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

Compressed SVG Representation of Raster Images Vectorized by DDT Triangulation

S. Battiato, F. Greco, S. Nicotra

Dipartimento di Matematica e Informatica, University of Catania, Italy
Viale A. Doria, 6 - 95125 Catania, Italy
{battiato, greco, snicotra} @ dmi.unict.it

Abstract

This paper presents a portable compression algorithm applied to raster data images properly triangulated by using DDT (Data Dependent Triangulation). In particular the input source data are encoded to be rendered by standard SVG (Scalable Vector Graphics) engine. The proposed compression strategy works reducing the overall entropy implementing some heuristic strategies to properly re-code the redundancy inside the mesh representation. The compressed data are enclosed into an SVG player able to both decompress and show the original image. Results show the effectiveness of the approach.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Applications

1. Introduction

The Scalable Vector Graphics (SVG) [SVG04] is an XML language for the description of bidimensional graphical objects. The SVG standard allows representing complex graphical scenes by a collection of graphic vectorial-based primitives, offering several advantages with respect to classical raster images such as: scalability, resolution independence, etc.. SVG format could find useful application in the world of mobile imaging devices, where typical camera capabilities should match with limited color/size resolutions displays. The core of the current SVG developments is the version 1.1. Actually the SVG 1.2 specification is under development and available in draft form. SVG Mobile Profiles (Basic and Tiny) devoted to resource-limited devices, are part of the 3GPP platform for third generation mobile phones. Major details can be found directly at the W3C site [SVG04]. An exhaustive overview of the recent SVG development and related applications can be found in the proceedings of SVG Open Conference [SOC05].

Recently, some advanced approaches have been proposed to vectorize raster images using an SVG engine as final rendering. See ([BBD*05],[BGM04],[BCDN05]) for more specific technical details about each method. A useful comparison is surveyed in [BDG*05].
In particular in [BBD*05] is presented a technique that approximates local pixel neighbourhood by making use of Data Dependent Triangulation (DDT) [DLR90]. The DDT replaces the input image with a set of triangles according to a specific cost function able to implicitly detect the edge details. The overall perceptual error is then minimized choosing a suitable cost function able to simplify triangulation. On the other hand the DDT is strictly connected to the original pixel positions; therefore the number of actual triangles is larger than the number of pixels. Although the quality achieved in this way is rather good the size of the resulting files may be very large. In [BBD*05] the DDT triangulation is furtherly processed by a polygonalization step devoted to minimize the dimensions of the resulting files by merging triangles together, according to specific similarity metrics, by reducing in this way the overall amount of redundancies.
In this paper we propose an alternative approach able to properly capture the redundancy of the mesh representation. The resulting file size of the vectorized images, also using native GZIP compression of SVG (SVGZ) is too big for practical purposes. The aim of this paper is to provide a portable and compressed version of the SVG output coming from DDT. The proposed compression uses both topological and physical redundancy used to represent the mesh. Our work is mainly inspired by [Ros99], [RS99] where several strategies and heuristics are proposed to compress a general triangular mesh. Further advanced approach for mesh optimization and compression can be found in [Ros03].
In our case starting from a triangular simplified 2D mesh, without holes, a simple visiting strategy that walks from one triangle onto an adjacent one has been designed. At each step it encodes whether the tip vertex and the left and right neighbours of the current triangle have already been visited. It encodes the complete connectivity of the triangle mesh in a sequence of symbols, by using a variable length encoding. Moreover the compressed information have been embedded into an SVG file that uses Ecmascript [SE99] to decompress and visualize the original data during execution, by using dynamic capabilities of the language.
The final result is a lossless compressed file, able to be decoded by any SVG rendering engine, that dynamically generates the corresponding vectorial images. Without any loss in terms of image quality the proposed technique reaches satisfactory results with respect to the final compression ratio; the overall bit-size is fully comparable with the original uncompressed raster image.

The paper is structured as follows: Section 2 briefly reviews the main details of DDT triangulation, while in Section 3 the overall mesh compression engine is described. Section 4 is devoted to the run-time

decomposition step. Section 5 reports some experimental results. Finally Sections 5 closes the paper addressing direction for future works.

