An Implementation of the Efros and Freeman Image Quilting Algorithm

Computer Science 766 Fall 2004 Final Project

Eric Robinson

Abstract

The focus of this project is an implementation of the Efros and Freeman "Image Quilting" Algorithm. The algorithm is designed to solve the texture synthesis problem in a unique way. It proposes to tile together patches of an input image in order to produce the output image. The novelty of the algorithm is in allowing neighboring patches to meet at edges that are not simply straight lines. I will specifically discuss the parameters that control the behavior of the algorithm and the experimental results that I obtained when implementing according to the original paper. After this I discuss some of the common problems that appear when applying the algorithm to structured and semi-structured textures, my thoughts on why they appear, and my proposals for how to compensate for them. I then discuss the experimental results obtained from implementing some extensions with specific emphasis on how effective the extensions were at achieving their intended goals. Finally I offer some concluding remarks on the algorithm itself, as well as on texture synthesis in general.

Introduction

In recent years advances in the technology and techniques for producing high-quality synthesized images has only been outstripped by the demand for such images. From special effects in films on the big screen to realistic virtual worlds in video games on the small screen, consumers are continually demanding more synthesized images of increasingly better quality. The additional trend of the proliferation of computing, and more specifically imaging, capabilities in places undreamed of even a decade ago (e.g. video games on cell phones) indicates that such demand will not abate any time soon. One technique researchers in computer graphics and computer vision have employed to meet this demand is texture mapping. In contrast to a three-dimensional reconstruction approach, a texture mapping approach generates a novel view from a smaller sample of the "real-world" and makes this view more realistic by mapping the appropriate visual texture onto the scene objects. Possibly the biggest advantage of this approach is that it is generally much faster and requires a smaller input dataset. To be successful, however, requires the availability of arbitrarily large patches of visual texture. This, in turn, drives the need for better, and faster, techniques for synthesizing large texture patches from smaller ones[1].

In this project I implement a relatively recently developed technique for solving the texture synthesis problem. The technique that I implemented is commonly known as "image quilting" and was first proposed in the SIGGRAPH '01 conference paper, "Image

Quilting for Texture Synthesis and Transfer" by A.A. Efros and W.T. Freeman. The algorithm, some extensions, and experimental results are described in greater detail in the rest of the paper. The main idea of the algorithm is to compose a large patch of texture by "stitching together" arbitrarily shaped smaller patched from the input texture – hence the term image quilting. Because I implemented an existing algorithm from a conference paper, this project report has a predictably strong reliance on the original paper and its cited sources.

Related Work

In [1], Efros and Freeman cite the long history of the interest in texture and how it is processed by the human visual system. Based on some of this research, in 1995 Heeger and Bergen introduced a statistical, histogram-based, approach to modeling and generating arbitrarily large patches of a stochastic texture from a given sample input texture [2]. Heeger and Bergen's work served as the basis for many subsequent global statistical pattern focused texture synthesis algorithms. While successful for stochastic textures, these approaches have foreseeable difficulties in modeling structured and semi-structured textures[1]. As a response to the local neighborhood problems suffered by the global statistical algorithms, Efros and Leung, in 1999, proposed a "pixel-at-a-time" approach to texture synthesis based on the conditional probability distribution of a pixel value given its already generated neighbors [3]. The major drawback of this approach is, of course, the run time. Several variations on this basic approach were proposed and implemented with varying levels of success. The observation that, for the basic algorithm, the main component of the running time was the need to search the entire input image for each pixel in the output image was what prompted Efros and Freeman to develop the "patch-at-a-time" approach in [1] that is the primary focus of this project. Since the image quilting algorithm described in [1] is not an optimization of the pixel-at-a-time approach, but rather an entirely new approach, it is able to achieve a speedup of a much greater order of magnitude.

Motivation

As stated in the introduction, modern techniques to meet the demand for high-quality synthetic images rely on the ability to generate arbitrarily large images of a texture from (possibly much) smaller input images. While stochastic and structured textures are challenging in their own right, the texture synthesis problem is most difficult for semi-structured textures. Early successful approaches at synthesizing structured and semi-structured textures, like the pixel-at-a-time approach in [3], were notable for their computational intensity and long running times. In [1], Efros and Freeman make the case that the full input image searches of the approach described in [3] are unnecessary for many of the pixels in structured textures. The "image quilting" idea, that is the idea of stitching together large arbitrarily shaped patches of the input image in order to generate the output image, has two main advantages over the pixel-at-a-time-approach. The first advantage is that, by transferring large patches at a time, it significantly reduces the number of searches required. This leads to significant decreases in run time which will become increasingly important as more and more non-traditional, and hence, less powerful, computing devices (e.g. phones, PDAs, etc…) could be called on to perform such tasks. The second advantage is that, again by transferring large patches at a time, it

guarantees visual consistency within any transferred patch. A potential pitfall of this approach is that it has an increased probability for visible boundaries in the generated texture at the points where the patches were stitched together.

Problem Statement

The goal of this project is to gain, through implementation, a thorough understanding of the Efros and Freeman image quilting algorithm and, by extension, of the issues that arise in texture synthesis in general.

