Texture Synthesis Using Graph Cuts

Ping Liu

Master of Science

Computer Animation and Visual Effects

Bournemouth University

Friday, August 29, 2014

Acknowledgment

I would like to start by recognizing the great education I have received in this institution and thank the NCCA staff for all the knowledge I have received during this year of studies which will help me grow in the industry. More specifically I want to express gratitude to the Professor Jon Macey, who gave me a lot of insight into the world of programming and scripting in CG. Mathieu Sanchez, who has been there for all of us during this year to help us with any problems and concerns during our assignments.

Contents

1. Introduction

Computer graphics is a very diverse field of research with many applications, including film, and visual effects, advertising, car and flight simulators, architecture, scientific simulations and computer games. These applications are the driving force behind computer graphics and the continuous demand for more quality and complexity in digitally synthesized images.

1.1 Main Goal

In [Kwatra, SIGGRAPH 2003], the authors introduce a new algorithm for image and video texture synthesis, as the later is only a generalization to a third dimension of the image process. Practically, we respectively want to build a large texturized area from a small pattern, and a longer video sequence from a short sample. The optimization of the process obviously consists in minimizing the visual inconsistencies. They also suggest to apply the algorithm to interactive merging and blending, i.e. to smoothly include a sample of an image in another.

The general idea is the following: we first select an offset to place the small pattern in the large output object, and then compute the parts that will finally be copied. In other words, we cut the input pattern in a way that optimizes the smoothness of the differences between the background and the new elements added. That is the essence of the algorithm, that we will describe in the rest of this document.

This project focuses on the 2D case, and does not provides implementation for video synthesis. Our goal is to produce an efficient implementation, while reaching results as good as the authors' ones, and trying to give some improvements. In addition, this project is very easy to use and install, since no special strange software or library is needed to build it from sources.

1.2 Terminology

Some recurrent terms will appear in the following, we briefly clarify their meaning according to our project:

- The **patch** is the input data given by the user, that is the small image we want to build a texture from,

- A **seam** is a cut around the patch, once it's placed, to select some pixels, and ignore some others,
- The **max-flow problem** in a graph consists in finding the path that max- imizes the flow given by the edges labels, it's equivalent to
- The **min-cut problem**, in which we want to divide the graph into two parts (the Source and the Sink parts), while minimizing the sum of the labels of the edges crossed. We'll see that finding the min-cut in a particular graph built from the images can help us to find a seam optimizing the smoothness of the border. That is, we'll look for the best graph cut.

We may also talk about input image, which is the given patch, and out-put image, which is the large texture generated, and, unless the process ends, may contain “empty” pixels, displayed as black ones on screen. The meaning of these words is sometimes confusing in the original article, since the target objects are not always the ones we ought to think to.

1.3 About the implementation

This project is programmed in C++ language, for any *nix system, thus all you need is a C++ compiler, you do not even require any image processing library.

So as to find the graph cut we need to compute the max-flow/min-cut, the authors (Y.Boykov, V.Kolmogorov) who first applied graph cuts to image processing produced a very efficient free implementation (BoykovMF,1995) of the max-flow computation, that we chose to reuse in this project.

The image processing interface we chose is CImg (TschumpCImg), the unique C++ header file is included in our project sources, thus no installation is needed for the user. It is portable to any common OS and the best (and the worst) point is that the whole CImg sources is only in one file of 600Kb.

2. Related Work

Texture synthesis techniques that generate an output texture from an example input can be roughly categorized into three classes. The first class uses a fixed number of parameters within a compact parametric model to describe a variety of textures. Heeger and Bergen (1995) use color histograms across frequency bands as a texture description. Portilla and Simoncelli's model (2000) includes a variety of wavelet features and their relationships, and is probably the best parametric model for image textures to date. Szummer and Picard (1996), Soatto et al. (2001), and Wang and Zhu (2002) have proposed parametric representations for video. Parametric models cannot synthesize as large a variety of textures as other models described here, but provide better model generalization and are more amenable to introspection and recognition (Saisan et al. 2001). They therefore perform well for analysis of textures and can provide a better understanding of the perceptual process.

The second class of texture synthesis methods is non-parametric, which means that rather than having a fixed number of parameters, they use a collection of *exemplars* to model the texture. DeBonet (1997), who pioneered this group of techniques, samples from a collection of multi-scale filter responses to generate textures. Efros and Leung (1999) were the first to use an even simpler approach, directly generating textures by copying pixels from the in- put texture. Wei and Levoy (2000) extended this approach to multiple frequency bands and used vector quantization to speed up the processing. These techniques all have in common that they generate textures one pixel at a time.

The third, most recent class of techniques generates textures by copying whole *patches* from the input. Ashikmin (2001) made an intermediate step towards copying patches by using a pixel-based technique that favors transfer of coherent patches. Liang et al. (2001), Guo et al. (2000), and Efros and Freeman (2001) explicitly copy whole patches of input texture at a time. Schodl et al. (2000) perform video synthesis by copying whole frames from the input sequence. This last class of techniques arguably creates the best synthesis results on the largest variety of textures. These methods, unlike the parametric methods described above, yield a limited amount of information for texture analysis.