2. Data Dependent Triangulation

A triangulation is a partition of a two-dimensional plane into a triangles set. The triangulation techniques, named Data Dependent Triangulation (DDT), try to optimize the approximation of a particular function, taking also advantage from the information obtained from the co-domain. The optimum triangulation research, by considering a fixed criterion, is a not well posed problem; however good results are obtained by considering locally optimal criterions. A triangulation is called "Optimal" if it is impossible to obtain a further improvement by swapping any edges.

The algorithm introduced by Lawson [Law77] is an iterative technique able to find a locally optimal triangulation. This approach inspects all the inner edges and exchanges those that reduce the total cost of the triangulation; the algorithm iterates the process until the total cost is still unchanged.

Another approach, based on Lawson's method and developed by Yu et alii [YMS01], has been used as starting point to develop the proposed approach:

1. Every pixel of the image is split into two triangles by using the diagonal with the lowest cost;
2. The obtained triangulation is furthermore adapted to the image data by executing *step 3* in an iterative process, which is stopped when a locally optimal triangulation is reached;
3. If the polygon formed by adjacent triangles is convex, the following look-ahead step is achieved:
   a. The diagonals are swapped if the swapping introduces a reduction of the edges cost sum;
   b. Otherwise, for each edge E_i of the polygon, with $i=2,...,5$ if the exchange of E_1 with its opposite diagonal E_1', and the simultaneous swapping of the edge E_i with its opposite diagonal E_i' decrease the costs of the edges of the polygon and the costs of the edges $E_6,\ldots, E_{13}$ of the adjacent triangles, then both exchanges are achieved (see Figure 1).

To estimate the triangulation, the optimality criterion is fixed using the following cost function:

$$Cost(E) = |\nabla P_1| \cdot |\nabla P_2| - \nabla P_1 \cdot \nabla P_2 \qquad (1)$$

where E is a common edge of two triangles and $\nabla P_i = (a_i, b_i)$ are the gradient of the planes P_i, that are the interpolating linear functions of the triangles. A few iterations are needed to guarantee an effective coupling between original edges distribution and related triangle vertexes map. In our case, to further speed-up the overall process we used the following approximation:

$$cost(e) = (|a_1| + |b_1|) \cdot (|a_2| + |b_2|) - (a_1 \cdot a_2 + b_1 \cdot b_2) \qquad (2)$$

that is cost effective. Such cost function is similar to the simplified cost function used in [DLR90] to describe the roughness of triangulated terrain (Sobolev seminorm).

Therefore the global triangulation cost is summarized in:

$$\text{cost}(T) = \sum_{\forall t_1, t_2 \text{ contiguous in T}} \left[(|a_1| + |b_1|) \cdot (|a_2| + |b_2|) - (a_1 \cdot a_2 + b_1 \cdot b_2) \right] \qquad (3)$$

The DDT approach aims to minimize the global triangulation cost. Three/Four iterations are usually needed to minimize the triangulation cost. Actually, after the fourth iteration the triangulation cost changes slightly introducing less improvement.

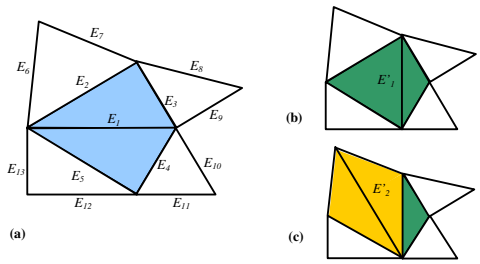

Figure 1 - *Example of look-ahead (a) Initial configuration, (b) Exchange of E_1 with the opposite diagonal E'_1. (c) Simultaneous exchange of E_1 with E'_1 and E_2 with E'_2.*

3. Triangular Mesh Compression

This section describes a generic compression scheme for digital images described by regular triangular mesh. Such technique will be applied to data coming from vectorization of raster images making use of DDT triangulation [BBD*05].

The main idea of the proposed approach is to compress data using the redundancy coming from the spatial information (triangles are connected), from semantic (objects are triangles with given features) and from storage describing object's syntax.

