Image Texture Tiling

Philip Reasa, Daniel Konen, Ben Mackenthun

December 18, 2013

Contents

Abstract

There has been extensive research done on the subject of texture synthesis. The main focus has been growing/continuing textures. We hope to explore the concept of texture synthesis beyond simply increasing the texture size, and into tessellating/tiling the texture. When a ”single unit” of texture can be tiled such that seems are not seen and borders are hidden, the texture can grow endlessly in all directions; something that current texture synthesis methods cannot achieve. Using techniques such as seam carving to produce minimum energy costs, our group hopes to be able to discover a ”single tile” of texture from an input image that contains the texture. This will be applicable in areas such as web development, and anywhere else were images are not processed to extend texture, but rather naively tiled.

Introduction

I Motivation

Texture Tiling

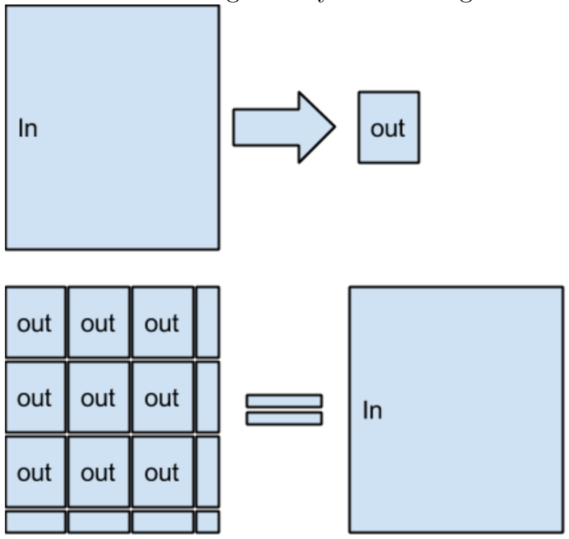
Although we have seen texture synthesis algorithms that can be applied in limited amounts (filling-in small areas), we wanted to create an algorithm that was able to work perpetually; specifically for its applications in web development. The first time that the need for such an algorithm was realized was during a summer job when there was a need to find a tiling portion of a texture. Per a mock-up, a background image had to be applied to a web-page, however the background image provided did not tile without artifacts. Although the pattern present in the background image was able to tile, the provided image in its current state did not. Without an algorithm to solve this problem a simple guess and check was the most practical for finding such a tiling image.

Image Reduction

Seeing this problem from a completely different perspective, this problem is also a problem of reduction. Specifically we are reducing a large image to a smaller image that can be used to accurately regenerate the larger image.

II Problem Statement

Given an input image (In) create an output image (out) such that when out is tiled it accurately reconstructs In, and can continue to tile to grow beyond the original size of In.



III Related Work

There has been related work on texture synthesis done by Efros and Freeman [2] and Efros and Leung [1]. These works are based off filling in texture holes, and in order to continue the texture synthesis substantial computing power must be used. Our approach differs as the initial computing is done all up front, and the continued synthesis can easily be carried out afterwards. That being said we do use SSD error detection as suggested by Efros-Leung[1] and compute the energy functions as done by Efros-Freeman [2]. We also solve a simpler problem than these two papers are solving. We assume that the input image contains a tile that can be perpetually repeated. This means that virtually no natural picture will satisfy our assumption, and only intentionally created patterns will work - obviously an easier problem, but still with many applications.

Our Method

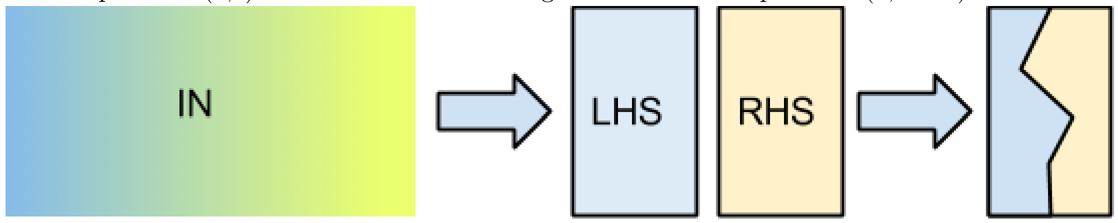
First: We have three specific parameters that are set for each run of our code.

1. Energy Mode - used to specify what energy function should be used. We chose to implement an SSD similar to Efros-Freeman [2]. Other options for this are listed in the results/experiments sections.
2. Seam Mode - specifies how the vertical seam within the energy image is computed. We currently have two methods implemented: (1) a straight seam, and (2) an 8-connect seam. The best results tended to be produced by a straight seam; more information in results/experiments sections.
3. Reduce Factor - specifies if we were going to find an global minimum seam or just a local minimum. As mentioned in the results/experiments sections, we used global as it produced better results.