Across different synthesis techniques, textures are often described as Markov Random Fields (DeBonet 1997; Efros and Leung 1999; Efros and Freeman 2001; Wei and Levoy 2000). MRFs have been studied extensively in the context of computer vision

(Li 1995). In our case, we use a graph cut technique to optimize the likelihood of the MRF. Among other techniques using graph cuts (Greig et al. 1989), we have chosen a technique by Boykov et al. (1999), which is particularly suited for the type of cost function found in texture synthesis.

3. Graph Cut Theory

A flow network $G(V,E)$ is formally defined as a fully connected directed graph where each edge $(u,\upsilon)\in E$ has a positive capacity $c(u,\upsilon)\geq 0$. Two special vertices in a flow network are designated the source *s* and the sink *t* respectively. A flow in G is a real-valued function $f: VXV \rightarrow R$ that satisfies the following three properties:

• Capacity Constraint:

For all $u,\upsilon \in V$, $f(u,\upsilon) \leq c(u,\upsilon)$.

• Skew Symmetry:

For all $u,\upsilon \in V$, $f(u,\upsilon) = -f(\upsilon,u)$.

• Flow Conservation:

For all $u \in (V-\{s,t\})$, $\sum_{\upsilon\in V} f(\mathrm{u},\upsilon) = 0$.

The value of a flow is defined as $|f| = \sum_{\upsilon\in V} f(\mathrm{s},\upsilon)$, the total flow out of the source in the flow network *G*.

3.1 The Max-Flow and Min-Cut Problem

The max-flow problem is to find the flow maximum value on a flow network *G*. A *s-t cut* or simply *cut* of a flow net- work *G* is a partition of *V* into *S* and *T* = *V* −*S* such that $s\in S$ and $t \in T$. For a given flow *f*, the *net flow* across the cut (*S*, *T*) is defined as $f(S,T) = \sum_{x\in S}\sum_{y\in T} f(\mathrm{x},\mathrm{y})$. Using a similar notation the capacity of a cut (*S*, *T*) is defined as $c(S,T) = \sum_{y\in T} c(\mathrm{x},\mathrm{y})$. A *minimum cut* of a flow network is a cut whose capacity is the least over all the s-t cuts of the network.

An example of a flow network with a valid flow is shown in Figure 1.

Theorem 1 *The max-flow min-cut theorem: If* f *is a flow in a flow network* G = (*V*, *E*) *with source* s *and sink* t *then the value of the maximum flow is equal to the capacity of a minimum cut. Refer to Cormen et. al. (1990) for the proof.*

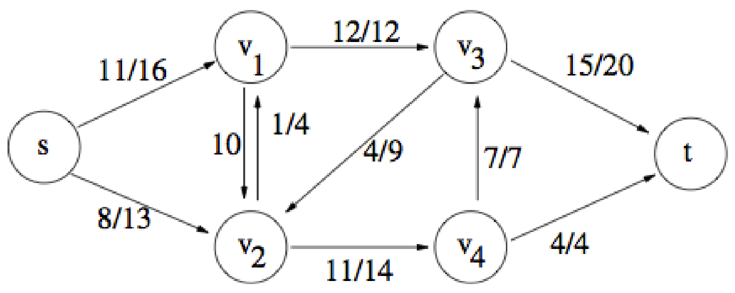


Figure 1: (a) The figure (taken from Cormen et. al. (1990) shows a flow network G(V,E) with a valid flow f. The values on the edges are f(u,v)/c(u,v). The current flow has value 19, it is not a maximum flow.

The intuition behind the proof is as follows. The maximum flow must saturate edges in the flow network such that no further flow can be pushed. These saturated edges must lie on one of the min-cuts. This result allows one to compute the minimum cut of a flow network by first solving for the max-flow, for which polynomial time algorithms exist.

The single-source single-sink max-flow problem described above is a specific case of the more general multi way cut problem where there are *k* terminals and a multi way cut is a minimum set of edges which separates each terminal from all the others. It has been shown that if $k \geq 3$, the problem is NP-Hard. Some of the graph cut applications that we shall investigate in this paper will indeed require approximation algorithms for the multi way cut problem.

3.2 Max-Flow and Min-Cut Algorithms

The polynomial algorithms for the single-source single-sink max-flow problem can be divided into two classes, algorithms based on the Ford Fulkerson method (1962) and those based on the "push relabel" method (1988). The two contrasting approaches are described below.