The spatial compression is achieved representing the triangles in a tree structure where nodes are the triangles while edges represent the connections. The mesh is connected, so it's possible to build a tree that cover the whole mesh. Moreover in a triangle each node is adjacent to its parent, that is they have two common vertices *p, q*, so it's convenient to store only the remaining vertex plus its orientation with respect to the father. Instead of storing absolutes coordinates of the point, we take into account the horizontal and vertical distance of the previous point in the triangle. Such representation allows saving space when triangulation is very fine, as in the case of images vectorized by DDT, where each original pixel belongs to 2 different triangles. Finally the tree representation is easily compressed by using an entropy encoder such as Huffman compression.

The input data is directly derived from a vectorial representation of a digital picture in an SVG format,

obtained applying the DDT triangulation using the technique described in ([BBD*05], [BGM04]) without considering the polygonalization step.
Triangular mesh are easily represented using the PATH SVG primitive; the input of the algorithm is defined by a set of lines, each of ones describe a triangle. For example the SVG primitive:

```
<path d="M 110 90 L 140 110 L120 110 Z"
fill="#ff0000">                                (4)
```

describes the triangle of vertexes (110,90 – 140,110 – 120,110) with red background color.
The complete syntax of the PATH primitive is given into Table 1.

Command: Name : Arguments : Description
M, **m** : moveto : x y : *Starts a new sub plot.* ***M*** *(uppercase) indicates that absolute coordinates will follow;* ***m*** *(lowercase) indicates that relative coordinates will follow*
L, **l** : lineto : x y : *Draws a line from actual point to (x,y).* ***L*** *(uppercase) indicates that absolute coordinates will follow;* ***l*** *(lowercase) indicates that relative coordinates will follow*
H, **h** : Horizontal lineto : *p* : *Draws an horizontal line from actual point (x,y) to point (x+p,y).* ***H*** *(uppercase) indicates that absolute coordinates will follow;* ***h*** *(lowercase) indicates that relative coordinates will follow.*
V, **v** : Vertical lineto : *p* : *Draws a vertical line from actual point (x,y) to point (x,y+p)* ***V*** *(uppercase) indicates that absolute coordinates will follow;* ***v*** *(lowercase) indicates that relative coordinates will follow.*
Z, **z** : Closepath : (none) : *Close the current subpath by drawing a straight line from the current point to current subpath's initial point.*

Table 1 - *Syntax of SVG PATH primitive.*

A triangular mesh is a geometrical structure composed by connected triangles having the following features:
- There aren't triangles with all vertices in the same line.
- Each triangle side can be adjacent with only a side of the neighbour triangle.
- If two triangles are adjacents then they have two common vertices.
- Each side on an internal triangle is adjacent to one side of the neighbour triangle.

An example of triangular mesh is given in Figure 2.

The overall process of the compression algorithm is composed by the following steps:

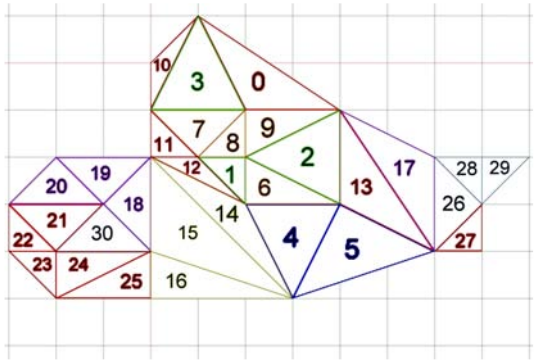

Figure 2 - *A triangular mesh.*

Step 1. Mesh Extraction from SVG files
Each triangle of the mesh is stored in a structure containing:
- vertices coordinates;
- RGB background color;

First image size is obtained from attributes *width* and *height* from the *svg* tag. For each PATH primitive, the *d* attribute is parsed obtaining the vertices coordinates.
Both absolute and relative movements into the canvas are taken into account checking if the object is a triangle and verifying also the triangular mesh constraints.
The background-color is picked-up from the *fill* attribute.

