# RestoCut: Reconstruction of regular images using graph cuts

Tagir Valeev
Institute of Informatics Systems,
Novosibirsk, Russia
lan@biorainbow.com

## Abstract

In this paper new approach to reconstruct images, removing unwanted objects from it, is described. The main idea of the introduced approach is to generate texture over unwanted object using graph cut based texture synthesis algorithm. Different texture synthesis techniques are briefly covered, while used technique is described in detail. After that, migration from the texture synthesis problem to the image reconstruction problem is explained, and some experimental results are shown.

***Keywords:*** *Image reconstruction, Texture synthesis, Graph cut.*

## 1. INTRODUCTION

Let's consider the following problem. We have some photo, where some kind of landscape is shown. Though there are some unwanted objects on the picture. For example, we have blooming field with some junk on it or stony beach with people resting, but our goal is to get the field without junk or unmanned beach. It's impossible sometimes to take the proper photo, so digital processing is required to get what we want.

We'll consider only those cases, where surrounding landscape has similar structure like that covered by unwanted object. So the easiest solution is to open our image in the favorite graphics editor package, copy similar image fragment (pattern) and place it over the object we want to remove. After feathering the borders we may get something like we wanted. Being simple this approach has many disadvantages. Here are some of them:

- Even after feathering the edges of copied fragment are often visible. Increasing feather radius may strike viewer's eye, because some features will become translucent;
- In case if unwanted object is rather big, we may not found similar image fragment large enough to cover the object completely. We can copy smaller fragment several times, but periodic structure of landscape will be striking;
- Rather often brightness is not constant across unwanted area and pattern as well, resulting in significant color difference on the edges of copied fragment. In this case feathering will not help at all.

In this paper we introduce rather new approach to solve this problem and describe how it was implemented in our software package, called RestoCut. The main idea was taken from texture synthesis techniques. In fact we can synthesize texture over object we want to remove and them seamlessly combine it with existing image.

Texture synthesis technique we are using was already introduced in [1]. Our method is based mainly on paper by V. Kwatra et al [2], where using of graph-cuts was suggested to generate textures. We have developed and modified their approach to fit our objective.

In the following section we'll cover the approach used to create textures using graph-cuts, pointing out the differences between methods described in [2] and our realization. After that we'll center on using texture synthesis to solve the considered problem. Then we'll describe RestoCut and show some experimental results.

## 2. TEXTURE SYNTHESIS

Currently there are many techniques exist to generate textures from sample image. Most of them doesn't take into account that sample image may contain non-periodical objects of real world (like flowers, stones, birds and so on). For example, Heeger and Bergen [3] used sample image to generate color-frequency distribution, and synthesize texture using this information. This algorithm was already implemented by Igehy and Pereira [4] in the work similar to this one. This is good enough to generate textures like wool, plastic, bitumen, etc., but such features as flowers will not be preserved. Image quilting technique, described by Efros and Freeman [5] is more oriented to keep the features from original sample, but still not good enough. In this approach the sample is divided into blocks which are copied into newly generated texture.

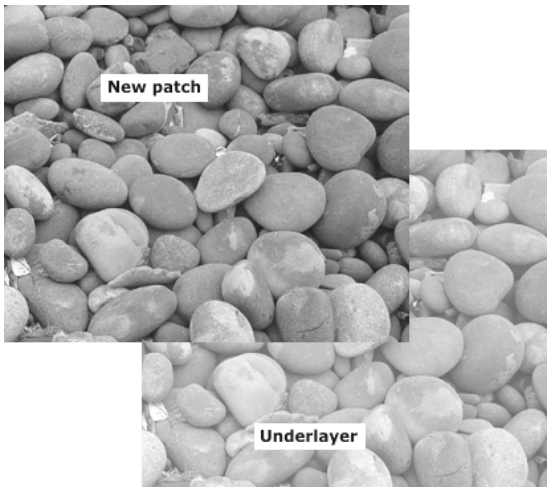


**Figure 1:** Patch placement.

In the method proposed in [2] whole patches from input are copied to synthesized texture, and then algorithm finds the best seam between newly copied patch, and patches copied before. We are using similar technique. However, in contrast to this work, all patches put to the new texture so far are merged together and considered as underlayer. This makes patch placement procedure more complicated, but simplifies seam finding and generalizes problem: in our solution underlayer may not consist of patches copied before only, but contain some other image fragments. Such generalization is much desired to solve the problem, discussed in introduction.