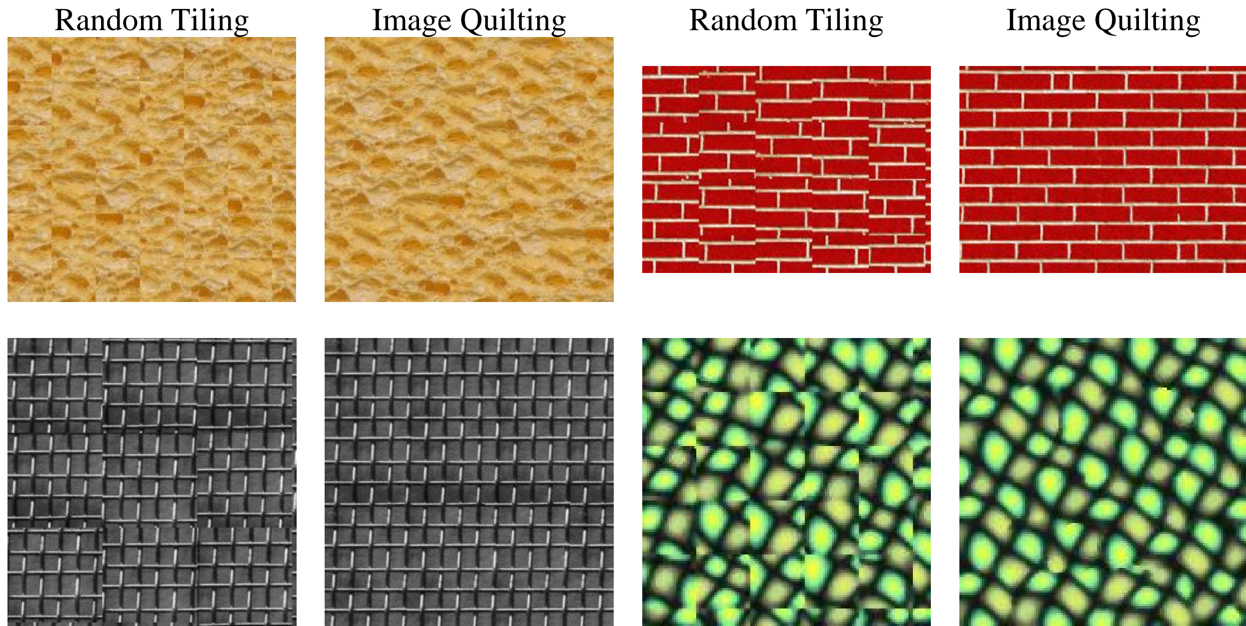
High-level Overview of the Algorithm

In this section I give a high-level description of the thought behind the algorithm. As discussed above, the main idea of the algorithm is to stitch together square texture "patches" from an input image in order to synthesize the output image. In [1], Efros and Freeman trace the evolution of the algorithm through three distinct steps. The basic initial idea is to tile together randomly selected square patches from the input image. This very naïve approach is actually quite successful for a very limited set of perfectly repetitive textures. In most semi-structures (and some stochastic) textures, though, the resulting images will have clear borders in them where neighboring patches meet. The next step in the approach is to allow neighboring patches of texture to overlap each other. Neighboring patches can now be selected based on some measure of the fit of their overlap. This will, in general, lead to borders that are less distinct but still noticeable. The final improvement to the basic algorithm is to permit the neighboring patches to have ragged borders. With this tweak to the algorithm, the demarcation between neighboring patches can now be chosen by selecting a path through the area of overlap that minimizes an error measure. This final improvement is the key innovation in the image quilting algorithm, is the basis for the algorithm's name, and significantly improves the quality of synthesized texture images. The diagram below (adapted directly from [1]) gives a visual overview of the algorithm:

Of course, the picture above sweeps many of the complexities of the algorithm under the rug. Fortunately, Efros and Freeman give a reasonably clear description of the parameters to set and the experimentally derived values that they used in their implementation. The next section contains a detailed discussion of the algorithmic complexities, including the error function used to determine the best-fit overlap, the definition of the error surface on which the minimum boundary cut is calculated and the size of the overlap window relative to the patch. Before tackling the details, however, it is beneficial to compare some experimental results achieved by random tiling with those achieved by image quilting. In this way we can consider whether or not the extra

complexity in the quilting algorithm relative to the random tiling method is worth introducing.

Below are several pairs of images. For each pair, the image on the left shows the synthesized output generated from the random tiling approach, while the image on the right shows the synthesized output generated from the quilting approach. It is important to note that, while the patch size may vary from pair to pair, it is constant within a given pair, so the results within a pair are comparable.



The four examples above are instructive for a few reasons. First, they represent several different, typical classes of texture. The texture in the upper left comes from a close-up image of a piece of bread and is an example of a stochastic texture. The texture in the upper right is from a synthesized image of a brick-wall, while the texture in the lower left is from a synthesized image of a fence. In both of these cases the images are synthesized to be regular, so the textures can be safely classified as fully structured. The image in the bottom right comes from a synthesized image of a pattern of scales. In this case, since the texture has some apparent, but irregular, structure, it is classified as semi-structured. It is interesting to note how much improvement is achievable with the two simple modifications described above.

Detailed Discussion of the Algorithm

The algorithm consists of the following steps:

- Initialize the upper-left corner of the output image with a randomly selected patch from the input image.
- Working left-to-right, top-to-bottom in the output image, repeat the following:
    a. Select the next patch to add from the input image from among the best-fit patches

b. Calculate the error surface between this new patch and its area of overlap with already processed patches
c. Calculate the minimum cost path through the error surface to determine the patch boundary and then add the new patch to the image

The initialization is self-explanatory and will not be discussed in detail here.

The heart of the algorithm is the left-to-right, top-to-bottom scan of the output image. To achieve the desired quilting behavior, the scan in this algorithm proceeds in steps of (patch_size – overlap_size) pixels. This is the key to achieving such dramatic decreases in run-time as compared to the previous, pixel-at-a-time approaches. Because of the left-to-right, top-to-bottom ordering of the scan, the area of overlap to consider is always contained in an L-shape composed of a horizontal strip across the top of the patch and a vertical strip along the left edge of the patch. Other than the initial patch (in which there is no overlap, thus our random choice), two special cases arise in which the overlap does not take on this characteristic L-shape. The first case is the top row of the output image. In this case there is no row of image data directly above, so the overlap area degenerates to a vertical strip along the left edge of the current patch. The second case arises in processing the first patch for every row. In this case we are at the left boundary of the image meaning that there is no image data directly to the left so the overlap area degenerates to a horizontal strip along the top of the current patch.

Step a: Patch selection

During each iteration of processing we need to select the best-fit patch to paste onto the current area of texture we are processing according to some error measure. The procedure for doing this is to perform a brute-force exhaustive search of all possible patches in the input image, calculate the error for each patch, retain all patches that meet the best-fit criteria, and, finally, randomly select one of the retained patches.

The first issue to consider in this step is the choice of what to use for our error measure. All that Efros and Freeman state regarding this issue is that their error measure was computed based on the L2 norm of pixel values. All of the images that I worked with were RGB color images, however, meaning that each pixel consists of not one, but three values. In my implementation I computed the error value at a given pixel to be the L2 norm of the weighted difference in corresponding pixel values between the current image data and the patch to paste in. The error value for a patch, then, was simply the sum of pixel error values across all pixels in the L-shaped overlap area of the patch. As an aside, I experimented with equal color channel weightings (i.e. 1/3 for each channel) and with standard luminance weightings (i.e. 0.30, 0.59, and 0.11 on the R, G and B channels respectively) and could see no significant difference in the output. In the end I chose to use the luminance weightings, but I do not think that this choice had any significant impact on the results.

The other issue to consider during patch selection is how strictly to interpret the concept of “best-fit.” Efros and Freeman state in their paper that in their implementation they

considered a patch to meet the best-fit criteria if its summed error was within ten percent of the minimum error patch. Following their lead, I used the same criteria in doing my best-fit calculations. I did not experiment with alternative bounds.

Step b: Calculate the error surface

Efros and Freeman define their error surface in terms of the squared difference between values at a given pixel. This differs from their overlap error calculation which was based on the L2 norm. In my implementation I chose to use the same weighted L2 norm as described above. Because of the methodology employed to determine the minimum error boundary cut, we can expect that using the L2 norm and using the squared difference will result in different error surfaces and, potentially, different cuts. Again, I experimented with using both measures and noticed no significant difference in the quality of the output images. In the end I chose to go with the channel weighted L2 norm (i.e. the same measure described in detail above) because it meant that I needed to calculate and store only a single value for each pixel for use in both the patch-selection phase and the error-surface calculation phase.

