Eurographics Italian Chapter Conference (2006)
G. Gallo and S. Battiato and F. Stanco (Editors)

SVG Vectorization by Statistical Region Merging

S. Battiato, G. M. Farinella, G. Puglisi

Dipartimento di Matematica e Informatica, University of Catania, Italy
Image Processing Laboratory
http://www.dmi.unict.it/~iplab

Abstract

In this paper a novel algorithm for raster to vector conversion is presented. The technique is mainly devoted to vectorize digital picture maintaining an high degree of photorealistic appearance specifically addressed to the human visual system. The algorithm makes use of an advanced segmentation strategy based on statistical region analysis together with a few ad-hoc heuristics devoted to track boundaries of segmented regions. The final output is rendered by Standard Vector Graphics. Experimental results confirm the effectiveness of the proposed approach both in terms of perceived and measured quality. Moreover, the final overall size of the vectorized images outperforms existing methods.

Categories and Subject Descriptors (according to ACM CCS): **General Terms**: Vectorization, Segmentation, SVG.

1. Introduction

The vectorial format is useful for describing complex graphical scenes using a collection of graphic vectorial primitives, offering the typical advantages of vectorial world: scalability, resolution independence, etc.. In this paper we propose a novel technique to cover the gap between the graphical vectorial world and the raster real world typical of digital photography. Standard Vector Graphics (SVG) is a standard language for describing two-dimensional graphics and graphical applications in XML ([DHH02], [Qui03]). This format could be very useful in the world of mobile imaging device where the typical capability of a camera needs to match with limited colour/dimension resolutions display. Another potential application is presented in [RB05] where an ad-hoc method for icon vectorization is applied emphasizing the good performances in terms of resolution scalability.

Recently, some commercial and free software have been developed using some solution to the "raster to SVG" problem (Vector Eye [VLDP03], Autotrace [Web02], Kvec [Kuh03]). Almost all of them are devoted to SVG rendering of graphic images (e.g. clip art, etc.) starting from different image formats, showing in such case good performances but also several perceptual drawbacks when applied to digital pictures acquired by consumer devices.

Considering photorealistic vectorization, a few advanced approaches are described in SWaterG [BCDN05], SVGenie and SVGWave [BBD*05] where ad-hoc segmentation strategies, making use respectively of watershed decomposition and adaptive triangulation, allow to obtain good performances both in terms of perceptual accuracy and overall compression size. An useful reviews is presented in [BDG*05]. We also mention the work presented in [PS05] where a region-contour segmentation scheme that takes into account contours resulting from colour gradients in an image, as well as their interrelationships in terms of bounding a common region, is described.

One of the main drawback of almost all cited approaches is the parameters tuning with respect to the final application. The correct trade-off between final perceived quality, scalability and the corresponding file size is a very challenging task, mainly for user that doesn't know the algorithmic details of each solution. Our proposed technique is based on a single input parameter, that fixes the degree of "coarseness" to be used in the segmentation step. An efficient algorithm has been used for the preliminary analysis of input raster data: Statistical Region Merging (SRM) [NN04]. Such segmentation algorithm is able to capture the main structural components of a digital image using a simple but effective statistical analysis.

Our technique consists of two main steps:

- the image is partitioned in polygonal regions using SRM;
- the borders of segmented regions are properly tracked and described by SVG primitives making also use of some smart tips.

The final results show how the proposed strategy outperforms state-of-art techniques in terms of visual perception, measured image quality and compression size. The overall performances are described in terms of Rate-Distortion plots, by varying the unique involved quality parameter.

The rest of the paper is organized as follows. In Section 2 we briefly review the main details of SRM. The successive Section describes how boundary representation is computed while Section 4 reports the generation of SVG representation by using suitable mapping strategies. In Section 5 several experimental results together with some comparisons with other techniques are reported. A final Section closes the paper tracking also direction for future works.