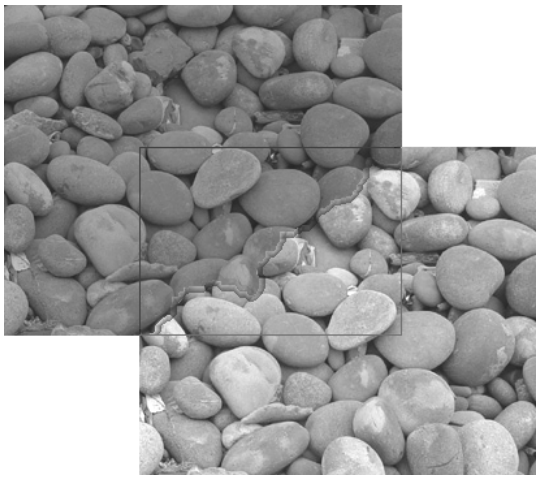

**Figure 2:** Finding the optimal seam.

Figure 1 shows an example of how new patch is merged with existing underlayer. New patch is placed so that it overlays underlayer. Patch placement strategy is discussed in section 2.2.

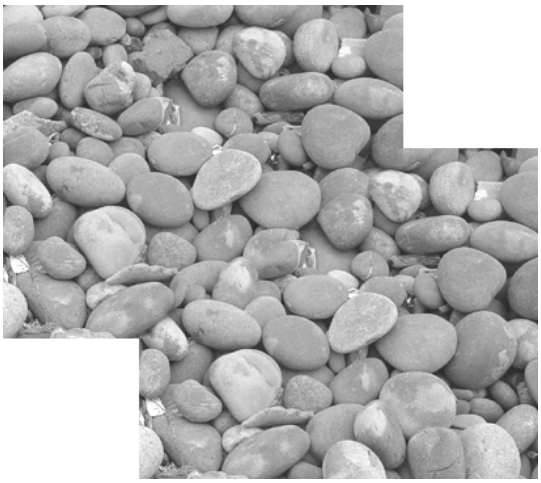

**Figure 3:** New patch and underlay after merge.

Figure 2 shows the optimal seam found in overlapping area; this procedure is described in section 2.1. Finally after slight feathering we'll get underlayer which consists of merged patches as shown on Figure 3. Applying these steps repeatedly will produce whole given area filled with sample image. An example of resulting texture is shown on Figure 4.

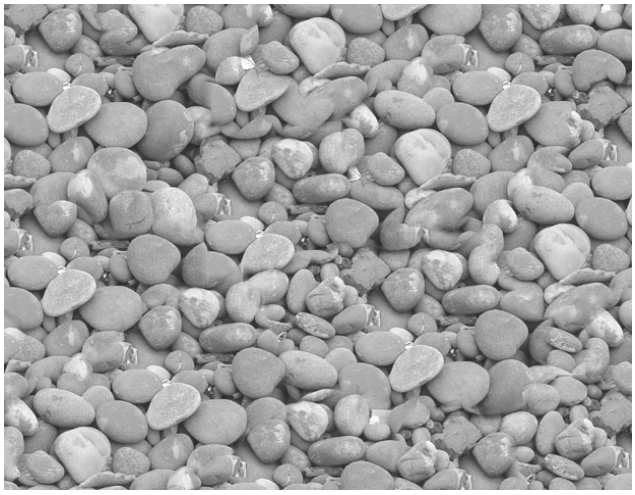

**Figure 4:** Synthesized texture.

Let's consider these steps in detail. We'll begin from finding the optimal seam rather than patch placement, because knowing how the seam is searched will help to see the problems of patch placement.

## 2.1 Using graph cuts to find the optimal seam

To find the optimal seam, which divides new patch and underlay, we should cut through overlapping area, so that pixels around the cutting line have the closest colors possible. This procedure is quite similar to one proposed in [2].