Step c: Determine minimum-cost boundary cut

Once the error surface is calculated, all of the information necessary to identify the minimum-cost boundary cut to use in order to paste the new patch into the output image is present. In their paper, Efros and Freeman define a cumulative path cost, E, based on pixel error values, $e_{i,j}$, on the error surface. Specifically, for a vertical portion of the overlap patch, they compute the minimum path cost, E, from a pixel in the first row to the current pixel recursively in terms of row index, i, and column index, j, as follows:

$$E_{i,j} = e_{i,j} + \min(e_{i-1,j-1}, e_{i-1,j}, e_{i-1,j+1}) \quad (i = 2 \ldots N)$$

with the appropriate arguments to the min function omitted at the edges of the overlap patch. The minimum error boundary cut can then be determined by searching the last (N^{th}) row of the overlap patch, identifying the minimum value, and tracing backwards. The formula for calculating the minimum cost path for a horizontal strip of the overlap patch, working left to right, is readily derivable with minor modifications to the i and j subscripts. For patches with both vertical and horizontal overlap (i.e. most patches), Efros and Freeman observe that the horizontal and vertical minimum cost paths will, by necessity, meet in the middle and the overall minimum can be chosen as the stopping point for the vertical and horizontal trace backs.

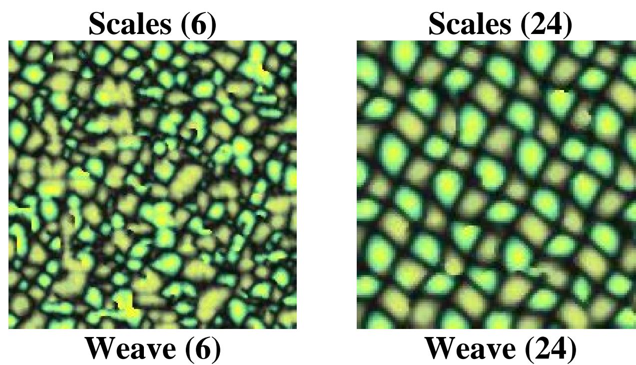
My implementation deviates from the description in the paper in one small, yet important way. Rather than initializing the top row (leftmost column) and working downwards (rightwards), I chose to initialize the bottom row (rightmost column) and work upwards (leftwards). For patches with non-L shaped error surfaces (i.e. patches with only horizontal or vertical overlap) there will be no difference in the calculated minimum cost boundary. For patches with L-shaped error surfaces, however, the minimum cost path could potentially be quite different. I believe that my right-to-left/bottom-to-top

algorithm will correctly calculate the global minimum cost path. To see why this is so, consider the boundary calculation phase of an L-shaped overlap patch. After determining the boundary to use, calculating the total error of the boundary cut requires only a summation of the pixel errors of every pixel on the path. Notice that some pixel in the "corner" of the L (i.e. the area where the horizontal patch intersects the vertical patch) is the "turning point" - the pixel at which the vertical portion of the path and the horizontal portion of the path intersect. It is easy to see that the cost of the whole path can be calculated by adding a) the minimum cost vertical path from the bottom row to this pixel and b) the minimum cost horizontal path from the rightmost column to this pixel and then subtracting out the pixel error at this pixel (it is double counted, once in each path). Thus, since the path calculation proceeds bottom-to-top and right-to-left, one way to assure selection of the global minimum cost boundary is by searching through the "corner" of the L for the pixel with the minimum value for the summation and difference just described. In the Efros and Freeman paper, however, the description seems to imply (because of the $i = 2 \ldots N$ in the definition of the path cost calculation, E) doing a left-to-right, top-to-bottom search. This will not, in general, give the minimum cost path that we desire. Having said this, the fact that the results shown in their paper are so good, and have relatively few boundary errors, I suspect that this error is only in their description of the algorithm, and not in their actual implementation.

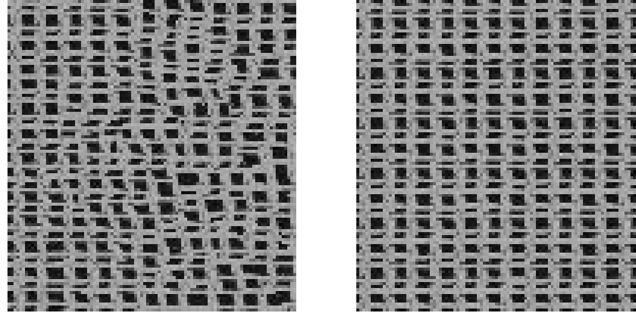
Other Implementation details:

Beyond those cited above, there are two other important parameters to consider – patch size and overlap size. In Efros and Freeman's original implementation, there is only one parameter for the user to set, patch size. The patch size is given in pixels, and the actual patch itself is assumed to be square (hence the single parameter). Getting the patch size right is essential for good performance. In general, there is a much higher penalty to pay for choosing too small of a patch size than there is for choosing too large of a patch size. The main danger of an overly large patch size is that the resulting image will be too repetitive. If the patch size is too small, however, key structures of the texture can be missed entirely. Below are some results showing how dramatic the effects of choosing too small of a patch size can be. The number in parentheses indicates the patch size, in pixels. In both examples we can see that the quilting algorithm is not able to capture the essence of the texture due to the small patch size.

Scales (6) **Scales (24)**



Weave (6) **Weave (24)**

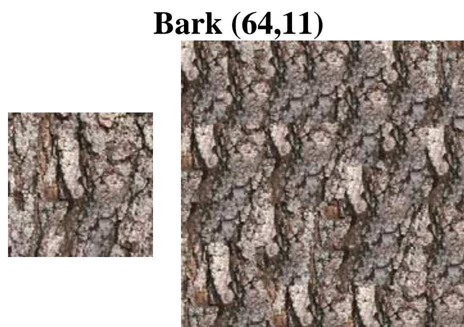


Moving on, for all of their experiments, Efros and Freeman fixed the overlap size at 1/6 of the patch size. In my initial experiments I followed this same paradigm. Later on I relaxed some of these constraints, and allowed the user to vary several of these parameters. Later in this paper I discuss the effects, for better or worse, that extending the implementation in these ways created.

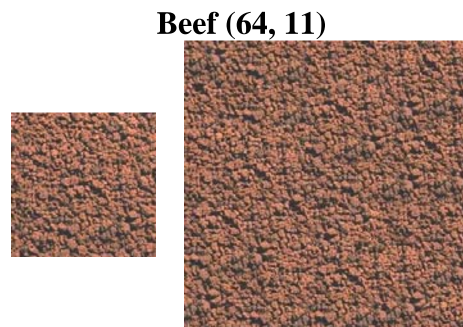
Experimental Results

Having explained in detail the basic algorithm and important aspects of my implementation of it, we are now ready to look at some of the results I was able to obtain. Below are several pairs of images. Each pair contains a small image, the input patch, and a larger image, the synthesized output patch. In each pair (except the apples), the output image is synthesized to have dimensions that are twice that of the input image – thus four times the area/number of pixels. The numbers in parentheses are the patch size and the overlap size respectively.