2. Statistical Region Merging

Segmentation is the process of partitioning an image into disjoint and homogeneous regions. A more formal definition can be given in the following way [LM01]: let I denote an image and let H define a certain homogeneity predicate; the segmentation of I is a partition P of I into a set of N regions R_1,R_2,...,R_N, such that:

- $\bigcup_{n=1}^{N} R_n = I$ with $R_n \cap R_m = \emptyset$, $n \neq m$;
- $H(R_n) = true \quad \forall n$;
- $H(R_n \cup R_m) = false \quad \forall R_n$ and R_m adjacent.

Recently, thanks to the increasing speed and decreasing cost of computation, many advanced techniques have been developed for segmentation of colour images. In particular we used the Statistical Region Merging [NN04] algorithm that belongs to the family of region growing techniques with statistical test for region fusion. SRM is based on the follow model of image: I is an image with $|I|$ pixels each containing three values (R, G, B) belonging to the set {1, 2, ..., g}. The model considers image I as an observation of perfect unknown scene $I*$ in which pixels are represented by a family of distributions from which each colour level is sampled. In particular, every colour level of each pixel of $I*$ is described by a set of Q independent random variables with values in [0, g/Q]. In $I*$ the optimal regions satisfy the following homogeneity properties:

- inside any statistical region and for any colour channel, statistical pixels have the same expectation value for this colour channel;
- The expectation value of adjacent regions is different for at least one colour channel.

From this model Nielsen and Nock obtain the following merging predicate:

$$P(R,R') = \begin{cases} true & if\ \forall a \in \{R,G,B\}, |\overline{R'_a} - \overline{R_a}| \\ & \leq b(R) + b(R') \\ false & otherwise \end{cases} \tag{1}$$

$$b(R) = g\sqrt{\frac{1}{2Q|R|}\left(\ln \frac{|R_{|R|}|}{\delta}\right)} \tag{2}$$

$\overline{R_a}$ denotes the observed average for color a in region R whereas $R_{|l|}$ is the set of regions with l pixels.

The order in which the tests of merging were done follows a simple invariant A:

- when any test between two true regions occurs, that means that all tests inside each region have previously occurred.

In the experiments, A is approximated by a simple algorithm based on gradient of nearby pixels. In particular Nielsen and Nock consider a function f defined as follow:

$$f(p,p') = \max_{a \in R,G,B} f_a(p,p') \tag{3}$$

A simple choice for f_a is:

$$f_a(p,p') = |p_a - p'_a| \tag{4}$$

More complex function that extends classical edge detection convolution kernels could be used to define f_a. In our case we used the Sobel gradient mask.

The pseudo-code of algorithm is the following [NN05]:

```
INPUT: an image $I$
Let $S_I$ the set of the 4-connexity couples of adjacent pixels
in image $I$
$S'_I$=orderIncreasing($S_I, f$);
for i = 1 to |$S'_I$| do
 if(($R(p_i) \neq R(p'_i)$) and $P(R(p_i),R(p'_i)) == true$) then
  merge($R(p_i),R(p'_i)$);
```

The set of the pairs of adjacent pixel (S_I) is sorted according to the value of (3). Afterwards the algorithm takes every couple of pixels (p,p') of S_I and if the regions to which they belong $(R(p)\ and\ R(p'))$ were not the same and satisfactory (1), it merges the two regions.

SRM algorithm gives, for each segmented region, the list of pixel belonging to it and the related mean colour. We use this output as starting point to create a vectorial representation of image.

3. Contouring

To obtain a vectorial representation we have to find the border of segmented regions. This could be done more easily if we consider the pixels belonging to several groups (fig. 1). First of all pixels are divided in:

- *internal pixels*: pixels with all neighbours (in a 4-connexity scheme) belonging to the same region;
- *border pixels*: remaining pixels.

Due to the overall complexity of border regions a further setting into two groups is required:

- *close pixels*: pixels with at least an internal pixel as neighbour (in a 8-connexity scheme);
- *open pixels*: remaining pixels.