The intuitive idea behind the Ford-Fulkerson method is that starting with zero flow ie. $f(u,\upsilon)=0$ for all $u,\upsilon \in V$, the flow can be gradually increased by finding a path from *s* to *t* along which more flow can be sent. Such a path is called an augmenting path, and once it has been found, the flow can be augmented along this path. The process if repeated, must end after a finite number of iterations after which no augmenting paths between s and t exist anymore. A typical algorithm of this type maintains for a given flow f, the residual graph of G, called Gf whose topology is identical to G but whose edge capacities stores the residual capacity of all the edges, given that there is already some flow in them. The search for an augmenting path at the i^{th} iteration is done on the current residual graph Gf_i . Once an augmenting path is found, the maximum amount of flow that can be sent down it, $fincr$ must saturate at least one of the edges of this augmented path. The new flow at the end of the iteration will be $f_i + fincr$.

The running time complexity of different algorithms will in general vary depending on how the augmenting path is chosen. Dinic algorithm that uses breadth-first search to find the shortest paths from s to t on the residual graph, has an overall worst case running time of $O(n^2m)$, n being the number of nodes and m being the number of edges.

In contrast to the Ford-Fulkerson method where augmenting the flow operates on the complete residual graph, the Push-Relabel algorithms operate locally on a vertex at a time, inspecting only its neighbours. Unlike the Ford- Fulkerson method, the flow conservation property is not satisfied during the algorithm's execution. The intuitive idea here is to associate a notion of height along with all the nodes in the network. The height of the source and sink are fixed at $|V|$ and 0 respectively while at the start all other vertices are at height 0. The algorithm starts by sending flow down from the source and the amount of flow sent, saturates all the outgoing edges. All intermediate nodes have a buffer or a reservoir that can store excess flow. Nodes with positive excess flow are said to be overflowing nodes. Overflowing nodes try to push the excess flow downhill. However when an overflowing node finds the edges to its neighbours at the same height as itself saturated, it increments its own height, a process which is called "relabeling". This allows it to get rid of the excess flow. The algorithm terminates when none of the nodes in V are overflowing. Often excess flow accumulated in the interior nodes are sent back to the source by relabeling these nodes with height beyond $|V|$.

The generic push-relabel algorithms thus have two basic operations - "push" flow and "relabel" an overflowing node and Cormen et. al. proves that a generic push-relabel style algorithm has a $O(n^2m)$ worst case running time, and there are certain $O(n^3)$ algorithms in this class. Goldberg and Tarjan provide details on these various algorithms, data structures chosen and practical trade-offs encountered in actual max-flow implementations.

3.3 Markov Random Fields

This section briefly introduces the theory of Markov Random Fields (MRF) as it will be relevant in understanding the common thread between the three papers chosen for study. Markov Random Fields is a generative model often used in Image Processing and Computer Vision to solve labeling problems. A Markov Random Field consists of three sets, a set S of sites, a neighbourhood system N and a set (also called field) of ramdom variables F. The neighborhood system $N = \{ N_i \mid i \in S \}$ where each N_i is a subset of sites of S which form the neighbourhood of site i. The random field $F = \{ F_i \mid i \in S \}$ consists of random variables F_i that take on a value f_i from a set of lables $L = \{l_1, l_2, ...\}$. A particular set of labels, often denoted by f (which can be thought of as the joint event $\{F_1 = f_1, F_2 = f_2, ...\}$) is called a configuration of F. The probability of a particular configuration f ie. $P(F = f)$ must satisfy the Markov property in order for F to be a Markov Random Field.

$$P(f_i \mid f_{S-i}) = P(f_i \mid f_{N_i}), \ \forall i \in S \,. \tag{3.1}$$

This means that the state of each random variable F_i depends on the state of its neighbours, ie. $F_{N_i} = \{ F_i \mid i \in N_i \}$. However it has also been shown that the probability of a particular configuration is proportional to the sum of *clique potential* V_C over all the cliques in N. The *clique potential* is obtained from prior probabilities of a particular labeling of the sites in the clique C.

Markov Random Fields are used to model labeling problems where an optimal labeling is desired. From a probabilistic perpective, one wishes to estimate the configuration f based on observed data D (could be noisy or incomplete) that maximises the likelihood function, $P(D|f)$. Using Bayes Theorem, this likelihood function can be expressed as an energy function $E(f)$ and the maximum a posterieri (MAP) estimate of f should maximize this energy function.

Different MRF's differ in a choice of the neighbourhood system and the prior probabilities. In vision and image processing, where S, the set of sites often coincides with the set of regular grid of pixels and voxels, 4-neighbourhood or 8-neighbourhood systems on a 2D grid or 6-neighbourhood or 26-neighbourhood systems on a 3D grid are common. Moreover, often a labeling with the following properties is desired; it should be locally constant or smooth but should also allow for discontinuities (at region boundaries). When one chooses clique potentials to make the desired labeling piecewise continuous, the resulting MRF is called Generalized Potts Model MRF (GPM-MRF). This formulation is useful for Vision and will be discussed in the next section. For more details on MRF's, refer to (HEEGER ,1995).

