed and appear photorealistic.

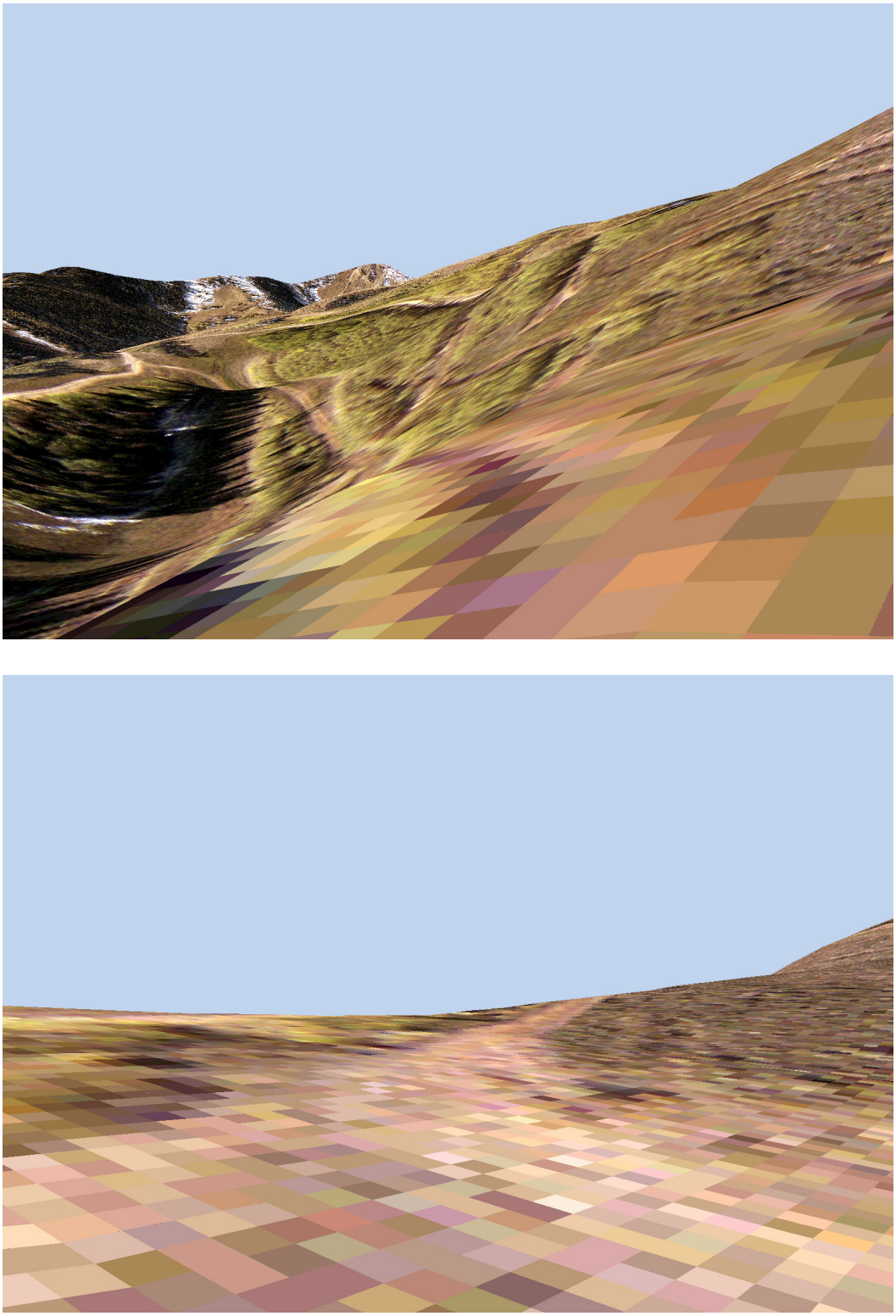


Figure 1.4: EView Failure. Images produced by EView when the viewpoint is near the surface of the terrain. The images lack detail and appear blocky. The research presented here will address this problem.

Chapter 2

Background

This chapter begins by introducing previous approaches to texture synthesis. We discuss the strengths and weaknesses of each approach as well as why texture synthesis alone does not solve the image enhancement problem. We then briefly introduce interpolation as it applies to this thesis. Two recently published solutions to the image enhancement problem are then discussed, including the limitations of these approaches and how they differ from our approach. An introduction to the approach taken in this thesis is then presented.

2.1 Introduction to Texture Synthesis

This section reviews previous work in texture synthesis and image interpolation. Research in texture synthesis began in the 1980s. Early attempts at texture synthesis focused on procedural method to synthesize two-dimensional and three-dimensional textures. Each procedure was written to generate a specific texture (ex. wood, marble, water, etc.). While this approach could produce good results, writing a new synthesizer for every type of desired texture is impractical.

Research next focused on sample-based texture analysis and synthesis. Sample based texture synthesis attempts to create a new texture similar to a sample. The analysis phase of this technique decomposes the sample texture to identify its statistical properties. The synthesis phase then uses the statistical characteristics to create a new texture with similar properties.

In [HB95], Heeger and Bergen introduced pyramid-based texture synthesis. Here a sample texture is decomposed using a steerable pyramid transform. A result texture is seeded with noise and is decomposed in a steerable pyramid. Iteratively, the histogram of the noise pyramid is coerced to match the histogram of the sample's pyramid. While this

approach produced reasonable results for some stochastic textures, it cannot synthesize structured textures.

Recent research has focused on two primary methods: pixel-based [Ash01, Bon97, EF99, HB95, IBG03, WL00, WL02] and patch-based [EF02, LL01, XGS00]. Both methods are based on the Markov Random Field (MRF). Pixel-based synthesis techniques generate images one pixel at a time. For each pixel in the destination image, the sample texture is searched for a pixel with a neighborhood matching the neighborhood in question. The destination pixel is then generated based on a "best fit" measure. Though pixel-based synthesis produces reasonable results for a wide variety of textures, it is computationally very expensive.

Efros and Leung introduced the neighborhood searching approach of pixel-based texture synthesis [EF99]. Their revolutionary method made previous techniques obsolete, because the neighborhood approach was much simpler and produced better results for more classes of textures. They showed that their technique worked very well on complex textures that contained both structure and stochastic elements.

Efros and Leung's approach takes as input a sample texture. The output is a synthesized texture that has the same local characteristics as the sample texture. The neighborhood approach synthesizes textures one pixel at a time from left to right, top to bottom. The top-left pixel in the synthesized texture is chosen randomly. For all other pixels in the synthesized texture, the algorithm searches the sample texture for a

neighborhood (32x32 block of pixels) that closely matches the neighborhood of the pixel in question. While this technique produces good results on many textures, it is very slow and on occasion can produce textures that reflect only part of a sample texture.

Wei and Levoy improved the pixel-based neighborhood approach by significantly accelerating that technique [WL00]. They succeeded in producing comparable results in a fraction of the time by using multi-resolution synthesis and tree-structured vector quantization.

Their approach synthesized the resulting texture several times at multiple resolutions. First, a low-resolution version of the sample was synthesized. This is similar to using large brush strokes to paint the general structure of the texture. Next smaller brush strokes will be used to add in more detail. After the lowest resolution is synthesized, higher resolution versions are created adding more detail. The advantage of multi-resolution synthesis is that it allows the use of a much smaller neighborhood when searching the sample texture. Using smaller neighborhoods decreases the time spent in the searching algorithm.

Wei and Levoy further improved the search performance by building a tree structured data object to store all possible neighborhoods from the sample texture. This data object allows for rapid searching for a matching neighborhood. When combining their tree structured searching with multi-resolution synthesis, Wei and Levoy produced good results much faster than the previous neighborhood technique.

Ashikhman further decreased the search time by limiting the neighborhoods to search based on results from previous searches [Ash01]. Ashikhman found that when synthesizing natural textures such as flowers and grass, this shortcut did not significantly affect the result's quality. However, in general, textures produced by [Ash01] are lower in quality than textures produced by other neighborhood techniques.

Pixel-based neighborhood texture synthesis seemed to be the defacto standard approach for all texture synthesis research until recently. In 2000, another revolutionary approach was published based on patch-based texture synthesis. Patch-based texture synthesis constructs an image by pasting square patches into the image at each step. Before the synthesis begins, a set of possible patches is extracted from the sample texture. Each of these patches is a small square piece of the sample texture. Patches are pasted into the destination image from left to right, then top to bottom.

The first patch is chosen randomly. To choose subsequent patches, the patch set is searched for the patch that best matches the neighborhood. Since the synthesis process generates a whole patch at each step, patch-based synthesis is an order of magnitude faster than pixel-based synthesis. For example, with a patch size of 32x32, a patch-based method would synthesize 1024 pixels each iteration, whereas pixel-based synthesis must perform a search for every pixel synthesized. Although the overhead cost of each iteration is more expensive for the patch-based system, the amount of texture generated stills give a significant overall performance improvement.

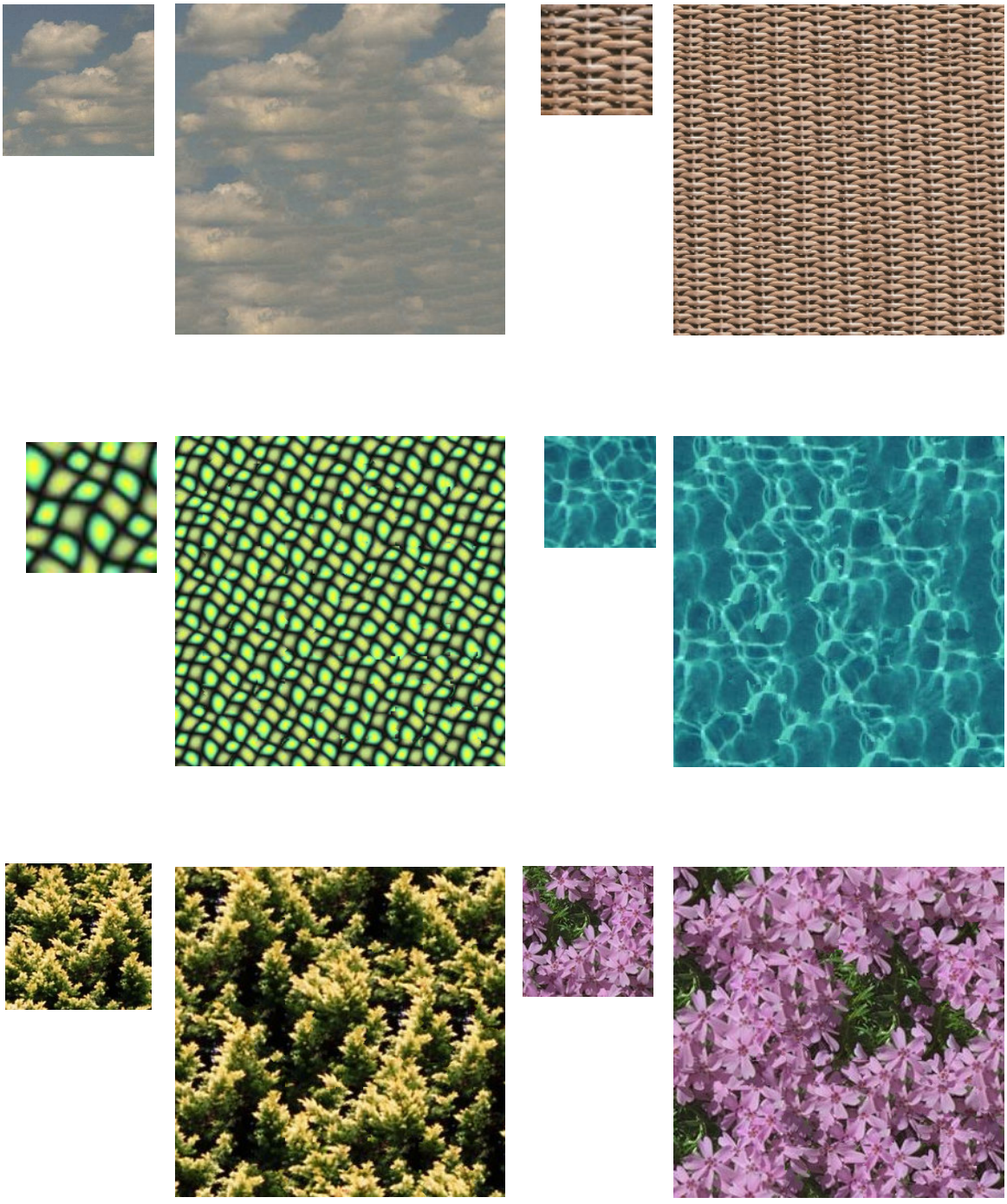


Figure 2.1: Texture Synthesis. Texture synthesis is the process of creating images with local characteristics like a sample image. The smaller images are the samples textures. The larger images are synthesized images created using a patch-based approach.

Patch-based synthesis was first proposed by Xu, Guo, and Shum [XGS00]. Patch-based synthesis was quite a breakthrough, because the reasonably simple algorithm produced very good results an order of magnitude faster than previous texture synthesis algorithms. While results were more repetitious than pixel-based methods, they were also much more predictable in quality.

Liang et al. simplified patch-based synthesis further and added three accelerators to synthesize moderate sized textures in real-time [LL01]. The first accelerator was to search for matching patches in a lower resolution. Then, patches that were marked as good prospects were searched in higher resolutions using an approximate nearest neighbor searching algorithm. This search was further accelerated using principle component analysis, which compares only some of the pixels from each patch that represent the greatest differences between the patches.

Efros and Freeman used patch-based synthesis for texture transfer [EF02]. This allows them to transfer one texture (e.g., an orange peel texture) onto another image (e.g., a photo of a banana).

Cohen et al. used Wang Tiles for patch-based synthesis [CSHD03]. This eliminated the high dimensional search and thus reduced synthesis time even more. This important performance enhancement will be discussed more in Chapter Four. After experimenting with both pixel and patch-based synthesis techniques, we have determined that a patch-based approach is more appropriate for our application. The patch-based

approach gives us the desired image quality and the required speed. Figure 2.1 shows textures created using patch-based texture synthesis.

2.2 Introduction to Image Interpolation

Image interpolation is a very mature topic that received much attention in the graphics community during the 1980s. Image scaling is the process of taking a source image and extending it to create a larger image. The primary problem with enlarging images using interpolation is that the larger result contains the same amount of discrete data as the smaller source image. Thus, visual discontinuities and artifacts are introduced into the enlarged image. Interpolation is the primary technique used for image scaling. Many interpolation formulas exist that provide varying levels of continuity when transitioning between data points. The type of interpolation chosen will determine the continuity and smoothness of the enlarged image.

The most common types of interpolation are bilinear and bicubic [Dod97]. Bilinear interpolation uses a 2x2 neighborhood of data points to calculate pixel color between data points. Bicubic interpolation uses a 4x4 neighborhood of data points to calculate pixel color. Although bicubic interpolation produces smoother, more continuous results, it is significantly more expensive than bilinear interpolation, and thus less practical for use in

real-time systems. Many other interpolation methods have been proposed (see [PKT83] for a comparison). Most of these were developed to achieve the smoothing of bicubic interpolation with less computation.

2.3 Similar Work in Detail Synthesis

Although the papers mentioned in the previous two sections address texture synthesis and image interpolation, none of them solves the problem of inserting detail into an image. To date we have discovered only two papers that attempts to solve the problem of detail insertion.

2.3.1 Image Hallucination

In [SZTS03], Sun et al. use a learning based approach to super-resolution. Their algorithm takes as input a low-resolution source image and several training images. Their algorithm produces a high-resolution version of the source image with edges both smooth and sharp. Their two-step approach begins by creating a training set of primitives and finishes with image synthesis.

In the first step, a set of image primitives is created from the natural training images. The primitives are obtained by using a primal sketch edge-

detection algorithm. Both high frequency and low frequency versions of each primitive are obtained.

In the second step, primitives are found in the low-resolution source image using the same technique as used on the training images. For each low frequency primitive found in the source image, the low frequency primitives in the training set are searched to find a similar structure. Once a low frequency primitive is found, the corresponding high-frequency version is used to enhance the low frequency source image with the high frequency data. This is only done for the intensity of the image at each pixel. The color channels are interpolated.

While the approach taken in [SZTS03] is very successful at enhancing images at a relatively low level of magnification (3x), their algorithm does not meet our needs in two areas. First, they algorithm is 3-4 orders of magnitude too slow for our needs. Their approach creates a high-resolution image for moderately sized input images in 20-100 seconds. Obviously, this cannot meet our needs of real-time enhancement.

Second, they do not attempt to insert detail on a large scale. Although their approach uses Markov techniques found in texture synthesis, the approach is targeting the problem of edge handling during super-resolution. This is more similar to edge directed interpolation or edge reconstruction than it is to texture synthesis. The resulting images do have both smooth and sharp edges, but the image does not have significant amounts of new detail.

2.3.2 Detail Synthesis for Image-based texturing

The second paper that addresses the problem of detail insertion has a strong emphasis on image-based modeling and rendering. In [IBG03] Ismer, Bala, and Greenberg use texture synthesis to insert detail. They use a multi-resolution pixel-based synthesis technique to coerce a poorly sampled texture to look more like a high quality sample texture. Their technique is very similar to Wei and Levoy's approach, but the comparison operator used during the search has been slightly modified. Instead of comparing the neighborhoods from the sample texture to only the synthesized pixel in the destination image, they also compare the prospective neighborhood against the corresponding area in the image to be enhanced with detail. This significantly increases the search time required for finding matching neighborhoods.

The problem and solution presented by Ismer, Bala, and Greenberg have several similarities as those presented here. The problem they address is enhancing low quality imagery and adding appropriate detail. They use texture synthesis techniques as a basis for their approach. However, there are three fundamental differences in our techniques.

The first difference is that [IBG03] uses a multi-resolution pixel-based synthesis approach. After implementing and evaluating two pixel-based methods and two patch-based methods, we have chosen to use a

patch-based approach. The patch-based approach produces visually pleasing results and is an order of magnitude faster than the most optimized pixel-based methods. Since real-time performance is one of the primary goals of this work, using a patch-based approach should provide significant speedup over the technique presented in [IBG03].

The second difference is that like other texture synthesis algorithms to date, [IBG03] does not address the need for using multiple sample textures. Their system requires the user to separate the source image into pieces of different textures. Each area of the image can then be enhanced separately. The algorithm presented here allows multiple sample textures as input and multiple texture classes in the resulting image.

The third difference is that Ismer, Bala, and Greenberg focus on inserting detail on a relatively low scale of magnification. Their examples show textures of bricks that appear fuzzy because the camera angle was not perpendicular to the surface or the camera was a significant distance from the brick wall. Their detail insertion algorithm cleans up the texture by coercing the fuzzy bricks to look more like bricks in a sample texture. In contrast, the algorithm presented here focuses on detail insertion at a much higher scale of magnification (scaling factors from four to 128).