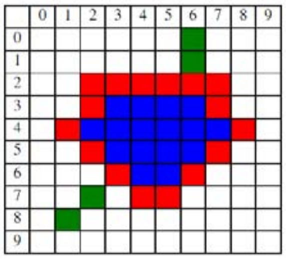

Figure 1: *An example of differend kind of pixels: internal (blue), close (red) and open (green).*

After having assigned each pixel to the corresponding category we describe regions in vectorial form. In particular there are two types of curves: *close curves* and *open curves*. In both cases we could approximate their boundaries through segments with eight possible directions (fig. 2).

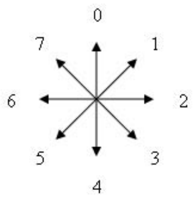

Figure 2: *Possible directions of segments that approximate the boundaries of regions.*

3.1. Close Curve

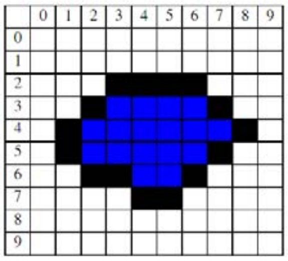

Figure 3: *An example of region with simple boundaries.*

A *close curve* is a curve made up of only *close pixels*. Initially we consider a simple configuration (fig. 3) to explain how segments could be found from *close pixels* list. The pseudo-code that describes the algorithm is the following:

```
initialPixel = findFirstPixel();
currentPixel = findNextPixel(initialPixel);
direction = findDirection(initialPixel, currentPixel);
segment = createNewSegment(initialPixel, direction);
while (currentPixel != initialPixel){
 oldCurrentPixel = currentPixel;
 oldDirection = Direction;
 currentPixel = findNextPixel(oldCurrentPixel);
 direction = findDirection(oldCurrentPixel, currentPixel);
 if (direction != oldDirection){
  setFinalCoordinate(segment, oldCurrentPixel);
  insertInSegmentsList(segment);
  segment = createNewSegment(oldCurrentPixel, direction);
 }
}
setFinalCoordinate(segment, currentPixel);
insertInSegmentsList(segment);
```

The functions are:

- `findFirstPixel()`: it chooses the top-left pixel as initial pixel.
- `findNextPixel(currentPixel)`: it looks for the next pixel in the neighbourhood following a counter clockwise direction.
- `createNewSegment(oldCurrentPixel, direction)`: it creates a new segment with first coordinate `oldCurrentPixel` and direction `direction`.
- `setFinalCoordinate(segment, oldCurrentPixel)`: it sets the final coordinate of segment `segment` at `oldCurrentPixel`.
- `insertInSegmentList(segment)`: it adds `segment` in the list of segments that describes the *close curve*.

Our algorithm chooses the top-left pixel of curve as initial pixel and following the boundary in counter clockwise direction creates the segments necessary for a vectorial description.

There are a series of complex configuration:

- regions with internal curve;
- ambiguity in the choice of next `currentPixel`;
- several *close curve* belonging to the same region;

To properly consider these cases it is needed to modify the algorithm described above in the following way:

```
while(closePixelsList contains some pixel);{
 currentCurve = createNewCurve();
 initialPixel = findFirstPixel();
 curveType = getCurveType(initialPixel);
 currentPixel = findNextPixel(initialPixel, curveType);
 direction = findDirection(initialPixel, currentPixel);
 segment = createNewSegment(initialPixel, direction);
 while (currentPixel != initialPixel);{
  oldCurrentPixel = currentPixel;
  oldDirection = direction;
  currentPixel = findNextPixel(oldCurrentPixel);
  direction = findDirection(oldCurrentPixel, currentPixel);
  if (direction != oldDirection){
   setFinalCoordinate(segment, oldCurrentPixel);
   addInCurve(currentCurve, segment);
   segment = createNewSegment(oldCurrentPixel, direction);
  }
 }
 setFinalCoordinate(segment, currentPixel);
 addInCurve(curve, segment);
 insertInCurvesList(currentCurve);
}
```