The problem of texture synthesis in graphics (Kwatra, 2003) can also be modeled as a MRF where sites of the MRF would typically be pixels or group of pixels in the output texture and the clique potentials in the MRF would depend on the similarity of pixel neighbourhoods in the input texture. The goal here is to generate an output texture perceptually similar to the input texture. Perceptual Similarity is essentially a labeling and grouping problem and is hence well modeled by MRFs.

4. Patch Placement

4.1 General Idea

The first operation in the process iterated for texture synthesis is the position- ning of the patch, that is a translation of the top left corner of our pattern, which must be rectangular. If the pattern is not a rectangular one, we may pad the empty areas by black or white pixels so as to it in the surrounding rectangle, then the algorithm should work by cutting the padded areas.

To measure the fitness of a translation, we'll consider the overlap between the patch and the existing pixels of the output image

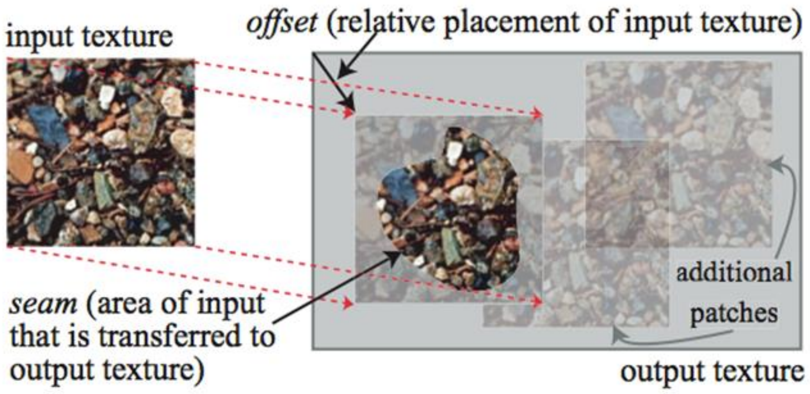

input texture
offset (relative placement of input texture)
seam (area of input that is transferred to output texture)
additional patches
output texture

Figure 2: Image texture synthesis by placing small patches at various offsets followed by the computation of a seam that enforces visual smoothness between the existing pixels and the newly placed patch.

4.2 Algorithms

4.2.1 Random Placement

The first and simplest algorithm consists in picking a uniform random position of the patch on the texture. That is, we randomly choose the two coordinates of the top-left corner of the patch, which can be seen as offsets of a translation from the (0, 0) position on the texture.

4.2.2 Entire Patch Matching

This algorithm is a bit more greedy than the previous one, we look for a trans- lation minimizing the normalized sum-of-squared differences (SSD) of the pixels amplitudes. As we work on color images, we consider the mean of this value on each color channel. Formally, the cost function can be written as

$$C(t) = \frac{1}{|A_t|} \sum_{p \in A_t} |I(\mathrm{p}) - \mathrm{O}(\mathrm{p})|^2 \tag{4.1}$$

where A_t is the overlapped area (and so $|A_t|$ the number of pixels in it), p a pixel in A_t, $I(p)$ the corresponding pixel in the patch, and $O(p)$ the one in the current texture synthesized. Note that here p does not stands for any explicit coordinate.

Then, as written in [Kwatra, SIGGRAPH 2003], we shall compute this cost for all possible translations, and randomly select one of those according to a probability function depending on the cost of the translations.

In practice, computing all translations' cost is very expensive in time, and we decided to modify the procedure: we simply select translations randomly, computing their cost, and finally electing the one with the least cost. The number of iteration is fixed before. For a certain number of iterations, we obtain results looking as good as the ones we had with exhaustive search, but with a huge time gain.

4.2.3 Bad Idea Tested: Alternative Sub-patch Matching

An original placement algorithm tested, which aim was to reduce time complexity, is now introduced: instead of looking for an optimal position of the full patch, we first randomly pick a sub-area of it, and search a good location like for the classical entire patch matching. Since the max-flow algorithm does not have a linear time complexity, by computing many small areas cost instead of full patch we shall gain some precious time. In practice, we indeed decrease the time needed to compute a full texture, but the render is much worse than before: as we have many small patches, it statistically increases the risk of “discontinuity” between the new sticked patch and the existing texture, and hence we finally obtain a bad texture. This algorithm is not part of the final implementation.

4.2.4 Sub-patch Matching

This is the default algorithm used by the program. Instead of looking for trans- lations on the whole texture, we only focus on a smaller sub-part of it. By reducing this area we increase our chances to find a good translation. In the original article, the cost function given is not normalized as the entire patch matching one is. As it does not change the results, we prefer to normalize it, and as before, we do not look for all possible translation, but for a fixed given number, lower than the total number of translations, and finally choose the best one.

$$C(\mathrm{t}) = \sum_{p \in S_0} \left| I(\mathrm{p}-\mathrm{t}) - \mathrm{O}(\mathrm{p}) \right|^2 \tag{4.2}$$

4.3 Improvements And Programming Tricks

4.3.1 Pre-computation

We see that the sum-of-squares matching function implies many computations of bytes' squares. We can do only 256 operations instead of several thousands at each iterations, by pre-computing a table of the squares from 0 to 255.

