MPRI 2-25
Graphcut Textures: Image and Video Synthesis Using Graph Cuts

Jean-Philippe Aumasson

15th January 2006

Abstract

This document describes my implementation of the algorithm for texture synthesis describe in [KwatraSIGGRAPH], it mainly focuses on the implementation aspect, since the detailed description of the algorithm can be found in the original article.

Contents

1 Introduction

This project was suggested by our teacher R.Keriven as an evaluation for the algorithmic vision course, at Paris 7's university research master in computer science (MPRI [MPRI]). In the following subsections we introduce the background knowledge needed to understand the next sections, in which we first describe the main stages of the algorithm (patch placement and seam cut by maxflow computation in a graph), before finally showing some results, and suggesting some improvements to the original paper. For a better understanding of this document, i strongly recommand to read [KwatraSIGGRAPH] before.

The last version of Kuva is downloadable from the author's page: *http://www.131002.net*.

1.1 Main goals

In [KwatraSIGGRAPH], 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.

The project Kuva 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, Kuva is distributed under the terms of the GPL licence, and 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 arround the patch, once it's placed, to select some pixels, and ignore some others,
- the **maxflow problem** in a graph consists in finding the path that maximizes 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 **output 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

Kuva 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 maxflow/min-cut, the authors (Y.Boykov, V.Kolmogorov) who first applyed graph cuts to image processing produced a very efficient free implementation [BoykovMF] of the maxflow computation, that we chose to reuse in Kuva.

The image processing interface we chose is *CImg* [TschumpCImg], the unique C++ header file is included in Kuva sources, thus no installation is needed for the user. It is portable to any common OS, and is distributed under the CeCILL licence, that is the French analog to the GPL. The best (and the worst) point is that the whole CImg sources is only in one file of 600Kb...

An implementation was already made last year by students of the MPRI, with the project *Texturize* [CornetTexturize], under the form of a plug-in for the program GIMP. As Kuva does, they only implemented the case of 2D textures, with a few additional functions.

2 Patch placement

2.1 General idea

The first operation in the process iterated for texture synthesis is the positionning 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 fit 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

2.2 Algorithms

2.2.1 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.

2.2.2 Entire patch matching

This algorithm is a bit more greedy than the previous one, we look for a translation 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

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

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 [KwatraSIGGRAPH], 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.

2.2.3 Bad idea tested: alternative subpatch 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 maxflow 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 henc we finally obtain a bad texture. This algorithm is not part of the final implementation.

2.2.4 Subpatch matching

This is the default algorithm used by the program. Instead of looking for translations 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.

2.3 Improvements and programming tricks

2.3.1 Precomputation

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

2.3.2 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 subpatch matching functions. Therefore Kuva has an option, which is not set defaultly, to automatically switch from one function to another when no advance is done during a certain time. In fact, if the function is the subpatch 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 sub-area of the texture. Then if we still do not fill the entire texture, Kuva switches to random placement, which quickly manages to end the process. The option to give in command line is `-sr`.

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.

2.3.3 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), and can be modified with the option `-ra`.

2.3.4 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. This coefficient can be modified with the options `-cx` and `-cy`.

2.3.5 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.

3 Seam computation

3.1 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.

3.2 Particular cases

3.2.1 Several overlap subareas

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.

3.2.2 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.

3.3 Graph building

The graph is built as described in the original paper, 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 applyed 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 [KwatraSIGGRAPH].

3.4 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. $\mathrm{A}(x)$ and $\mathrm{B}(x)$ respectively stands for the color of the pixel x in the patch and in the output texture. As we only consider overlapping pixels, $\mathrm{B}(x)$ cannot be empty.

3.4.1 Basic function

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

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

Like for the placement algorithms, we consider the mean on the three color channels when computing the total cost.

3.4.2 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:

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

3.5 Improvements and programming tricks

3.5.1 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.

3.5.2 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 $65536/4 = 16384$.

3.5.3 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 graphcut computation, and small enough to stay under the maximum value allowed.

3.5.4 Seams display

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

3.5.5 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.

3.5.6 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.

4 Results

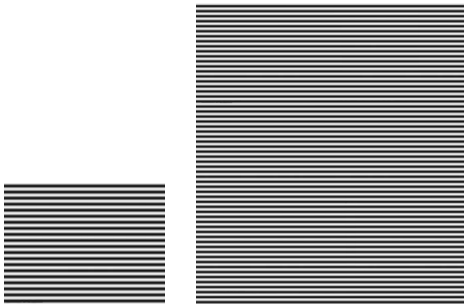
We only present here a very few examples, but you may go to Kuva's website to see much more, including large sized textures:

http://www.131002.net/softs/kuva

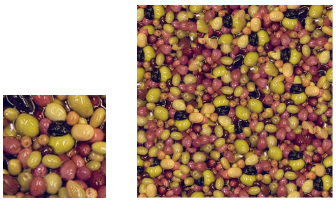
4.1 Simple patterns

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).

Horizontal stripes



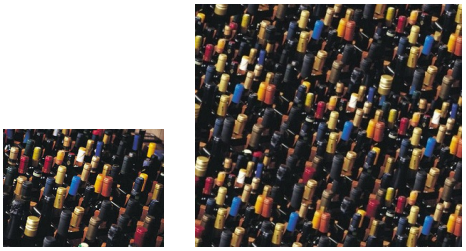
Olives



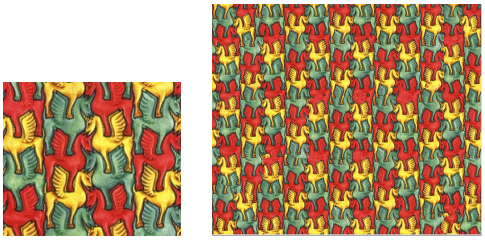
4.2 Complex patterns

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:

Bottles



Horses



4.3 Synthesis time

The proceeding time is a weakness of those algorithms, we tryed 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 20 seconds, but for a full screen render with subpatch matching you can wait 5 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.

5 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 someones give better results with the authors' own implementation.

We modified a bit some algorithms, introduced some constraints and parameters, and precompute 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, while allowing them to control many parameters in command line, so as to *play* testing the algorithm introduced in [KwatraSIGGRAPH].

References

[KwatraSIGGRAPH] V.Kwatra, A.Schödl, I.Essa, G.Turk, A.Bobick, *Graphcut Textures: Image and Video Synthesis Using Graph Cuts*, *http//www.cc.gatech.edu/cpl/projects/graphcuttextures*.

[BoykovMF] Y.Boykov, V.Kolmogorov, *MAXFLOW - software for computing mincut/maxflow in a graph, Version 2.2*, *http://www.cs.cornell.edu/People/vnk/software.html*.

[TschumpCImg] D.Tschumperlé , *CImg library*, *http://cimg.sf.net*.

[CornetTexturize] M.Cornet, J.B.Rouquier, Texturize plugin, *http://www.manucornet.net/Informatique/Texturize.php*.

[MPRI] MPRI website, *http://www.mpri.master.univ-paris7.fr/*.

[EfrosFreeman] A.A. Efros, W.T. Freeman, *Image quilting for texture synthesis and transfer*, Proceedings of SIGGRAPH 2001.