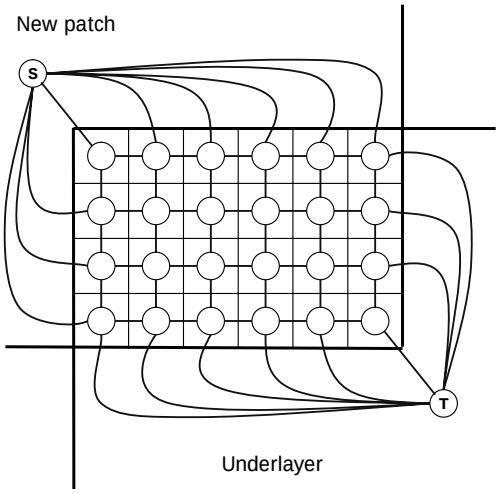


**Figure 5:** Representing of overlapping area as graph.

First we have to build weighted graph, which fits the following conditions. Each node corresponds to single pixel in overlapping area (Figure 5), edges connect neighbor pixels. Also two special nodes are added: S (source) and T (sink), which are connected to all boundary pixels from the patch side and underlay side respectively.

Now we should assign weights to the edges. If the edge connects two regular nodes corresponding to pixels A and B, then edge weight will be

$$W_{AB} = \left\| C_A - C_A^* \right\|_1 + \left\| C_B - C_B^* \right\|_1, \qquad (1)$$

where $C_A, C_A^*$ are color vectors of pixel A, corresponding to patch and underlayer respectively ($C_B, C_B^*$ are the same for pixel B). In other case, when the edge connects regular node with source or sink, weight assigned to it is some constant, which is bigger than maximal value of (1).

Finally we can split this graph into two sub-graphs, so that one of them contains source node, other contains sink node, and weight sum of edges connecting nodes from different sub-graphs has the minimal possible value. This problem is well-known in graph theory as min-cut problem, and can be reduced to network flow problem, which is solved by Ford-Fulkerson algorithm [6]. To perform this step we have used C++ code developed by Boykov and Kolmogorov (it was introduced in [7] and freely available on the authors' website). This code is well-optimized and perfectly suits for such tasks.

Note that underlayer may already contain several patches or some other fragments, so the overlapping area is not necessarily rectangular. It may be in any form, have holes or even consist of several unconnected areas. These cases are handled in the same way like described above. In [8] underlying patches are handled separately, and additional nodes are created along the old seam to refine it when new patch is placed. We've simplified this step to make possible generalization mentioned above.

## 2.2 Patch placement strategy

The simplest way of patch placement is to place them randomly. This approach works in some rare cases, when sample image fits strict requirements. Of course, graph-cut algorithm, described above, finds the best possible seam for specified patch position, but in many cases even the best seam is not good enough. So we should place patches using special strategy to lighten the graph-cut work. Each patch placement is characterized by offset of its left-top angle. We can assign some weight function to patch position and minimize it. To speed up minimization we are using genetic algorithm [8] where new patch position (x and y coordinates) is considered as chromosome.

Now we should define that weight function to distinct between good and bad patch placement. Here are some constraints which should be reflected in weight function:

1. Overlapping area should be thick enough everywhere, otherwise graph-cut algorithm have too small space to select from;
2. Position where new patch touches underlayer, should be considered as very bad;
3. New patch placement shouldn't disrupt large-scale structure if any;
4. New placement should cover enough uncovered pixels, otherwise algorithm may work quite long (or even infinite) time.

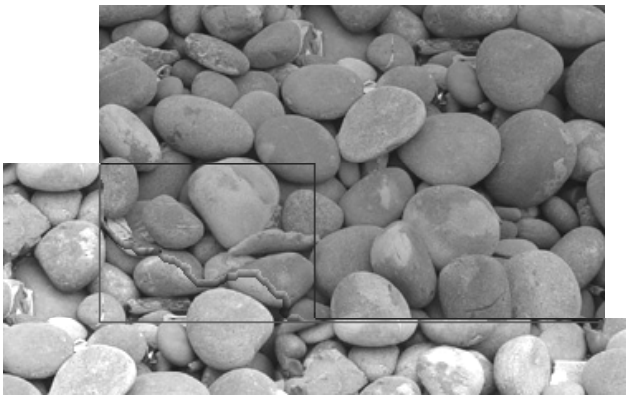

**Figure 6:** Thin overlapping area

Figure 6 illustrates first constraint. New patch (on top) intersects underlayer, but right side of overlapping area is only two pixels thick. This means that graph-cut have no choice but to draw seam between these two pixels, even if it's not good enough (in shown example it's really not good; the stones above and below thin area are quite different). So it's evident, that this case should be considered bad and have rather big weight.