For example the triangle defined in (4), showed by using label 0 in Figure 2, is described as:

x1	y1	x2	y2	x3	y3	**R**	**G**	**B**
110	90	140	110	120	110	**255**	**0**	**0**

Step 2: Preprocessing Steps
At each step of the compression process, a triangle is extracted from the set. As already mentioned, it's important to know the orientation of the triangle with respect to the plane.
So the description of the triangles is rearranged to contain the coordinates of the basis, the right side and the left side respectively. The triangle 0 becomes:

x1	y1	x2	y2	X3	y3	**R**	**G**	**B**
140	110	110	90	120	110	**255**	**0**	**0**

In the meanwhile, the record that describes the triangle is updated adding pointers to its neighbours as showed below:

...	T1	T2	T3
…	Triangle 9	Triangle 3	Emply

Step 3. Mesh Visit
The role of the visit consists in the creation of the following elements:

- BASE_VERTEX: Coordinates of the starting node;
- VERTEX: Array of vertices of the triangles;
- CLASSIFY: Array of the triangles classification;
- R,G,B: Arrays of the background colours;

A triangle during the visit can be classified (see Figure 3 for a visual explanation) as follows:

- **Complete** (C): It is possible to return to the same triangle choosing always the right triangle.
- **Left** (L): The triangle has only a right adjacent triangle.
- **Right** (R): The triangle has only a left adjacent triangle.
- **Split** (S): A triangle that is not complete and has both left and right neighbours.
- **Empty** (E): The triangle has not neighbours.

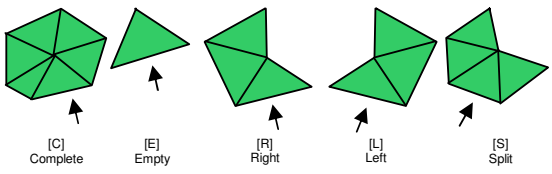

Figure 3 - *Triangle classification based on neighbourhood configuration.*

The pseudo code of the tree building is the following:

```
Procedure Visit_Mesh()
Let T the first triangle having at most
two adjacents.
Let P a stack;
BASE_VERTEX := Base(T);
Node := T
For i := 1 to #Triangles – 1;
VERTEX += opposite_to_gate_edge(Node);
Classification := Classify(Node);
UPDATE_NEIGHBOURS(Node);
CLASSIFY += Classification;
R += RED(Node);
G += GREEN(Node);
B += Blue(Node);
Switch Class
Case “C”: Next := Right(T);
Case “R”: Next := Left(T);
Case “L”: Next := Right(T);
Case “S”: Push(T,P); Next := Right(T)
Case “E”: Next := Left(Pop(T,P));
End
```

The UPDATE_NEIGHBOURS procedure removes from neighbours records, the pointers to the selected `Node`, while `opposite_to_gate_edge(Node)` function returns the coordinate of the point opposite to the edge adjacent to `Node`.

For example the compressed elements of the mesh in Figure 2 are:
BASE_VERTEX = ”140 110 30 20 20 0 255 0 0 C ”
VERTEX = ”20 0 | 0 10 | 10 -10 | 10 0 | -10 -10 | -10 -20 | 30 10 | 10 10 | 10 10 | 20 0 | 10 -10 | 10 0 | 0 -10 | -20 0 | 0 -10 | -30 -10 | -10 10 | -20 0 | -20 -10 | 0 20 | -20 -10 | -10 -10 | 0 -10 | -10 0 | -10 0 | 20 10 | 0 10
10 0 | 0 -10 | -20 -10 |”
CLASSIFY = ”S | E | C | R | C | S | C | C | R | R | C | R | R | S | R | E | E | C | R | S | L | S | E | L | E | C | R | R | R | E”
RED = ”255 | 1 | 1 | 0 | 0 | 14 | 0 | 71 | 0 | 0 | 171 | 0 | 0 | 0 | 0 | 14 | 84 | 241 | 0 | 1 | 85 | 255 | 172 | 0 | 0 | 255 | 1 | 0 | 255 | 1” (the GREEN and BLUE elements are similar)
The algorithm chooses the triangle #0 as starting point (it has two adjacents), such triangle is complete (C), since is it possible to return choosing the right triangle (i.e, #3,#7,#8,#9). So the BASE_VERTEX contains the coordinates: (140,110) that is the base point p; (30,20) the q distance from p; (20,0) the l distance from p. The other collected information are the RGB background color of the triangle and its classification.
The second triangle is #3, because it is **Right** with respect to the triangle #0. Such triangle is a **Split** triangle (S): it is not complete and has both left and right adjacents.
The node is pushed into the stack and the visit continues with triangle #10, that is clearly an **Empty** triangle (E). A pop from the stack returns the triangle #3 and the visit continues with **Left** with respect to #3 (i.e. triangle #7). Such triangle is **Complete**, while its right (triangle #11) is a **Right** triangle since it has only a left adjacent.
The complete sequence of visited triangles is:
| 0 | 3 | 10 | 7 | 11 | 12 | 14 | 15 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 16 | 30 | 4 | 5 | 13 | 17 | 26 | 27 | 28 | 29 | 2 | 9 | 8 | 1 | 6 |