Some functions have been modified and other are new, in particular:

- `createNewCurve()`: it creates a new curve;
- `addInCurve(currentCurve,segment)`: it inserts in `currentCurve` the segment `segment`;
- `insertInCurvesList(currentCurve)`: it inserts `currentCurve` in the list of curves that describes the boundary of regions;
- `getCurveType(initialPixel)`: analyzing neighbours of `initialPixel` it returns the type of curve (internal or external);
- `findNextPixel(initialPixel,curveType)`: it looks for the next pixel in the neighbourhood following a clockwise or counter clockwise direction based on `curveType` value. If there is an ambiguity the pixel on the left is chosen. Moreover each pixel have a multiplicity equals to the number of regions they are near to. When a pixel is chosen its multiplicity is decremented by one and when it becomes 0 that pixel is deleted from the list of *close pixel*.

3.2. Open Curve

Even if a good segmentation algorithm should create regions with simple boundaries and not ragged this is not always the case. For this reason we have divided border pixels into two groups: *close pixels* and *open pixels*. The last are the pixels devoted to describe the ragged above mentioned.

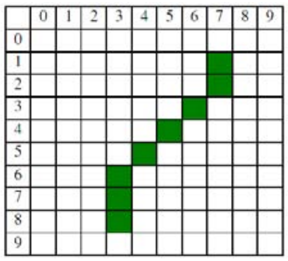

Figure 4: *An example of simple open curve.*

For simple configurations (fig. 4) we could use the following algorithm:

```
initialPixel = findFirstPixel();
currentPixel = findNextPixel(initialPixel);
direction = findDirection(initialPixel, currentPixel);
segment = createNewSegment(initialPixel, direction);
while (it is possible to find new nearby pixels){
 oldCurrentPixel = currentPixel;
 oldDirection = direction;
 currentPixel = findNextPixel(oldCurrentPixel);
 direction = findDirection(oldCurrentPixel, currentPixel);
 if (direction != oldDirection){
  setFinalCoordinate(segment, oldCurrentPixel)
  insertInSegmentsList(segment)
  segment = createNewSegment(oldCurrentPixel, direction)
 }
}
setFinalCoordinate(segment, currentPixel);
insertInSegmentsList(segment);
```

It is very similar to the one described for the *close curve*, but the following function has a different behaviour:

- `findFirstPixel()`: it choices a pixel with only a neighbour as initial pixel. Moreover when a pixel is chosen it is deleted from the list of *open pixels*.

For complex configuration is needed a more sophisticated algorithm:

```
while(openPixelsList contains some pixels){
 currentCurve = createNewCurve();
 initialPixel = findFirstPixel();
 currentPixel = findNextPixel(initialPixel);
 direction = findDirection(initialPixel, currentPixel);
 segment = createNewSegment(initialPixel, direction);
 while (secondList is not empty or it's the first
execution){
  if(there isn't new currentPixel and it isn't the first
execution){
   initialPixel = getPixelFromSecondList();
   currentPixel = findNextPixel(initialPixel);
   direction = findDirection(initialPixel, currentPixel);
   segment = createNewSegment(oldInitialPixel, direction);
  }
  while (it's possible to find a new currentPixel){
   oldCurrentPixel = currentPixel;
   oldDirection = direction;
   currentPixel = findNextPixel(currentPixel);
   direction = findDirection(oldCurrentPixel, currentPixel);
   if (direction != oldDirection){
    setFinalCoordinate(segment, oldCurrentPixel);
    addInCurve(currentCurve, segment);
    segment =createNewSegment(oldCurrentPixel, direction);
   }
  }
  setFinalCoordinate(segment, currentPixel)
  addInCurve(currentCurve, segment);
 }
 insertInCurvesList(currentCurve);
}
```

Some functions have been modified and other are new, in particular:

- `findNextPixel(initialPixel)`: it is similar to the precedent version, but it also adds some pixel in the `secondList` every time there is an ambiguity in the choice of the next pixel.
- `getPixelFromSecondList()`: it takes a pixel from the `secondList`.