2.4 Approach Presented in this Thesis

In order to solve the problem of real-time, detail synthesizing image enhancement, we use a combination of interpolation and texture synthesis. Our approach uses bilinear or bicubic interpolation as in traditional scaling algorithms to avoid aliasing and to provide smooth transitions between different classes of textures. In addition, it uses texture synthesis to solve two important problems:

1. It allows the user to specify the detail to be inserted into the output image by providing a representative sample for the system to replicate. These sample textures can be hand drawn, scanned images, or digital photos. There are no special requirements imposed on the input samples (i.e., the sample textures do not need to be able to tile seamlessly).
2. Applying texture synthesis appropriately will allow us to avoid the unnatural repetition of texture tiles that can occur with standard texture mapping.

The primary contribution of our approach is that it allows real-time detail insertion from multiple sample textures. This chapter has described several partial solutions to this problem, but none of the current approaches provide real-time image enhancement using texture synthesis. In addition,

none of the texture synthesis papers allows the use of multiple sample images.

Chapter 3

Overview: RETS

This chapter presents an overview of our approach to real-time image enhancement using texture synthesis (RETS). We begin by stating the requirements a solution must meet. We then define the inputs and outputs of the algorithm. Finally, we present a brief overview of our approach to solve the problem stated.

3.1 Solution Requirements

As previously stated, the purpose of the algorithm presented here is to enhance a source image with texture in real-time. Before discussing how we do this, we first list goals we want the system to accomplish:

1. The ability to synthesize a high-resolution version of a low-resolution source image.
2. The system should be able to insert appropriate detail from sample textures.
3. The system must run in real-time.
4. There should be no popping when transitioning from a source image to a synthesized image when used in MIP mapping systems.
5. The system should support multiple sample textures.
6. The system should allow for multi-resolution output.

Each one of the stated goals is very important in ensuring this algorithm can be used in real world applications. The first goal is the base objective of this thesis. The second goal allows the user to input sample textures easily. The second goal also requires that new detail be added. Adding detail is critical for large levels of magnification. The third goal of real-time performance is critical so that this technology can be used in interactive applications. Together, goals four and goal six allow RETS to be used with rendering systems that support MIP-Mapping (eg. OpenGL and DirectX). Goal five will enable our system to synthesize images with multiple texture classes. This is in contrast with all current texture synthesis algorithms that support only one input texture.

3.2 Inputs and Output

A user or program utilizing our enhancement technique is required to provide two items of input. The first is a low-resolution image to be enhanced. Our algorithm has been designed around the specific application of enhancing low-resolution aerial-imagery. Aerial imagery has the special property of being locally similar, meaning if a small 32x32 sub-image of an aerial photo is selected, it is very likely that you could find many other 32x32 sub-images in the same image that look very similar. For example, if you selected a sub-image of grass, you could find many other sub-images of grass that look very much like the first, but not exactly the same. Though this thesis will focus on enhancing aerial images, the technique could be applied to other classes of imagery as well.

The second item of input is a set of sample textures. The term texture as used in this thesis defines a special class of images. Textures should be locally similar. If you select any two sub-images of a texture (above a certain size), those two sub-images should always look similar. Thus, a digital photograph of a person would not be a texture as we are using the term, because a sub-image of the person's arm would not look like a sub-image of the person's nose. However, a digital photograph of a wheat field would meet our definition of a texture, because it is likely that any two sub-images

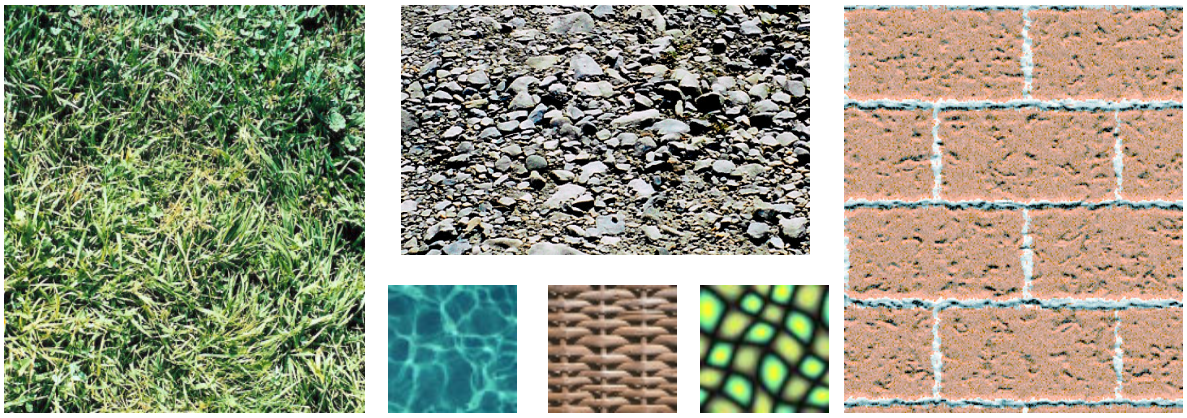


Figure 3.1: Examples of texture classes.

would look very similar. Figure 3.1 shows an example of several sample textures. Notice that each of these passes the test of being locally similar.

In traditional patch-based synthesis, the user is allowed to input only one texture sample to be used in synthesis. However, in our version of texture synthesis the user supplies multiple sample textures. The user should supply one sample texture for each type of texture that will occur in the destination image.

As shown in Figure 3.2, the output of RETS is a single high-resolution image that has the general structure of the source image and the texture qualities of the high-resolution sample textures.

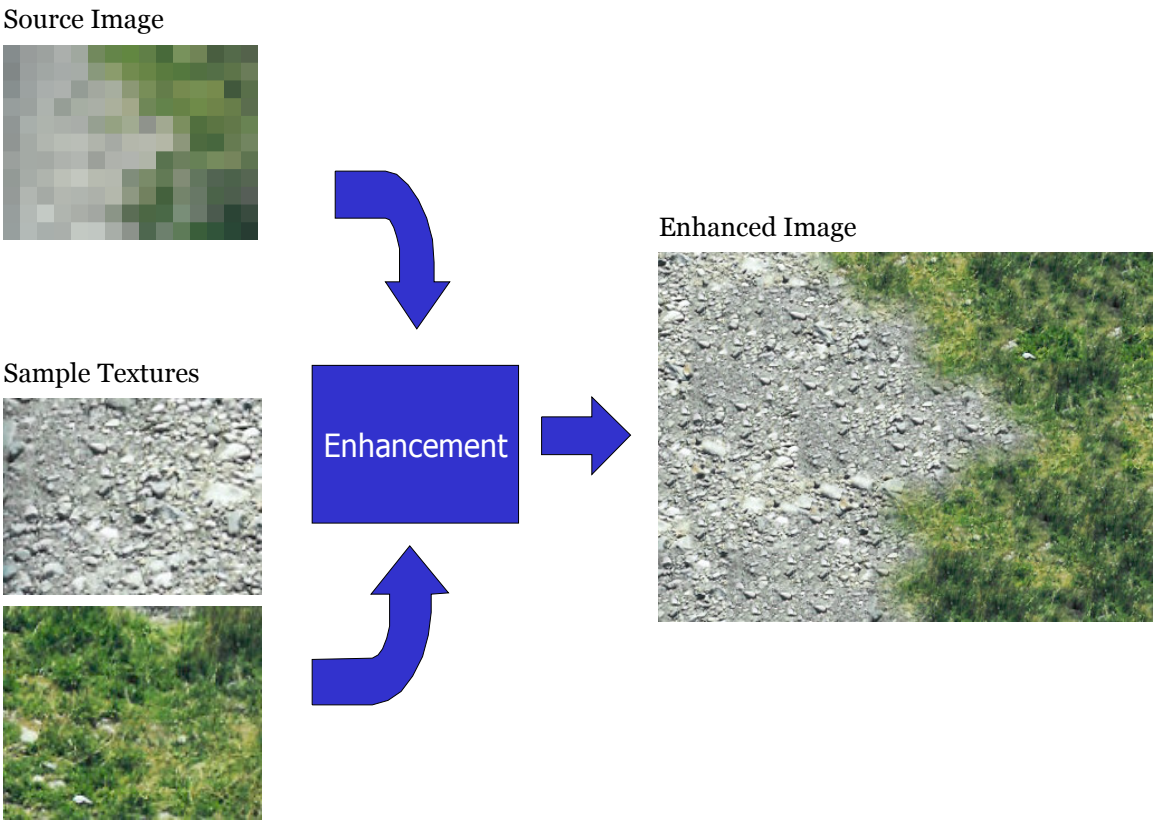


Figure 3.2: RETS Inputs and Outputs. The enhancement process takes two types of inputs. In the top left is the low-resolution source image. In the bottom left are two high-resolution input textures. On the right is the high-resolution enhanced image. The enhanced image has the general structure of the source image, but with detail consistent with the sample textures.

3.3 Algorithm Overview

The process begins by extracting one patch set for each sample texture supplied by the user. A patch set is a set of smaller sub-images created from a sample texture. Next, we classify the source image into areas

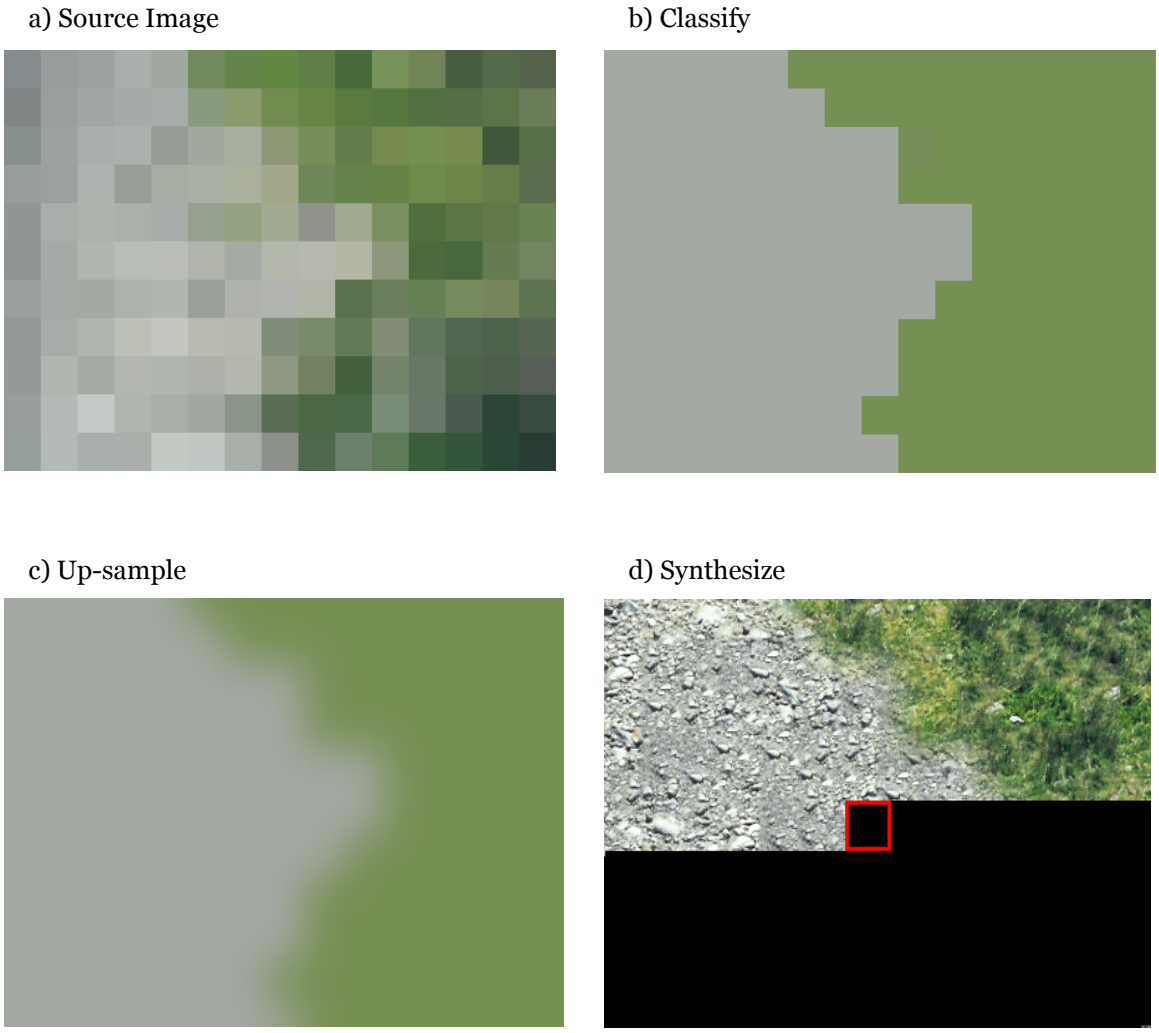


Figure 3.3: RETS Overview. (a) The low-resolution source image to be enhanced. (b) The source image is classified into areas of different texture classes. (c) The classified area and the original image are up-sampled. The up-sampled classified area is used to smooth transitions between areas of different texture classes. The up-sampled version of the original is used to reduce aliasing. (d) The last step uses a modified patch-based texture synthesis algorithm to create the high-resolution result.

of different texture classes. Here, the image is divided into regions (water, grass, and trees, etc.). Finally, we use a modified patch-based texture synthesis technique to synthesize a high-resolution image. We start in the top left corner. From our classification step, we know the texture class the area in question is best represented by, so we select a patch from the matching set. We then proceed through the image left to right, top to bottom selecting patches that match the area in question (see Figure 3.3). These steps will be discussed in the following sections as of this thesis:

1. Extract patches from sample textures (Chapter 4)
2. Classify source image (Section 5.1)
3. Up-sample the source image (Section 5.2)
4. Synthesize high-resolution result (Sections 5.3-5.7)

Chapter 4

Extracting Patch Sets

Patch extraction is the first step in the enhancement process. This step can be done as a preprocess step and need not be a real-time operation. This chapter will describe how most patch-based texture synthesis techniques extract patch sets and a weakness of this approach. The majority of the chapter will then discuss in detail an alternative approach for creating optimized patch sets, including modifications needed to support image enhancement (in contrast to texture synthesis alone). This alternative approach is based on Wang Tiles, which we will describe in this chapter.

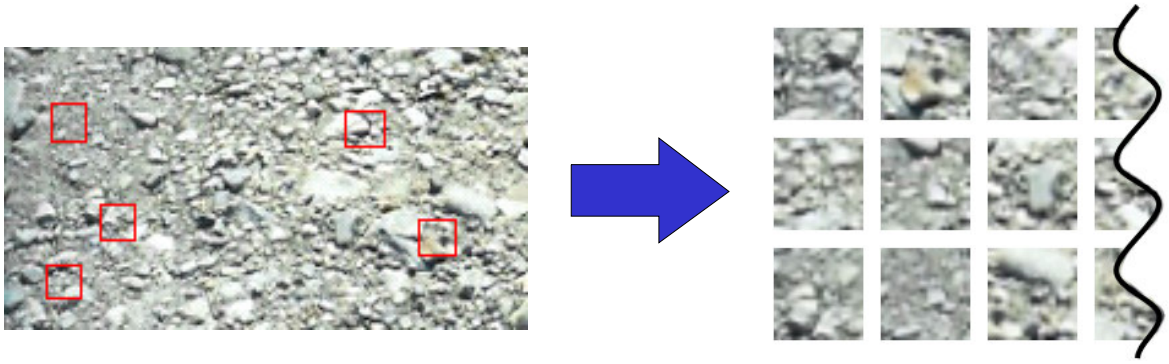


Figure 4.1: Traditional Patch Extraction. Traditional patch extraction selects all sub-images of a given size (ex. 32x32). This approaches is trivial to implement but creates very large patch sets.

4.1 Traditional Approach

Most patch-based texture synthesis techniques create the patch set by simply extracting all sub-images of a given size (e.g., 32x32) [EF02, LL01, XFS00]. While this is very easy to implement, it creates patch sets of a very large size. For example, if the texture was of size 512x512 and the patch size was 32x32, the resulting patch set would contain 230,400 patches.

In most patch-based texture synthesis algorithms, very large patch sets increase image synthesis time. Later, during the synthesis step, we must often search through patch sets for a patch best matching our current boundary conditions. In performing the search, we must compare the edge of each patch with the edge of our current boundaries. The first patch-based

texture synthesis technique used brute force searches. Although the first patch-based approach was much faster than pixel-based synthesis techniques, this technique was still very inefficient. Subsequent papers focused on optimizing the search with approximate nearest neighbor (ANN) searches, multi-resolution searches, and principal component analysis (PCA). Although the enhancements helped tremendously, the algorithm was still limited to small texture samples. It could not support large texture samples and still synthesize results in real-time.

Recently, in [CSHD03], Cohen et al. presented a technique for building optimized patch sets from a sample texture. These patch sets have the advantage of being very small, in terms of the number of patches used, when compared with patch sets created using the previous technique. Small patch sets improve performance during the enhancement process. Even though the patch sets are small, the patches work well together and can tile with little disparity. The technique used by Cohen et al. is based on Wang Tiles.

4.2 Introduction to Wang Tile Sets

A Wang Tile set is a collection of square texture tiles with each edge color-coded based on the properties of the image at that edge. The tile set

Figure 4.2: Wang Tile Set. Wang Tiles have color-coded edges constraining which tiles can be placed next to one another. Wang Tile Sets can be used to tile a plane with texture.

has a fixed number of color-codes for the horizontal edges and a fixed number of color-codes for the vertical edges. For example, a Wang Tile set could have two vertical color-codes and two horizontal color-codes. In this case, sixteen different tiles can be created by combining the color-coded edges. With h as the number of horizontal color-codes, and v as the number of vertical color-codes, we can calculate the number of possible tiles and the smallest valid tile set as defined below.

$$\text{Number of tiles possible} = \frac{2^{h} * 2^{v}}{hv}$$

$$\text{Smallest valid tile set} = hv$$

Wang tile sets do not need to be of any particular size. With two horizontal and two vertical color-codes, one could create a valid tile set ranging in size from four to sixteen tiles. If one chose a tile set of size eight, one must then choose eight of the sixteen possible tiles. Thus, many

different sets can be created from the same two horizontal and vertical color-codes.

Wang Tiles are used to tile portions of a plane. Tiles are laid onto the plane in raster scan ordering, beginning in the top left. The first tile is chosen randomly. Subsequent tiles must have edges that match the color-codes of the tiles already laid down.

To ensure that we can tile the plane successfully, we must guarantee that there exists a tile for all possible top-left color-code combinations. As long as this criterion is met, we are guaranteed that we will have at least one tile that can be placed in every location. Thus, we will always we able to tile a plane without encountering a boundary condition for which there exists no appropriate patch.

In order to avoid repetition caused by periodic tiling, we must have at least *two* tiles whose color-code edges match all top-left combinations. This assures us that our tiling can be non-periodic, because at each step we will always have a minimum of two tiles that match the current boundary areas. By randomly choosing between two or more matching tiles, we introduce the randomness required to ensure our tiling will be non-periodic.

Once we have a set of tiles that meet the criteria specified above, we can tile the plane in the following manner (see Figure 4.3):

1. Randomly select a tile for the top left corner.

2. Tile the first row by choosing tiles that have left edges matching the right edge of the previous tile.
3. Select a tile for the first column of the next row by matching the top edge to the bottom edge of the above tile.
4. Tile the remainder of the row by selecting tiles that match both the tile to the left and above the area in question.
5. Repeat steps three and four for each row until finished.