Step 4 Huffman Compression
The overall data collected in the previous steps, have sensibly reduced the redundancy of the mesh representation coding a topological compression of the triangulation. In order to achieve better results each element is compressed separately using classical Huffman algorithm. The splitting does not give problems and it is based on a few simple consideration:

- VERTICES contains only distances from a point to another.
- CLASSIFY contains only 4 kinds of letters
- Both RED, GREEN, BLUE elements contains number between 0 and 255.

Such data together with the dictionaries to be used in the decoding phase, are the compressed information of the original image.

4. Decompression Step
The aim of the decompression step is to reconstruct the original mesh using an SVG representation starting from the compressed information.
In this paper, we present two schemes of decompression: the first approach follows back the steps described in the previous section: Huffman decompression of the elements, mesh visit of the tree starting from the root obtaining the full set of triangles, writing of triangles using SVG PATH primitive. This method has been named *offline decompression* and requires an external decoder that after having properly read the data performs all the steps before getting the final result.
The second and more innovative method is a general approach to store and process compressed elements within an SVG file.
All array elements contain description of a single triangle and in order to reconstruct the overall set of triangles only the coordinates of the adjacent sides are required. So,

starting from the mesh root, described by the BASE_VERTEX element, is it possible to:

- read the information relative to the successive triangle in the compressed arrays;
- decode the information using the element's dictionary;
- compute the triangle coordinates using the coordinates of adjacent edges in the preceding triangle and the corresponding type.
- draw the triangle in SVG canvas.

The underlying idea of the method is the storing of the compressed elements into an SVG file, implementing an auto-extracting system that while reads information, prints out the triangles in the SVG canvas. Then the decompression occurs without any external software to decode the compressed information and each triangle is visualized independently during the tree visit; for this reason the method has been defined as *online decompression*. In the following some details about the implementation are reported.

4.1 Writing Compressed Elements in SVG

SVG is an XML language, so SVG instructions are XML elements having attributes and content. Each compressed element can be inserted into the SVG file defining a new XML element having CDATA (Character data) content. In our application, the content of the element is the ISO 8859-1 (Latin-1) representation of the Huffman binary coding, while the attribute *id* describes the name of the vector. For example the array element VERTEX is defined as:

```
<my_data id="vertex">
<![CDATA[>_;]¿ UFv³X}1á?ßp²Öp<!]]>
</my_data>
```

4.2 SVG Scripting Capabilities

The SVG language uses ECMAScript as client scripting language [SE99]. ECMAScript is an object-oriented programming language for performing computations and manipulating computational objects within a host environment.

An important feature of ECMAScript is that is possible to read and write SVG elements trough the Document Object Model (DOM). The function to read XML elements and put into a variable is :

```
function read_Node(id){
 node=svgDocument.getElementById(id);
 node_content=node.getFirstChild();
 node_content_data= node_content.data;
 return node_content; }
```

while the function to create a new triangle is:

```
function
build_mesh(x1,y1,x2,y2,x3,y3,r,g,b) {
var d ="M"+x1+","+y1+" "+x2+","+y2+"
"+x3+","+y3+" Z";
var shape=
svgDocument.createElementNS(svgns,
"path");
var color="rgb("+r+","+g+","+b+")";
 shape.setAttributeNS(null, "d", d);
 shape.setAttributeNS(null,
"fill",color);
 shape.setAttributeNS(null,
"stroke",color);
svgDocument.documentElement.appendChild
(shape);}
```

