**Order-Independent Texture Synthesis with Image Parallel Computing Acceleration**

University of Wisconsin-Madison

Jieru Hu

**Abstract**

Recent years have witnessed significant progress in texture synthesis algorithms, but more efforts are needed to develop an algorithm that is both efficient and capable of producing high quality results. This study aims to tackle the challenging task by designing a high-performance pixel-based algorithm for synthesis of images composed by textures. We improve the quality of the algorithm’s generated output by operating the synthesis through multiresolution pyramids. The multiresolution strategy allows a bottom-up, hierarchical representation of images, through which we are able to filter and extract features of convoluted images and apply them to the consecutive output. On each pyramid level, we choose pixels for the synthesis from the input pixels in the sample that share the highest neighborhood resemblance. To further enhance the efficiency of the algorithm, we employ the GPU parallel computing. The intensive computational work is divided through MATLAB’s parallel toolbox and executed on the multiprocessor concurrently. Implications for practice and suggestions for future research are discussed.

**Introduction**

Across various fields of scientific study, such as computational photography, material science, and so on, the question of how to ideally reproduce a large output from a tiny sample input has regularly received attention. My study focuses on the development of a texture synthesis algorithm- a technique of great efficiency and capability, for the purpose of constructing large output images based on an arbitrary user-defined input. This technique has a wide range of applications. For example, it can be used to repair a broken image or to reconstruct damaged materials using textural fragments from the unimpaired parts. Moreover, we can apply this technique on different dimensions by mapping textures onto 2D- or 3D- object surfaces, which can be extremely useful in geometric modeling.

**Related Works**

There have been a great many studies on texture synthesis from a diverse range of subjects including statistics, computer modeling, psychology, image processing, and material science. In the field of psychology, research has examined how the human visual system perceives texture patterns (Balas, 2012). The visual system’s sensitivity to higher-order statistics in natural textures remains acute even after texture synthesis has removed some major perceptual features of the original textures (Balas, 2012). In the area of computer science, researchers and scholars have proposed a number of methods on how to perform texture synthesis targeting different materials, and their efforts have proved successful. Typically, the key of an algorithm is to enforce each local element to preserve the global patterns from the sample input and formulate an efficient way of sampling each unit. One popular approach for sampling presented by Efros and Freeman (2001) is to use a user-specified block of elements as a unit, named as “patch”. Each time, a patch-based searching mechanism is applied to iterate through every location so as to select a patch to properly overlap onto the previously constructed graph. Afterwards, a minimum error boundary is determined to cut through the overlapped region to enable a smooth and natural transition between patches.

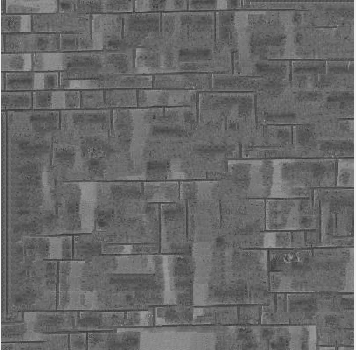
Another prevailing means among all search-based synthesis algorithms is using a pixel-based unit for synthesizing. As the name implies, one single pixel is selected to synthesis at a time. Wei and Levoy (2000) developed an algorithm on pixel synthesis operated on a frame composed by multi-resolution pyramids. Markov random Field is used to model the texture as a random selection process. A specially designed technique called the Tree-structured vector quantization is used to accelerate the selection of the best-matched pixel. Compared with the patch-based algorithm, the pixel-based algorithm can produce more favorable synthesis results, but requires more computational cost to carry out. The architecture of Wei and Levoy’s (2000) algorithm provides a solid foundation for my study. I developed the texture synthesis algorithm between levels of pyramids with different resolutions, which work similarly as the Gaussian pyramids. By synthesizing on a filled graph, we are no longer restricted to the scanline order since the neighborhood of each pixel can be determined. Once the texture synthesis is processed with an arbitrary order, we could make use of GPU parallel computing to break the computational work into parts in order to accelerate the synthesis process.

**Overview**

To describe the texture synthesis algorithm, the first thing is to define what texture is in our technique. The term “texture” comes from a method in computer graphics called “texture mapping,” which involves applying color or bitmap onto a computer-generated model. However, the definition of “texture” in our technique could be much broader than a colored map. The term can be used to describe all kinds of surface quality characteristics from either nature- or computer-generated images with repeated pattern properties (Wei, Lefebvre, Kwatra, &Turk, 2009). In this research, textures are images composed by repeated patterns. Texture synthesis, as the name suggests, is an approach to creating more textures based on the given input. The basic idea of the synthesis algorithm is to begin with a random noise image. Each time we choose the best-fitted unit from the input image and stitch it onto the output graph, eventually filling the output canvas in the scanline order. The discussion of this simple process comprises two parts: the size of the unit and the best objective function to evaluate the next candidate unit.