4.3 Using Wang Tiles to Build Patch Sets

Now that Wang Tiles have been introduced, this section will show how one can use the concept of Wang Tiles to create optimized patch sets. First, we must choose the number of horizontal and vertical color- codes. The number of horizontal color-codes does not need to be equal to the number of vertical color-codes. Increasing the number of color-codes can improve the quality of the synthesized result, but also increases the minimum size of the set and increases the computation time required to build the set.

Second, we must choose how many tiles will match each top-left edge combination. From the rules stated in the previous section, we know that

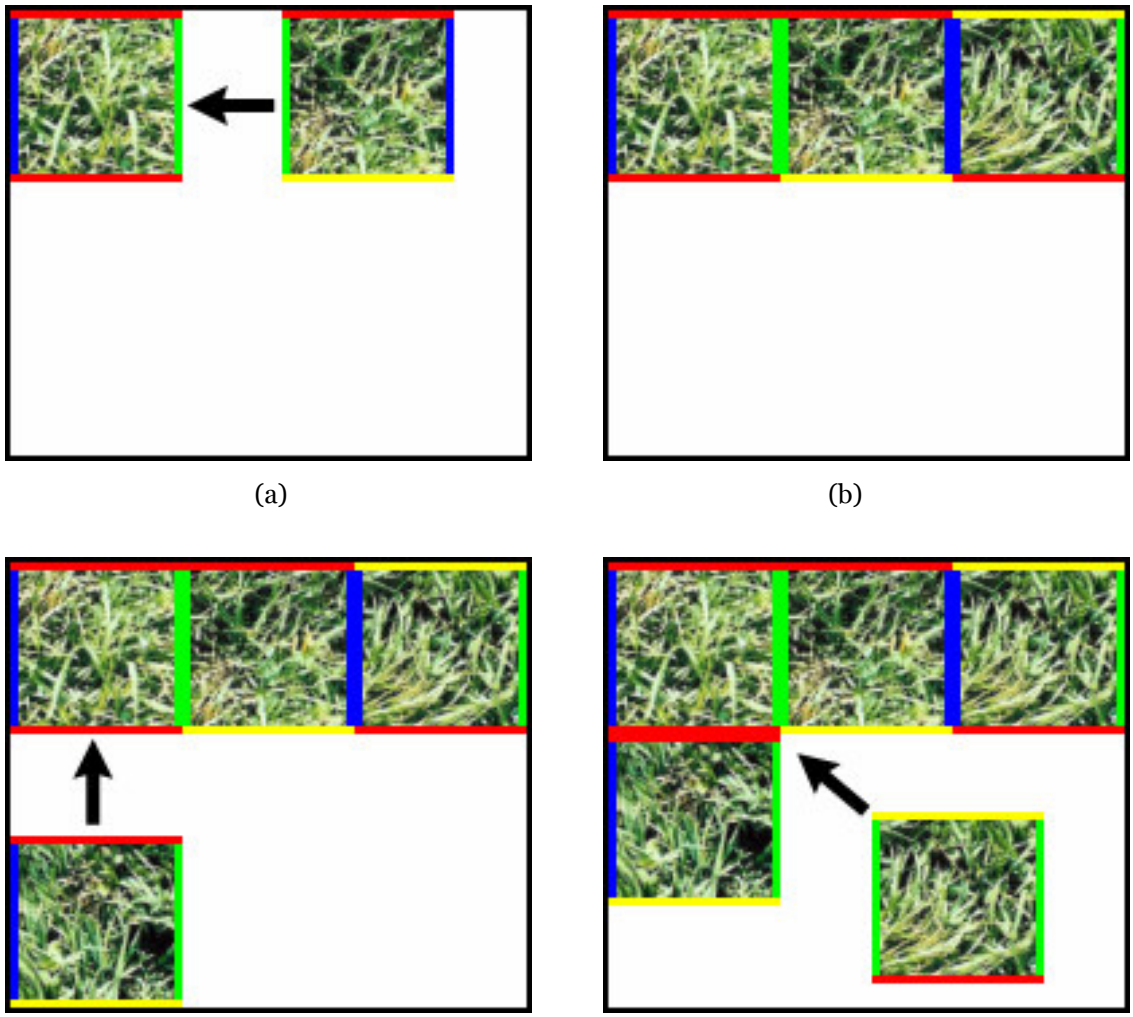


Figure 4.3: Wang Tiling. Above are the steps to tile a plane using Wang Tiles. (a) The top left tile is chosen randomly. The next tile is chosen to match the right edge of the previously tile. (b) The first row is completed by continuing to match the right edge of the previous tile. (c) The first tile of the second row is chosen to match the bottom edge of the above tile. (d) Subsequent tiles must match both left and top boundary conditions.

we must have at least two tiles that match each possible top-left combination. As with the number of color-codes, using three or more tiles to match all constraints can increase the quality of the resulting texture, but at the cost of a larger set size and generation time. Note, however, that

patch sets can be pre-computed. Thus, neither the number of color-codes nor the number of tiles to match each top-left combination have an effect on the speed at which an image is enhanced. We have found that five horizontal and five vertical color-codes combined with two tiles for each top-left combination produces high-quality results.

Once we have chosen the above-mentioned parameters, we can calculate the size of the resulting patch set. The size of the patch set is defined by the following formula:

Patch set size = $h * v * n$

where

h = number of horizontal color-codes

v = number of vertical color-codes

n = number of tiles to match each top-left combination

So, for h=5, v=5, n=2 we have a patch set of size fifty. Normally, the size of the patch set is less than the number of possible tiles. In that case, only a portion of the possible tiles will be selected for inclusion in the patch set.

Once we have determined the patch set size, we then build the patch set. The steps taken in building the patch sets are (see Figure 4.4):

1. Randomly select $h + v$ diagonal sub-images from the sample texture (Figure 4.4a). For the example shown in the figure h=2 and v=2, thus

we have selected two diagonal images for horizontal edges and two diagonals images for vertical edges.

2. For each top-left color-code combination find the best n combinations. This is done by first selecting a diagonal sub-image for the top and another diagonal sub-image for the left edge of the tiles. In Figure 4.4b, there are four top-left combinations possible by combining our horizontal and vertical diagonals.
3. Try all possible combinations for the right and bottom diagonals. As can be seen in Figure 4.4c, we can create four different combinations for each top-left combination. Thus, sixteen tiles are created.
4. Calculate the error of each tile by comparing the boundary areas where the diagonal regions overlap. The error between overlapping boundary areas is equal to the difference between the corresponding RGB values.
5. After calculating the error for all possible combinations, select the tiles with the least error and add these tiles to the prospective set. Remember, a Wang Tile set is only a subset of all possible tiles that can be created using the diagonals selected in step one. We want our set to have the least possible overall error, so we select tiles that have the least disparity along diagonal boundaries.
6. Now that we have a complete patch set, we calculate the error of the entire prospective patch set. This is done by summing the error of all tiles that were added to the prospective set in step 2.

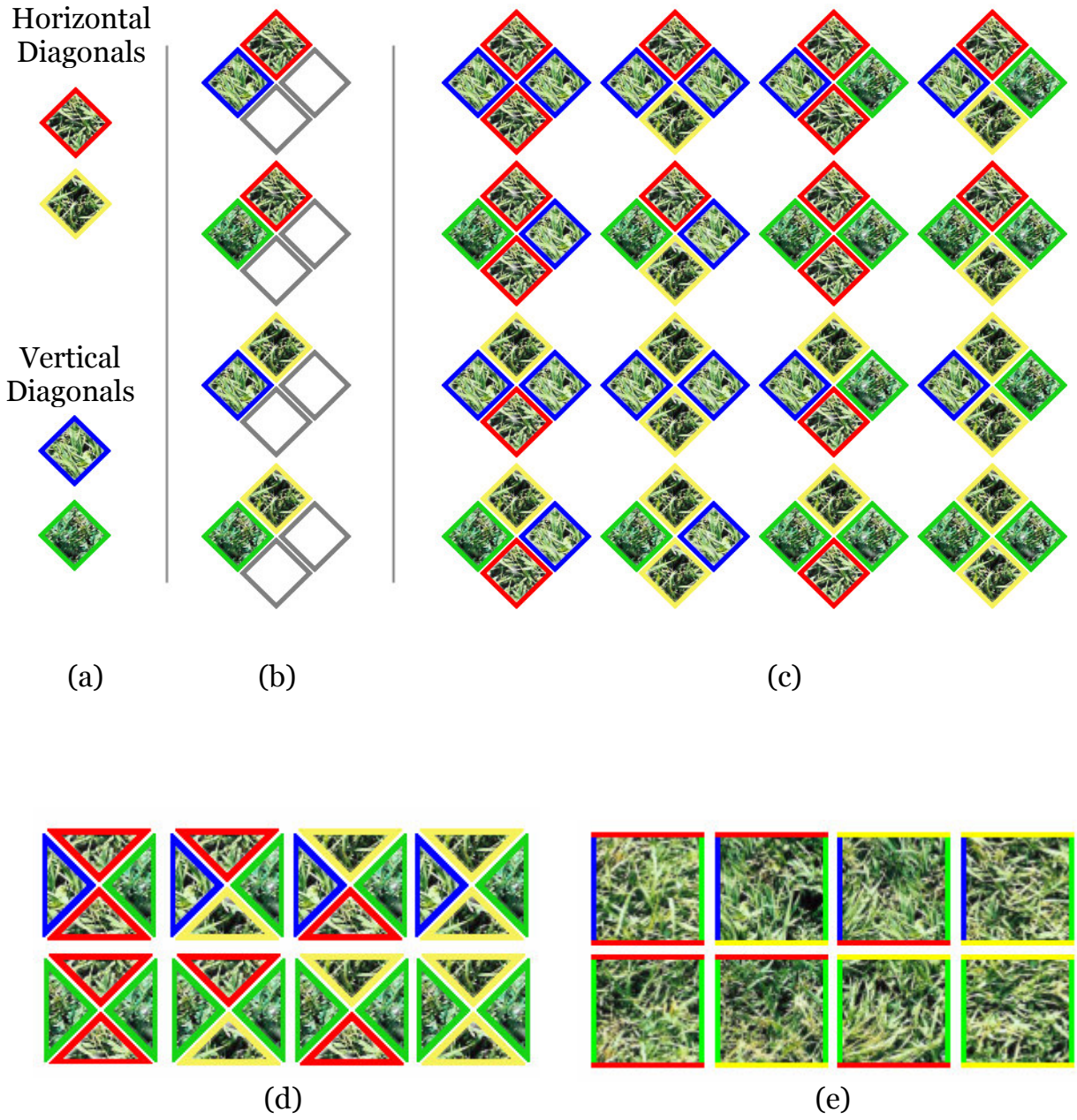


Figure 4.4: Automatic Wang Tile creation. (a) Select diagonal sub-images from the sample texture. (b) Group diagonals into all top-right edge color combinations. (c) For each top-right color combination, try all combinations. Select those with least disparity where diagonals meet. (d) Trim large diagonals into square patches (e) Stitch pieces into patches.

7. If the error is less than a user-defined threshold, keep the set and exit. If the error exceeds the threshold, but has the least error thus far, store the set. If we have surpassed a user-defined number of iterations, return the best set thus far. If none of the above conditions have been met, go back to step one and repeat.

4.4 Non-periodic Repetition

By ensuring at least two tiles from each set match all top-left edge color combinations, we guarantee that a tiling can be non-periodic. However, certain tile sets can have a sub-set of tiles that can take over the tiling. In Figure 4.5, we see four tiles that were part of a much larger tile set. However, during our research we noticed that when the complete tile set was used for texture synthesis, over fifty percent of the resulting image was synthesized using only these four tiles. The other forty-six tiles were rarely used. On closer inspection, we found that once these tiles were first used, the rest of the image used *only* these four tiles. When this occurs, the resulting texture appears very repetitive even though it is technically non-periodic.

You will notice that these four tiles alone are a valid Wang Tile set. They can tile a plane without ever finding boundary conditions they cannot

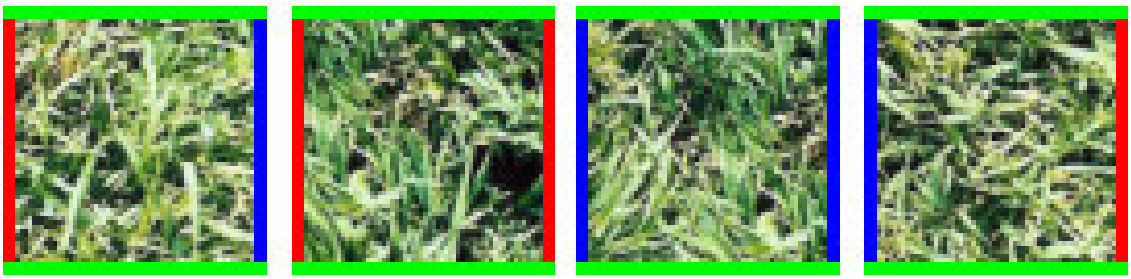


Figure 4.5: Cyclic Repetition. These four tiles, which were generated as part of a much larger set of tiles, cause cyclic repetition. Notice that all four tiles have green top and bottom edges. Avoiding tiles with two edges of the same color drastically reduces the occurrence of these subsets.

match. The set also matches all its own boundary conditions with two possible tiles. Since the large set containing these tiles was created with only two tiles matching each top-left edge color combination, these tiles are the only tiles that can match boundary conditions produced by this subset. Thus, randomly choosing between the two tiles that match the boundary condition does not allow us to escape the repetition caused by this cyclic tiling.

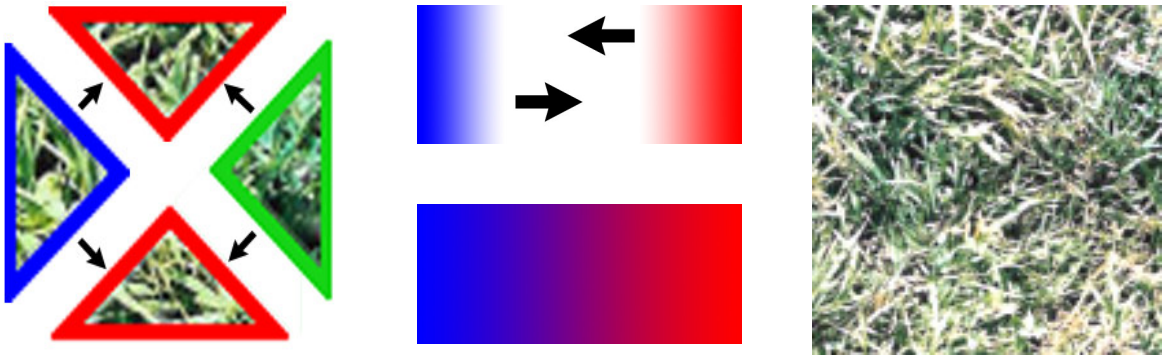
Since we cannot avoid repetition if a tile set has a subset like the one previously described, we must ensure that our tile sets never contain such subsets. Notice that all four tiles have green top and bottom edges. By limiting our Wang Tiles to have unique colors for its top, bottom, left, and right edges, we assure that small cyclic sets like Figure 4.4 cannot exist.

Much larger cyclic subsets are still possible, but are several orders of magnitude less probable.

4.4 Stitching Diagonals into Patches

Once we have selected the diagonals to use and the formation in which they will be placed, the diagonals must be stitched together and trimmed into square patches. Simply pasting the diagonals together will cause noticeable discontinuities along the two interior diagonal seams. Although diagonals are selected in a manner to maximize the continuity across boundaries, they will not be perfect. To further minimize the discontinuities, a stitching technique is used to join the diagonals.

In [CSHD03], Cohen et al. use the stitching technique presented by Efros and Freeman in [EF02]. The technique is called minimum error boundary cut. Dynamic programming is used to find the least cost path through the overlapping boundary region of two diagonal pieces. The cost function is the difference between RGB values at the point in question. The difference function is calculated as the sum of the squared differences of the three components. Although using a minimum error boundary cut produces much better results than simply pasting the diagonals together, it has disadvantages. First, the results are visually mediocre at best. This



(a) (b) (c)

Figure 4.6: Diagonal Stitching. (a) Stitching the diagonal pieces to form a patch. (b) Feathering is used to fade out one edge as the next edge fades in. (c) The result is a smooth transition between diagonals.

approach performs poorly on structured textures with patterns that are not axis aligned and textures with smooth transitions in color. Second, this technique is computationally more expensive than other proposed solutions. An alternative method, used in [LL01], produces comparable results, is more efficient, and is much simpler to implement.

In [LL01], Liang et al. use cross edge filtering to join pieces of an image. The technique was originally used by Szeliski and Shum to stitch together multiple digital photographs to create a larger panoramic picture [SS97]. As can be seen in Figure 4.6, pixels are simply weighted according to their distance from the edge of the diagonal. Pixels closer to the edge receive less weight. This feathering approach fades out one diagonal as the next diagonal is faded in. This simple blending approach is easy to implement but produces very good results. We have experimented with

combining both the least error path and cross edge filtering, but the results were not significantly better than cross edge filtering alone. Thus, the technique we use is cross-edge filtering.

Once the diagonal pieces have been stitched together, the large diagonal image created is then trimmed into a square patch. We do this for each patch. At this point, we also calculate the average color of each patch in the set. Later this is used to transfer the texture or detail represented by these patches to other areas of color space. For example, patches of dirt can be used to synthesize dark brown dirt, light brown dirt, or dirt with a reddish tint. Finally, the patch set including the patches, average color, and parameters with which the set was built are written to a file to be utilized by the real-time enhancement process.

Chapter 5

Image Enhancement

This chapter details the real-time process of enhancing the source image. The first section discusses how we classify the source image into areas matching the texture classes represented by the input sample textures. The second section describes how to up-sample the source image to the desired output resolution. In the third section, we will overview the process of tiling the patches into the destination image. The fourth section details how we choose an appropriate patch at each step. The fifth section discusses the need for custom patches and how they are to be constructed. Finally, we will discuss how this approach supports multi-resolution enhancements.

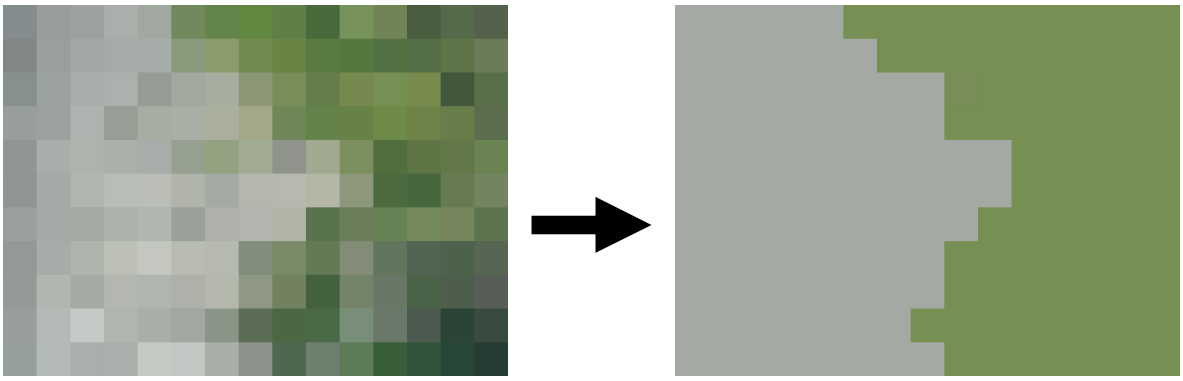


Figure 5.1: Classification. Classification is the process of dividing the source image into areas of different texture classes.