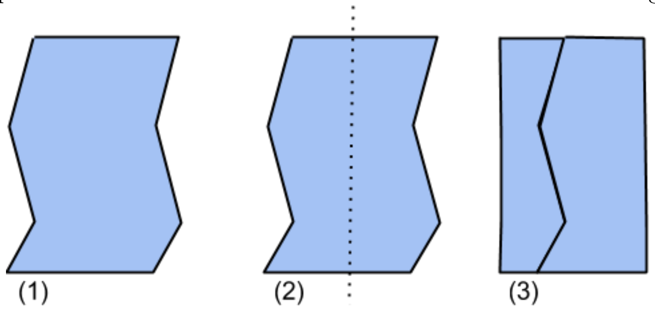
Our basic tile finding algorithm (vastly simplified) is as follows

```
1. given image $In$
2. for all possible $outputImage\ width$s
    (a) $overlapingPartsOfImages$ = getOverlap($in$, $width$)
    (b) $e$ = energyFn($overlapingPartsOfImages$)
    (c) $s$ = findBestSeam($overlapingPartsOfImages$, $e$)
    (d) $currentBestMin$ = min(energyOfSeam($s$),$currentBestMin$)
3. $outputImage$ = squareImage($In$,$s$)
4. rotate image, re-run with $in = outputImage$
```

To clarify our algorithm we will briefly explain each step, however the exact algorithm can be found in our code. Part 1 is simply accepting input from the user. Part 2 is very complex. In essence we are trying to trim the input image so that its right-hand side perfectly matches its left hand side. We do this like they do for image quilting. We choose a certain amount of overlap (multiple different amounts are iterated through at part 2), and then find the best seam so that the pixel at (r,c) on the RHS will look good next to the pixel at $(r,c+1)$ on the LHS.



Step 3 is mostly an artifact of our implementation, but still is worth mentioning. We are left with an image like (1) after we find the best seam. At that point we have to make the image regular, so we merge the two jagged sides (which are mirrors of one another). In (3) it is important to note that any "artifacts" that will be created during the tiling can be clearly seen in the middle of this picture. This makes validation of the success of the algorithm much quicker.



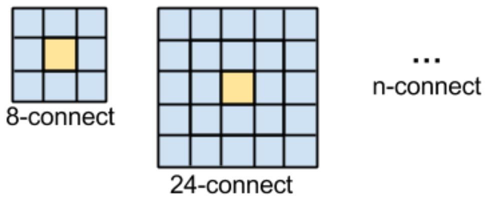
After step 3 we have an image that should tile nicely in the horizontal direction, so step 4 just rotates and repeats in order to provide good tiling in the vertical direction. It is important to note at this point, that a truly optimal algorithm would solve the vertical and horizontal seam at the same time, however due to computational complexity we separated it into two distinct steps. Also note that our output image will have to be rotated back before our algorithm returns.

Experiments

There are a number of things we could have done differently while calculating the seams. There were many tradeoffs we had to decide on, and many things which we would like to implemented when we get more time. Carving the seams was one of the bigger experiments because of the way tessellating tiles worked. The way we calculated the energy of the tessellating image we were given is something else that could be changed. There are also two functions in particular that could increase the overall effectiveness of our program. These functions are finding the minimum tile and verifying that no loss of data occurred.

I Seams

When calculating the minimum energy seams, we implemented it two ways: locally and globally. Calculating the minimum energy seam locally has the bonus of being fast, however it is not guaranteed to be the best seam whereas calculating the seam globally is slower, but it will be the very best seam available. We saw the best results for global seams, and the test images we ran it on had no noticeable slow down due to the the global search. Another area to experiment with came up when we were calculating the seam was how the seam-line was formed. We initially thought the best way to do it would be a vertical 8-connected path of pixels in the image from top to bottom with only one pixel in each row of the image. We quickly realized that if we are given a texture that is already known to be tile-able (i.e. there is a perfect and distinct pattern present in the input image), a straight seam will produce much better results. We tested this idea and found that the straight seams did produce much smoother tiles than the 8-connected seams in the general case. When the input images were not perfect (there was not an exact tile to be found) the 8-connect seam had more potential to produce a good result, however the errors were still very noticeable. An area for further exploration is n-connect seams. instead of limiting our seams movement to 8 pixels, we could try arbitrary amounts (like 24)



II Energy Functions

We calculated the energy using the sum of squared difference technique. Because of this, when we rotate the image to find the horizontal seams we end up losing previously weighted pixels in the final image. A way to mitigate this loss this would be to use a Gaussian weighted sum of squared difference energy function. This would keep pixels in the center of the image at a higher weight (as they are more likely to be in the output image), and limit the loss of weighted pixels during step 4 of the algorithm. Another experiment we wanted to try but were unable due to computational complexity is the have a gradient energy function.