To determine the best unit for the synthesis algorithm, it is essential to first become familiar with two terms in digital imaging: “pixel” and “patch.” Pixel is the smallest controllable element in an image. In MATLAB, an image is usually stored as matrix of multiple number types. The most common type is (64-bit) double floating numbers. As an example, a commonly used RGB image can be represented by a three dimensional matrix M\*N\*3, where each element in the 2D array corresponds to a color channel vector of length 3. The color characteristic for each pixel is determined by the combination of each numeric component in the color channel. To save memory space and computational cost, images are sometimes stored as integers in the image processing. For this study, we used uin8 integer with each pixel ranging from 0 to 256. RGB images with three color channels are converted into grayscale images to reduce the computing cost.

A patch is a small cluster of pixels combined together under different geometric shapes. In the previous work, I used a patch-based algorithm to reproduce the synthesis. The advantage of using patch instead of pixel as a unit is the significant reduction on computational cost, since one single calculation performed on a large-scale texture can be much faster than multiple unit calculations. Despite the benefits imposed by the use of patch as a unit, the quality of outputs from the patch-based algorithm cannot compete with the one created by the more exhaustive pixel-based synthesis. One reason is that the sampling based on the overlapped edge can only guarantee the smooth transition between patches but fails to represent or predict the locality of the pixels within patches. Logically speaking, in practice, it is usually unreasonable for the user to prioritize reducing costs at the expense of quality. Therefore, patch based synthesis algorithm is not considered as a perfect candidate for high quality texture reproduction.

Sample input Output

Figure 1: Result from L-shaped patch-based texture synthesis. The sample input is an image of a brick with size 192\*192 pixels. The output is an image of a brick with size 353\*353 pixels. Both are gray scaled. This reproduction is synthesized with patch of size 10 and L-shaped overlap region size of 3.

To evaluate each subsequent candidate unit, we employ Markov Random Field, which is a model widely used in physics and probability for describing a set of variables sharing the Markov properties in a network. In texture synthesis, we use it to describe the correlation between pixels. Each pixel belongs to the local neighborhood based on its location on the output graph. Therefore, every pixel is characterized by all the pixels within its neighborhood. The patch-based algorithm I developed previously adopted an L-shaped neighborhood, consisting of the patches towards the left and the top side. For our algorithm, we use a squared neighborhood N (p) to describe the locality of each pixel and the sum of difference (SSD) to measure the distance between each neighborhood.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | **More Pixels to the right** |
|  |  |  |  |  |
|  |  | P |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| **More Pixels on the bottom** | | | | |

N (p)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  | P | **→** | **→** | **→** | **→** |  |
|  | **↓** |  |  |  | **↓** |  |
|  | **↓** |  |  |  | **↓** |  |
|  | **↓** |  |  |  | **↓** |  |
|  | **↓** | **→** | **→** | **→** | **↓** |  |
|  |  |  |  |  |  |  |

Sample input with size 7 The output image after synthesis

Figure 2: Pixel sampling from the input with sample size of 7 and neighborhood size of 3. Each pixel is described by a number of neighbor pixels. We looked at each possible neighborhood in the input and compared its distance to the current neighborhood of the pixel in the output image. In the case above, pixels in the output image is selected from candidates in 5\*5 square of pixels, surrounded by arrow.

**Order-Independent Synthesis**

Although the technique described above lends itself well to generating a valid algorithm for performing texture synthesis, the process can be time-consuming since each candidate pixel requires thousands of times of calculation using the database from the input image. Nevertheless, we could simply improve the efficiency and quality of the original technique by applying the texture synthesis on levels of multiresolution pyramids. The idea is to downgrade the input image in terms of size and operate texture synthesis on levels with increasing size. In each pyramid level i, each input image Pin,i performs texture synthesis with the corresponding output image Pout,i . We start from the basis level with images of the smallest size and proceed to the original-sized level. In each iteration, after finishing synthesizing a level of pyramid, we enlarge the image into the next level with larger size and lower resolution and then continue the texture synthesis process on the enlarged image. The algorithm could significantly improve the running time of the algorithm. This is because at each level, instead of selecting candidate pixels from a relatively large input image, we look through the pixels in a comparatively small size sublevel image after convolution and perform calculation to select the best candidate. Besides, Wei and Levoy (2002) introduced a way to remove cyclic neighborhood dependency: in the process of texture synthesizing, instead of overwriting the old pixels with new ones, the new technique creates a new blank output graph and keeps the old one at the same time. This technique could also be applied to my algorithm when synthesizing at each level to enhance the quality of the image.