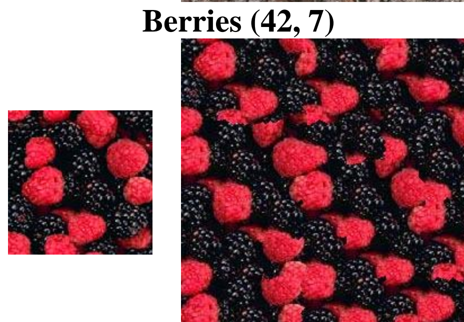
Bark (64,11)



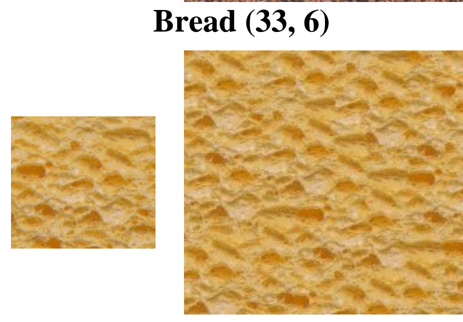
Beef (64, 11)



Berries (42, 7)

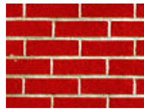


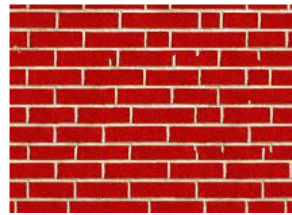
Bread (33, 6)



Concrete (26, 4)

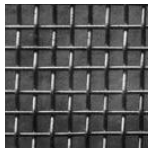
Brick Wall (42, 7)

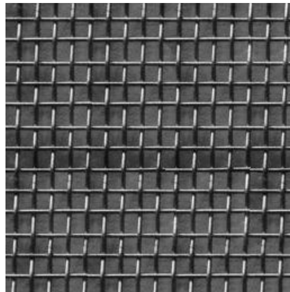


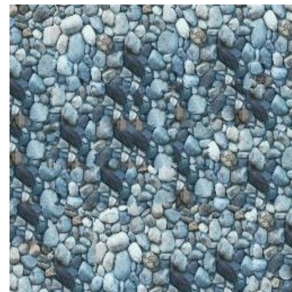


Fence (84, 14)

Pebbles (42, 7)

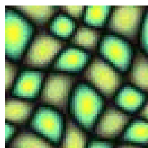


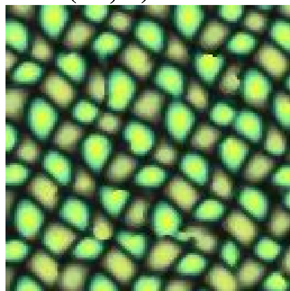




Scales (24, 4)

Text (24, 4)



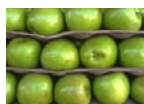


ut it becomes harder to lau
round itself, at "this daily
ving rooms," as House De
escribed it last fall. He fai
ut he left a ringing questio
nore years of Monica Lewi
inda Tripp?" That now see
Political comedian Al Fra
ext phase of the story will

Housed it last fall. He fatself, at "this dhe left a ringi
ll. Heft a ringing questiooms," as House years of Mo
g quears of Monica Lewid it last fall. Ha Tripp?" Tha
ica Lripp?" That now seeft a ringing quonica Lewie le
nowal comedian Al Frans of Monica Lat now see ye
?" That now seself, at "this daied it last fall. He fall.
omedian Al Frams," as House left a ringing questng
elf, anda Tlow sit last fall. He ringof Monica Lewnica
ms," ...
it lasda Trlast s of Monica Le?" Thomedian Al Fran
g queself, at "this dars of Monedian Al That now see
ica Loms," as Houseipp?" Thathe story sedian Al Fra
now. it last fall. Hal comedians," at lashe left a ra ri
d itself, at a ringing as of quest last a ringing rs of to
rooms," a of Monicipp?" a Lera rins of Monipp?" dai
bed it last years of Monica Lering rooms," as House D
left a ring Tripp?" That now scribed it last fall. He f

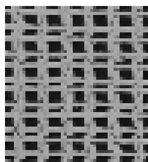
Wood (42, 7)

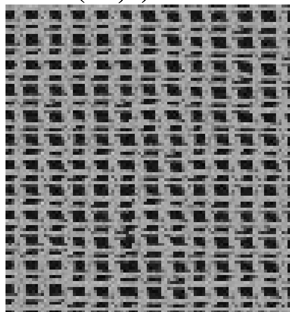
Apples (42, 7)

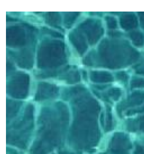


Weave (24,4)

Water (48, 8)







We can make several interesting observations about the synthesized textures. First, it is clear that the algorithm performs quite well on both stochastic and fully structured input. I consider both the "weave" and "fence" textures to be fully structured. The results on these textures were excellent, though we would expect this to be the case since it is not hard to imagine the naïve tiling approach to perform well on a fully structured, basically rectangular, texture like these. Due to the nature of the textures in question, we expect quite a lot of near-perfect repetition. I consider "beef" and "concrete" to be stochastic textures. In both of these cases, the rough, stochastic nature of the textures is ideally suited to the quilting algorithm. As long as the boundaries are reasonable, they get masked by the texture itself, resulting in apparently seamless synthesized images.

The other images fall into the "semi-structured" category. In [1], Efros and Freeman make the intuitively sensible claim that images in the semi-structured category were always the most difficult for the statistical methods to handle. On the "bark" and "bread" textures, my implementation performed quite well. There are no obvious boundaries, and the textures look natural. The "text" image looks reasonable as well. A close examination shows that the synthesized image is obviously gibberish, but at a glance no obvious non-Roman characters jump out and the text looks realistic. There are no obvious boundary issues, either, which is probably due to the presence of a lot of white space of more or less constant color and intensity that the algorithm is able to exploit to make a good minimum-error cut.

Four semi-structured images, "berries", "pebbles", "wood", and "water" show repetition effects. In the case of berries and water, the repetition is not necessarily undesirable – the water looks natural, and the berries in the input texture do appear to be in rows (it is not clear whether the output texture should be rows of berries or an unordered pile of berries). My implementation of the algorithm does not perform very well on the wood or pebbles textures. In both cases (though most absurdly with the knot in the wood), there appears to be some attraction to dark spots in the input texture. Indeed, though they take up a small amount of the total area, the knots in the wood and the black stones in the pebbles are repeated excessively and certainly capture a viewer's attention.

Finally, three of the images, "apples", "scales", and "brick wall" show significant boundary artifacts. For the scales and, to a lesser extent, the apples textures, median smoothing (not shown here) is able to mask some of the more obvious boundaries. In the brick wall, however, the issue is not one of smoothing as much as it is of inadequately capturing the texture.