III Image Reduction

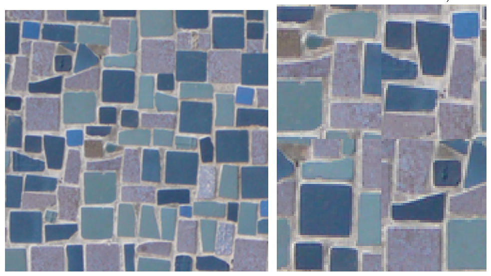
The initial idea behind our project was to make a program that would find the smallest tile possible while keeping the texture tile-able. Originally, however, the smallest possible tile we could produce was 1/4 the size of the input image. We resolved this by running our program through a loop. Our program would find a smaller tile if possible, then compare the error values to previous error values, and once those error values stopped improving we stopped looping and returned the previous tile. This gets us the smallest tile possible if our program is running correctly. Another function we are hoping to add is a function that would compare the original tile's repeating pattern with the new tile's repeating pattern, and determine if they were a perfect match. Obviously, this is fairly easy to determine by eye in most cases, but it would make finding the smallest possible tile easier. With more time we would like to implement a function that did the comparison for us, and would return a boolean to see if we lost data. With a function like this in place we would actually be able to use our project as a compression tool for images! This could be a great utility for companies that handle large amounts of image data.

Results

Our Results can be found on our website, and you can generate your own results using our code (website and code can be found in Work Details Section). There were quite a few interesting results. They can be broken down into the following categories:

1. Non-Tileable - Some images were simple impossible to tile. An example of this is the blue bird MC Escher painting that is on our website. Although there is a distinct and obvious pattern present, our algorithm cannot pick it up. This is because there is no "box" that would produce a perfect tile present in the image. The pattern never fully repeats in the painting, and although our human eye can extrapolate the image, there isn't enough data for our algorithm to work.

2. Seam - Our best results were produced by straight seams. That being said, if a perfect pattern was not present in the input image 8-connect seams did produce better results. The image below is an example of no perfect tile begin present, and the straight seam output. You can see how a better seam could be found, even if it wasn't a perfect seam.



3. Wrong Pattern - Occasionally our algorithm would pick up the wrong pattern. Although the output it produced did tile perfectly the pattern was not what was input into the program. There are multiple examples of this on our website. This is part of the motivation behind wanting to add a function to check if the image produced was a perfect representation of the input image.

4. Perfect tile - It has been alluded to multiple times that an input that contains a perfect tile produces the best results. Not only is the pattern important, however, but the color is too. Our yellow MC Escher painting example (again on our website) is an example of this. Although our algorithm picked up the pattern perfectly Escher's painting had a gradient from top to bottom that produces an obvious artifact in the results.

Work Details

I Approximate lines of code

PHP: 50

HTML/CSS/JS: 300

MATLAB: 350

C(MEX): 150

II Work Per Person

Idea exploration: Phil, Dan, and Ben

Initial Repo Setup/Project Start: Phil and Ben

Website core: Phil and Ben

Website style: Phil and Dan

Matlab code core: Phil

Matlab experiments: Phil, Dan and Ben

Paper: Phil, Dan and Ben

Presentation: Phil and Dan

III Data Locations

Website located at: `http://pages.cs.wisc.edu/~preasa/534`

Examples Images located at: `http://pages.cs.wisc.edu/~preasa/534/output.php`

Code repo (made public on 12/18) located at: `https://bitbucket.org/534team/final-project/`

Conclusion

I Further Work

1. isPerfect function
2. More energy function (Possibly gaussian weighted SSD)
3. n-connect seam
4. more results

II Assessment

Our algorithm seems to solve the problem for almost all images for which a tile is clearly present. Although this is a very small subset of all images, we do have a viable solution. Our algorithm does fail under certain cases, such as the minimal energy providing visually unappealing break in the pattern. Overall, however, we are satisfied with the results and look forward to future research and implementation both into patterned image reduction and patterned image tiling.

Bibliography

[1] Cecilia Aguerrebere, Yann Gousseau, and Guillaume Tartavel. Exemplar-based Texture Synthesis: the Efros-Leung Algorithm. *Image Processing On Line*, 2013:213–231, 2013.

[2] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. *Proceedings of SIGGRAPH 2001*, pages 341–346, August 2001.

Code Contributions: All code was written by the authors specifically for this project save the attributions below:

1. Website skeleton: Boilerplate HTML5 (`http://html5boilerplate.com/`)
2. seam_overlay.m: modified from project 3 code