4. SVG Generation Code

After tracking the curve boundaries is necessary to map the data by SVG primitives. In particular a generic *close curve* could be represented in the following way:

```
<path d="M x1,x1 L x2,y2 Lx3,y3 Z" stroke ="#RRGGBB"
fill"#RRGGBB" />
```

where `x1,y1,x2,y2,x3,y3` are the vertexes coordinates and `RR, GG, BB` are respectively the hex representation of red, green, blue mean value of the region that *close curve* belong to.

An *open curve* could be represented in the following way:

```
<path d="M x1,x1 L x2,y2 Lx3,y3" stroke ="#RRGGBB"
fill="none" />
```

Open curves are no filled (`fill = "none"`) and start point is not equal to final point (`Z` parameter is absent).

In order to obtain small file size some optimization could be done [Wor03]:

- `path` element permits to eliminate some separator characters, to use relative coordinate (`m, l` command) and `h, v` command for horizontal and vertical lines respectively;
- `<g>` elements is used to properly ensemble common graphic properties of various primitives.

To explain how these optimizations work we consider Figure (1). The SVG representation is the follow:

```
<path d="M1,4 L2,3 L2,2 L7,2 L7,3 L8,4 L5,7 L4,7 z"
stroke="#RRGGBB" fill="#RRGGBB"/>

<path d="M6,2 L6,0 M3,6 L1,8" stroke="#RRGGBB" fill="none"/>
```

The overall code length used is 148 characters. By using some optimizations we obtain:

```
<path d="M1,4l1,-1v-1h5v1l1,1l-3,3h-1z" stroke="#RRGGBB"
fill="#RRGGBB"/>

<path d="M6,2v-2M3,6l-2,2" stroke="#RRGGBB" fill="none"/>
```

The length of code is 133 characters. It is easy to see that both `path` element have the `stroke` colour in common. Using `<g>` properties we can write all code in the following way:

```
<g stroke="#RRGGBB">
 <path d="M1,4l1,-1v-1h5v1l1,1l-3,3h-1z" fill="#RRGGBB"/>
 <path d="M6,2v-2M3,6l-2,2" fill="none"/>
</g>
```

A further gain of 11 characters is obtained. Finally we use gzip compression on SVG documents to obtain a SVGZ file.

The overall segmentation described above, mainly when used to obtain a photorealistic representation, gives as output an high number of regions. The final gain of SVG optimization, in terms of the overall bit size, is proportional to such number.

5. Experimental Results

We have done several experiments in order to study the behaviour of our solution as the Q parameter increases. In particular we used two indexes of performance: PSNR and bit per pixel. The first index is tied with the quality of the image, the second with the file size.

We use Lena image to show the relation between Q parameter and some indexes of performance (PSNR, bpp). However analogous results have been found for other images. Figures (5) and (6) show that both quality and image size grow as the Q parameter increases. In fact increasing Q, SRM produces more regions and more details need to be represented. Moreover it is not useful increasing the Q parameter over a certain threshold (see Figure 7).

In Figure (8) Lena image vectorized with our solution and the original uncompressed bitmap image are showed. The quality of Lena vectorized is good enough and the file dimension is less than original image.

For sake of comparison SVG images obtained with our algorithm have been compared with other techniques such as Vector Eye [VLDP03], SWaterG [BCDN05], SVGenie and SVGWave [BBD*05]. The set of images used for comparison are of different kinds: graphics, textual, colours and grey levels. The parameters of different techniques have been set in order to obtain images visually agreeable. Figure (9) shows similar PSNR values for all the techniques. Fixing the quality, the file size produced by different techniques changes. In particular our algorithm outperforms the others (see Figure 10 and 11) producing files of lower size.