Some Possible Extensions to the Basic Algorithm

As discussed in the previous section, while my implementation of the image quilting algorithm performed quite well on many of the textures that I applied it to, it performed poorly on some others. Consistent with Efros and Freeman's observations in [1], unsuccessful texture synthesis seemed to stem from excessive repetition, boundary issues, or both. I identified the following three extensions to try to improve performance:

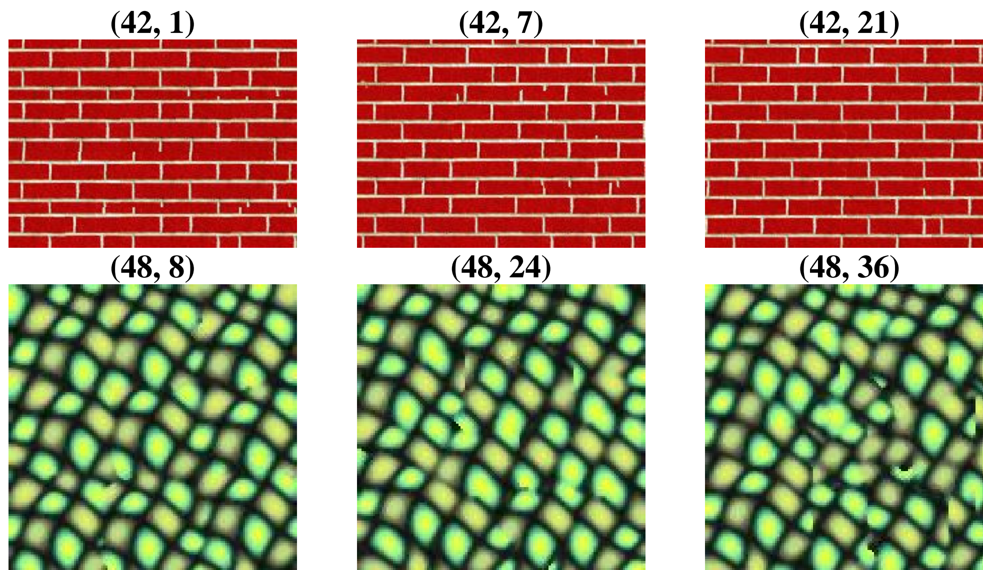
a. Allow variable overlap size

b. Allow rectangular patches
c. Consider flipped and/or rotated versions of the input image in the patch selection phase

The first two extensions, allowing variable overlap size, and allowing rectangular patches, are intended to attack the boundary problem. Allowing variable overlap size expands the space available to search for a minimum-cut boundary. This expanded search space should lead, though at the cost of increased processing time in the patch-selection and path computation phases, to better (i.e. lower error) boundaries. The final extension I identified, considering flipped and/or rotated versions of the input image, is intended to address the excessive repetition problem by giving eight times as much input data (the eight possible orientations) from the single input image. I implemented extensions a and b and the results are discussed below.

Experimental Results of the Implemented Extensions

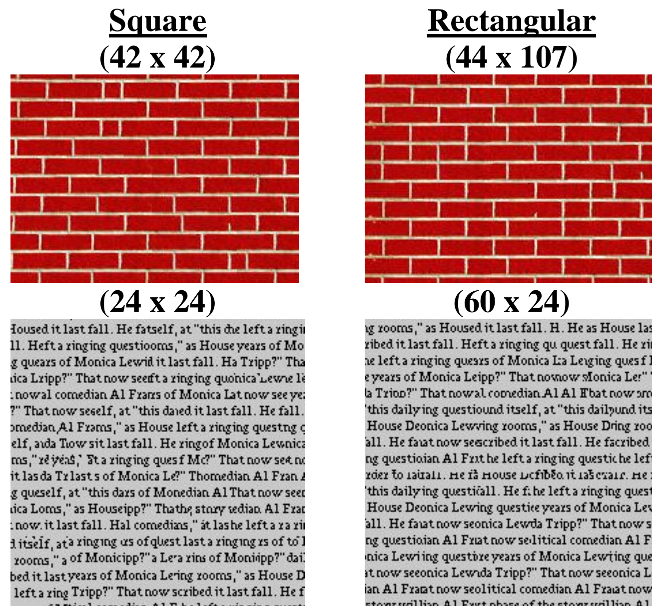
The first proposed extension is to allow the user to specify the size of the overlap area rather than fixing it at 1/6 the patch size. Below are two examples that illustrate some effects that varying the overlap area (while holding the patch size constant) can have on image synthesis. The numbers in parentheses are the patch size and the overlap window (both in pixels) respectively:



The interesting thing to note here is that a larger overlap size is **not** always desirable. My initial assumption was that since a larger overlap size will, in general, result in a lower cost cut, we will always be better off with as large of an overlap size as possible. Indeed, with the brick wall we see that as we increase overlap size we are able to achieve better and better performance. With the scales pattern, however, the smaller overlap size performs better than the larger overlap size. In the image with the largest overlap size we can see a significant amount of "clumping" of the bright green patches. In this case, the

tendency to cut along closely matching pixels coupled with the significantly expanded search space resulted in an unanticipated negative effect on the quality of the output image. This observation that an increased overlap window is beneficial for some texture classes but detrimental to others underscores the idea that allowing the overlap window to be a user-specified parameter is probably preferable to the one-size-fits-all approach taken in the base implementation as described in the original paper.

The other proposed extension that I implemented was to allow rectangular patches rather than simply square patches. The idea behind this extension was to improve results in textures that are composed of objects in rows by allowing long rectangular patches that would, hopefully, capture some of the "row-ness" of the input image. To investigate if this extension would fit that need I tested it on textures with obvious rows – such as the bricks and the newspaper text. Obviously, since a square is a rectangle, this extension to the implementation does not preclude the use of square patches. The spirit of the experiment, however, was to check the effects of using rectangular patches that were not square or "almost square." Below are some results that compare the best output I was able to achieve with square patches (i.e. the results from the main portion of the paper) with the best output I was able to achieve with a rectangular (but not square or almost square) patch. The left column shows the square patch results and the right column shows the rectangular patch results. The numbers in parentheses give the dimensions of the patch.



Interestingly enough, on both of these textures this extension does not seem to enhance performance. The brick wall texture synthesized with the rectangular patch is arguably inferior to the one synthesized with the square patch. The patch width required, though, was the full width of the input texture which seems like cheating. The hope was that the "small" bricks and the partial mortar lines between bricks would no longer be present - they still are, though. For the newspaper text, the results of each methodology are comparable.

Unfortunately the bricks and the newspaper were the two most successful textures for this extension to the algorithm. As a matter of fact, I tested this extension on the rows of apples texture as well (results not shown) and in every test that I performed, the synthesized texture was significantly less impressive than that achieved with the standard square patch. While I would want to run more experiments on a wider variety of textures before firmly concluding that the rectangular patch extension is useless, the preliminary results shown here do strongly suggest that it is probably not a fruitful avenue to pursue and is not worth the time spent to implement this extension. It is not at all clear to me why the experimental results for the apples texture would be worse with the rectangular extension than with the original square-patch restriction. The only explanation that I can come up with is that perhaps, since there are some random elements of the algorithm (initial patch, choice among the "best-fit" population), I happened to get lucky with my square patch results.

Concluding Remarks

In this paper we have analyzed an implementation of the image quilting algorithm first proposed by Efros and Freeman in 2001. We discussed the theory behind the algorithm, the implementation specific issues that must be addressed and the error functions and user defined parameters that determine the algorithm's operation. Observing experimental data helped us to identify the strengths and weaknesses of the algorithm as well as possible extensions to improve performance on specific classes of input texture. We then examined the results obtained by extending the implementation in some of the ways described. Overall, the basic algorithm performs quite well on a broad class of textures. It produces high-quality synthesized texture images very quickly. There are undoubtedly many more possible extensions that could improve the results on some of the semi-structured textures. I believe, however, that the next "great leap forward" in this area will come not from a purely image based approach, but rather through some approach that is able to more explicitly, and accurately, model the texture elements themselves. Perhaps one of my classmate's suggestions, given after the class presentation, to incorporate edge-detection techniques in order to overcome some of the boundary issues could be a good first step in that direction and open up a new avenue of research in this area.