4.3.2 Automatic Function Switch

When reaching the end of the synthesis process, only a very few pixels of the output image remain empty, and we may wait a long time before filling them with the entire and sub-patch matching functions. Therefore our implementation has an option, which is not set default, to automatically switch from one function to another when no advance is done during a certain time. In fact, if the function is the sub-patch matching one (default function), it allows the program to switch to the entire matching function after a given time with no new pixel filled. Indeed, entire matching has more chances to fill the remaining pixels as it does not first discriminate over a subarea of the texture. Then if we still do not fill the entire texture, the implementation switches to random placement, which quickly manages to end the process.

We shall note that this process is only made to speed up the program, and not a solution to a never-ending-algorithm, since the process theoretically converges (it can always finish the texture), but this may take a (very) long time, as the placement is stochastic.

4.3.3 Overlap Ratio

Another detail we decided to add is a minimal treshold for the size of the overlapping area, in order to have a minimal real correspondance between the two images, and not only three or four same pixels which may false the results. We call this value the ratio, it is defaultly set to 5% (0.05).

4.3.4 Mask And Dimensions

We can also note that the program uses in memory a mask image, which has the same dimensions as the output texture, and is used to know which pixels are empty and which are not. We cannot use the fact that a pixel on the texture is still at (0, 0, 0) (initial value) in the case of pixels filled in black.

The initial dimension of the texture is set to three times the patch one, in both width and height.

4.3.5 Help Entire Patch Matching To Fill Empty Areas

So as to fill some empty pixels at each iteration, and not to wait to long to finish synthesis when only a few empty pixels remain, we decrease the cost of a placement when it helps to fill some pixels. This simple tip really speed up the process, and we don't wait one or two minutes to complete the job anymore. In fact, we ignore placement which fully overlap while we have not finished the synthesis, and allow them during the refinement step.

5. Seam Computation

5.1 General Idea

The principle is quite simple, we need to:

1. Build a graph whose vertices are overlapping pixels, and edges are created between each neighbour, labelled with a cost function, and also with two additional nodes, Source and Sink, such that pixels on the border close to the patch are linked with the Source, and those close to the texture are linked with the Sink,

2. Compute a min-cut in this graph,

3. Only copy to the output image the pixels belonging to the Source part of the graph.

We will also account for old seams, by adding some nodes in the graph corresponding to the previous cuts, so as to improve the correction of areas where we may observe inconsistencies. In fact we want to minimize the final number of "visible" seams, as we can see on the seam image displayed at the end of the process.

5.2 Particular Cases

5.2.1 Several Overlap Cases

A problem faced in the construction of the graph occured when we have several isolated overlap areas. In this case we can think that several graphs are needed, to find the best cut on each one, but it's not. It is easy to see that the problem is solved the same way if we only build one large graph, since the overlapping parts are not linked together, but all with the Source and Sink.

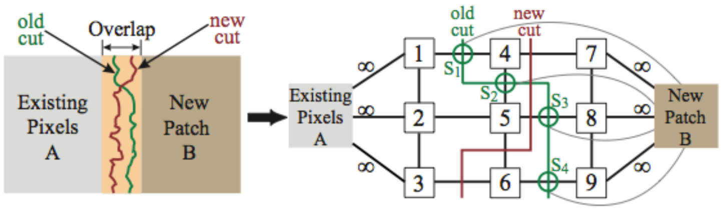


Figure 3: (Left) Finding the best new cut (red) with an old seam (green) already present. (Right) Graph formulation with old seams present. Nodes s_1 to s_4 and their arcs to **B** encode the cost of the old seam.

5.2.2 Case Of Surrounded Regions

When the full patch overlaps the output image, we have to notice that no pixel may be linked with the source, and that the source part of the graph can be a set of nodes inside the graph, and not, as the classical examples of graph-cut show, two blocks on the left and right, each one standing for the Source or the Sink part. This remarkable property allows the algorithm to refine a fully filled texture by replacing some pixels by other ones.

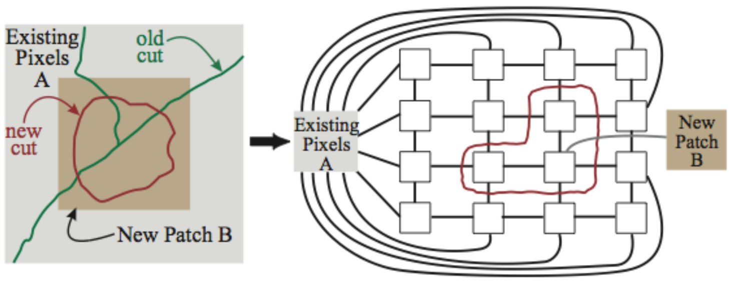


Figure 4: (Left) Placing a patch surrounded by already *filled* pixels. Old seams (green) are partially overwritten by the new patch (bordered in red). (Right) Graph formulation of the problem. Constraint arcs to **A** force the border pixels to come from old image. Seam nodes and their arcs are not shown in this image for clarity.