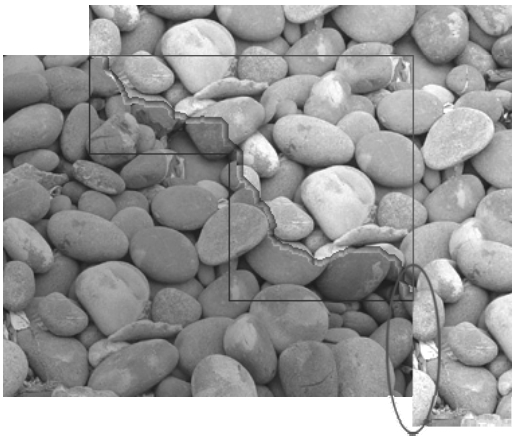

**Figure 7:** New patch touches underlayer

Second constraint is rather similar to the first one. It's shown on Figure 7. New patch was placed so that it touches underlayer in the area, highlighted by ellipse. In this case graph-cut cannot do anything, because contact area is outside of the overlay.

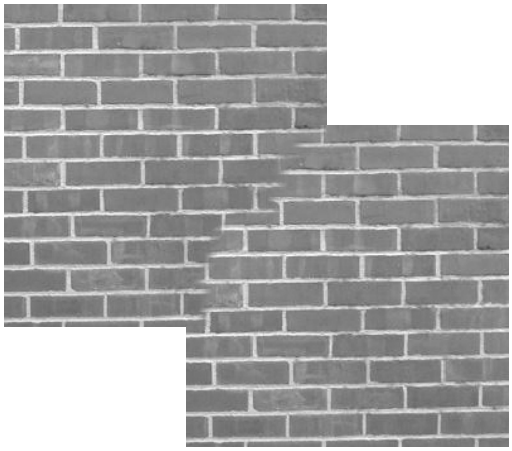

**Figure 8:** Large-scale structure

Figure 8 illustrates the large-scale of sample image (bricks). In the given case offset between patches is not the integral number of brick periods, thus graph-cut cannot merge patches seamlessly, so it's impossible to synthesize good texture if we allow such patch positioning.

Fourth constraint was introduced to reduce processing time. Sometimes best weight according to three other constraints is assigned to placement, when patch entirely lies on underlayer or adds only several pixels to it, so it will take too many steps to fill in whole given area.

According to all these constraints, we empirically have created penalty functions. $L(x)$ is responsible for touches and thin overlapping areas, and $B(x)$ is responsible for constraint 4. Third constraint makes the base of weight function, for which penalties are applied. So the weight function looks like this:

$$N = B\left( L\left( \frac{1}{|S|D} \sum_{p \in S} \left\| C_p - C_p^* \right\|_2^2 \right) \right), \qquad (2)$$

where $S$ is set of overlapping area pixels and $D$ is color deviation of sample image pixels:

$$D = \frac{\sum_{p \in T} \left\| C_p - \overline{C} \right\|_2^2}{|T| - 1}, \qquad (3)$$

where $T$ is the set of pixels in sample image (patch), thus $|T|$ is the patch size. $C_p$, $C_p^*$ are color vectors of pixel $p$ corresponding to patch and underlayer respectively.

Argument of $L$ is penalty value without taking into account constraints 1, 2, 4. Functions $L$ and $B$ can enlarge penalty value if selected patch position is unsatisfactory from the point of these constraints. Value of $L$ depends on minimal thickness of overlapping area. When overlapping area is thick enough, $L(x)$ returns $x$, otherwise it rapidly grows and returns maximal value when constraint 2 takes place. $B(x)$ is constructed similarly: it returns $x$, when new patch placement adds to underlayer $|T|/2$ pixels or more, and comes to its maximal value, when new placement doesn't add to underlayer even a single pixel.