References

1. A.A. Efros and W.T. Freeman, Image Quilting for Texture Synthesis and Transfer, SIGGRAPH 01.
2. D.J. Heeger and J.R. Bergen, Pyramid-Based Texture Analysis/Synthesis, SIGGRAPH 95.
3. A.A. Efros and T.K. Leung, Texture Synthesis by Non-parametric Sampling, ICCV 99.

Appendix: Original Java Source code

Note: I implemented in Java because it was easier for me to work on the project from home that way. The original implementation described in the paper was in MATLAB and this one could undoubtedly be ported to MATLAB as well, though Java's object orientation make much of the code (in my opinion) easier to understand, though at the cost of being considerably less concise. Note that this code is not "clean" in that it has a lot of commented out debugging code, etc…

ImageFile.java – Used for the "processing"

```
import java.io.*;
import java.util.*;

class ImageFile {
	protected String magicNumber;
	protected int myRows;
	protected int myCols;
	protected int myChannels;
	protected int myMaxVal;
	protected byte[][][] myData;

	//these are for synthesizing new images
	protected int newRows;
	protected int newCols;
	//protected int patchSize;
	protected int patchRows;
	protected int patchCols;
	//protected int ovlpSize;
	protected int ovlpSizeR;
	protected int ovlpSizeC;
	protected byte[][][] newData;

	static byte[][][] copyArray(byte[][][] source, int r, int c, int ch) {
		byte[][][] rVal = new byte[r][c][ch];
		for (int i = 0; i < r; i++) {
			for (int j = 0; j < c; j++) {
				for (int k = 0; k < ch; k++) {
					rVal[i][j][k] = source[i][j][k];
				}
			}
		}
		return rVal;
	}
	class Patch {
		byte[][][] data;
		int rows;
		int cols;
		int channels;

		public Patch(int r, int c, byte[][][] d, int ch) {
			rows = r;
			cols = c;
			//data = copyArray(d, r, c, ch);;
			data = d;
			channels = ch;
		}
	}

	class TraceBackNode {
		double myError;
		int predOffset;
```

```
		public TraceBackNode(double error, int offset) {
			myError = error;
			predOffset = offset;
		}
	}

	class Anchor {
		byte[][][] data;
		int ovlpRows;
		int ovlpCols;
		int totalRows;
		int totalCols;
		int channels;

		public void stitch(Patch p) {
			//we have the anchor and the patch to add to it
			//let's just paste the patch over for now to test things
			//data = copyArray(p.data, totalRows, totalCols, channels);
			//data = p.data;

			//let's do the horizontal traceback first
			TraceBackNode[][] horizontal = new
TraceBackNode[ovlpRows][totalCols];
			//do the path finding - need to go right to left...
			//do the rightmost column first
			for (int i = 0; i < ovlpRows; i++) {
				double error = calcPixelError(p, i, totalCols - 1);
				horizontal[i][totalCols - 1] = new TraceBackNode(error, -
100);
			}
			for (int j = totalCols - 2; j >= 0; j--) {
				for (int i = 0; i < ovlpRows; i++) {
					double error = calcPixelError(p, i, j);
					double aboveError = Double.POSITIVE_INFINITY;
					if (i!=0) aboveError = horizontal[i-1][j+1].myError;
					double belowError = Double.POSITIVE_INFINITY;
					if (i+1!=ovlpRows) belowError =
horizontal[i+1][j+1].myError;
					double bestError = horizontal[i][j+1].myError;
					int offset = 0;
					if (aboveError < bestError) {
						bestError = aboveError;
						offset = -1;
					}
					if (belowError < bestError) {
						bestError = belowError;
						offset = 1;
					}
					horizontal[i][j] = new TraceBackNode(error +
bestError, offset);
				}
			}

			//now do the vertical traceback
			TraceBackNode[][] vertical = new
TraceBackNode[totalRows][ovlpCols];
			//do the path finding - need to go bottom to top...
			//do the bottom row first
			for (int j = 0; j < ovlpCols; j++) {
				double error = calcPixelError(p, totalRows-1, j);
				vertical[totalRows-1][j] = new TraceBackNode(error, -100);
			}
			for (int i = totalRows - 2; i >= 0; i--) {
				for (int j = 0; j < ovlpCols; j++) {
					double error = calcPixelError(p, i, j);
					double leftError = Double.POSITIVE_INFINITY;
					if (j!=0) leftError = vertical[i+1][j-1].myError;
					double rightError = Double.POSITIVE_INFINITY;
					if (j+1!=ovlpCols) rightError =
vertical[i+1][j+1].myError;
					double bestError = vertical[i+1][j].myError;
```

```
                                        int offset = 0;
                                        if (leftError < bestError) {
                                                bestError = leftError;
                                                offset = -1;
                                        }
                                        if (rightError < bestError) {
                                                bestError = rightError;
                                                offset = 1;
                                        }
                                        vertical[i][j] = new TraceBackNode(error +
bestError, offset);
                                }
                        }
                        boolean[][] keepPatch = getCutMap(horizontal, vertical);
                        //so now our keeppatch is complete.
                        //traverse it and fill in the anchor as necessary
                        for (int i = 0; i < totalRows; i++) {
                                for (int j = 0; j < totalCols; j++) {
                                        if (keepPatch[i][j]) data[i][j] = p.data[i][j];
                                }
                        }

                }

                public boolean[][] getCutMap(TraceBackNode[][] horizontal,
TraceBackNode[][] vertical) {
                        //this is how we will decide where we are relative to the path
                        //assume keeping the whole patch by default
                        boolean keepPatch[][] = new boolean[totalRows][totalCols];
                        for (int i = 0; i < totalRows; i++) {
                                for (int j = 0; j < totalCols; j++) {
                                        keepPatch[i][j] = true;
                                }
                        }

                        //check the various cases
                        if (ovlpRows == 0 && ovlpCols == 0) {
                                return keepPatch;
                        }
                        if (ovlpRows == 0) {
                                //only vertical overlap
                                //find the min value in the top row of vertical
                                //and follow it back to the bottom
                                double minErr = Double.POSITIVE_INFINITY;
                                int bestJ = -1;
                                for (int j = 0; j < ovlpCols; j++) {
                                        double curr = vertical[0][j].myError;
                                        if (curr < minErr) {
                                                minErr = curr;
                                                bestJ = j;
                                        }
                                }

                                //now follow it
                                keepPatch = verticalTB(vertical, keepPatch, 0, bestJ);
                        } else if (ovlpCols == 0) {
                                //only horizontal overlap
                                //find the min value in the left col of horizontal
                                //and follow it back to the right
                                double minErr = Double.POSITIVE_INFINITY;