A different kind of experiment has been done with the recent VISTA technique [PS05]. In particular we have used for suitable comparison the data (original, SVG code) downloadable from `http://www.svgopen.org/2005/papers/Prasad_Abstract_R2V4SVG/`. For sake of comparison we fix the overall bit size of each image in order to properly tune our Q parameter. The final quality, measured (PSNR)

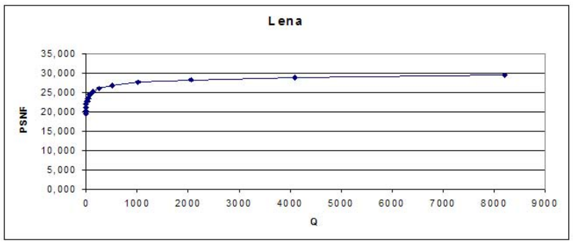

Figure 5: *Relation between PSNR and Q for Lena image.*

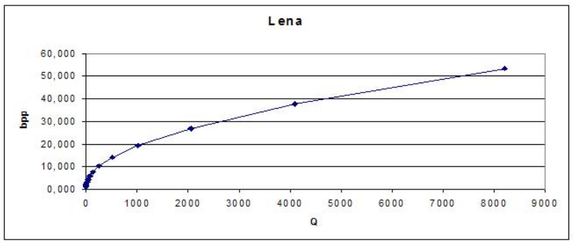

Figure 6: *Relation between bpp and Q for Lena image.*

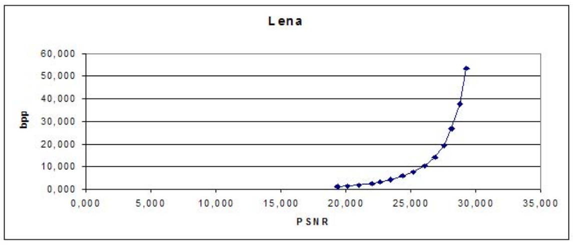

Figure 7: *Relation between PSNR and bpp for Lena image.*

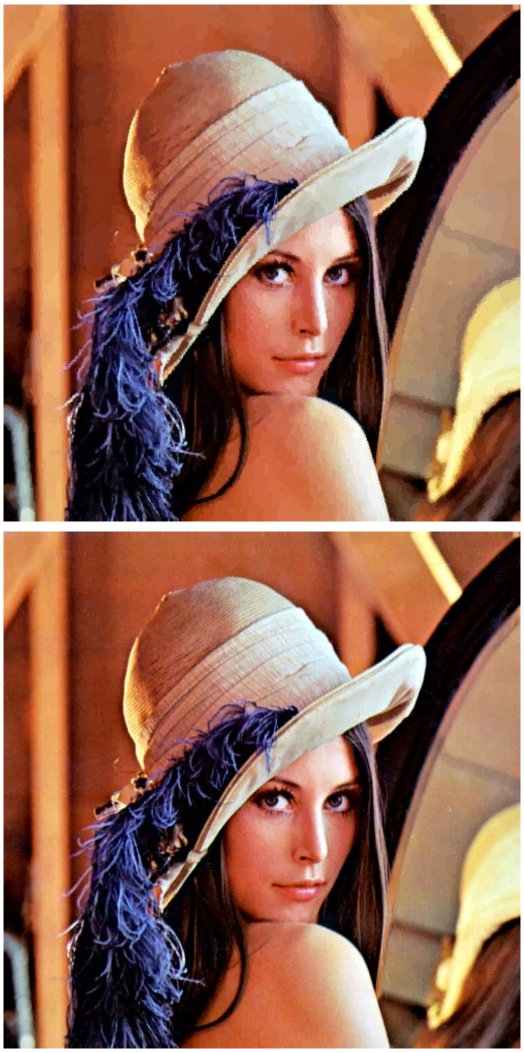

Figure 8: *Visual comparison between image Lena vectorized with our solution (up) and original bmp uncompressed image (down). The quality of image produced with our technique is good enough (PSNR = 27,62) and the svgz file size is 162 KB whereas original bmp file size is 768 KB.*