5.1 Classification of the Source Image

The first step in the real-time portion of the image enhancement process is to classify the source image (see Figure 5.1). The classifier can be as simple as finding the sample texture whose average color value is nearest the color value of the source pixels. The classifier can also be a more robust Bayesian classifier, taking into account pixel color, average neighbor color, maximum neighbor color, minimum neighbor color, etc.

The accuracy of the classifier is dependant on the type of image that is being enhanced. There have been many classifiers built for specific types of images. One example is a classifier built at the University of Utah by Premoze et al. for classifying panchromatic aerial imagery [PTS]. Their

classifier used pixel brightness, average neighborhood brightness, minimum neighborhood brightness, maximum neighborhood brightness, elevation, slope, aspect, and the angle to the southern border as features for the Bayesian classifier.

After implementing both the simple approach (nearest average color) and a Bayesian classifier, we chose to use the simple approach. The simple approach is to classify each pixel to the texture with an average color nearest to the pixel's color. Distance is computed as the sum of the squared differences of the three color components. This simple approach has two advantages. The first advantage is that color shifting is reduced during the synthesis process. Section 5.3 will give details on why this is the case. The second advantage of using the simpler classification method is that it is computationally less expensive than alternatives. This is important in that the focus of this thesis is *real-time* image enhancement.

5.2 Up-sampling the Source Image

Previously we stated that a requirement of our solution is that the solution must synthesize enhanced images that would not cause visual popping when transitioning between the original and the synthesized image. This means that in rendering systems that support MIP mapping, the original image could be used at one level of the MIP map and enhanced

images could be used at higher levels of the MIP map. To ensure that there is no noticeable visual popping when the rendering system transitions from the original source image to an enhanced image, we must guarantee that a down-sampled version of the enhanced image has minimal differences in colors at each pixel when compared to the original source image. Essentially this means that for a given area in the source image, the corresponding area in the destination image should have the same average color.

In addition to reducing the popping problem, a darker area of grass found in the source image should be enhanced to become a more detailed, darker area of grass regardless of the color of the grass in the sample texture. Likewise, a light area of grass in the source image should be enhanced to become a more detailed light area of grass.

The two problems stated above can be solved by guaranteeing that the average color of an area in the destination image matches the average color of the corresponding area in the source image. We do this by up-sampling the source image to the desired output resolution and then inserting the texture or detail into that image. The process of inserting detail into the up-sampled data modifies individual pixel values to add texture, but average color values for an area must still match the color values of the corresponding area in the source image. Section 5.3.1 details how we can insert texture into an image without changing the average color values of local areas.

When up-sampling the source image to a higher resolution there are many re-sampling functions that could be used. Nearest neighbor re-sampling simply stretches the image and would produce harsh discontinuities and jagged edges. Bilinear and bicubic interpolating functions create smooth transitions and can be used on images to avoid the blocky appearance caused by nearest neighbor re-sampling. Many other interpolating functions can be used to resample an image. Each produces a different interpolation of the source image and each has different computational costs. Due to their relatively low computational cost and acceptable quality of the results, we have implemented bilinear and bicubic re-sampling in our system.

Bicubic interpolation uses a 4x4 grid of sixteen data points from the original image to calculate the data for each point in the re-sampled image. Using this data the bicubic function creates C2 continuous output. Bilinear interpolation uses only a 2x2 grid of four data points. Using this data, bilinear interpolation creates only a level one continuous image. Following are both the bicubic and bilinear interpolating functions.

$$bicubicInterpolation(x, y) = \sum_{i=0}^{3}\sum_{j=0}^{3}[w_{ij} \cdot original(i, j)]$$

$$w_{ij} = R(i - dx) \cdot R(dy - j)$$

$$R(x) = \frac{1}{6}[P(x+2)^3 - 4P(x+1)^3 + 6P(x)^3 - 4P(x-1)^3]$$

$$P(x) = \begin{Bmatrix} x > 0 \rightarrow x \\ x \leq 0 \rightarrow 0 \end{Bmatrix}$$

$$bilinearInterpolation(x', y') = \sum_{i=0}^{1}\sum_{j=0}^{1}[w_{ij} \bullet original(x+i, y+j)]$$

$$w_{ij} = R(i - dx) \bullet R(dy - j)$$

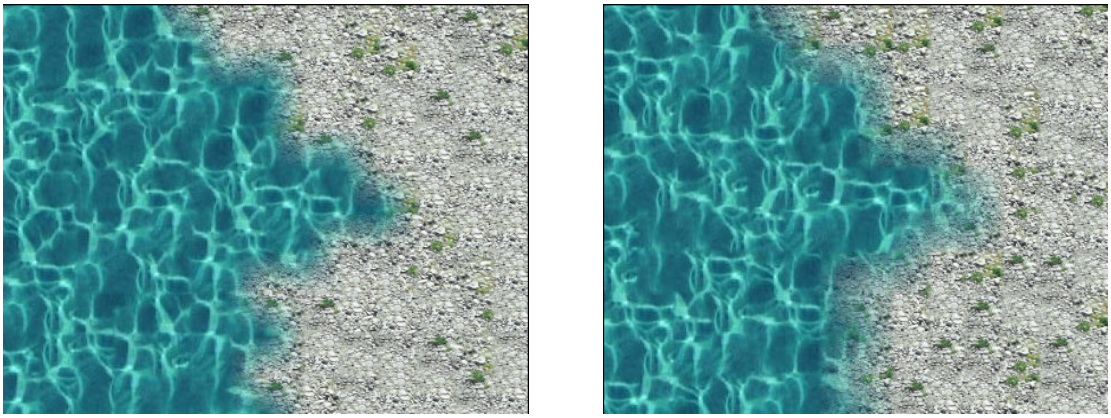
$$R(x) = \begin{Bmatrix} -1 \leq x \leq 0 \rightarrow x+1 \\ 0 \leq x \leq 1 \rightarrow 1-x \end{Bmatrix}$$

As can be seen from the above functions, bicubic interpolation is significantly more expensive than bilinear interpolation. Though the bilinear results are not as smooth as the bicubic results, they are adequate for our needs and provide significant performance benefits over bicubic interpolation. Thus, our implementation uses the bilinear functions.

Since the red, green, and blue components must be interpolated for every pixel in the destination image, this step is the most computationally expensive part of the enhancement process. To improve performance, we build lookup tables containing the blending weights (w_{ij} in the above equations). This significantly reduces the amount of computation necessary

for interpolating each pixel. Once we have up-sampled the source image we have a larger image that has smooth color transitions. The image appears to be a larger, blurry version of the source image, but lacks detail to look realistic.

(a) (b) (c)



(d) (e)

Figure 5.2: Interpolation Comparison. Comparison of bilinear and bicubic interpolation. (a) Original image (magnified using nearest-neighbor interpolation). (b) Bilinear interpolation of a. Notice the jagged edges have been removed. (c) Bicubic interpolation of a. (d) Bilinear interpolation has been used to smooth the seams between textures classes. (e) Example of texture synthesis using bicubic interpolation between texture classes.

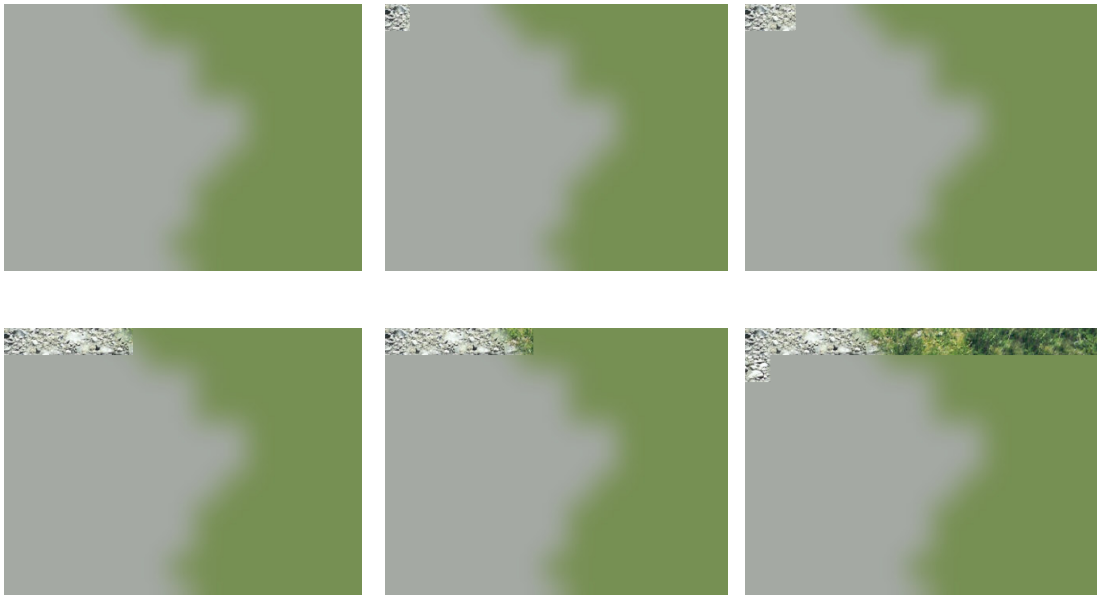
5.3 Pasting Patches

Once the tiles have been built and the original image has been up-sampled and interpolated, we then synthesize the destination image. The synthesis process combines the texture from patches and the colors of the up-sampled source image to create the destination image. This process creates the destination image one block at a time in raster scan ordering. We begin in the upper left corner of the destination image and move left to right until we complete a row. After completing the first row, subsequent rows are synthesized moving top to bottom.

Each block pasted into the destination image will correspond to a square area the size of a pixel in the source image. This corresponding area is not centered on a pixel. Instead, the area will be one-forth of four pixels with the area's corners corresponding to a pixel center in the source image. This distinction is necessary for creating custom patches to smooth transitions between areas of different texture classes (discussed in Section 5.5). Since each patch pasted into the destination image will always correspond to the area of one pixel in the source image, we will vary the patch size used for synthesis to accommodate our goal of multi-resolution output (discussed more at the end of this section).

(a) Patch Sets

(b) Synthesis



(c) Result

Figure 5.3: Synthesis Steps. (a) Patch sets used for texture. One set for each texture class is needed. (b) Appropriate patches are pasted into the result, left to right, top to bottom. Patches are chosen such that their top and left edges match the bottom left edges of patches already pasted into the image. (c) The resulting image.

For each block of the destination image synthesized, we must select an appropriate patch. This patch is chosen from the patch sets built, as described in Chapter Four, or a custom patch is created by combining multiple existing patches.

The method for choosing an appropriate patch is the subject of the following section. Once a patch has been selected, we chose the proper resolution of the patch.

Previously it was stated that a patch used in the destination image would always correspond to an area the size of one pixel in the source image. Thus, the patch size required is dependant upon the level of magnification being used in the current enhancement operation. For a two times magnification we would use a patch of size 2x2. For a four times magnification we would use a patch size of 4x4. Thus, for all patches we need access to versions of the patch at various resolutions. This is accomplished by computing and storing scaled versions of a patch. Scaling is done with the interpolating function previously mentioned.

5.3.1 Transferring Detail

Once a patch has been chosen and we have obtained the appropriate version based on the level of magnification, we must combine the texture from the patch with the color from the up-sampled source image.

Combining these two pieces can be thought of as transferring the detail or texture from the patch to the up-sampled image. We are inserting detail from the patch into the blurry, detail lacking, up-sampled source image.

Figure 5.4 shows an example of detail transfer. Image (a) shows the original texture. Image (c) represents the blurry image needing detail. Image (d) shows the result of transferring detail from image (a) into image (c). Figure 5.5 shows more results of detail transfer.

We have used two methods to transfer the detail from the patches into the up-sampled source image. One method works in RGB color space and the other in HSV color space. First, the RGB approach will be discussed followed by the HSV alternative approach. The discussion will include a comparison of the resulting image quality and performance.

5.3.2 RGB Detail Transfer Approach

The RGB approach begins by creating detail patches from the texture patches previously built. A detail patch is like a texture patch except that it does not contain absolute colors. Instead, the values stored in a detail patch are the differences between the corresponding value in the texture patch and the average color of the patch. Whereas a texture patch can tell us that at position (3,5) the pixel value is red [r=255, g=0, b=0], a detail patch will tell us that the pixel is 10 units more red and 5 units less green than the average

color of the original texture patch. The texture patches contain color intensities ranging from zero to 255, while the detail patches contain color intensity offsets ranging from -255 to 255.

To combine a detail patch with a block from our up-sampled source image, we simply add the value from the detail patch to the corresponding component from the up-sampled source image. Some pixels will be darkened; others will be lightened. Some pixels will become more blue and others more red. However, because the sum of all differences between each pixel in the texture patch and the average color of the texture patch is near zero in all three color bands, the sum of all components for each color band in the detail patch will be near zero. So, while we lighten some pixels and darken others, it has little net effect on the overall color of the area. This preserves the average colors of the source image while adding detail. Thus, when a RETS enhanced image is used with MIP mapping, popping artifacts can be avoided when transitioning between original imagery and enhanced imagery.

5.3.3 Color Shifting

While zero net change in color to a local area is ideal, we do introduce a small amount of color shifting caused by overflow. When adding a component from a detail patch to a component from the up-sampled source

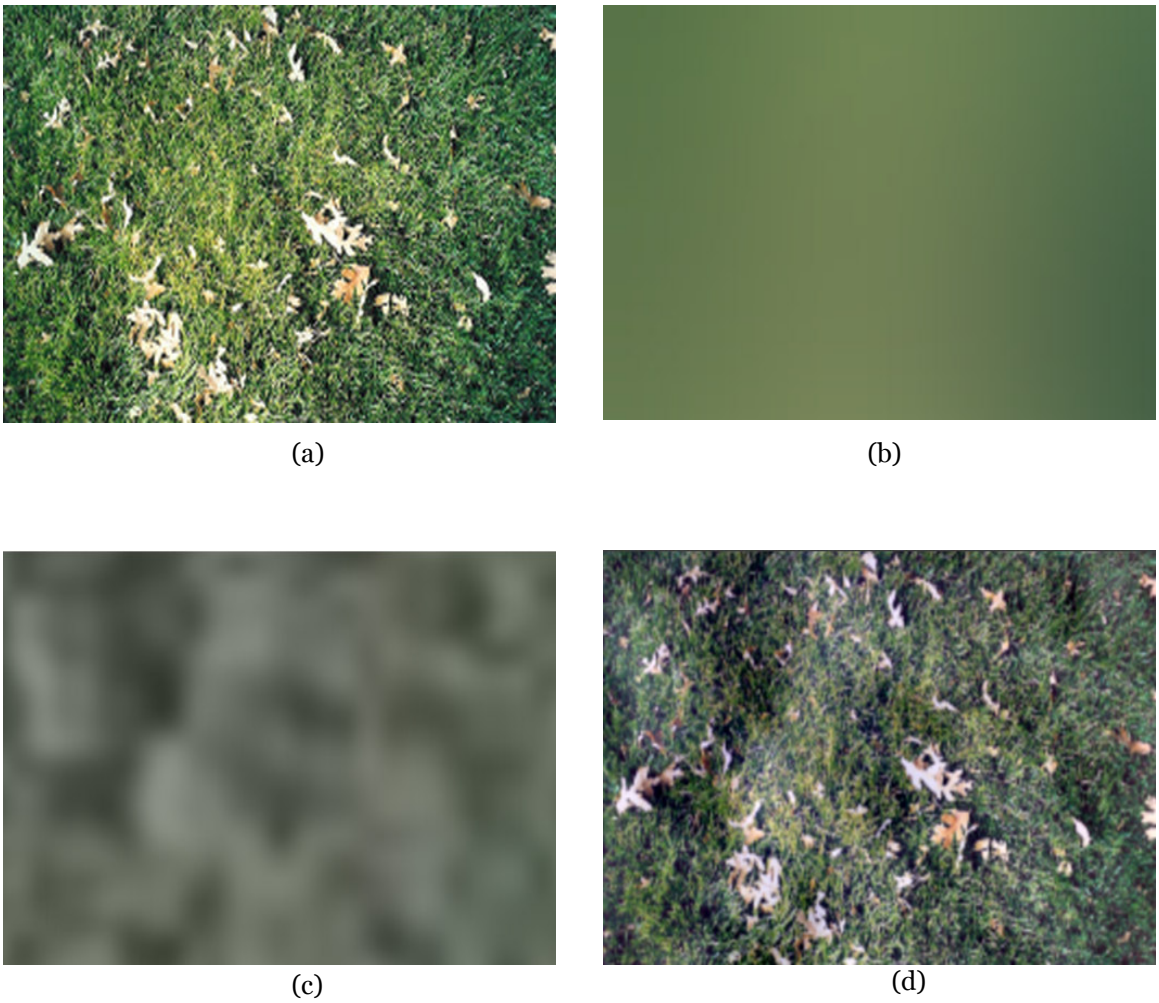


Figure 5.4: Detail Transfer and color shift. (a) Original texture. **(b)** Average color of the texture (a) **(c)** The image containing the desired texture. **(d)** The resulting image created by differencing image (a) and image (b), and then adding the result to image (c). Image (d) has the texture characteristics of image (a), but the average color of image (c). Image (d) was created using the RGB texture transfer approach. Notice the pink tint of the leaves.