## 2.3 Seam feathering

Though graph-cut finds best seam possible, sometimes seam is still visible. In such cases feathering of nearby pixels will help to hide it. We are using simple linear feathering. New color for the pixel $A$ near seam is calculated using the following formulae:

$$C_{A\mathrm{Res}} = \frac{1}{2l}\left( C_A \cdot (l - d) + C_A^* \cdot (l + d) \right), \qquad (4)$$

where $d$ is the distance between pixel $A$ and seam, and $l$ is feathering radius (usually it's about 5 pixels). Distance between seam and current pixel is calculated in rather simple way. Two subgraphs found by min-cut algorithm are considered separately, source and sink are removed, but special seam-node is added instead. The seam-node is connected to all the nodes which were bound with other subgraph. After that distance between each node and the seam-node (if it doesn't exceed $l$) is calculated using breadth-first algorithm.

## 3. RECONSTRUCTION OF IMAGES

Now we can apply texture synthesis approach to reconstruct image fragments. Consider the sample image "Meadow" (Figure 9). Our goal will be to remove a man from the picture, leaving meadow and forest as it is.

As you can see, the man in the "Meadow" image covers some flowers. So it is good idea to synthesize flower field texture over the man. First we should define the patch for the texture synthesis algorithm. We can select any rectangular area nearby, which contains flowers. However it is not preferred to select flowers above (or below) unwanted object because or perspective. Also we shouldn't select too small patch, in order to generate non-monotonous texture. Example of patch selection is shown by dotted line on Figure 10.

**Figure 9:** Source image "Meadow". Image is courtesy of Vladimir Logutenko

**Figure 10:** Mask and sample selection

Second we should mark the area which should be filled with new texture. Actually we select two areas: texture area, which will be replaced fully by newly synthesized texture, and buffer area, which can be replaced partially. Typically texture area should be inside buffer area (see Figure 10). In the most of cases buffer area can be selected automatically, by expanding texture area to the fixed number of pixels (this function presents in most of modern graphic editors). The "Meadow" image has resolution of 800×600 pixels, and buffer area was selected by expanding texture area by 40 pixels. Also user can select texture area rather roughly, high accuracy is unnecessary here, so defining areas is quite fast procedure.

**Figure 11:** Processed "Meadow" image.

Finally we can launch texture generation algorithm. It considers buffer area as area, which was already filled by generated texture. Though some its pixels may be replaced, if graph-cut algorithm decides to draw seam over them.

Pixels outside of selected areas are not modified at all. If patch placed so that it juts out of buffer area, it's cropped before graph-cut applying. Figure 11 shows the result of processing the "Meadow" image.

## 4. BRIGHTNESS COMPENSATION

An improvement of introduced algorithm is described below. Sometimes though image has regular structure, image brightness is not constant. Typically it's applied to sky images (e.g. birds in the sky), indoor photos (especially when flash bulb was used) and so on. In these cases brightness of patch opposite sides is quite different, so graph-cut algorithm will unable to produce good result.

**Figure 12:** Sample image "Wall".

In this case brightness compensation may be performed in the initial and final stages. On initial stage brightness of patch is equalized as well as brightness of buffer area. On the final stage brightness of filled area is reversed to original.

Here we should define brightness compensation function, namely the function of coordinates which we can subtract from source image brightness to get brightness-equalized image. Light intensity is inversely proportional to the square of distance between light source and observer, so we assume that brightness function looks like the polynomial of degree two:

$$f(x,y) = Ax^2 + By^2 + Cxy + Dx + Ey + F \qquad (5)$$

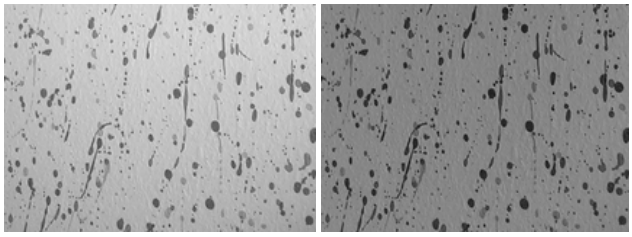

**Figure 13:** Selected patch before and after brightness compensation.

Coefficients *A*–*F* in (5) are defined by minimizing sum of squares of differences between the $f(x,y)$ value and the actual pixel brightness for the whole area:

$$\sum_S \left(Y(x,y) - f(x,y)\right)^2 \to \min \qquad (6)$$

Minimization problem is solved by using least squares method and Gauss-Jordan elimination function as it is described in [9]. We calculate coefficients *A–F* for patch and for buffer area separately. Then we equalize buffer area and patch, subtracting $f(x,y)$ value from pixel brightness using obtained coefficients, and generate texture. After that we restore original brightness, adding $f(x,y)$ value to all of the buffer area and texture area pixels, using coefficients for buffer area. Equalization is performed in YUV color space. Only luma is modified, while chroma components remain unchanged.