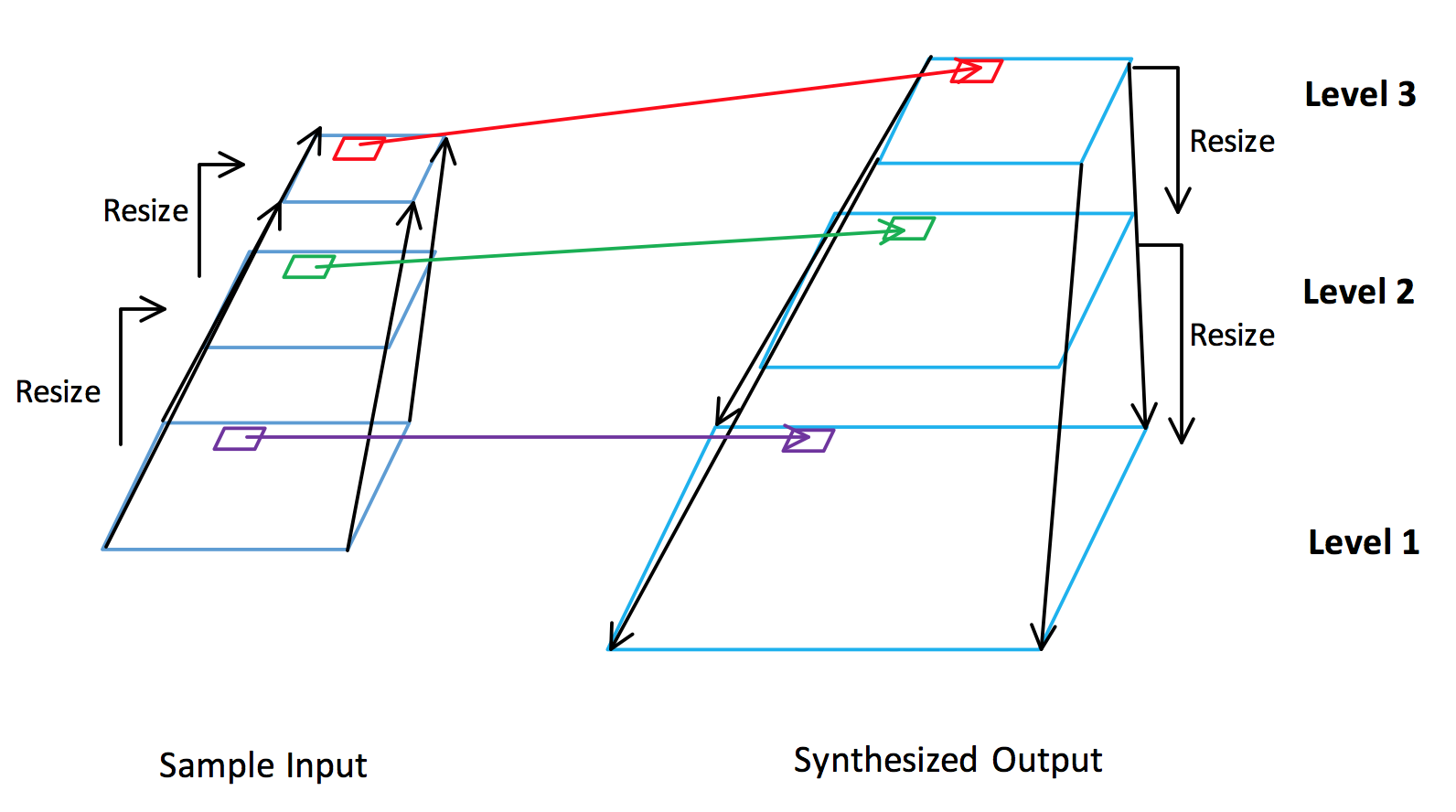


Figure 3: A three-level pyramid model is shown. Each output image level only selects pixels from the corresponding input image pool. Synthesis proceeds from the highest level to the lowest level.