                                int bestI = -1;
                                for (int i = 0; i < ovlpRows; i++) {
                                        double curr = horizontal[i][0].myError;
                                        if (curr < minErr) {
                                                minErr = curr;
                                                bestI = i;
                                        }
                                }

                                //now follow it
                                keepPatch = horizontalTB(horizontal, keepPatch, bestI, 0);
```

```
			} else {
				//normal case - we need to find the min summed cost in
				//the UL ovlp patch
				int bestI = -1;
				int bestJ = -1;
				double minErr = Double.POSITIVE_INFINITY;
				for (int i = 0; i < ovlpRows; i++) {
					for (int j = 0; j < ovlpCols; j++) {
						double err = horizontal[i][j].myError +
vertical[i][j].myError;
						if (err < minErr) {
							bestI = i;
							bestJ = j;
							minErr = err;
						}
					}
				}
				//now mark everything above and left of the best spot
				for (int i = 0; i <= bestI; i++) {
					for (int j = 0; j <= bestJ; j++) {
						keepPatch[i][j] = false;
					}
				}
				//and finally do the tracebacks
				keepPatch = verticalTB(vertical, keepPatch, bestI, bestJ);
				keepPatch = horizontalTB(horizontal, keepPatch, bestI,
bestJ);
			}
			return keepPatch;
		}

		public boolean[][] verticalTB(TraceBackNode[][] v, boolean[][] keep, int
i, int j) {
			while (i < totalRows) {
				for (int colCounter = 0; colCounter <= j; colCounter++) {
					keep[i][colCounter] = false;
				}
				TraceBackNode tmp = v[i][j];
				i++;
				j+=tmp.predOffset;
			}
			return keep;
		}

		public boolean[][] horizontalTB(TraceBackNode[][] h, boolean[][] keep, int
i, int j) {
			while (j < totalCols) {
				for (int rowCounter = 0; rowCounter <= i; rowCounter++) {
					keep[rowCounter][j] = false;
				}
				TraceBackNode tmp = h[i][j];
				j++;
				i+=tmp.predOffset;
			}
			return keep;
		}
		public double calcError(Patch p) {
			if (ovlpRows == 0 && ovlpCols == 0) return 0.0;
			//first calculate the dual overlap (upper left corner)
			double cumError = 0.0;
			for (int i = 0; i < ovlpRows; i++) {
				for (int j = 0; j < ovlpCols; j++) {
					//double curError = 0.0;
					//for (int k = 0; k < channels; k++) {
					//	double currDiff = data[i][j][k] -
p.data[i][j][k];
					//	curError += currDiff*currDiff;
					//}
					//cumError += Math.sqrt(curError);
					cumError += calcPixelError(p, i, j);
				}
```

```
			}
			//now do the row only (ie top strip)
			for (int i = 0; i < ovlpRows; i++) {
				for (int j = ovlpCols; j < totalCols; j++) {
					//double curError = 0.0;
					//for (int k = 0; k < channels; k++) {
					//	double currDiff = data[i][j][k] -
p.data[i][j][k];
					//	curError += currDiff*currDiff;
					//}
					//cumError += Math.sqrt(curError);
					cumError += calcPixelError(p, i, j);
				}
			}
			//now do the col only (ie left strip)
			for (int i = ovlpRows; i < totalRows; i++) {
				for (int j = 0; j < ovlpCols; j++) {
					//double curError = 0.0;
					//for (int k = 0; k < channels; k++) {
					//	double currDiff = data[i][j][k] -
p.data[i][j][k];
					//	curError += currDiff*currDiff;
					//}
					//cumError += Math.sqrt(curError);
					cumError += calcPixelError(p, i, j);
				}
			}
			return cumError;
		}

		private double calcPixelError(Patch p, int row, int col) {
			double rVal = 0.0;
			//for (int k = 0; k < channels; k++) {
				//double currDiff = data[row][col][k] -
p.data[row][col][k];
				//rVal += currDiff*currDiff;
			//}
			double Rdiff = .3*Math.pow((data[row][col][0] -
p.data[row][col][0]),2);
			double Gdiff = .59*Math.pow((data[row][col][1] -
p.data[row][col][1]),2);
			double Bdiff = .11*Math.pow((data[row][col][2] -
p.data[row][col][2]),2);
			rVal += (Rdiff + Gdiff + Bdiff);
			return Math.sqrt(rVal);
		}

		public Anchor(int tR, int tC, int oR, int oC, byte[][][] d, int c) {
			totalRows = tR;
			totalCols = tC;
			ovlpRows = oR;
			ovlpCols = oC;
			//data = (d, tR, tC, c);
			data = d;
			channels = c;
		}
	}

	private Patch selectPatch(Anchor anchor) {
		//the Patch itself will be the same size as the anchor
		int rCount = anchor.totalRows;
		int cCount = anchor.totalCols;
		int maxRow = myRows - rCount + 1;
		int maxCol = myCols - cCount + 1;
		double minError = Double.POSITIVE_INFINITY;
		double[][] errorTrack = new double[maxRow][maxCol];
		//now we need to zip through the original (this) image
		//and create patches
		for (int i = 0; i < maxRow; i++) {
			for (int j = 0; j < maxCol; j++) {
```

```
                              byte[][][] pData = extractArray(i, rCount, j, cCount,
myData, myChannels);
                              Patch curPatch = new Patch(rCount, cCount, pData,
myChannels);

                              //now have the anchor calculate the error measure
                              double curError = anchor.calcError(curPatch);
                              errorTrack[i][j] = curError;
                              if (curError < minError) minError = curError;
                       }
               }

               double tolerance = 0.1;
               double threshold = minError + tolerance*minError;
               int possible = 0;

               for (int i = 0; i < maxRow; i++) {
                       for (int j = 0; j < maxCol; j++) {
                              if (errorTrack[i][j] <= threshold) possible++;

                       }
               }

               //System.out.println("Possible Choices:" + possible);
               int toTake = (int)(Math.floor(possible*Math.random()));
               //System.out.println("ToTake:" + toTake);
               int chosenI = -1;
               int chosenJ = -1;
               possible = 0;
               for (int i = 0; i < maxRow && possible <= toTake; i++) {
                       for (int j = 0; j < maxCol && possible <= toTake; j++) {
                              if (errorTrack[i][j] <= threshold) {
                                     if (possible == toTake) {
                                            chosenI = i;
                                            chosenJ = j;
                                     }
                                     possible++;
                              }
                       }
               }
               //System.out.println("Used:" + chosenI + "," + chosenJ);
               byte[][][] pData = extractArray(chosenI, rCount, chosenJ, cCount, myData,
myChannels);
               return new Patch(rCount, cCount, pData, myChannels);


        }

        private void addPatch(int curRow, int curCol) {
               System.out.println("Called add Patch: " + curRow + "," + curCol);
               //first figure out the actual size of the patch we will be adding
               //could vary due to boundary conditions
               int pRows = Math.min(patchRows, newRows - curRow);
               int pCols = Math.min(patchCols, newCols - curCol);
               //if either dimension is <= ovlp size this is unnecessary, keep what's
there
               if (pRows <= ovlpSizeR || pCols <= ovlpSizeC) return;
               //System.out.println("Hi!");
               //now extract the area we are filling in
               byte aData[][][] = extractArray(curRow, pRows, curCol, pCols, newData,