**Figure 14:** Processed "Wall" image.

Brightness compensation is illustrated on the sample image "Wall" (Figure 12). Our goal is to remove the picture on the right side of the "Wall" image. Figure 13 shows patch (actually it was selected below the picture) and the same patch after brightness equalization. Figure 14 shows the processed image.

## 5. COMPARISON WITH OTHER WORKS

Recently using graph-cut approach to generate textures became quite popular so several implementations of it can be found through the Internet. Optimal seam search described in 2.1 above is quite similar in these works. Edge weight is calculated using formulae similar or exactly the same as (1). Also to solve min-cut problem most of developers use the same Boykov and Kolmogorov's code. The main difference between the approaches is patch placement strategy. This step can be performed in many different ways.

We have compared our implementations with two others which are available in the Internet:

- Texture Synthesis project of the National University of Singapore (authors are: Guo Dong and Zhuo Shaojie); available from: http://www.comp.nus.edu.sg/~guodong/syntxt/
- The Texturize plugin for GIMP; available from: http://www.manucornet.net/Informatique/Texturize.php

Both of these works are based on [2]. Though they don't allow to replace fragments of image by generated texture like RestoCut, they can be compared with RestoCut texture synthesis module. Also these products don't have brightness compensation implemented, so comparison was performed for patches without notable brightness gradients.

The Texturize plugin for GIMP uses patch placement strategy which is rather similar to the one described in [2]. Texture Synthesis places patches in slightly different manner. It places to the output area not patches, but samples, rectangular subpictures of patch with fixed width and height, but offset of sample inside patch may differ. User may specify parameters *x* and *y* which define ratio between sample and patch size (for width and height respectively). By default *w* and *h* both equal 0.5. Places for new samples are fixed; coordinates (*a*, *b*) of new sample left-top corner are defined as:

$$\begin{aligned} a &= Wx(1-l)i \\ b &= Hy(1-l)j \end{aligned}, \qquad (7)$$

where *l* is overlapping area size, *W* and *H* are patch width and height respectively. Values of *i* and *j* are changed from 0 to *w* and *h* in the loop (*w* and *h* are also specified by user indirectly defining output picture size). For each pair (*i*, *j*) the best offset of sample in the patch is selected and chosen sample is combined with generated so far picture using min-cut.

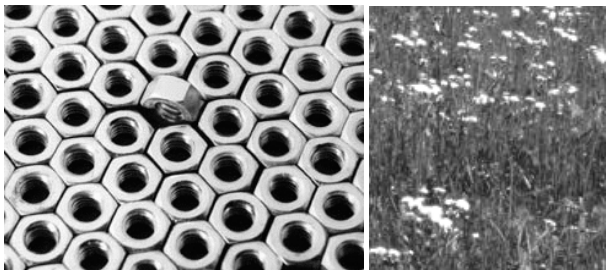

**Figure 15:** Patches "Nuts" and "Flowers".
"Nuts" is courtesy of VisTex.

This approach is easier to implement, but results not always look well. Also there are many parameters and sometimes it's necessary to tune them in order to get satisfactory results.

Texturize gives the best results for patches with regular structure. Consider an example patch "Nuts" shown on Figure 15. Synthesized result (Figure 16, left) is almost perfect, only slight artifacts are visible. Result of RestoCut (Figure 16, right) is not so good. Result generated by Texture Synthesis (Figure 17) has no artifacts, but turned nuts are not spread all over the texture which makes result the worst.