4.3 Online Decompression

Let *c_triangle(x1,y1,x2,y2,x3,y3,r,g,b,type)* the array containing the information to build the current triangle. The array *c_triangle* is initialized directly from the BASE_VERTEX array. To reconstruct the others *N-1* triangles (where *N* is the number of triangles) the following procedure is used:

```
Function get_triangle(i)
(x,y) = get_coordinates(VERTEX)
k = get_classification(CLASSIFY)
(r,g,b)=get_colors(R,G,B)

switch (c_triangle[9])
case "C" or "L"://Right of preceding
c_triangle[0]=c_triangle[4];
c_triangle[1]=c_triangle[5];
c_triangle[4]=c_triangle[4]-x;
c_triangle[5]=c_triangle[5]-y
c_triangle[6]=r:c_triangle[7]=g;
c_triangle[8]=b; c_triangle[9]=k;
case "R": // Left of the preceding
c_triangle[2]=c_triangle[4];
c_triangle[3]=c_triangle[5];
c_triangle[4]=c_triangle[0]-x;
c_triangle[5]=c_triangle[1]-y;
c_triangle[6]=r:c_triangle[7]=g;
c_triangle[8]=b; c_triangle[9]=k;
case "S": // Derives from a split
push(triangle_c); //save into a stack
c_triangle[0]=c_triangle[4];
c_triangle[1]=c_triangle[5];
c_triangle[4]=c_triangle[4]-x;
c_triangle[5]=c_triangle[5]-y;
c_triangle[6]=r:c_triangle[7]=g;
c_triangle[8]=b; c_triangle[9]=k;
case "E" // After an empty.
c_triangle=pop; // get from the stack
c_triangle[4]= c_triangle[0]-x;
c_triangle[5]= c_triangle[1]-y;
c_triangle[6]= r; c_triangle[7]=g;
c_triangle[8]=b; c_triangle[9]=k;
}
build_mesh(c_triangle);
```

In the pseudo code, the *get_* functions are designed to get information about the next triangle to elaborate. The input parameter is a set of array element. For example *get_classification(CLASSIFY)* returns the type of the next triangle to be processed. Circular buffers are used to optimize memory space.

5. Experimental Results

The proposed method of coding triangular mesh through SVG primitives has been implemented and tested over a demonstrative set of digital images. All images have been vectorized by DDT triangulation, without using merging, obtaining the corresponding SVG version. Afterwards the overall set has been compressed using the technique described in Section 3.

Table 2 summarizes the overall results, showing also the thumbnails of the compressed images. For the offline decompression scheme, five different files, containing the compressed elements are properly generated. The computed overall bit-size of the compression is obtained by summing each file size, while the final bit-size of the online compression is the bit-size of the auto-extracting SVG(z) file.

In Figure 4(a) a graphical comparison of the compression ratio between the SVG file produced by DDT transformation with respect to the simple gzipped version (SVGZ), the offline method, the online compression and its gzipped version is shown. The proposed method produces encouraging results clearly outperforming the widely used GZIP compression because it is able to capture the remaining redundancy of the mesh representation. It's important to notice how the auto-extracting script in the online compressed SVG produces a small size overhead that is almost null when a further GZIP compression is applied. Figure 4(b) compares graphically the bit per pixels (bpp) of the coded file proving how the SVG compressed DDT transformation produces files whose size are comparable with standard raster image format both compressed and not.

The only drawback of the on-line method is the overall running time because its performances are mainly related with the rendering engine implementation. Actually about 60-90 seconds are needed to draw a single image, with medium resolution (e.g., 400*400, 24 bit-depth). On the contrary the off-line counterpart that we have implemented in ANSI C-language run in almost a few seconds.

6. Conclusions and Future Works

In this paper a lossless compression method able to reduce the entropy of a regular SVG mesh of data is presented. In particular the proposed technique has been applied to vectorial images obtained by using DDT triangulation as described in ([BGM04], [BCDN05]).

The decompression scheme has been implemented by using the dynamic characteristics of the SVG standard making use of EcmaScript capabilities. For each compressed image an auto-extracting *.svgz* file able to reconstruct the original data is built. Experimental results confirm the effectiveness of the proposed method.

Future works will include the generalization of the proposed method to any kind of image partitioning together with a detailed analysis of the timing performances of the final Script code in order to speed-up, whenever possible, the run-time execution.

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 Proceedings of IEEE CAMP'05, International Workshop on Computer Architecture for Machine Perception – (2005).

[BGM04] BATTIATO S., GALLO G., MESSINA G., SVG Rendering of Real Images Using Data Dependent Triangulation. In Proceedings of ACM SCCG'04 - Spring Conferences on Computer Graphics, Budmerice, Slovak Republic, (2004).