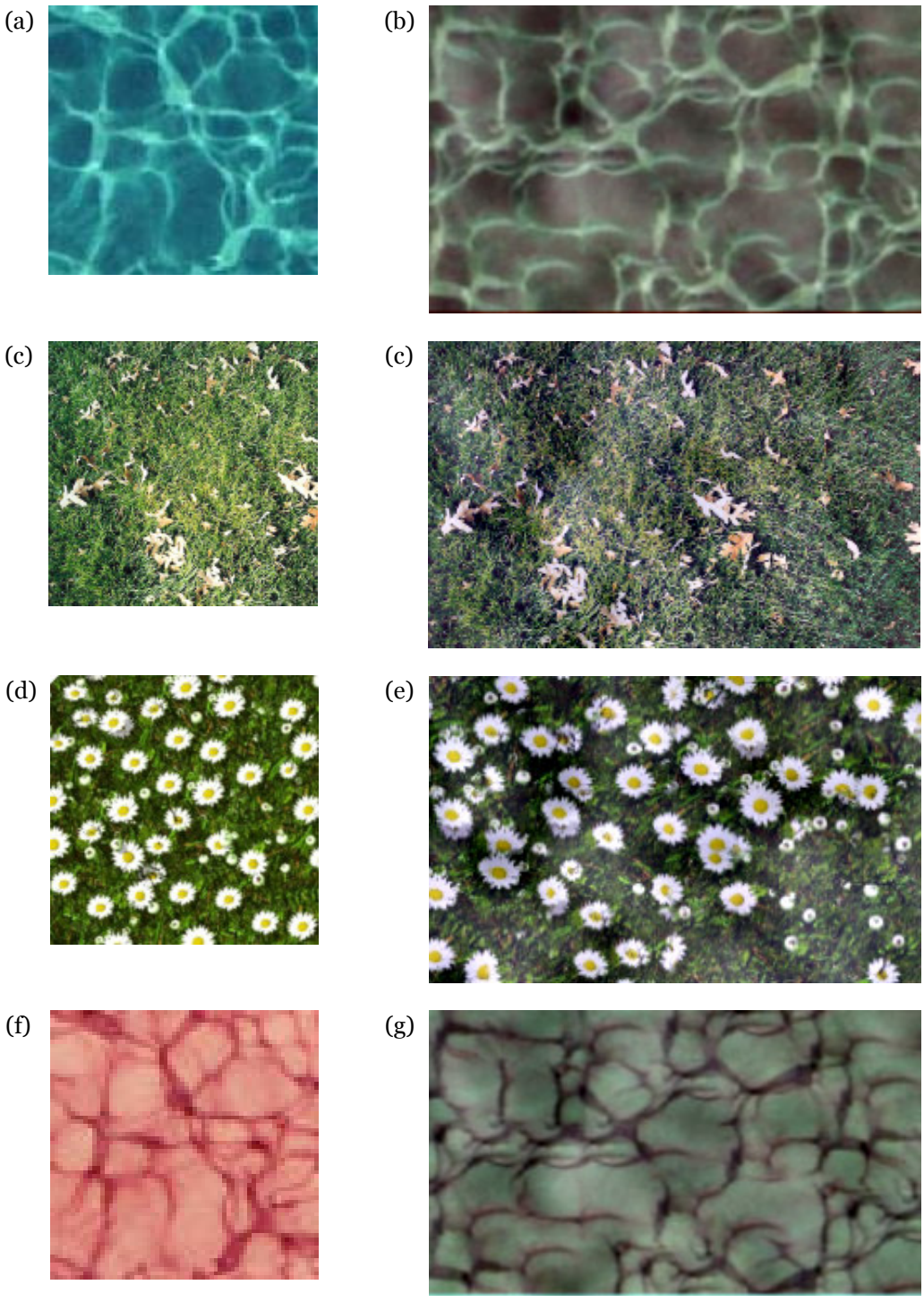


Figure 5.5: Detail Transfer. Examples of various texture being transferred to different area of color space. The original sample textures are the smaller images on the left. The large images on the right are the resulting images created using the RGB approach to detail transfer. The image used as the destination for the detail transfer can be seen as image (b) in Figure 5.6.

image the result can be less than zero or greater than 255. When this happens, we must clamp the value at the limit and discarded the excess value. Exceeding these limits is caused by two contributing factors. First, overflow will happen more often the greater the deviance for pixels in the texture patch from the average color of the texture patch. Second, overflow will occur more often the greater the distance between the average color of the patch and the color of the area the detail is being transferred to.

For example, if the texture patch has a value at some pixel of [r = 10, g = 10, b = 10] and the patch has an average color of [r = 150, g = 150, b = 150], the detail patch would have a corresponding value [r = -140, g = -140, b = -140]. If this is applied to a pixel from the up-sampled source image with a value of [r = 40, g = 40, b = 40], the resulting value will be [r=-100, g=-100, b=-100]. This value must be adjusted to [r=0, g=0, b=0].

In general, if we transfer a texture to an area that is significantly more intense in any color band than the patch average for that color band, we will reduce the overall intensity of that color band for that area of the image. While this can be a problem and should be taken into consideration when building patch sets, the error introduced is usually insignificant. In our tests, overflow normally causes a 1.3% shift in each color band, but does not seem to be enough to cause any noticeable popping when transitioning between the original image and an enhanced image when using MIP mapping.

In addition to the color shift introduced by overflow, the hue of our texture is often affected during the transfer. Figure 5.4 shows an example of the hue being shifted. Hue is affected most when a texture has areas of very different hue combined with high intensity and saturation levels. An example of this was found when using a texture of green grass partially covered by large light brown and orange leaves. When this texture was transferred to a darker area the net RGB changes were near zero, but at any given pixel the ratio of red to green to blue was different from both the up-sampled source and the texture. This ratio is what defines the hue. Therefore, in this experiment we found that the leaves had a slight twinge of pink instead of being purely brown and orange. This hue shifting can be avoided by working in HSV color space instead of RGB space. As is usually the case, the benefits gained by the HSV approach come with additional costs.

5.3.4 HSV Detail Transfer Approach

Our HSV approach to transferring detail preserves the hue found in the original texture sample at the cost of not being quite as accurate at preserving the local average color of the source image. The HSV approach is also computationally more expensive. In the HSV approach, a detail patch

does not contain RGB differences between the texture patch and the average color of the patch. Instead, each pixel in the detail patch contains:

1. The hue from the texture patch
2. The difference between the texture patch saturation at that pixel and the patch's average saturation
3. The difference between the texture patch intensity at that pixel and the patch's average intensity

Note that the hue is not stored as an offset from the average hue of the entire patch, but instead is simply the hue found at that location in the texture patch.

To combine the up-sampled source image with our HSV detail patch, we first convert the RGB value of our up-sampled source pixel to HSV space. We then add the saturation component to the saturation offset found in the HSV detail patch. Then, we do likewise for the intensity. However, for the hue, we use only the value from the HSV detail patch. Thus, the hue found in the source image does not have any effect on this part of the image enhancement. Lastly, we must convert this HSV value to RGB color space.

The HSV approach does succeed in eliminating the hue shifting seen using the RGB approach. Although it does increase the amount of color shifting away from the up-sampled source image, the error was still within a

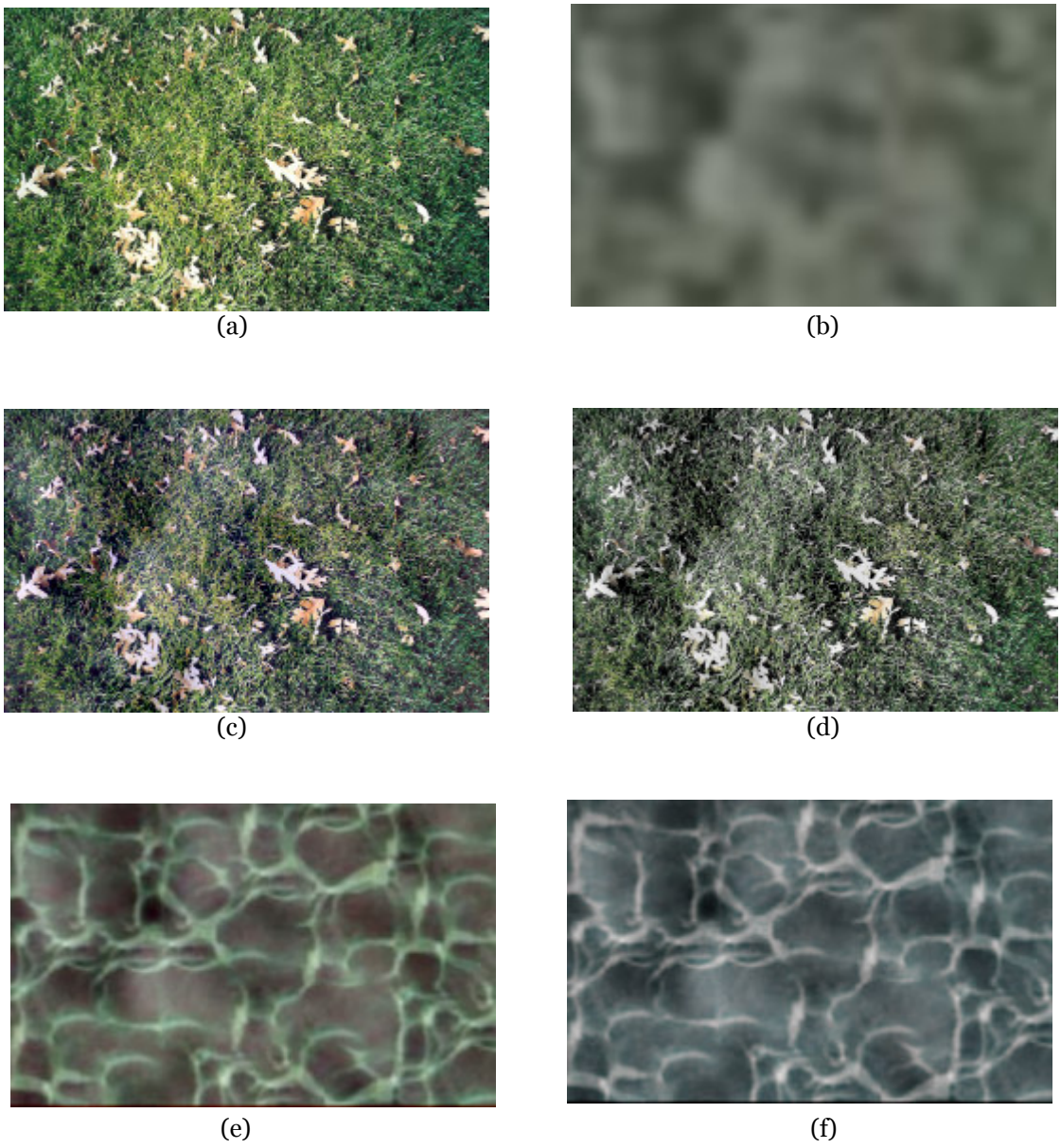
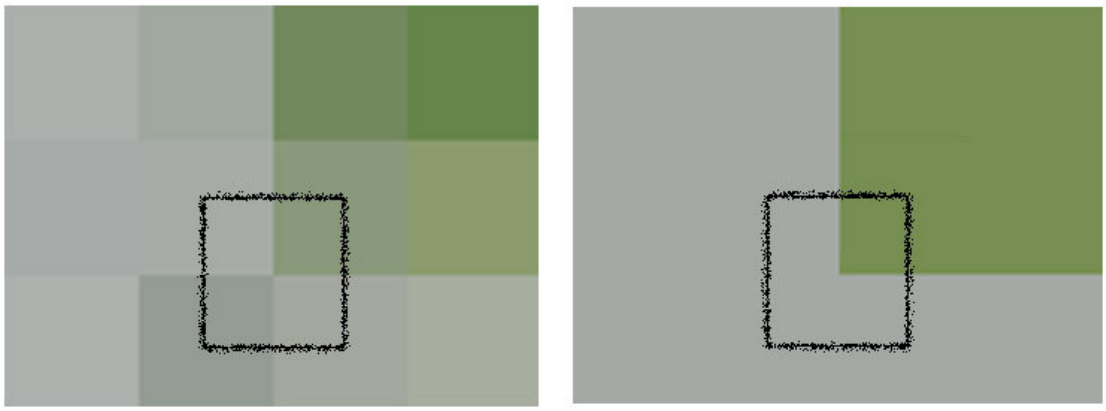


Figure 5.6: RGB vs. HSV Detail Transfer. Comparison of RGB detail transfer and the HSV alternative. (a) Sample texture. (b) Image lacking detail. (c) Combination of items a and b using RGB detail transfer. Notice the pink tint seen in the leaves. (d) Result created using HSV detail transfer. (e) Another example using RGB transfer. (f) An image created using the HSV approach.



(a) (b)

Figure 5.7: Custom Patch or Extracted Patch. (a) Part of a source image. Each patch will represent an area the size of one pixel in the source image. The area is shown by the black box. (b) The same area in the classified source image. Note that two texture classes are represented in this area. Thus, a custom patch will be needed for this area.

reasonable tolerance. Our limited testing shows a color shift of 1.8%. The biggest drawback of the HSV detail transfer approach is that it costs more computationally. Using the HSV approach increases the computation time of the enhancement algorithm by three to four times. Because of the high cost of the HSV approach, we have chosen to use the RGB detail transfer approach for all real-time image enhancement operations. Figure 5.6 shows a comparison of RGB and HSV detail transfer.

5.4 Custom Patch or Extracted Patch

Up to this point, we have not described the details of choosing an appropriate patch for each step in the pasting process. In this section, we will address that topic. Patches pasted into the image will be either a patch chosen from one of the sample sets or a custom-built patch to handle seams between areas of different texture classes. To decide whether the patch will come from a stored set or will be custom built, we reference the classified version of the source image. We analyze the four pixels in the source image corresponding to the area the patch will cover and count the number of unique texture classes represented by those four pixels. See Figure 5.7. There are two possibilities:

1. All four pixels are of the same texture class.
2. Two or more classes are represented in the four pixels.

If all four source-pixels are of the same class, we do not need to interpolate this patch. This means that the patch is not involved in a seam between texture classes. In this case, we select an extracted patch from the appropriate set that best matches the boundary areas above and to the left of the patch. We will discuss this in detail in the next section. If two or more texture classes are represented by the four pixels, then we must build a

custom patch to smooth the transition between areas of different texture classes. Custom patches will be discussed in Section 5.7.

5.5 Selecting a Patch from the Set

There are several methods for selecting an appropriate patch from a given set. In [EF02], Efros and Freeman use a brute force search. Their method compares the boundaries of every patch in the set against the known boundaries. As they do this, they create a qualifying set of patches that satisfy the boundary constraints within some specified error tolerance. Then they randomly select a patch from the qualifying set. The random selection is very important to image quality ensuring that replication and patterns are minimized. If no patches match the boundary areas within the defined error tolerance, the patch with the least error is used. Though the exhaustive search approach is very easy to implement and produces good results, it is much too slow for our needs.

In [LL01] Liang et al. use three significant performance accelerators to decrease the amount of time necessary in choosing a matching patch from a set. First, they build a kd-tree [LL01, Mou98] to organize the patches. This enables the use of an Approximate Nearest Neighbor (ANN) search to find the qualifying set of patches. The second enhancement is to first search a lower-resolution version of the data points creating a set of candidates.

The higher-resolution versions of these candidates are then searched to find the qualifying set. The final acceleration technique is to use principal component analysis (PCA) to reduce the number of data points for the query vector. These three steps produce a qualifying set. One patch is randomly chosen from this set to be pasted into the destination image. The three improvements reduced the synthesis time by 99%.

Another approach we investigated is to pre-compute all possible searches and store this information in a lookup table. While this would be the quickest at runtime, it is unrealistic for most patch-based texture synthesis algorithms because of the size of the patch set. Using traditional patch-based techniques, a look-up table for a single sample of 64x64 pixels requires approximately eight megabytes of memory. Though this may be reasonable for some applications, the table size grows exponentially. A 256x256 sample requires approximately 20 gigabytes of memory. This is clearly not reasonable.

Fortunately, we are able to build optimized patch sets using Wang Tiles as discussed earlier. This enables us to build small patch sets even when the input sample textures are large. Because of the small number of patches contained in each set, we can build look-up tables without using large amounts of memory. For each patch set, we build one table of dimension n by n (where n is the number of patches in the set) and two tables n by one. Remember we should have a patch set for every texture class in our source image. In the first table, we store a pointer to the patch

that should be used for each top left boundary condition (patches above and to the left of the area in question). The entry found at table[top,left] will be a list of patches that can be pasted seamlessly next to its neighbors. The two smaller tables are used for the first row and first column when we have only a patch above or to the left of the location in question, but not both. All three of these tables are built when the patch set is loaded. They are created using information stored in the patch set files about the patch edge color-coding. Using these lookup tables, RETS can have instant access to the correct patches for any boundary condition without having to perform a costly search.

5.6 Custom Patches using Interpolation

If the four pixels spoken of in Section 5.4 represent two or more texture classes, the corresponding patch is involved in a seam between areas of different texture classes. To reduce discontinuities along seams, we construct custom patches. Figure 5.8 shows an image enhanced without custom patches and a second image enhanced using custom patches. The second image looks significantly better. Figure 5.9 shows an example in which a custom patch is required.

In preparation to construct this patch, we select one patch from the appropriate set for each of the classes that will be involved in this patch.

Figure 5.8: Texture Class Interpolation. The top image demonstrates the problem of seams between areas of different texture classes. The discontinuities can be reduced by creating custom patches using bilinear interpolation.

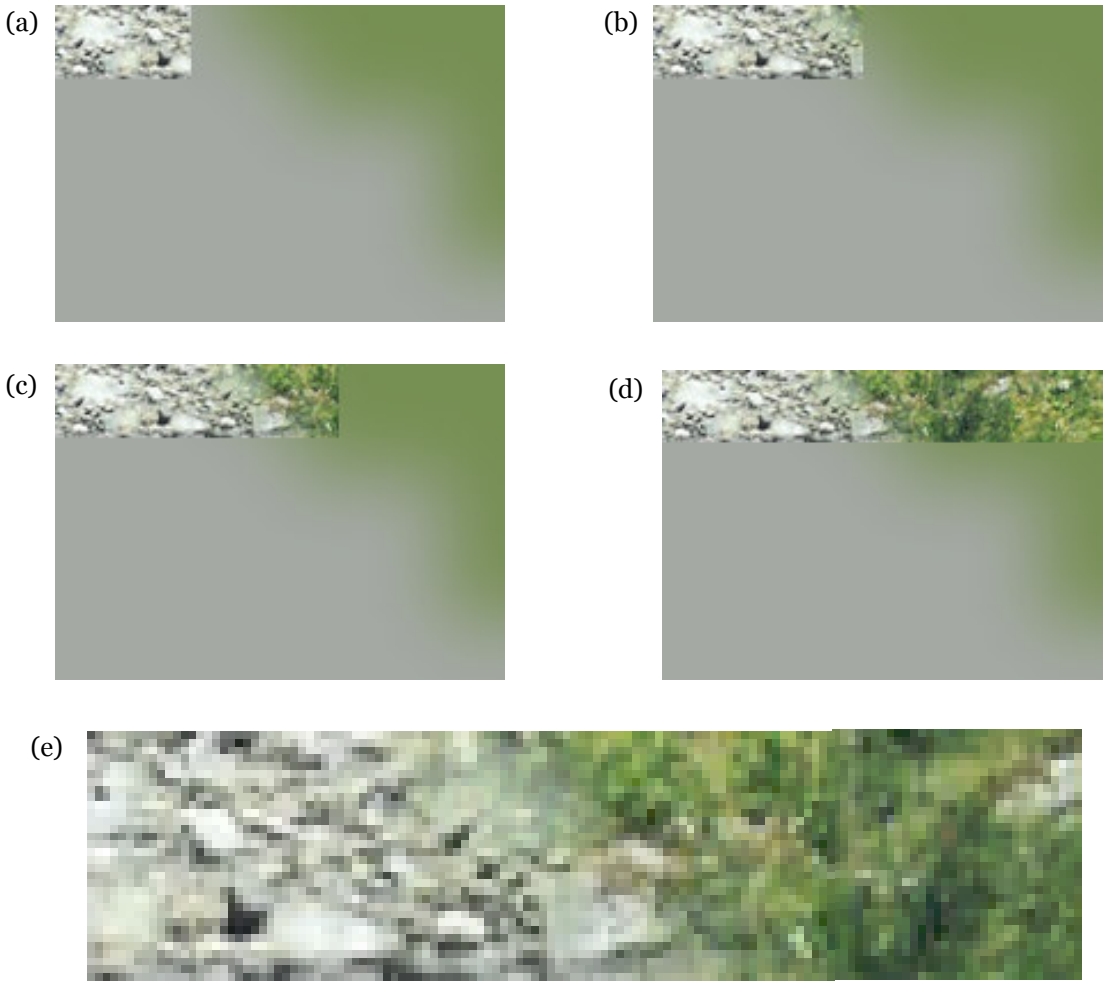


Figure 5.9: Custom Patches. (a) The first two patches are purely rock and thus are selected from a set. (b) The third patch is mostly rock, but the top right corner is green and thus should be grass. For this patch, we use bilinear interpolation to transition from rock to grass. (c) The fourth patch is mostly grass with a little rock. (e) Enlarged version of the four patches. Seams are not noticeable.