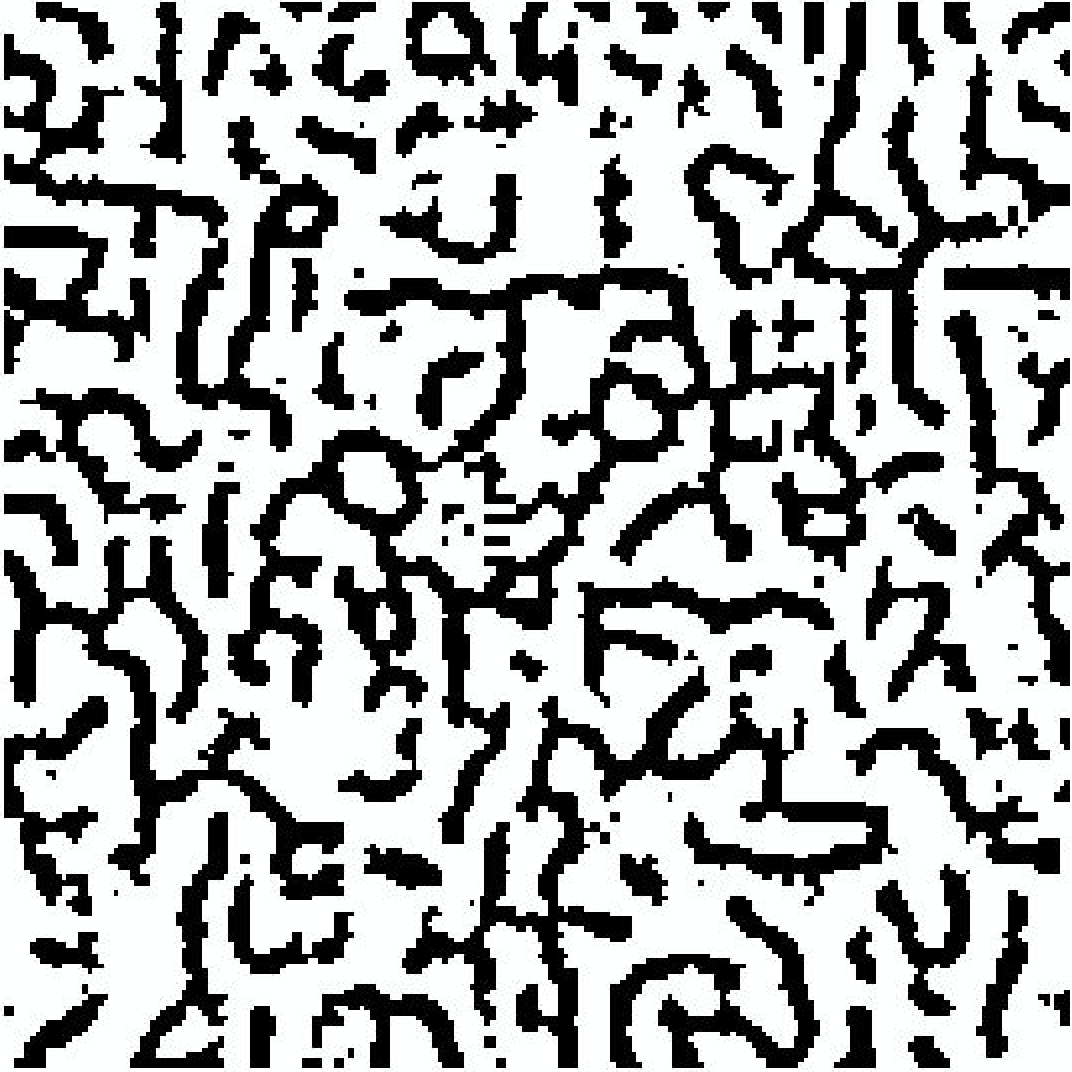
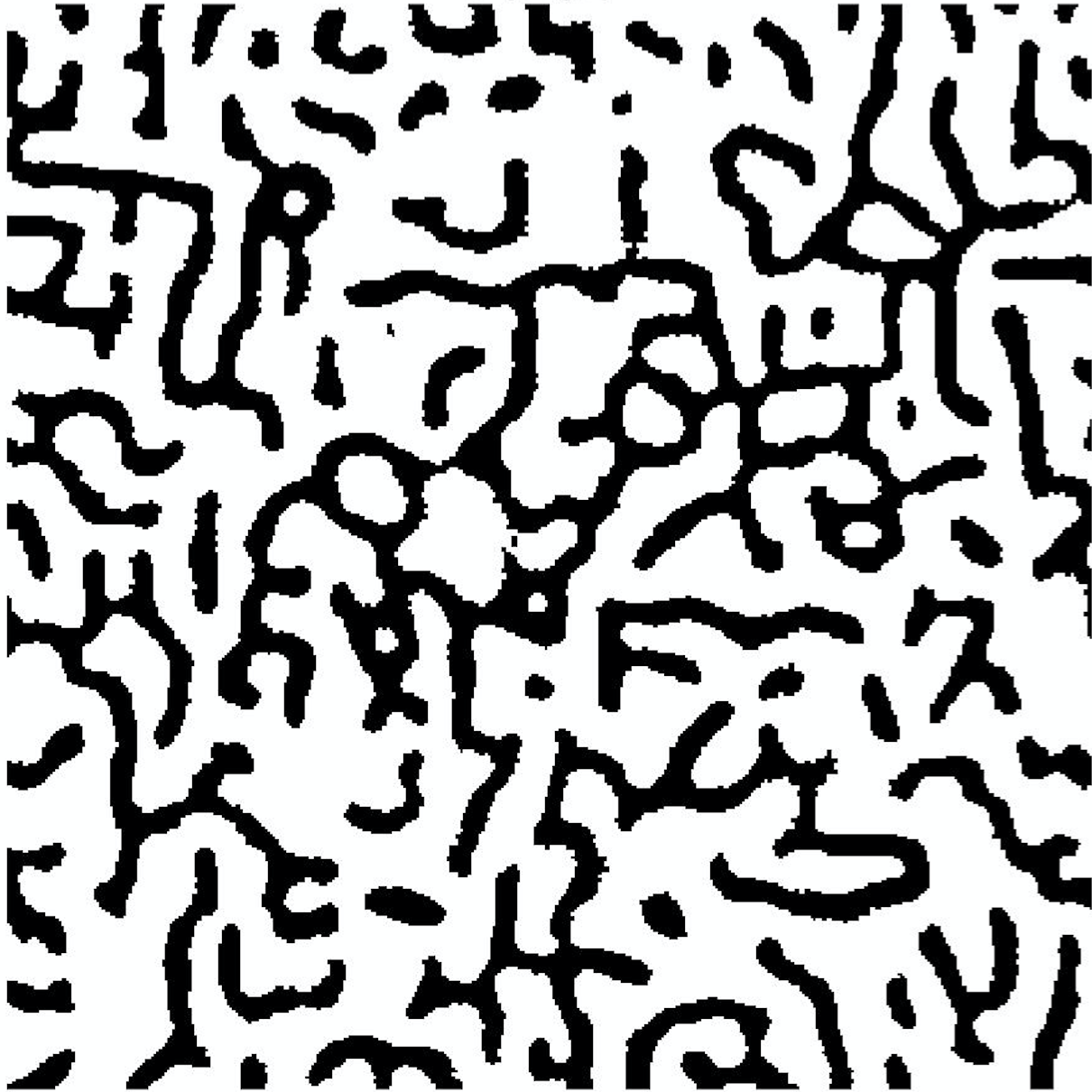
Most rescaling algorithms use interpolation and decimation to compute the newly resized image (Turkowski, 1990). In image processing, interpolation represents a process of determining a numeric pixel based on multiple samples. On the contrary, generating more pixels from a single one is decimation. Different interpolation approaches are introduced by Reddy & Reddy (2013), including the nearest neighbor, linear interpolation, and bicubic interpolation. The mechanisms adopted by these algorithms are similar. Within each neighborhood, the user defines an appropriate filter to compute an output pixel using all pixels in the neighborhood. For example, the simplest mapping could be taking the average of all pixels’ numeric in the neighborhood. Therefore, when a locally distinct cluster of pixel is rescaled into a single pixel or a cluster of pixel is interpolated from a single pixel through some filtering process, biases may emerge locally but the global properties remain comparatively unchanged.