myChannels);
               //and then determine the position specific overlap stuff...
               int oRows = ovlpSizeR;
               if (curRow == 0) oRows = 0;
               int oCols = ovlpSizeC;
               if (curCol == 0) oCols = 0;
               //and create the actual anchor for where we are...
               Anchor anchor = new Anchor(pRows, pCols, oRows, oCols, aData, myChannels);
               Patch toAdd = selectPatch(anchor);
               //with the anchor and a patch to attach to it we can stitch it in
               anchor.stitch(toAdd);
```

```
			//and then copy the anchor data back to the newImage array
			for (int i = 0; i < pRows; i++) {
				int rowIndex = curRow + i;
				for (int j = 0; j < pCols; j++) {
					int colIndex = curCol + j;
					for (int k = 0; k < myChannels; k++) {
						newData[rowIndex][colIndex][k] =
anchor.data[i][j][k];
					}
				}
			}
	}

	public ImageFile synthesize(int patchRowCount, int patchColCount,
				int ovlpR, int ovlpC, int r, int c) {
			newRows = r;
			newCols = c;
			//patchSize = patch;
			patchRows = patchRowCount;
			patchCols = patchColCount;
			//ovlpSize = ovlp;
			ovlpSizeR = ovlpR;
			ovlpSizeC = ovlpC;
			//int step = patchSize - ovlpSize;
			int rowStep = patchRows - ovlpSizeR;
			int colStep = patchCols - ovlpSizeC;
			newData = new byte[newRows][newCols][myChannels];

			for (int i = 0; i < newRows; i += rowStep) {
				for (int j = 0; j < newCols; j += colStep) {
					addPatch(i, j);
					if (j + patchCols >= newCols) j = newCols;
				}
				if (i + patchRows >= newRows) i = newRows;
			}
			return new ImageFile(newRows, newCols, newData);
	}

	public static byte[][][] extractArray(
				int rowStart, int rowCount, int colStart, int colCount, byte[][][]
source,
				int channelCount) {
			byte[][][] rVal = new byte[rowCount][colCount][channelCount];
			for (int i = 0; i < rowCount; i++) {
				int rowIndex = rowStart + i;
				for (int j = 0; j < colCount; j++) {
					int colIndex = colStart + j;
					for (int k = 0; k < channelCount; k++) {
						rVal[i][j][k] = source[rowIndex][colIndex][k];
					}
				}
			}
			return rVal;
	}

	public int getRows() {
			return myRows;
	}

	public int getCols() {
			return myCols;
	}

	public int getChannels() {
			return myChannels;
	}

	public byte[][][] getData() {
			return myData;
	}
```

```
	public String toString() {
		return "magicNumber: " + magicNumber + "; rows:" + myRows +
			"; cols:" + myCols + "; channels:" + myChannels + "; maxVal:" +
myMaxVal;
	}

	public ImageFile(int rows, int cols, byte[][][] data) {
		myRows = rows;
		myCols = cols;
		myData = data;
		magicNumber = "P6";
		myChannels = 3;
		myMaxVal = 255;
	}

	public ImageFile(String fileName) throws IOException{
		BufferedInputStream in = new BufferedInputStream(new
FileInputStream(fileName));
		magicNumber = readMagicNumber(in);
		//System.out.println(magicNumber);
		myCols = readInt(in);
		//System.out.println("Rows: " + rows);
		myRows = readInt(in);
		myMaxVal = readInt(in);
		myChannels = 3;
		myData = new byte[myRows][myCols][myChannels];
		for (int i = 0; i < myRows; i++) {
			for (int j = 0; j < myCols; j++) {
				for (int k = 0; k < myChannels; k++) {
					myData[i][j][k] = (byte)(in.read());
					if (i == 0 & j == 0) {
						System.out.println(myData[i][j][k]);
					}
				}
			}
		}

		int curByte = in.read();
		while (curByte != -1) {
			if (curByte != 32 && curByte != 10) {
				throw new IOException("data at end of file?");
			}
		}
	}

	public void write(String filename) throws IOException {
		BufferedOutputStream out = new BufferedOutputStream(new
FileOutputStream(filename));
		byte[] mN = magicNumber.getBytes();
		out.write(mN, 0, mN.length);
		out.write(10);

		byte[] c = Integer.toString(myCols).getBytes();
		out.write(c, 0, c.length);
		out.write(32);

		byte[] r = Integer.toString(myRows).getBytes();
		out.write(r, 0, r.length);
		out.write(32);

		byte[] mV = Integer.toString(myMaxVal).getBytes();
		out.write(mV, 0, mV.length);
		out.write(10);

		for (int i = 0; i < myRows; i++) {
			for (int j = 0; j < myCols; j++) {
				for (int k = 0; k < myChannels; k++) {
					out.write(myData[i][j][k]);
				}
			}
		}
```

```
                out.flush();
                out.close();
        }

        private int readInt(BufferedInputStream in) throws IOException{
                int curByte = in.read();
                //System.out.println(curByte);
                while (curByte == 35) {
                        //System.out.println("In a comment");
                        curByte = in.read();
                        while (curByte != 10) {
                                //System.out.println(curByte);
                                curByte = in.read();
                        }
                        //System.out.println(curByte);
                        curByte = in.read();
                        //System.out.println(curByte);
                }
                int rVal = 0;
                while (curByte != 32 && curByte != 10) {
                        if (curByte < 48 || curByte > 57) {
                                throw new IOException("Got a non-digit");
                        }
                        curByte = curByte - 48;
                        rVal *= 10;
                        rVal += curByte;
                        curByte = in.read();
                }

                return rVal;

        }

        private String readMagicNumber(BufferedInputStream in) throws IOException{
                int curByte = in.read();
                if (curByte != 80) throw new IOException();
                curByte = in.read();
                if (curByte != 54) throw new IOException();
                curByte = in.read();
                if (curByte != 10) throw new IOException();
                return "P6";
        }
}
```

ImageQuilt.java – used for parsing the command line args

```
import java.io.*;
import java.util.*;

public class ImageQuilt{
        static String ext = ".ppm";
        static String inPath = "input/";
        static String outPath = "outputL/";


        public static void main(String args[]) {
                try {
                        String inFileName = args[0];
                        //int patchSize = Integer.parseInt(args[1]);
                        int patchRows = Integer.parseInt(args[1]);
                        int patchCols = Integer.parseInt(args[2]);
                        int ovlpRows = Integer.parseInt(args[3]);
                        int ovlpCols = Integer.parseInt(args[4]);
                        int outputRows = Integer.parseInt(args[5]);
                        int outputCols = Integer.parseInt(args[6]);
                        String outFileName = args[7];

                        ImageFile imF = new ImageFile(inFileName);
                        ImageFile imF2 = imF.synthesize(patchRows, patchCols,
                                ovlpRows, ovlpCols, outputRows, outputCols);
                        imF2.write(outFileName);

                } catch (FileNotFoundException fnfe) {
                        System.err.println("Unexpected fnfe in main");
                        System.exit(-1);
                } catch (IOException ioe) {
                        System.err.println("Unexpected ioe in main");
                        System.err.println(ioe);
                        System.exit(-1);
                }

        }
}
```