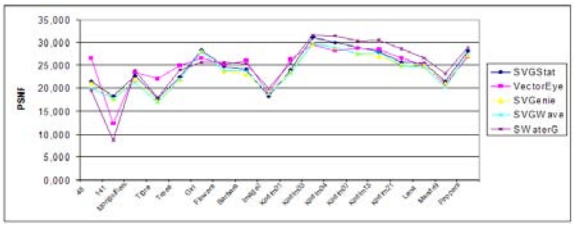

Figure 9: *PSNR comparison of dataset images outputs.*

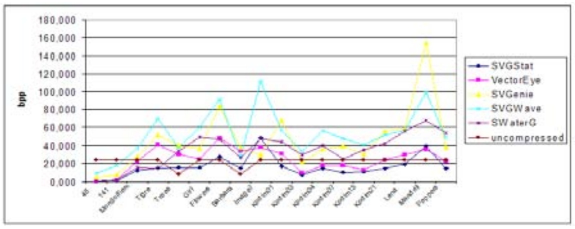

Figure 10: *Bit per pixels comparison of dataset images outputs.*

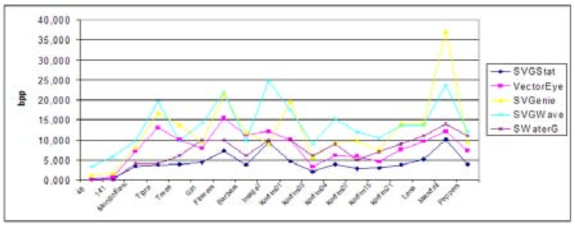

Figure 11: *Bit per pixels comparison of dataset images compressed outputs (SVGZ embedded coding).*

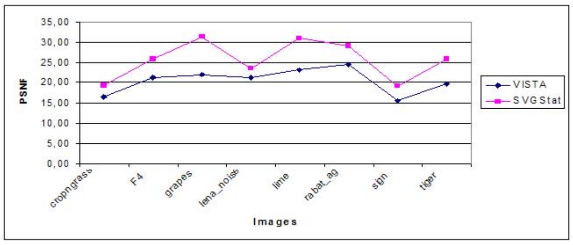

Figure 12: *PSNR comparison of our solution with respect to VISTA dataset images outputs.*

and perceptual, is clearly better then corresponding VISTA images as showed in Figure (12) and (13)

It is useful to consider that our algorithm has only a parameter (Q) that is easily tuneable to obtain the desired quality-size trade-off. Further experiments can be found at the following web address: `http://www.dmi.unict.it/~iplab`.