An advantage of operating the texture synthesis on multiresolution levels is that the neighborhood effect on each pixel from higher pyramid level is inherited and magnified as the synthesis proceeds downward. As we use Markov Random Field to describe the relationships between pixels, each pixel shares network connections with all pixels in the neighborhood. As the current synthesized image grows into the next level, the neighborhood size of pixels in the current level also grows with the same factor. For example, in figure 2, each pixel is characterized by a 3 by 3 square neighborhood size. Suppose the output is then enlarged by a factor of 2, the neighborhood size of each pixel grows into a 6 by 6-pixel square. Therefore, the neighborhood of each pixel expands in size but still retains valid connection with the corresponded pixel. With a larger neighborhood size, we could ensure that each pixel is more comprehensively and closely bounded to the whole synthesized image on each level, thus improving the overall quality of the output.

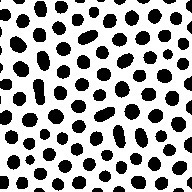
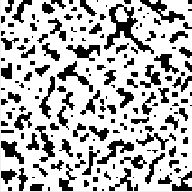
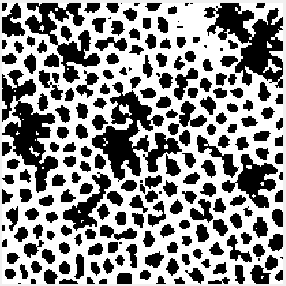
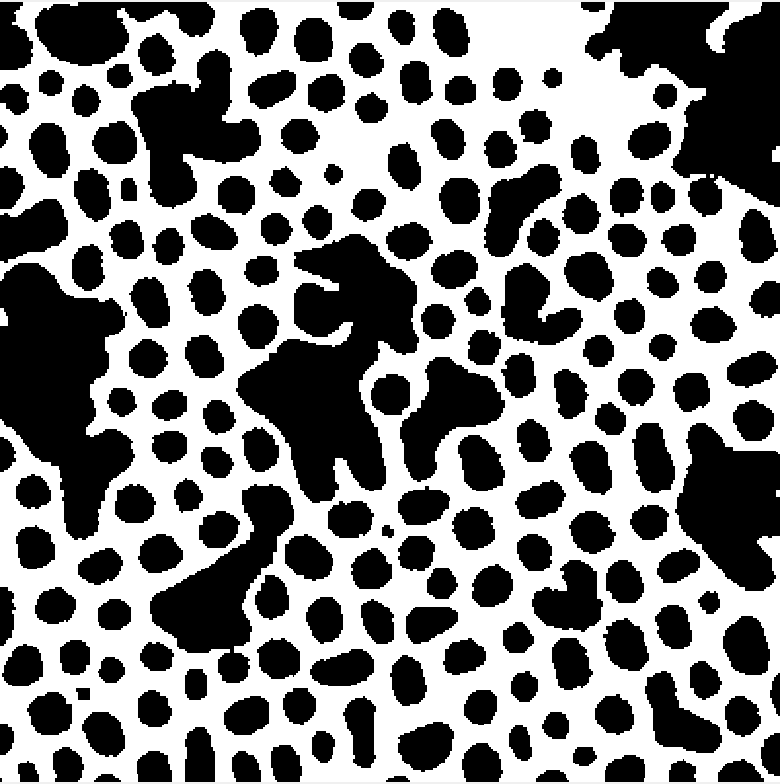
The detailed algorithm implemented in this study reconstructs the two-phase structure on a 2D image. A two-phase structure composes pixels with numerical value equal to either 0 (Pb) or 256 (Pw), corresponding to the color black and white respectively. The algorithm requires all input samples to be converted be gray-scale images before sampling and synthesizing. Starting synthesizing from the highest level of the pyramids, we need to construct an image filled with random noise of black and white pixels. To be more precise, our algorithm takes 200 samples from a number of pixels in the input graph and separates them into two different sets based on their numerical value. We approximate the ratio for the initial noise image by determining the ratio of black pixel and white pixel in the sample.

After we finished each level of synthesizing, a simple filter scans through all pixel elements and resets values in the image matrix:

In this case, a two-phase structure is always maintained after each level of synthesizing.

Input texture Random Noise Level 1 Level 2 Level 3

Input texture Random Noise Level 1 Level 2 Level 3

Figure 4: Synthesis results on different pyramid levels. The two-phase synthesis is processed from a random noise image and executed on three different pyramid levels. Each level is increase by a size factor of two. We did three iterations of texture synthesis on each pyramid level. The input texture size contains 192\*192 pixels. The corresponding size of each level is 100\*100, 200\*200, and 400\*400.

Assume that we are synthesizing an image with size L from a sample input with size A, operated on a neighborhood size of b. Each pixel is determined through calculating the sum of squared differences of A\*A neighborhoods, where each SSD requires B\*B matrix arithmetic calculation. Eventually, we traverse the SSD array and select the corresponding pixel within a proper error range of the minimum value. As the neighborhood size is usually small, we consider each SSD calculation as constant time operation (Turkowski, 1990). Therefore, the total runtime is approximately O (L2\*A2). In some real cases, when the size of the input image is extremely smaller than the output, the algorithm ends up with a quadratic running time O (L2).

The application of similar algorithms and analyses extends to the realm of 3D image reconstruction. The only difference between 2D and 3D reconstruction is that for each pixel selection, we perform arithmetic calculation on 3D shape neighborhoods, such as cubic, octahedron ball-shaped, and other 3D convex hulls. In practice, the computational cost can grow exponentially in higher dimensional image reconstruction. Therefore, we are in need of a more powerful and efficient tool to process the intensive synthesis computations.