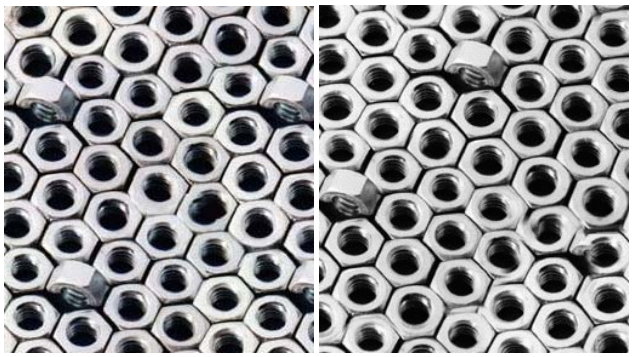

**Figure 16:** Texture “Nuts” generated by Texturize (left) and RestoCut (right)

The second example is “Flowers” (Figure 15). Generated restuls are shown on Figure 18. Result of Texturize (in the middle) have somewhat regular structure, thus looks unnatural. Images generated by RestoCut and Texture Synthesis look good, though image generated by Texture Synthesis with default parameter set was really bad, so parameter tuning was necessary.

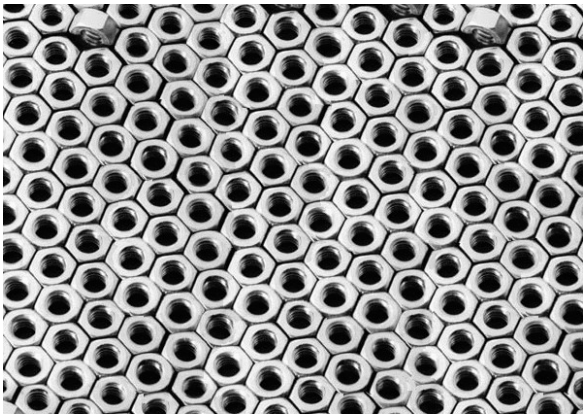

**Figure 17:** Texture “Nuts” generated by Texture Synthesis.

On some textures generated by Texturize and Texture Synthesis seams similar to one shown on Figure 7 can be noticed. In total seven images were involved in comparison, though we cannot observe all the generated textures here. More detailed results are available from http://skypiece.iis.nsk.su/~lan/restocut/

Table 1 shows time spent to generate textures by different realizations. All times are in seconds and were measured on PC with AMD Athlon XP 2500+ CPU and 512 Mb RAM. For RestoCut also time spent to graph-cut and patch placement steps is calculated separately. Total time is higher than sum of them because it includes also file loading and storing, feathering and other steps.