5.3 Graph Building

The graph is built as described in (Kwatra,2003), including the accounting of the old seams, so as to reduce discontinuities. The implementation was quite easy but tedious, since we have to manage several cases for each node (pixel), and look for:

• A top neighbour, in which case we compute the cost of the edge, using one of the cost functions below,

• A left neighbour, idem.

We must decide whether the pixel has to be linked to the SOURCE or the SINK, for that we use some simple tests to know if the pixel is located on the border of the patch, or on the one of the texture. Then we add an infinite valued node to the source or the SINK, respectively if the pixel is on the boundary between the overlap and the applied patch, or on the one close to the texture synthesized.

We also look for previous seams between the current pixel and its top and left neighbours, if there are some we only add them to the graph, following the description of (Kwatra, 2003).

5.4 Cost Functions

The cost function must be a measure of the color difference between two pixels, to label the edges between neighbours in the graph. A(x) and B(x) respectively stands for the color of the pixel x in the patch and in the output texture. As we only consider overlapping pixels, B(x) cannot be empty.

5.4.1 Basic Function

The naive function, the simplest too, is the following:

$$M(s,t,A,B) = |A(s) - B(s)| + |A(t) - B(t)| \tag{5.1}$$

Also it can be described as the following figure:

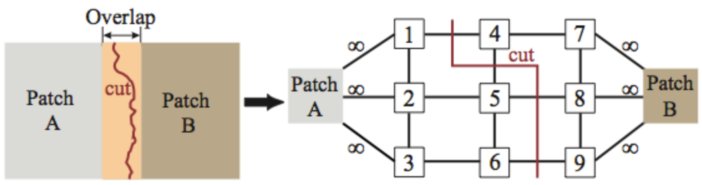


Figure 5: (Left) Schematic showing the overlapping region between two patches. (Right) Graph formulation of the seam finding problem, with the red line showing the minimum cost cut.

5.4.2 Optimized Function

This function takes into account the fact that discontinuities or seam boundaries are more prominent in low frequency areas than in high frequency ones. By dividing by the gradient, we'll decrease when the frequency grows:

$$M'(s,t,A,B) = \frac{M(s,t,A,B)}{|A(s)-A(t)|+|B(s)-B(t)|} \tag{5.2}$$

5.5 Improvements And Programming Tricks

5.5.1 Saving Node Numbers

While building the graph, we need to remember which node corresponds to a given pixel. For that, we use the mask image again, by saving the node number +1 at each pixel position, so as to retreive the node after min-cut computing. If the node does not belong to the Source part, the value is reset to zero.

5.5.2 Reduce Infinite Value

When adding infinite valued edges to the SOURCE or SINK nodes, one should not use the maximal value, since the data type has limited capacity (we use 32bits signed integers). We choose to use 1/4 of the maximal value allowed, that is 65536/4 = 16384.

5.5.3 Cost Reduction

One of the arguments of the cost functions is the cost reduction, which is by default set to 20, its goal is to reduce the value of the cost computed, so as not to overcome the data type capacity. In fact, the default value give costs between 1 and 20, which are large enough for a correct graph cut computation, and small enough to stay under the maximum value allowed.

5.5.4 Seams Display

In addition to the patch and the texture, our implementation also displays the seams of the final results.

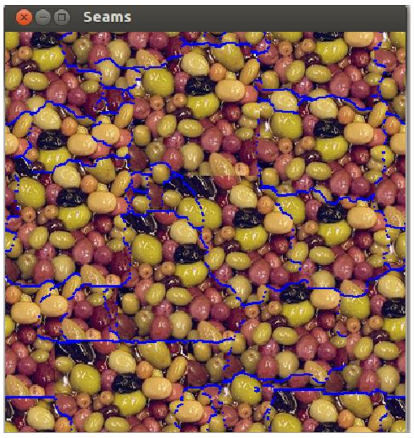


Figure 6: Seam Disply

5.5.5 Tileable Texture

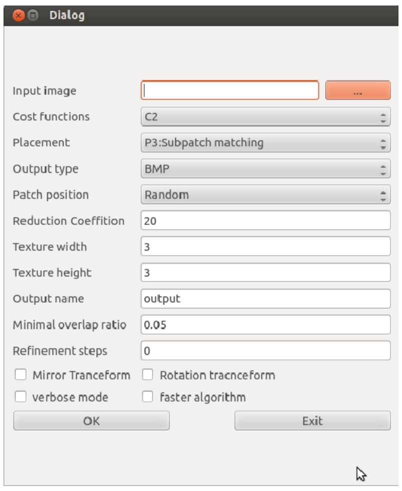
The textures are synthesized such that they can be used as mosaic (e.g. for a wallpaper) with minimized discontinuities, since the image is viewed as a sphere during the computation.