Thus, we will select between two and four patches. In selecting each of the patches, we may or may not need to match the boundaries of the top and left patches currently in place. In order to determine whether the patch must match the top and/or left patches currently in place, the following criteria are used:

1. Do either of the left two pixels have the same texture class as the patch being selected? If so, this patch must match the patch to the left.
2. Do either of the top two pixels have the same texture class as the patch being selected? If so, this patch must match the patch above it.

Once we have determined the matching requirement for this patch, we can then pick the patches we will use to create the custom patch. After the patches have been selected, a custom patch is produced by blending the chosen patches extracted from our sample texture. To smooth seams and reduce aliasing we use bicubic interpolation. For each pixel in the custom patch, all of the selected patches contribute to the pixel according to a bilinear blending function. The color of each pixel is:

$$customPatch(x', y') = \sum_{i=0}^{1}\sum_{j=0}^{1}[w_{ij} \bullet patch_{ij}(x, y)]$$

$$w_{ij} = R(i - dx) \bullet R(dy - j)$$

$$R(x) = \begin{Bmatrix} -1 \le x \le 0 \rightarrow x + 1 \\ 0 \le x \le 1 \rightarrow 1 - x \end{Bmatrix}$$

This interpolation produces smooth transitions that blend patches of different classes. After the patch is created, it can be converted to a detail patch and pasted into the destination image just as those found in the patch sets extracted from the original sample textures. Figure 5.9 shows an example of using custom patches to smooth transitions between texture classes.

5.7 Multi-resolution Support

The sixth requirement for our solution as defined in Chapter Three is that it must have multi-resolution support. By this, we mean that our enhancement process should be able to synthesize enhanced versions of a source image at different resolutions. All enhanced versions of a source image should look alike, albeit with different sizes and levels of detail. A simple but unacceptable approach is to synthesize the highest supported level of magnification and then down-sample to the various requested

resolutions. While this would solve the problem, it is unacceptable from the perspective of performance. If we request to enhance a 256x256 image to 512x512, we would not want to synthesize an image 32,768 pixels wide and then down sample the result to 512x512.

Fortunately, our synthesis technique lends itself to multi-resolution support. To accomplish this, for each patch, we have access to scaled versions of each patch at sizes from 2x2 to 256x256. Based on the requested level of magnification, we choose the appropriate version of each patch. Since each patch will replace an area the size of a pixel in the source image, using the 2x2 version of the patch will create an enhanced image twice the width and height of the source image. Using a patch size of 256x256 will create an enhanced image 256 times wider and taller than the source image. This approach allows multi-resolution enhancement without performance degradation.

Chapter 6

EView

In this chapter we will introduce EView, an application using the image enhancement approach presented in this thesis. We will then discuss three special problems encountered in integrating the enhancement algorithm into EView: massive image size, remembering previous results, and filling holes.

6.1 Introduction to EView

As part of this thesis, we have integrated RETS into a real-time virtual environment application named EView. EView, developed at Brigham Young University, automatically combines elevation data and aerial imagery to produce massive virtual environments covering hundreds of square miles.

Originally designed to allow users to fly around the environment, it has since been enhanced to allow users to bike or hike along mountain trails.

As stated in the practical example found in the introduction of this thesis, the views produced by EView look very good when the viewpoint is far from the surface of the terrain. However, when the viewpoint is moved near the surface, as in the hiking or biking usage of EView, the terrain becomes much less attractive. As can be seen in Figure 7.4, the low-resolution aerial imagery creates a blocky terrain, which lacks detail.

6.2 The Size Problem

Adding image enhancement to EView presents several complications. The complications are caused by the sheer size of the images used by EView. EView essentially has one large continuous image that represents the terrain of the virtual environment. Normally the size of this image ranges from 600 MB to multiple GB. If we enhanced this entire image with a magnification of 128, the resulting image would require approximate 16,384 GB of storage space.

Obviously, enhancing all of the terrain at once is not a realistic approach. Instead, EView requests enhancement of small pieces of the low-resolution aerial imagery as the viewpoint moves near those areas. Often, EView will request that the same area be enhanced several times. First, it

may request that a subsection of the terrain be enhanced with a magnification of two. Later, when the user's viewpoint is moved closer to that subsection, an enhancement with a magnification of eight may be requested. When the viewpoint is moved to within a few feet of an area, the area could be enhanced by a magnification factor of 128.

While the approach of synthesizing only areas of the source image requested by EView does alleviate memory constraint problems, it does present complications. First, the synthesis process must remember areas of the source image that have already been synthesized. Then when a higher-resolution version is requested, the synthesis process must create a new image like the previously synthesized version, but now at a higher level of magnification. This is accomplished using the multi-resolution support discussed in Section 5.7 coupled with a hierarchal map to store information about areas that have been enhanced.

6.3 Remembering Previous Results

As stated in the previous paragraph, we must remember areas previously enhanced so that we can duplicate those results later. Because of memory constraints, we cannot simply store all enhanced images. However, we can store all information about an enhanced area required to duplicate

previous results. To do this we build a hierarchal map. In a two-level map, the first level is a two-dimensional array that points to blocks. Each block, only allocated when first needed, contains a 2048 by 2048 array of entries. Each entry corresponds to an area the size of one pixel in the source image. Thus each block represents an area 2048 pixels by 2048 pixels in the source image. Each entry stores the patch to be used in enhancing the corresponding area of the source image. The patch information and the color values from the source area being enhanced are the only information needed to synthesize a high-resolution result.

6.4 Filling Holes

A second complication created by enhancing small areas in non-raster scan order is the creation of holes. If EView requests enhancements for areas near each other but not bordering one another and later requests an enhancement for the hole between the previously enhanced areas, difficulties arise. Figure 6.1 shows an example of a hole.

The difficulty caused by holes stems from the Wang Tile approach to texture synthesis. Although the Wang Tile approach is much faster than any other approach, it does not support filling holes without discontinuities. Most other pixel-based and patch-based techniques do support filling holes.

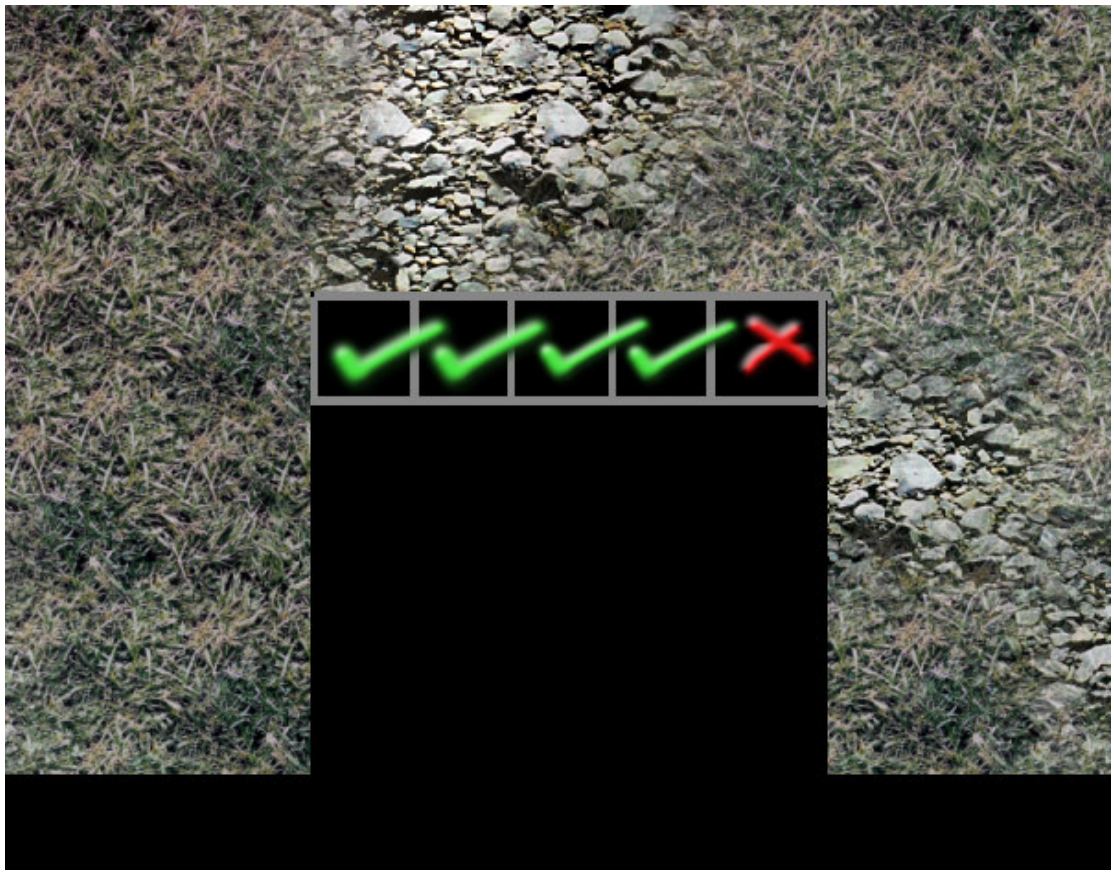


Figure 6.1: Filling Holes. In this example, EView has requested areas to be synthesized in a sequence leaving a hole, which has not been enhanced. Enhancing the remaining area is problematic. The process can begin in raster scan order as normal. The first four patches are OK. Then we get the patch marked with the red x. No patch exists in the available patch set to match all three boundary conditions.

Using the Wang Tile approach, we can match boundaries above and to the left of the area seamlessly. However, when we get to the point where we have boundary conditions below or to the right of our patches, we cannot match them. This is because we built the patch sets to have at least two

matches for all top-left combinations. However, our patches will not match all top-left-bottom-right combinations. Technically we could build a patch set that would match all of these combinations, but at the expense of very large patch sets. Since the reason we use the Wang Tile approach is for small patch sets, this is counterproductive.

Although the Wang Tile approach to texture synthesis cannot fill holes well, we can drastically reduce how often these holes occur. To do this we use the hierarchal map discussed previously. When EView requests an area of the imagery enhanced, we find the block the area is contained in. We then map out what patches will be used to replace each pixel from the source image for the entire block. We do not actually synthesize the resulting high-resolution image for the whole block, but we map out exactly what the image will look like. Using this approach, holes can only occur on block boundaries. Since mapping out a block is cost-effective in terms of CPU time and relatively cost-effective in terms of memory usage, we map out blocks above and to the left of the current block. This further reduces the occurrence of holes.

Now that holes rarely occur, there are two options in dealing with them. The first is to use our Wang Tile based synthesis technique to fill the holes and accept the small discontinuity along the bottom and right of the result. For many applications, the discontinuities are barely noticeable even when looking for them. These discontinuities can be further reduced by blending the edges of the patches along the seam that does not match.

The second option is to use traditional patch-based synthesis on the last row and last column of the hole. This requires a second large patch set for each texture class. This patch set can be created from the original texture sample by extracting all sub-images of a given size. We can then search this large patch set for patches that will fit nicely into the right column and bottom row. Because this search can be optimized and will occur rarely, it should have little negative impact on overall performance. However when this approach is used, areas of the image that require hole filling will take significantly longer than areas that don't require hole filling.

In our application of RETS used in EView, we have chosen not to use traditional patch-based synthesis for hole filling. Instead, we use the Wang Tile approach and accept the small discontinuities along the hole boundaries. The reasoning for our choice is specific to the EView application. EView loads and synthesizes terrain textures on a separate thread from the rest of the program. However, the current EView implementation does not allow multiple textures to be loaded or synthesized in parallel. Thus, if synthesizing one texture takes a long time, no other textures can be loaded or synthesized until the first is completed. This delay will likely cause unacceptable popping artifacts.

Chapter 7

Results

This chapter reviews the results of our implementation of RETS. The first section presents a performance analysis including both the performance strengths and weaknesses of our current implementation. The second section shows images from a stand-alone version of RETS and images rendered by EView using enhanced aerial imagery.

7.1 Performance

In approaching the image enhancement problem, we have focused on *real-time* large-scale magnification of aerial imagery. As can be seen in Figure 7.1, RETS can enhance images in real-time. At a magnification factor

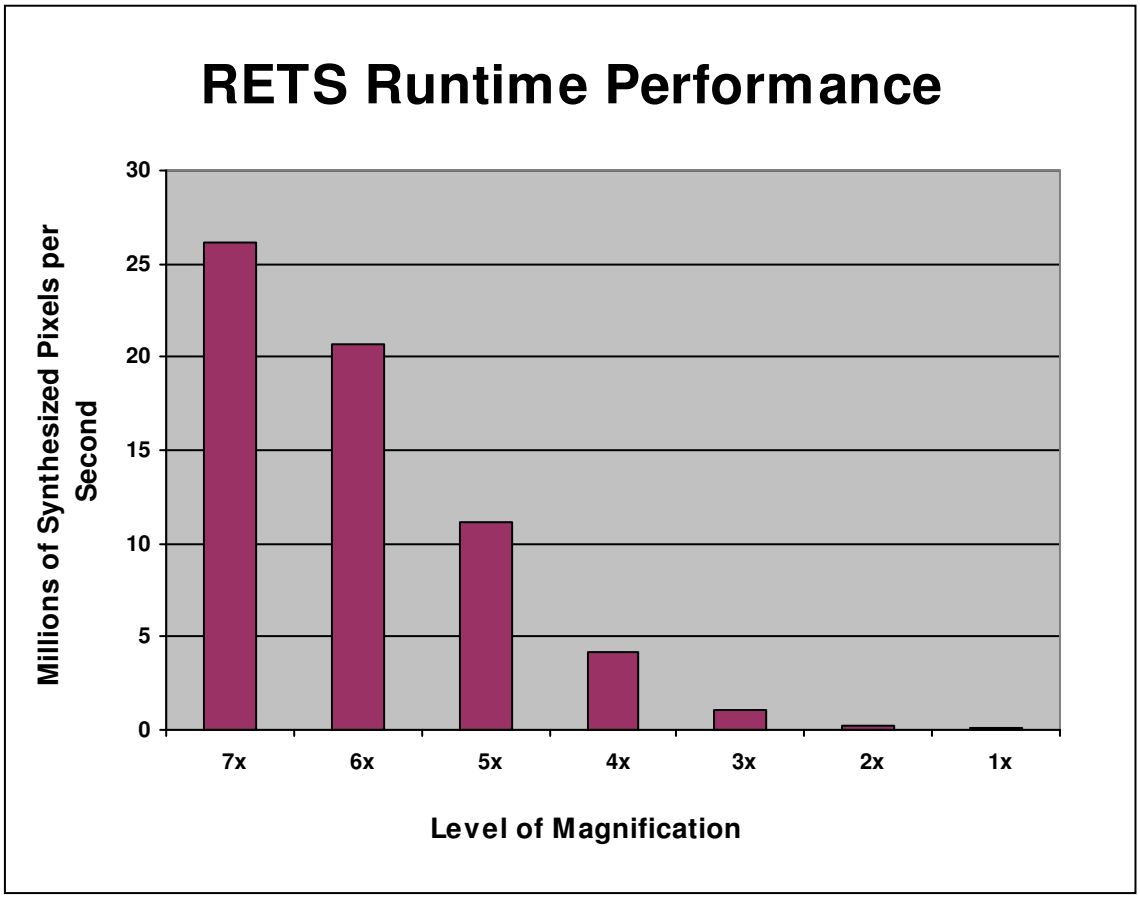


Figure 7.1: RETS Runtime Performance. This chart shows that RETS performs fastest at higher levels of magnification. At 7x magnification, RETS replaces every pixel in the source image with a patch with size 128x128. At this level RETS can synthesize 26 million pixels per second. However, at 2x magnification, RETS can synthesize less than 1 million pixels per second. Tests were done on a 3.0 GHz processor.

of 128 times the original width, RETS can produce 27 million pixels per second. The total expense of the enhancement algorithm is dependant upon two factors. The first is the size of the input image. The second is the

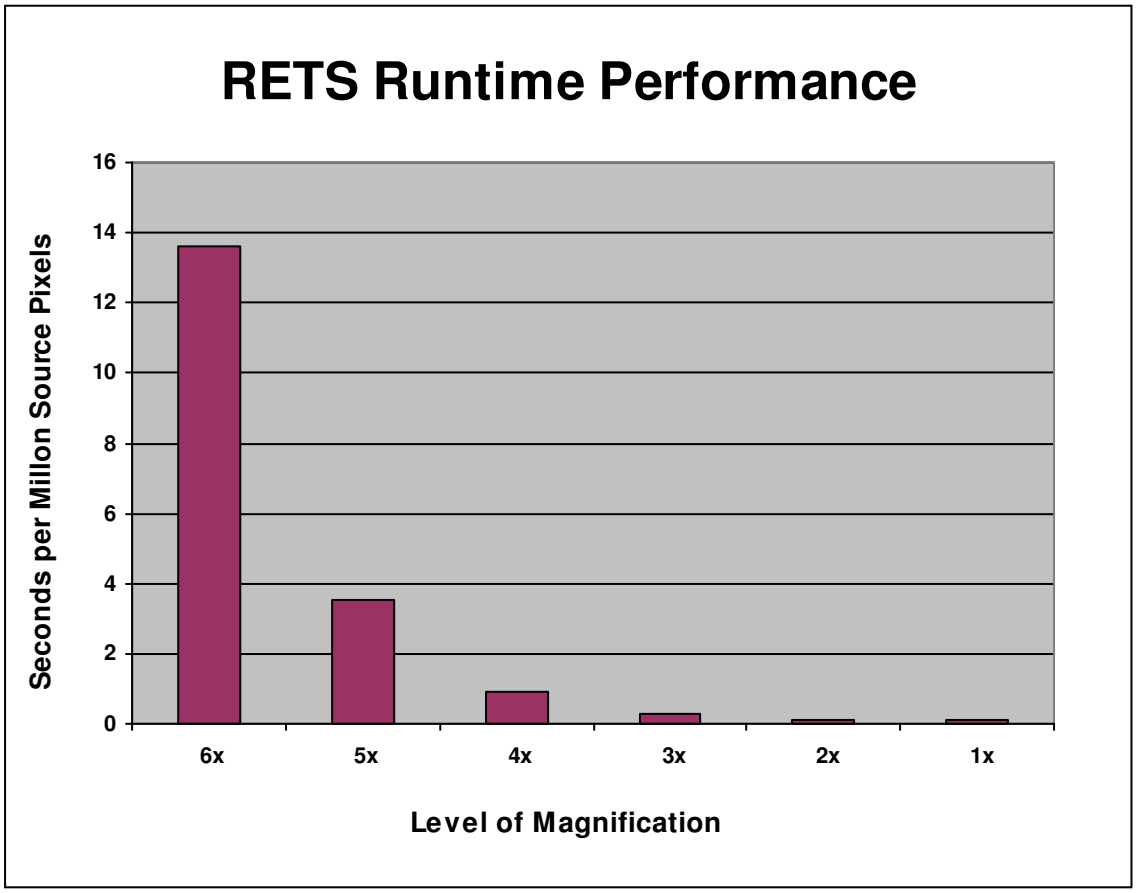


Figure 7.2: RETS Runtime Performance. This chart shows the time it takes to enhance one million pixels at different magnification factors. At 1x through 4x magnification, increasing the magnification level does not significantly increase the amount of time required to enhance an image. Enhancing at 2x magnification only costs 27% more than 1x magnification even though the result is 4 times larger. At magnification levels above 5x, the synthesis time goes up exponentially, but at a slower rate than the size of the enhanced image.