[DLR90] DYN N., LEVIN N., RIPPA S., Data Dependent Triangulation for Piecewise Linear Interpolation. IMAJ. Numerical Analysis, vol.10, pages 137-154, (1990).

[Law77] LAWSON C. L., Software for C1 Surface Interpolation. Mathematical Software III, J.R. Rice, Academic Press, pages 161-194, New York, (1977).

[Ros99] ROSSIGNAC J., Edgebreaker: Connectivity compression for triangle meshes. IEEE Transactions on Visualization and Computer Graphics, vol. 5, No. 1, pp. 47-61, (1999).

[Ros03] ROSSIGNAC J., Surface simplification and 3D geometry compression. Chapter 54 in Handbook of Discrete and Computational Geometry (second edition), CRC Press, Editors: J. E. Goodman and J. O'Rourke. 2003.

[RS99] ROSSIGNAC J., SZYMCZAK A., Wrap&Zip decompression of the connectivity of triangle meshes compressed with Edgebreaker. Journal of Computational Geometry, Theory and Applications, Volume 14, Issue 1-3, pp. 119-135,(1999).

[SE99] Standard ECMA-262 - ECMAScript Language Specification, http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf ,1999.

[SOC05] SVG OPEN CONFERENCE AND EXHIBITION, Annual Conference on Scalable Vector Graphics, http://www.svgopen.com, 2005.

[SVG04] SCALABLE VECTOR GRAPHICS (SVG), XML GRAPHICS FOR THE WEB http://www.w3c.org/Graphics/SVG, 2004.

[YMS01] YU X., MORSE B.S., SEDERBERG T.W., Image Reconstruction Using Data Dependent Triangulation, Journal IEEE CG&A, vol. 21 (3), pages 62-68, 2001.

Filename	Thumbnail	RESOLUTIO	PPM RAW	PNG	DDT SVG	DDT SVGZ	OFF LINE	ONLINE	SVGZ ONLINE
Kodim15		400x266	319238	166254	3929061	557490	118895	148451	117110
Lena		400x400	480038	301316	5790565	970851	203282	250424	204284
Peppers		400x400	480038	269451	5790562	954099	186464	229911	187893
Kodim23		400x266	319238	157694	3829077	605742	121774	151860	121285
Flowers		400x350	360038	228072	4326777	748225	159256	198644	162801

Table 2 – *Final bit size (in bytes) of 5 digital images coded in different modalities.*

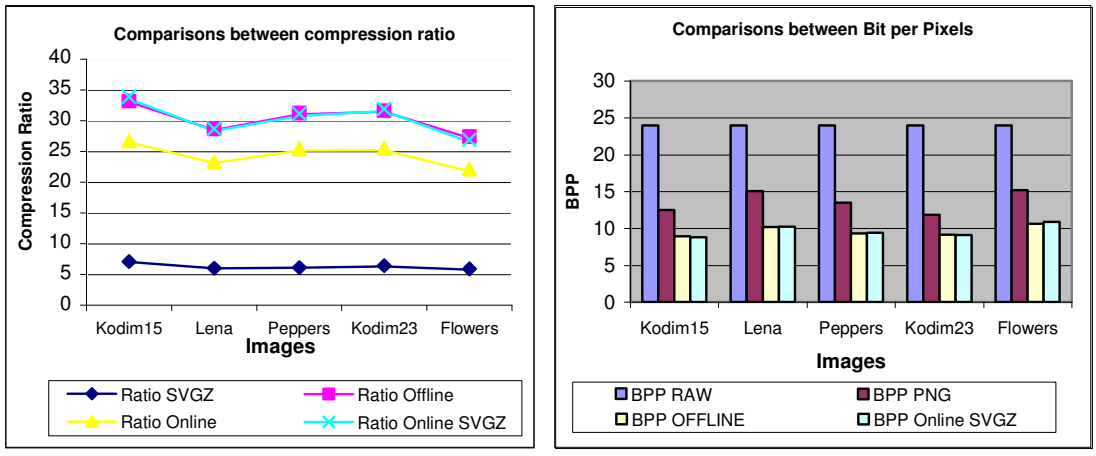

Figure 4 – *Experimental results: a) Comparisons of compression ratio with respect to the file size obtained by DDT Triangulation b) Final bit-size comparison in terms of bit per pixel.*