5.5.6 Refinement

The refinement steps come after the whole texture is synthesized, the user can choose how many stages of refinement he wants (each stage is 10 iterations). The most adapted placement is the entire patch matching, since we want to improve the largest part of our texture.

6. Results

We only present here a very few examples, but more results can be found in output archive, including large sized textures.



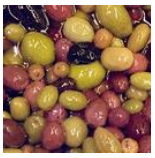
Input image	
Cost functions	C2
Placement	P3:Subpatch matching
Output type	BMP
Patch position	Random
Reduction Coeffition	20
Texture width	3
Texture height	3
Output name	output
Minimal overlap ratio	0.05
Refinement steps	0

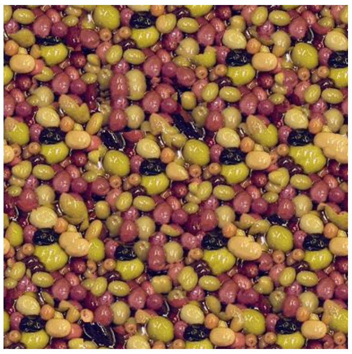
☐ Mirror Tranceform ☐ Rotation tracnceform
☐ verbose mode ☐ faster algorithm

Figure 7: User Interface

6.1 Simple Patterns

What we call simple patterns are relatively homogenous patches, from which one can sometimes obtain a good result even with random placement . Texture of Horizontal stripes was obtained with entired patch matching, Olives, with subpatch matching (while applying mirror and reverse operations).





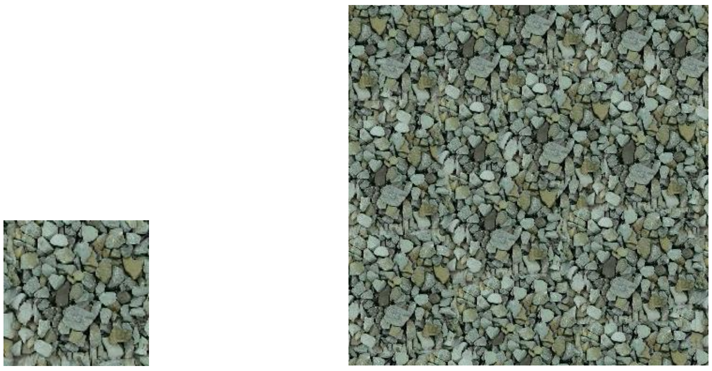


Figure 6: The input texture images(left) and the synthesis results(right)

6.2 Complex Patterns

Those patterns include a particular structure and the risk of visual inconsistencies is much increased, the best results were obtained with the entire patch matching algorithm, you can see below two textures obtained this way.

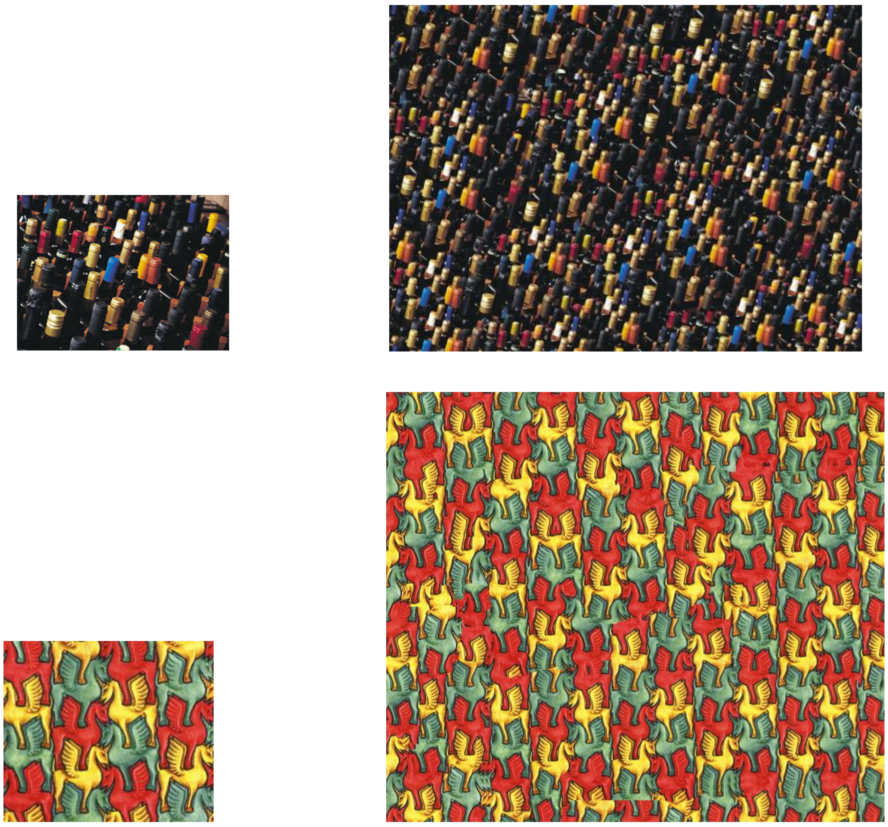


Figure 7: The input complex pattern texture images(left) and the synthesis results(right)