magnification factor. Because the cost of part of the algorithm is dependant upon the size of the input image, increasing the magnification factor has a greater impact upon the size of the resulting than it does upon the total

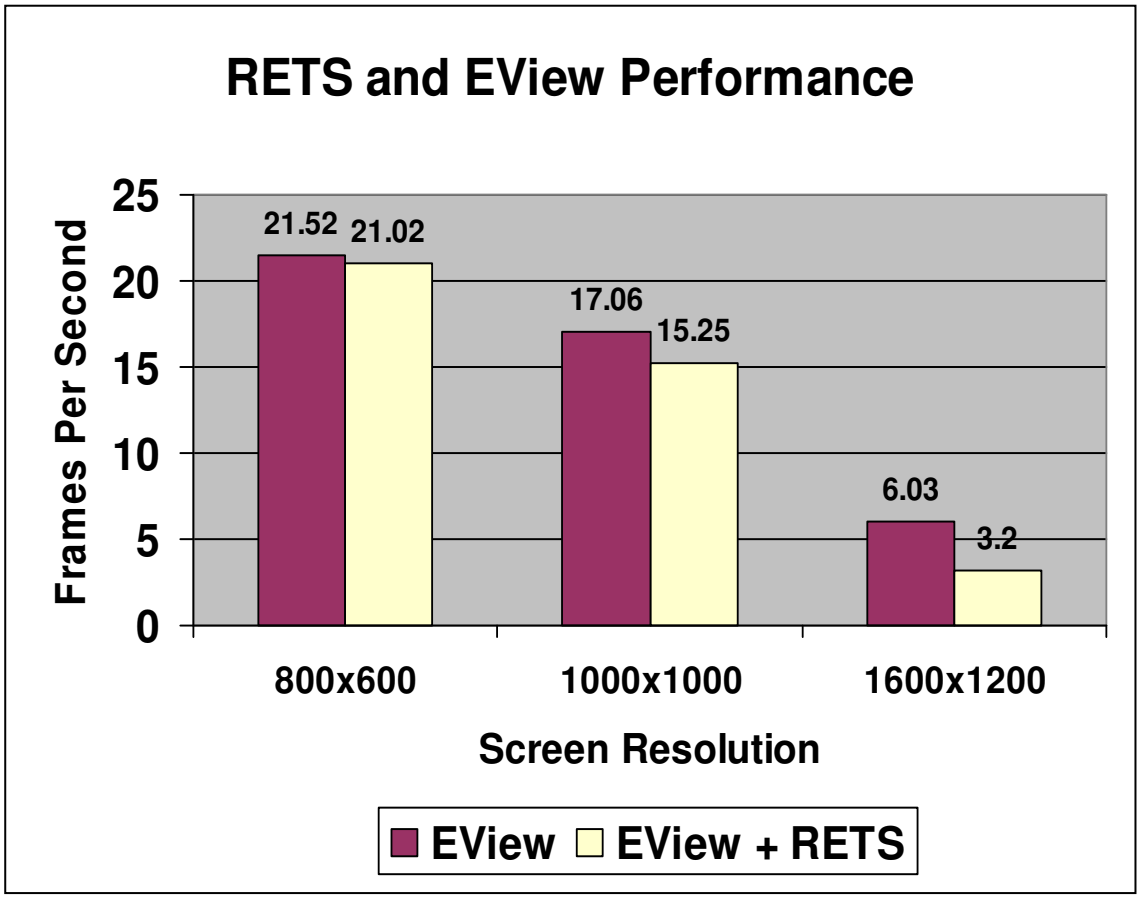


Figure 7.2: RETS and EView Performance. The chart shows the impact of adding RETS to a real-time virtual environment program. The red bars show EView's average frames per second without using any image enhancement. The yellow bars show that EView performs only slightly slower when RETS is added.

synthesis time. Thus, if two images are synthesized from the same source image, the first at 1x magnification and a second at 3x magnification, the second resulting image will not take significantly longer to synthesize even though the resulting image is significantly larger than the first synthesized.

As a result, the number of pixels produced per second increases as the level of magnification becomes larger.

To prove that using RETS in a real-time interactive application is feasible, RETS was added to EView. As can be seen in Figure 7.2, RETS had little negative impact on EView's performance for most display resolutions. At the default resolution (800x600), EView ran only 1% slower with RETS than without. At the highest resolution tested (1600x1200), EView performed at half the frame rate with RETS incorporated into it, as compared with running EView without RETS. While this seems to suggest that RETS drastically slowed down EView, profiling the application showed that the RETS process took up less that ten percent of the CPU time. The majority of the decrease in frame rate was caused by supplying the rendering process with 16,384 times more texture data.

Unlike other texture synthesis techniques, the computational cost of the Wang Tiles approach is not dependant upon the size of the input sample, the number of samples, or the size of the patch sets used for synthesis. Thus the only inputs that affect the runtime performance of RETS are the size of the source image and the magnification factor.

7.2 Image Quality

The quality of enhanced images produced by RETS is dependant on several parameters. These parameters include the quality and number of input sample textures, the number of patches in each set, and the time used to generate the patch set.

The first input affecting image quality is the quality and the number of input sample textures. As previously stated, the user should supply one sample texture for each texture class represented in the source image. If the user does not provide enough sample images, areas will be enhanced with the wrong texture.

In addition to providing an appropriate number of samples, the user must carefully choose the samples to supply. Sample textures that have small portions of the image that drastically stand out should rarely be used. All patch-based texture synthesis techniques use verbatim copies of the sample texture in the synthesized result. Areas that drastically stand out from the rest of the texture will create noticeable repetition in the synthesized result. While all patch-based techniques suffer from this ailment, the Wang Tile approach compounds the problem because of the small number of patches in each set. In image (b) of Figure 7.5, there is noticeable repetition found in the lower right corner. This is because one rock in the image stands out from the rest of the texture.

Second, the quality of the synthesized images is dependant on the size of the patch set. We derived our synthesis technique from the Wang Tile approach, because this allowed minimal patch set size. The small patch set size allows for real-time enhancement without consuming massive amounts of memory. However, the cost of small patch set size is repetition. For many textures, repetition is acceptable. Examples include wicker mats, tile floors, brick walls, and shingled roofs. For many other textures, exact repetition is not acceptable. Examples of these textures include rocks, dirt roads, bushes on a hill, and trees in a meadow. While it is acceptable for two bricks to look near identical, rocks are usually unique in size, shape, and color.

To minimize exact repetition the size of the patch set must be increased. In our tests, patch sets containing between 50 and 100 patches produce little or no noticeable repetition. Figure 7.5 shows two renderings of the same scene. The bottom image was enhanced with patch sets of size 32. Notice the visible repetition in the lower right corner. The top image shows the same scene enhanced with patch sets of size 50. Notice that the repetition is drastically reduced.

Third, the quality of the enhanced image is dependant upon the amount of time used to generate the patch sets from the sample textures. Section 4.3 detailed how patch sets are automatically created from sample textures. This process uses a costly brute force search to find patch sets that tile together with little disparity. The greater the amount of time this

process is given the higher the probability of finding higher quality patch sets. This means that there will be less disparity where two patches meet in the resulting synthesized image.

7.3 Synthesis Results

The goal of this research was to enhance low-resolution images; to make them look better; and to do this in real-time. In this, we have succeeded. Figure 7.3 shows three versions of an image containing grass and rocks. The top image is the original digital photograph. The middle image is a down-sampled version of the photograph. The down-sampled version is representative of what the area would look like from low-resolution aerial imagery. The bottom version is an enhanced version of the low-resolution image. While the bottom image does not look exactly like the top one or even quite as realistic as the top image, it looks much better and more realistic than the low-resolution image.

Figure 7.4 shows the problem RETS is targeted at solving. This figure contains two renderings of the same scene in EView. The top image has very noticeable aliasing due to the low-resolution imagery. In the bottom image, bilinear interpolation has been used to smooth the image. While this has reduced the aliasing, the scene still lacks details. Figures 7.5, 7.6, and 7.7 show the same scene as in Figure 7.4, but the aerial imagery was enhanced

with RETS. Notice how much more detail the scenes contain. These results show that RETS can be successfully used to enhance aerial imagery in real-time. Figure 7.8 shows results from RETS not rendered in a three dimensional application like EView. This figure shows both the source image and the result, so it can be seen that lighter areas of the source image have been enhanced as gray rock and darker areas of the source image have been enhanced to be a dark green grass.

(a)

(b)

(c)

Figure 7.3: Results. **(a)** Original digital photograph. **(b)** Down-sampled version of image (a). **(c)** enhanced version of image (b). Note that image (c) does not look exactly like the original, but looks much better than the input source (b).

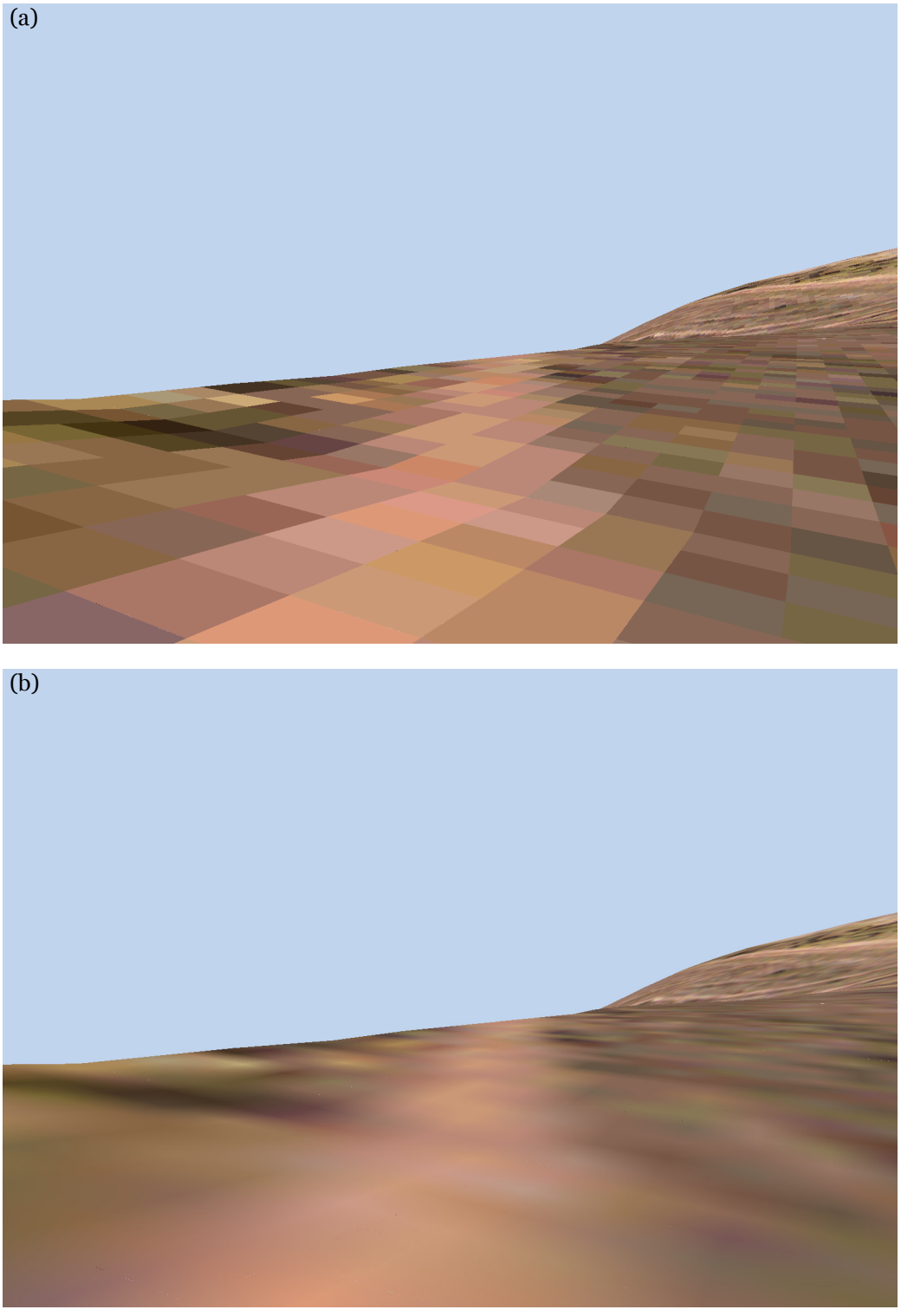
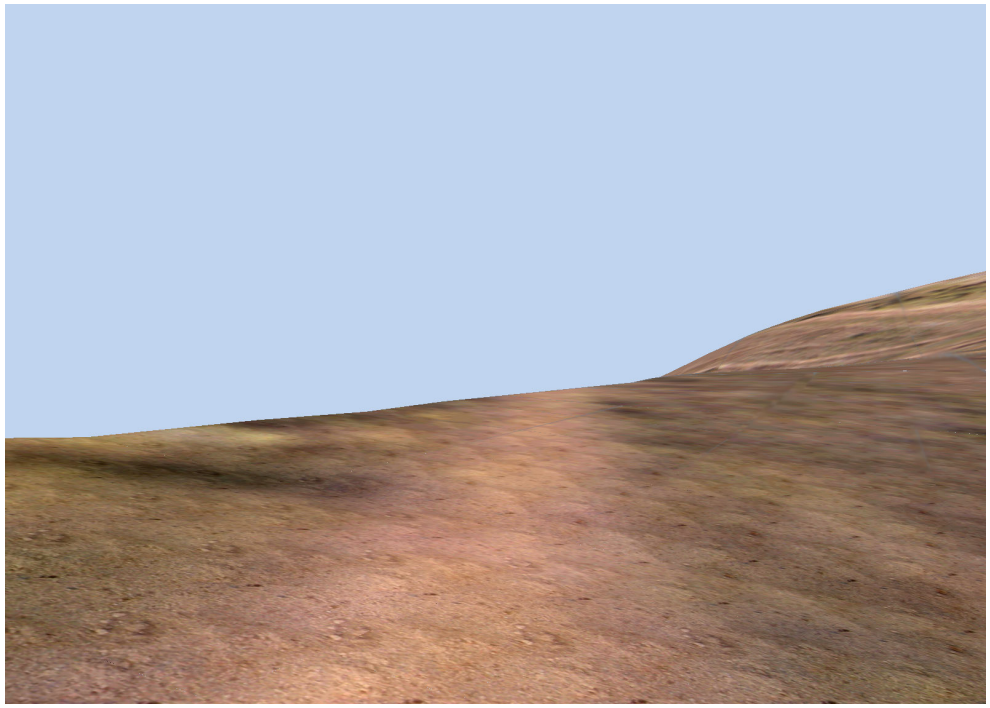


Figure 7.4: EView Problems. The top image shows a scene from EView rendered without image enhancement or image interpolation. The bottom image is the same scene rendered with bilinear interpolation.

(a)



(b)

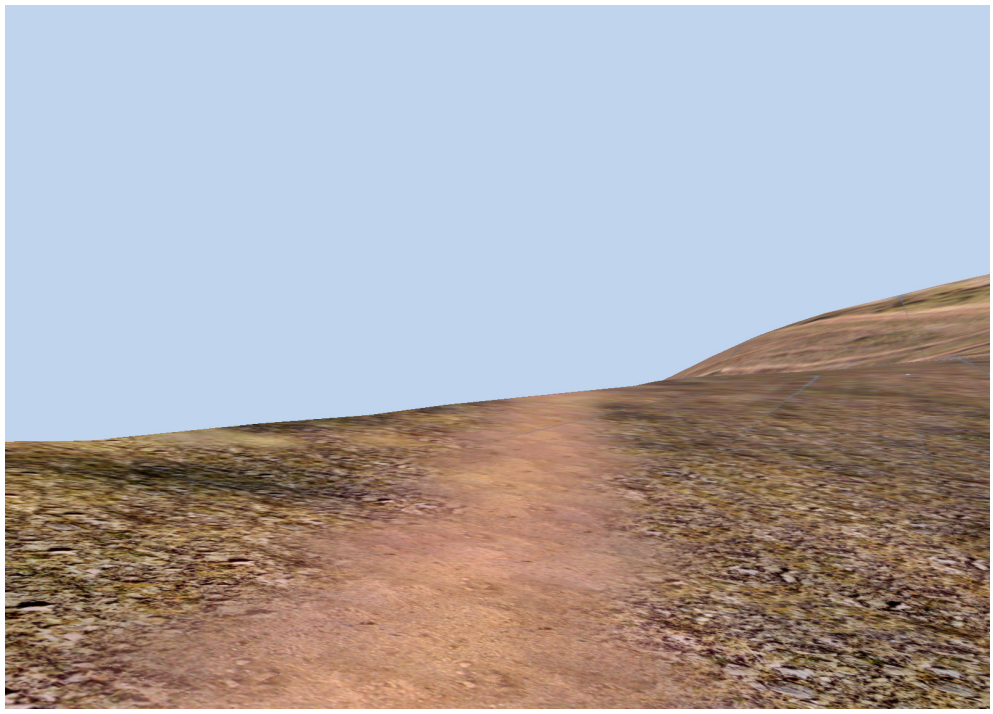
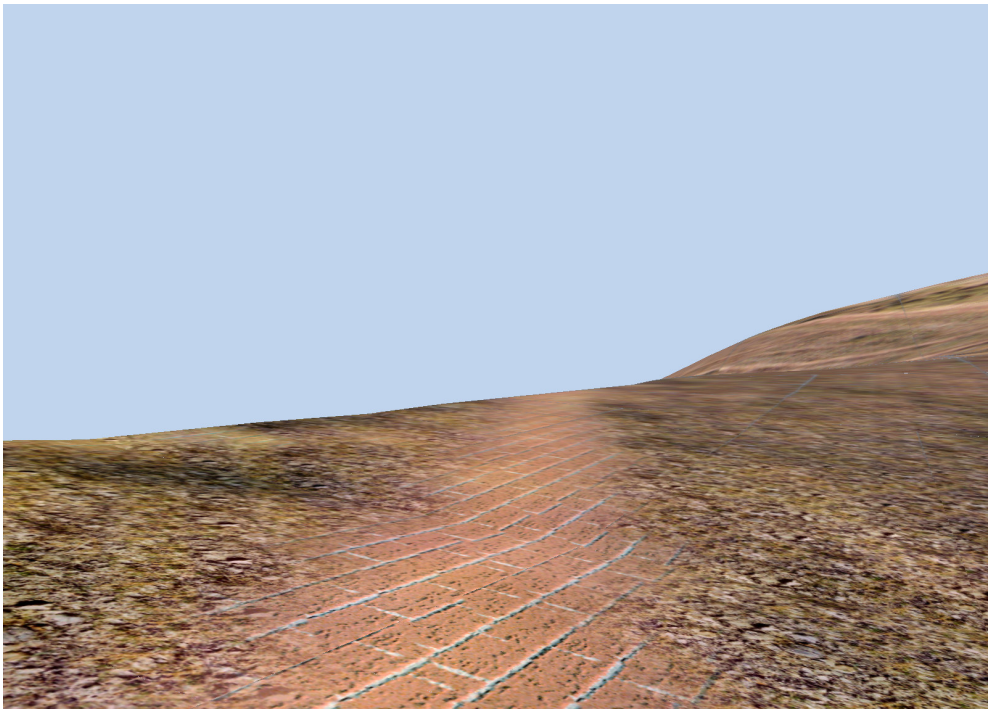


Figure 7.5: EView with RETS. Scenes rendered by EView using imagery enhanced by RETS. Image (a) was enhanced with a single texture sample. Notice that using only one sample texture, RETS synthesizes very light areas and dark areas. Image (b) was enhanced using two sample textures.