**Parallel Computing Acceleration**

To further improve the speed of the algorithm, we could utilize the Graphical Processing Unit (GPU) parallel processing in MATLAB to run the algorithm. GPU is different from CPU in the way that it has hundreds of microprocessors to perform high-speed computing on a parallel array of data. For this study, I run the program in MATLAB based on support for NVIDIA CUDA-Enabled GPUs (Reese & Zaranek, 2011). The benefit of using GPU is straightforward, which is to accelerate the computing speed of the algorithm. However, there are two major requirements for a computing program to be suitable to run on GPU. First, utilizing GPU computing requires users to transfer data back and forth from CPU and GPU at regular intervals throughout the computation. Therefore, to guarantee that GPU is superior to CPU, computation time should be significantly longer than data transfer time between CPU and GPU. In our case, transferring data from the invariably small input image should take an exceptionally shorter time than computing each optimal pixel for the output. The second constraint is that the computation should be divided into thousands of small units of work for GPU to operate in parallel. Our algorithm could work well, because at each level of pyramid the computational work for each pixel is specified by the enlarged output and the corresponding input. Instead of looping through each pixel for the output, we could break the computing task into separate parts and make use of GPU for parallel processing.

Specifically, MATLAB has its own extension named parallel computing toolbox, which takes advantage of the multicore and multiprocessor to solve those computationally intensive problems. The MATLAB parallel toolbox executes the computing session following the procedure in figure 5. In this process, the large computation work from the MATLAB session is called the “job.” At first, the client sends the job to the scheduler for evaluation, after which the separated tasks are distributed to the workers in the parallel working pool. During their lifetime, all workers execute the assigned task simultaneously and they are able to communicate with each other at the same time. The user could set the preferred number of workers for the parallel pool, but the actual working number of workers depends on the available cores in the operating system.

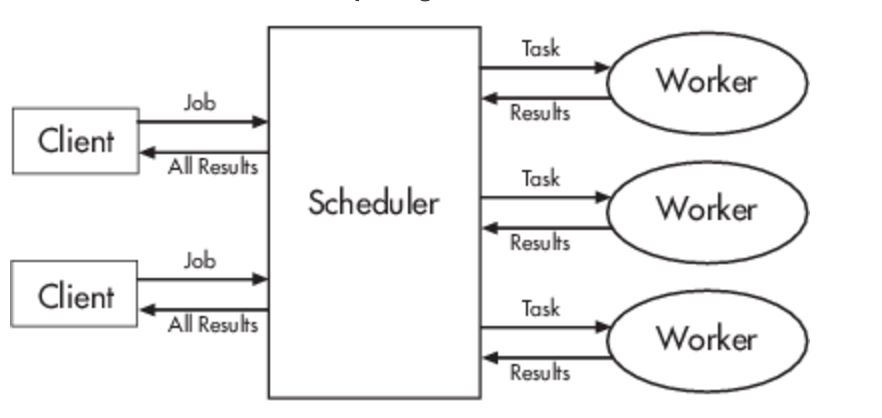


Figure 5. The flow diagram of how MATLAB process the job from the user, and then separate and distribute it to different workers available.

Table 1. Comparison on the time consumed at different part of the texture synthesis

Part Name Time consumed

a Sampling and Initialization 2.9121e-04 s

b Pixel selection and synthesis 4.7350e+03 s

c Others 4.9971 e-01 s

In the time-record test, 192\*192 input texture size is used to generate 400\*400 pixels’ image through 3-level pyramid frame, using a neighborhood size of 5.

As the diagram depicted, the most time-consuming and computationally intensive part in the whole program is the scanline-order pixel selection based on neighborhood-SSD calculation and comparison. The operating time could take even longer time when the output size is larger. We utilize the parallel computing toolbox to divide the computational task of synthesizing texture on each pyramid level to workers in the pool. The reason why parallel computing is applicable to the texture synthesis is based on the nature of the order dependent synthesis. The parallel computing requires that computation outputs have no correlation with each other. Our synthesis model satisfies the requirement because the algorithm performs pixel selection from a previously resized input pixel pool, based on a produced neighborhood. Specifically, parfor loop is used to activate the parallel computing pool to execute the synthesis. Since the functionality of MATLAB’s parfor loop is limited on one-dimensional array, we store all pixels of the image in a one-dimensional array before synthesizing and reshape the linear array to the 2-dimensional array to retrieve the image as soon as the synthesis ends at each level. As the time recorded from the MATLAB shows below, the execution time of synthesis is significantly reduced with the help of the parallel computing toolbox.