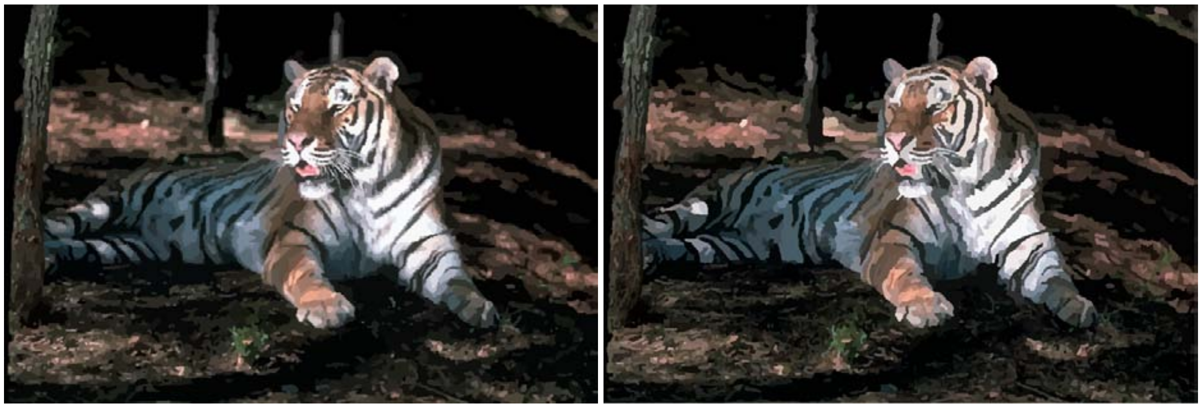

Figure 13: *Visual comparison between tiger image vectorized with our solution (left) (99 KB, PSNR=26,51) and VISTA SVG image (right) (94 KB, PSNR=19,69). The file size is almost the same but the quality of image produced with our tecnique is clearly better.*

6. Conclusion and Future Works

In this paper we have proposed a novel technique able to convert raster images in a SVG format. Moreover, we have carried out several experiments showing that our algorithm outperforms other similar techniques. Future researches will be devoted to studying advanced region merging heuristics, to use of Bezier curves and filter enhancement.

References

[BBD*05] BATTIATO S., BARBERA G., DI BLASI G., GALLO G., MESSINA G.: Advanced SVG Triangulation Polygonalization of Digital Images. In *Proceeding of SPIE Electronic Imaging-Internet Imaging VI-* (2005), vol. 5670.1.

[BCDN05] BATTIATO S., COSTANZO A., DI BLASI G., NICOTRA S.: SVG Rendering by Watershed Decomposition. In *Proceeding of SPIE Electronic Imaging-Internet Imaging VI-* (2005), vol. 5670.3.

[BDG*05] BATTIATO S., DI BLASI G., GALLO G., MESSINA G., NICOTRA S.: SVG Rendering for Internet Imaging. In *Prooceding of IEEE CAMP'05, International Workshop on Computer Architecture for Machine Perception* (July 2005), Palermo(Italy), pp. 333–338.

[DHH02] DUCE D., HERMAN I., HOPGOOD B.: Web 2D Graphics File Format. *Computer Graphics forum 21*, 1 (2002), 43–64.

[Kuh03] KUHL K.: Kvec 2.99, 2003. Copyright KK-Software, http://www.kvec.de.

[LM01] LUCCHESE L., MITRA S.: Color Image Segmentation: A State-of-the-Art Survey. In *Proc. of the Indian National Science Academy(INSA-A)* (Mar. 2001), vol. 67 A, pp. 207–221.

[NN04] NOCK R., NIELSEN F.: Statistical Region Merging. *IEEE Transaction on Pattern Analysis and Machine Intelligence 26*, 11 (NOVEMBER 2004), 1452–1458.

[NN05] NOCK R., NIELSEN F.: Semi-supervised statistical region refinement for color image segmentation. *Pattern Recognition 38*, 6 (June 2005), 835–846.

[PS05] PRASAD L., SKOURIKHINE A.: Raster to Vector Conversion of Images for Efficient SVG Representation. In *Proceedings of SVGOpen'05* (August 2005), NL.

[Qui03] QUINT A.: Scalable Vector Graphics. *IEEE Multimedia 3* (2003), 99–101.

[RB05] RABAUD V., BELONGIE S.: Big Little Icons. In *CVAVI* (2005), San Diego CA.

[VLDP03] VANTIGHEM C., LAURENT N., DECKEUR D., PLANTINET V.: Vector eye 1.0.7.6, 2003. Copyright SIAME e Celinea, http://www.siame.com, http://www.celinea.com.

[Web02] WEBER M.: Autotrace 0.31, 2002. GNU General Public License, http://www.autotrace.sourceforge.net.

[Wor03] WORLD WIDE WEB CONSORTIUM: *Scalable Vector Graphics (SVG) 1.1 Specification*, 2003. http://www.w3.org/TR/2003/REC-SVG11-20030114/.