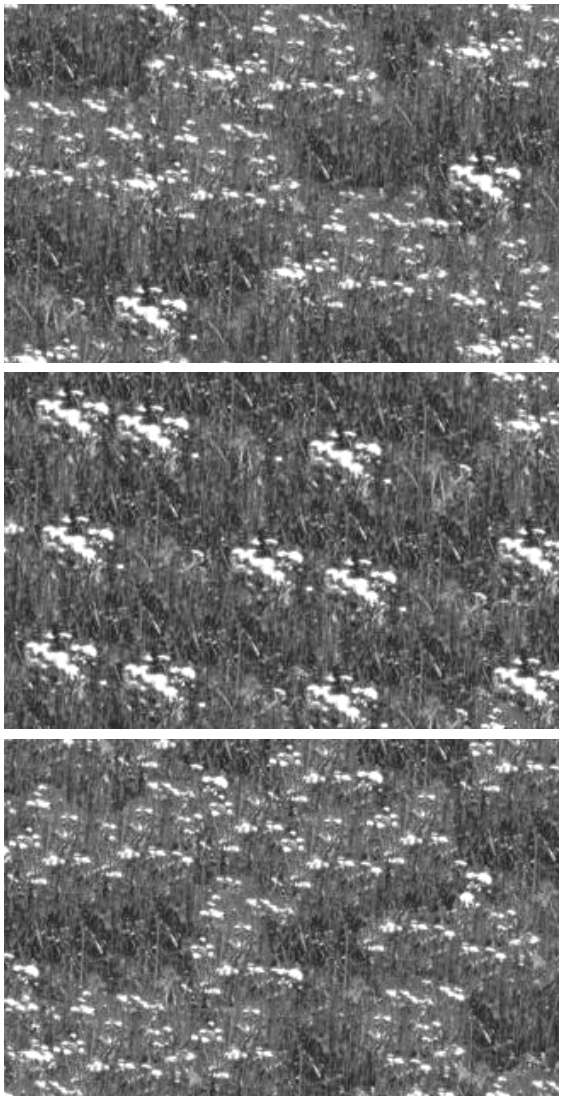

**Figure 18:** Texture “Flowers” generated by RestoCut (top), Texturize (middle) and Texture Synthesis (bottom).

**Table 1:** Texture synthesis speed.

| Source pattern | Source size | Target size | RestoCut | | | Texturize | Texture Synthesis |
|---|---|---|---|---|---|---|---|
| | | | Placement | Graph-cut | Total | | |
| brickwall | 256×256 | 578×578 | 25.8 | 1.7 | 29.2 | 27.9 | 30.8 |
| daisy | 176×234 | 646×855 | 52.5 | 1.3 | 57.4 | 135.3 | 81.8 |
| icystraw | 200×150 | 450×340 | 14.3 | 0.8 | 15.9 | 15.8 | 23.4 |
| input | 256×256 | 578×578 | 32.0 | 1.0 | 34.7 | 42.9 | 78.8 |
| flowers | 145×161 | 327×360 | 11.3 | 0.7 | 12.6 | 32.6 | 25.9 |
| Nuts | 293×217 | 661×488 | 30.9 | 0.6 | 31.2 | 49.6 | 26.6 |
| field | 566×585 | 880×907 | 67.9 | 3.2 | 74.1 | 35.0 | 234.8 |

Table 2 below shows which additional features are supported by these implementations. Tiling texture is the texture which can be seamlessly stitched to itself (useful for webpage or desktop wallpapers and so on). Rotation means that software has an option

to generate more diversified textures by rotating original patch by 90°, 180° or 270°. Rotation and mirroring was also introduced in [2].

**Table 2:** Features supported by different implementations.

| Feature | RestoCut | Texture Synthesis | Texturize |
|---|---|---|---|
| Brightness compensation | + | – | – |
| Tilable textures | + | – | + |
| Rotation | – | + | – |

As for texture synthesizers which are not graphcut-based, we have considered that results are much worse visually, thus not so interesting. In [2] you can find comparison of graphcut-based algorithm with image quilting based.

## 6. CONCLUSION AND FUTURE WORK

The method, proposed in this paper, is suitable for solving the problem of reconstructing image fragments, which are somewhat regular by its nature.

Future improvements of this idea may include modifications of current placement procedure and adding new features. In some special cases placement may be improved so result will look more natural. New features may include using patch modifications (e. g. rotations or mirroring) as well as several patches to synthesize more varied texture. Also some algorithms to take the perspective into account may be analyzed and included into this work.

## 7. REFERENCES


[1] Valeev T. F. *GRAPHCUT: Using graph-cuts in texture synthesis. // Proceedings of the conference-and-contest of young scientists "Microsoft Technologies in the Theory and Practice of Programming" Feb 22-24, 2006; pp.51-53.*

[2] Kwatra, V., Schödl, A., Essa, I. et. al. *Graphcut Textures: Image and Video Synthesis Using Graph Cuts // Proc. of SIGGRAPH 2003.*

[3] Heeger, D. J., Bergenn, J. R. *Pyramid-based texture analysis/synthesis. // Proc. of SIGGRAPH 1995, pp. 229–238.*

[4] Igehy, H., Pereira, L. *Image replacement through texture synthesis. // Proc. of the 1997 international Conference on Image Processing (ICIP '97) Oct. 26 - 29, 1997; 3-Volume Set-Volume 3, pp. 186-189.*

[5] Efros, A., Freeman, W. *Image quilting for texture synthesis and transfer. // Proc. of SIGGRAPH 2001.*

[6] Ford, L., Fulkerson, D. *Flows in Networks. // Princeton University Press. 1962.*

[7] Boykov Yu., Kolmogorov V. *An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. // IEEE Transactions on Pattern Analysis and Machine Intelligence, Sept. 2004; vol. 26, no. 9, pp. 1124-1137.*

[8] Goldberg, David E, *Genetic Algorithms in Search, Optimization and Machine Learning. // Kluwer Academic Publishers, Boston, MA., 1989.*

[9] Press, W.H. et al, *Numerical recipes in C: The Art of Scientific Computing. // Cambridge University Press, 1992.*


## About the author


Tagir Valeev is a Ph.D. student at Institute of Informatics Systems, Russian Academy of Sciences, Siberian Branch, Complex Systems Modeling Group. Also he is the leading developer of BioRainbow scientific software group. His contact email is lan@biorainbow.com.