Figure 6. Run time difference on synthesis with and without parallel computing. Three output images with their sizes ranging from 100, 200 and 400 pixels are produced with and without parallel computing toolbox in MATLAB. Parallel computing is Time consumed for synthesis is recorded using tic and toc commands in MATLAB.

Based on the comparison in figure 6, we could conclude that parallel computing greatly improves the speed of the texture synthesis. Besides, as the size of the output reproduction increases, the accelerating effect brought by parallel computing becomes even larger. To reproduce a 400 \* 400 pixels output, the texture synthesis with parallel computing only consumes of the time required by the synthesis with no acceleration. Therefore, the parallel computing acceleration proves to be successfully employed on the large-scale texture synthesis because it is efficient and easy to implement.

**Future work**

Texture synthesis has huge potential to grow into different directions. So far we have analyzed the synthesis algorithm on homogeneous input images. However, in real life, we are more likely to process images with variant patterns. Zhang and colleauges (2003) from Microsoft research introduced a type of texture called progressively-variant texture: the texture with locally constant pattern in small domain changes smoothly on a global scale. The variation on the texture property could range from direction, density, color, etc. Ideally, these changing properties could be captured by user control. This could be a huge implication for the industry, because we could not only reproduce massive output based on a single input, but also create numerous kinds of textures with widely applicable properties.

Similar techniques of texture synthesis could also apply to the reconstruction of material. Liu and Shapiro (2015) were the ones who put forward the notion of the material descriptor. Different material properties enclosed by the manually defined unit space, such as 3D- convex hull, could all be represented as the material descriptor. These include volume fraction, stiffness, conductivity, and some other useful properties. Processing the collection of material input is similar to synthesizing images with higher dimensions. However, it is not easy to model a material while retaining all its properties. For example, while reconstructing a heat transfer model of a material, the high inter-relatedness between some properties, such as conductivity, density, and heat capacity, could make it difficult to predict each microstructure. Moreover, determining boundary conditions between each unit space can be computationally costly for some properties such as stiffness (Liu & Shapiro, 2015).

**Conclusion**

To sum up, our algorithm uses the notion of neighborhood for describing the property and the locality of each pixel to perform synthesis through multiresolution pyramids. The whole process is accelerated by MATLAB GPU’s parallel processing. The algorithm produces a technique that has huge growth potential in the future. A possible future direction is imposing some user control on the output to change some aspects of the texture property, such as direction, density, color etc. We could even extend the application of the algorithm to synthesis of moving textures such as water flow and moving sand. Similar algorithms could also advance techniques for mapping 2D textures onto 3D model surface, which could make a substantial impact on industrial production.

**References**

Balas, B. (2012). Contrast negation and texture synthesis differentially disrupt natural texture appearance. *Frontiers in Psychology, 3*. <http://dx.doi.org/10.3389/fpsyg.2>  
 012.00515

Efros, A. A., & Freeman, W. T. (2001). Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques in SIGGRAPH '01: *Image Quilting for Texture Synthesis and Transfer.* New York, NY: ACM.

Liu, X., & Shapiro, V. (2015). Random heterogeneous materials via texture synthesis. *Computational Materials Science. 99*, 177-189. <http://dx.doi.org/10.1016/j.com>  
 matsci.2014.12.017

Reddy, K. S. & Reddy, K. R. (2013). Enlargement of image based upon interpolation techniques. I*nternational Journal of Advanced Research in Computer and Communication Engineering, 2*(12), 4631-4640.

Reese, J., & Zaranek, S. (2011). GPU Programming in MATLAB. *Mathworks.*

Turkowski, K. (1990). Filters for common resampling tasks. In Glassier, A. S. (Ed.), *Graphics gems* (pp. 147-165). Cambridge, MA: Academic Press Inc.

Wei, L., & Levoy, M. (2000). Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques in SIGGRAPH ’00: *Fast Texture Synthesis Using Tree-structured Vector Quantization.* New York, NY: ACM Press/Addison- Wesley Publishing Co.

Wei, L., & Levoy, M. (2002). Order-Independent texture synthesis. *Technical Report TR- 2002-01.*

Wei, L., Lefebvre, S., Kwatra, V, & Turk, G. (2009). Eurographics 2009, State of the Art Report, EG-START: *State of the Art in Example-based Texture Synthesis.* Munich: Eurographics Association.

Zhang, J., Zhou, K., Velho, L., Guo, B., & Shum, H. (2003). Synthesis of progressively- variant textures on arbitrary surfaces. *ACM Trans. Graph. 22*(3), 295-302.