6.3 Synthesis Time

The proceeding time is a weakness of those algorithms, we tried to optimize our implementation so as to find the best balance between performances and computation time. For an easy texture with random placement it can take less than 8 seconds, but for a full screen render with sub-patch matching you can wait 3 minutes, and more with entire patch matching. These indicatives values are given for a recent processor, and are based on my tests with several different patches.

7. Conclusion

Our implementation performs well for most of the textures, as long as we choose good parameters (placement algorithm, etc.). But we still notice some defects on certain complex images, and someone give better results with the authors' own implementation.

We modified a bit some algorithms, introduced some constraints and pa- rameters, and pre-compute some values so as to speed up the computations. Unfortunately, we cannot compare the time performances of our implementation and the one of the authors.

Finally, this software may help people who need a simple interface for texture synthesis.

Future work can be focused on exploring the implementation in 3D to perform video texture synthesis and also using GPU to accelerating the synthesis process.

Bibliography

A.Blum,J. Lafferty,M. R.Rwebangira and R.Reddy, “Semi-Supervised Learning Using Randomized Mincuts”, *In Proceedings of the 21st International Conference on Machine Learning*, Banff, Canada 2004.

ASHIKHMIN, M. 2001. Synthesizing natural textures. *2001 ACM Symposium on Interactive 3D Graphics*, 217–226.

A.V Goldberg and R. E. Tarjan, “A new approach to the maximum-flow problem”, *Journal of the Association for Computing Machinery*, vol 35, no. 4, 921-940, 1988.

BOYKOV, Y., VEKSLER, O., AND ZABIH, R. 1999. Fast approximate energy minimization via graph cuts. In *International Conference on Computer Vision*, 377–384.

DEBONET, J. S. 1997. Multiresolution sampling procedure for analysis and synthesis of texture images. *In of SIGGRAPH 1997*, 361–368.

EFROS, A. A., AND FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. *In SIGGRAPH 2001*, 341– 346.

EFROS, A., AND LEUNG, T. 1999. Texture synthesis by non-parametric sampling. In *International Conference on Computer Vision*, 1033–1038.

HEEGER, D. J., AND BERGEN, J. R. 1995. Pyramid-based texture analysis/synthesis. *In SIGGRAPH 1995*, 229–238.

L. Ford and D. Fulkerson, “Flow in Networks”, *Princeton University Press, 1962.*

LI, S. Z. 1995. *Markov Random Field Modeling in Computer Vision*. Springer-Verlag.

LIANG, L., LIU, C., XU, Y.Q., GUO, B., AND SHUM, H.Y. 2001. Real- time texture synthesis by patch-based sampling. *ACM Transactions on Graphics Vol. 20, No. 3* (July), 127–150.

PORTILLA, J., AND SIMONCELLI, E. P. 2000. A parametric texture model based on joint statistics of complex wavelet coefficients. *International Journal of Computer Vision 40*, 1 (October), 49–70.

SAISAN, P., DORETTO, G., WU, Y., AND SOATTO, S. 2001. Dynamic texture recognition. In *Proceeding of IEEE Conference on Computer Vision and Pattern Recognition (CVPR),* II:58–63.

SCHO¨DL, A., SZELISKI, R., SALESIN, D. H., AND ESSA, I. 2000. Video textures. *Proceedings of SIGGRAPH 2000* (July), 489–498.

SOATTO, S., DORETTO, G., AND WU, Y. 2001. Dynamic textures. In *Proceeding of IEEE International Conference on Computer Vision 2001,* II: 439–446.

SZUMMER, M., AND PICARD, R. 1996. Temporal texture modeling. In *Proceeding of IEEE International Conference on Image Processing 1996,* vol. 3, 823–826.

S. Z. Li, "Markov Random Field Modeling in Computer Vision", *Springer Verlag, 1995.*

T.H. Cormen, C.E. Leiserson and R.L. Rivest, "Introduction to Algorithms", *McGraw-Hill, 1990.*

V. Kwatra, A. Schodl, I. Essa, G. Turk and A. Bobick, "Graphcut Textures: Image and Video Synthesis Using Graph Cuts", *In SIGGRAPH 2003,* 277-286.

WANG, Y., AND ZHU, S. 2002. A generative method for textured motion: Analysis and synthesis. *In European Conference on Computer Vision*.

WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree- structured vector quantization. *In SIGGRAPH 2000,* 479–488.