(a)



(b)

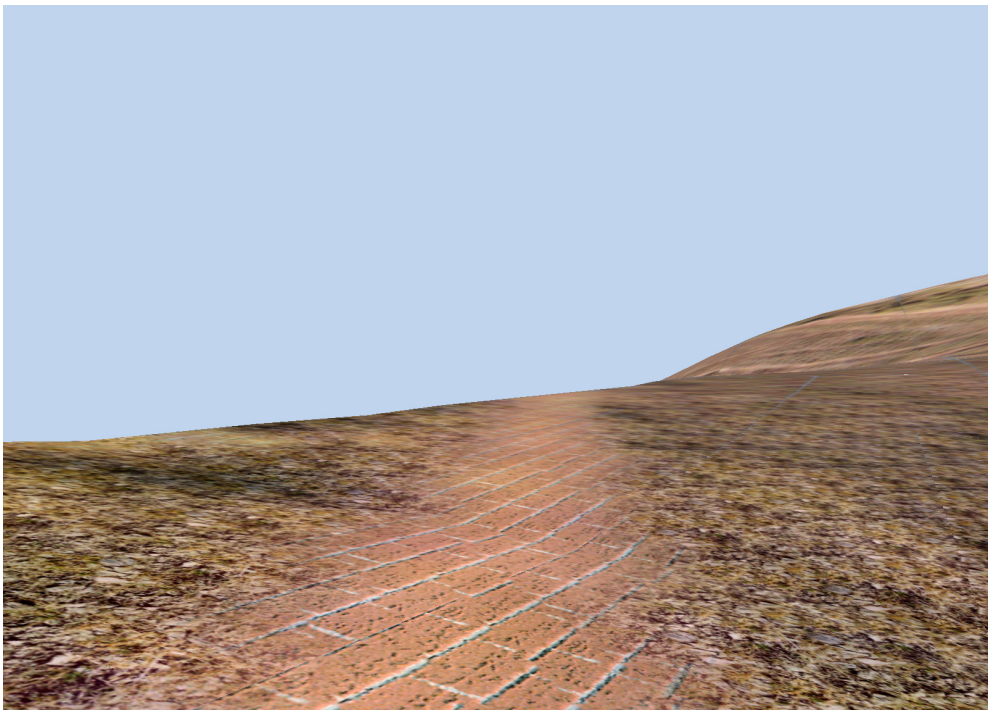
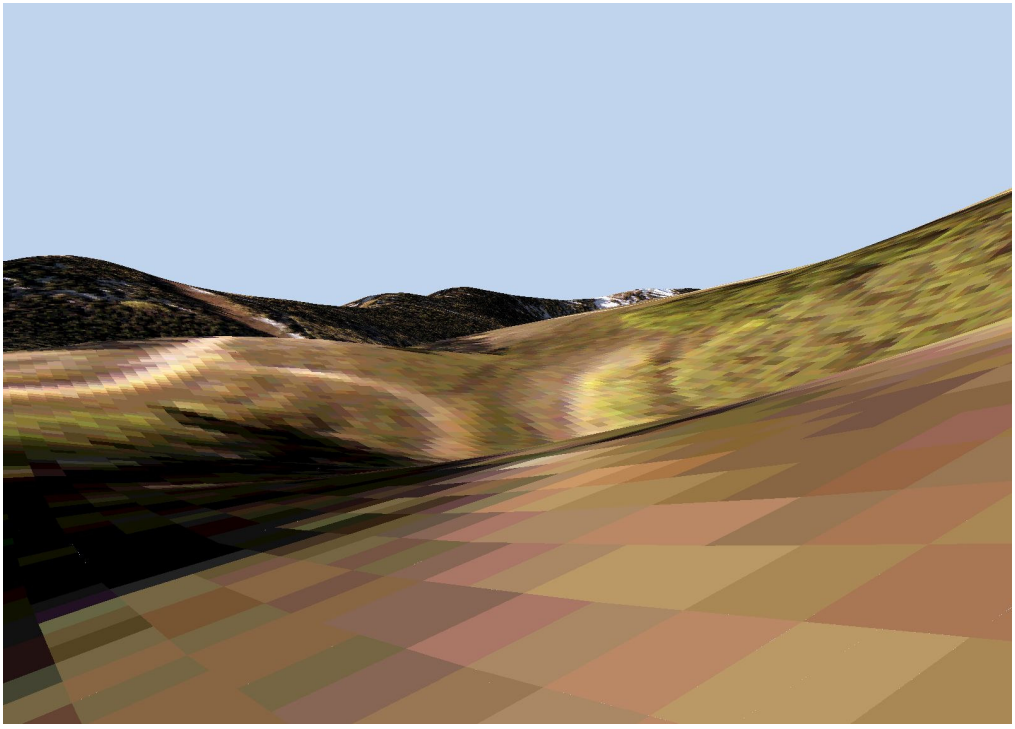


Figure 7.6: Repetition. (a) The same scene as in Figure 7.4, but RETS has been used to enhance the low-resolution imagery. (b) RETS has been used, but noticeable repetition is visible in the lower right quadrant. The repetition is caused by using a patch set of insufficient size. While the portion of the image with bricks also suffers from repetition, it is acceptable for this type of texture.

(a)



(b)

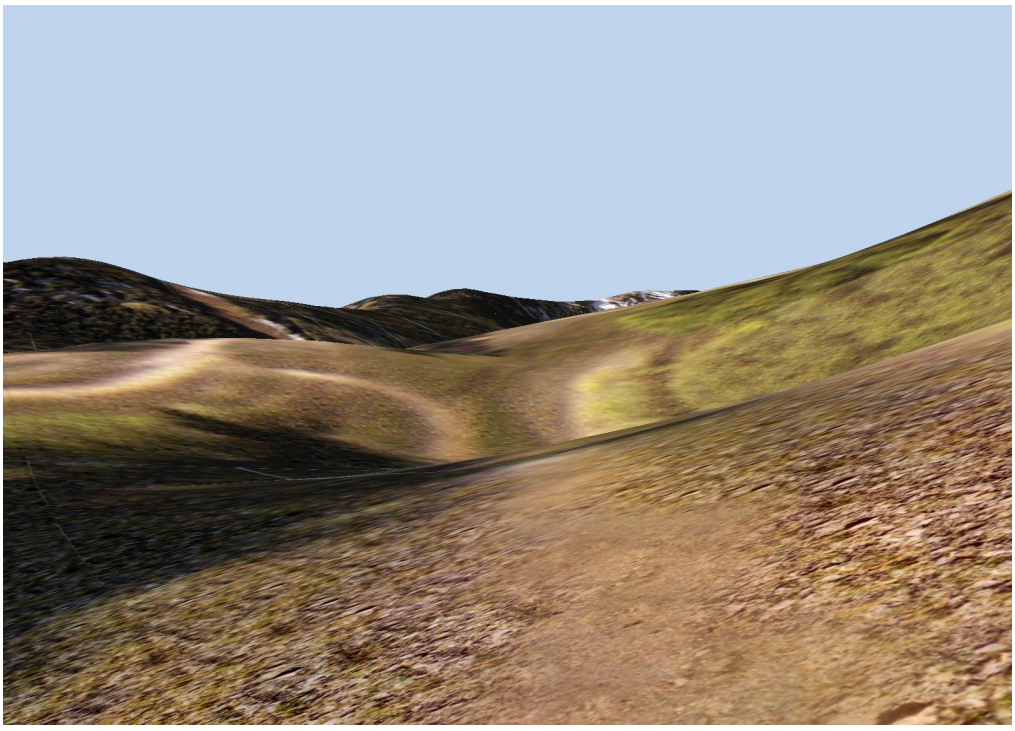


Figure 7.7: Comparison. Image (a) was rendered by EView without enhancement. Image (b) was rendered by EView using RETS enhanced imagery.

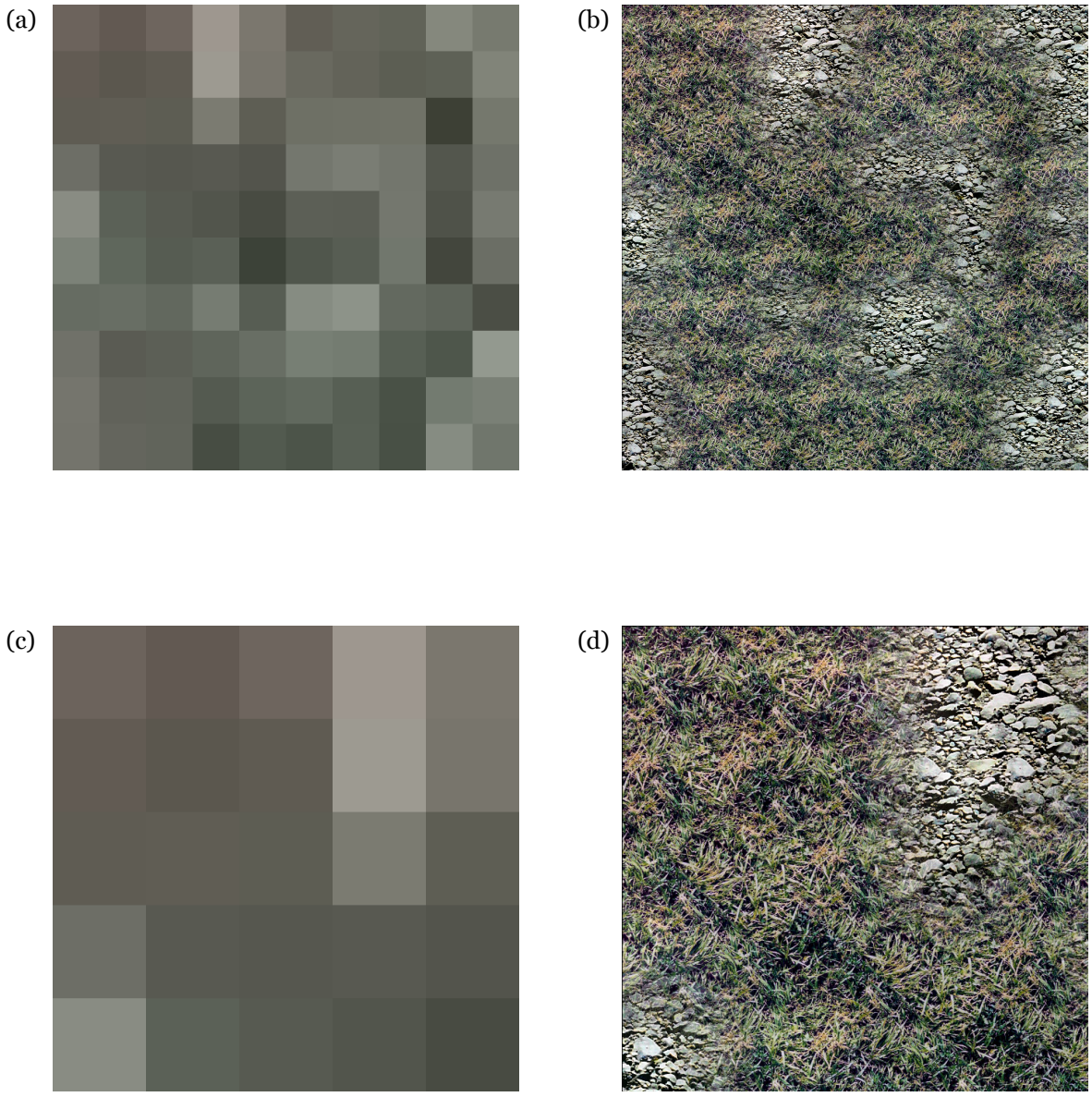


Figure 7.8: Enhancement Examples. Images (a) and (c) are pieces of an aerial photo. Images (b) and (d) are pieces of the high-resolution version of the aerial photos which have been enhanced using RETS. Notice that where the source image appears light in color, the corresponding high-resolution image has gray rock. Darker areas of the source image have been enhanced to a dark green grass.

Chapter 8

Conclusion

This thesis has presented a combination of classic interpolation and patch-based texture synthesis to enhance low-resolution images while they are being enlarged. We successfully designed and implemented a solution to the real-time image enhancement problem.

RETS fulfilled each goal as specified in Chapter Three. RETS does synthesize a high-resolution version of a low-resolution input image. Detail is inserted in the high-resolution version consistent with sample textures. As proven by integrating RETS into EView, RETS can be accomplished in real-time. Furthermore, RETS is fast enough to only take up a small portion of the CPU while EView is running. EView uses RETS to create high-resolution images for use with MIP-mapping. No noticeable visual popping artifacts are caused when transitioning from the original image to a

synthesized image. Lastly, RETS supports image enhancement at multiple resolutions.

The original approach presented in this thesis addresses several problems not solved in other papers. The following are new and original techniques used by RETS:

- RETS combines patch-based texture synthesis with image interpolation.
- RETS enlarges images locally similar to supplied sample images in real-time.
- The algorithm presented here also handles multiple texture classes appearing in the same image.

8.1 Future Work

Although the goals we set when this research began have been met, there is much room for future research. One area of particular interest is real-time enhancement using *oriented* texture synthesis. The texture synthesis approach we use, like the algorithms it is based on, only orients the texture as supplied in the input sample. Therefore, if a texture sample shows waves moving from the top of the texture to the bottom, the synthesis results will always contain waves moving top to bottom. However, in the case of a coastline, waves should be perpendicular with the coastline

regardless of the orientation of the waves in the sample texture. Oriented texture synthesis for use in image enhancement is a promising area for future research.

Coupled with oriented texture synthesis is the need for a classification algorithm that not only classifies aerial imagery but also assigns each location an orientation vector. For example, a good classification algorithm would not only identify certain pixels as belonging to a river but also classify directions for each pixel. The same concept applies for roads, lakes, and other items that have a direction component.

A third area of interest is that of oriented motion synthesis. Whereas RETS synthesizes static textures, there is a need for motion synthesis. Examples of motion synthesis are: waves moving towards shore, a stream flowing, grass blowing in the wind, etc.

Bibliography

[Ash01] M. Ashikhmin. Synthesizing Natural Textures. In *Symposium On Interactive 3D Graphics 2001*, p.217-226, March 2001.

[CSHD03] M. Cohen, J. Shade, S. Hiller, O. Deussen. Wang Tiles for Image and Texture Generation. In *Proceedings of SIGGRAPH 2003*, p.287-294, August 2003.

[Bou01] P. Bourke. Bicubic Interpolation for Image Scaling. http://astronomy.swin.edu.au/~pbourke/colour/bicubic/, May 2001.

[Dod97] N. A. Dodgson. Quadratic Interpolation for Image Resampling. *IEEE Transactions on Image Processing*, p.1322-1326, September 1997.

[EF02] A. A. Efros and W. T. Freeman. Image Quilting for Texture Synthesis and Transfer. In *Proceedings of SIGGRAPH 2001*, p.341-346, August 2002.

[Bon97] J. S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proceedings of SIGGRAPH 1997*, p.61–368, August 1997.

[EF99] A. A. Efros and T. K. Leung. Texture Synthesis by Non-Parametric Sampling. In *Proceedings of International Conference on Computer Vision*, p.1033-1038, September 1999.

[H2001] A. Hertzmann et al., Image Analogies, In *Proceedings of SIGGRAPH 2001*, p.327-340, August 2001.

[HB95] D. J. Heeger and J. R. Bergen. Pyramid-Based Texture Analysis/Synthesis. In *Computer Graphics Proceedings, Annual Conference Series*, p.229–238, July 1995.

[IBG03] R. Ismert, K. Bala, and D. Greenberg. Detail Synthesis for Image-based texturing. *Interactive 3D Graphics*, p.171-175, 2003.

[LL01] L. Liang, C. Liu, Y. Xu, B. Guo, and H. Shum. Real-Time Texture Synthesis By Patch-Based Sampling. *Technical Report at Microsoft Research*, March 2001.

[Mak96] M. Makivic. Bicubic Interpolation. www.npac.syr.edu/projects/nasa/MILOJE/final/node36.html, July 1996.

[Mou98] D. M. Mount. *ANN Programming Manual*. University of Maryland Computer Science Department, 1998.

[PFH00] E. Praun, A. Finkelstein, and H. Hoppe. Lapped Texture. In *Computer Graphics Proceedings, Annual Conference Series*, p.465–470, July 2000.

[PKT83] J. Parker, R. Kenyon, and D. Troxel. Comparison of interpolation methods for image resampling. *IEEE Transactions on Medical Imaging*. Vol. 2, p.258-272, March 1983.

[PTS99] S. Premoze, W. B. Thompson, and P. Shirley. Geospecific rendering of alpine terrain. In *Eurographics Workshop on Rendering*, p.107-118, 1999.

[SS97] R. Szeliski and H.-Y. Shum. Creating full view panoramic mosaics and environment maps. In *Proceedings of SIGGRAPH 97*, p.251–258, August 1997.

[SZTS03] J. Sun, N. Zheng, H. Tao, and H. Shum. Image Hallucination with Primal Sketch Priors. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, p.729-736, 2003.

[WL00] L. Y. Wei and M. Levoy. Fast Texture Synthesis Using Tree-Structured Vector Quantization. In *Computer Graphics Proceedings, Annual Conference Series*, p.479–488, July 2000.

[WL02] L. Y. Wei and M. Levoy. Order-Independent Texture Synthesis. *Standford Computer Science Department TR-2002-01*, January 2002.

[WL01] L. Y. Wei and M. Levoy. Texture Synthesis over Arbitray Manifold Surfaces. In *Proceedings of International Conference on Computer Vision* p.355–360, August 2001.

[XGS00] Y. Q. Xu, B. Guo, and H. Y. Shum. Chaos Mosaic: Fast and Memory Efficient Texture Synthesis. In *Microsoft Research Technical Report MSR-TR-2000-32*, April 2000.