◊ VI.4

Efficient and Robust 2D Shape Vectorization

Jean-François Doué
HEC
Paris, France

Ruben Gonzalez Rubio
University of Sherbrooke
Canada

◊ Introduction ◊

In computer graphics applications, it is often convenient to be able to convert a shape from its bitmap representation to a vectorized form. Indeed, a vectorized form has many advantages over a bitmap: It can be transformed without loss of resolution, and it is sometimes better suited for the needs of computer vision. This gem presents an algorithm to perform the vectorization of arbitrary 2D shapes (a shape is defined here as "a set of contiguous pixels of the same color"). We developed the algorithm for a computer vision application: The purpose was to study how to train a neural network to recognize arbitrarily scaled handwritten digits and required the network to be presented with a more meaningful representation of the data than just a raw bitmap (Michaud *et al.* 1993).

The first part of this gem summarizes the problem, explains how it is usually solved, and shows the limits of this technique. The second part presents a simple yet very useful extension of the algorithm, based on pattern matching, that solves the limitations raised in the first part. It uses postprocessing rules somewhat similar to the production rules found in formal systems. Finally, a third section makes some remarks about vectorization that can be useful.

◊ Summary of the Problem ◊

Vectorization transforms a shape into a collection of vectors that delimits its boundaries. Since vectorization starts from a coarse approximation of the shape (its rasterization on a grid of pixels), it can only produce a limited number of vectors, which can be encoded as integers in the $[0, 7]$ range (see Figure 1). The collection is also sometimes

Copyright © 1995 by Academic Press, Inc.
All rights of reproduction in any form reserved.
IBM ISBN 0-12-543455-3
Macintosh ISBN 0-12-543457-X

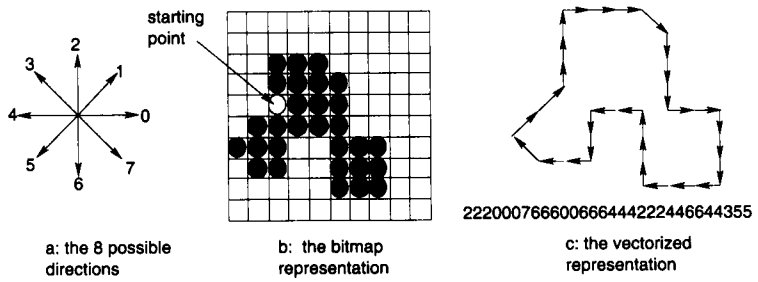


Figure 1. The original algorithm.

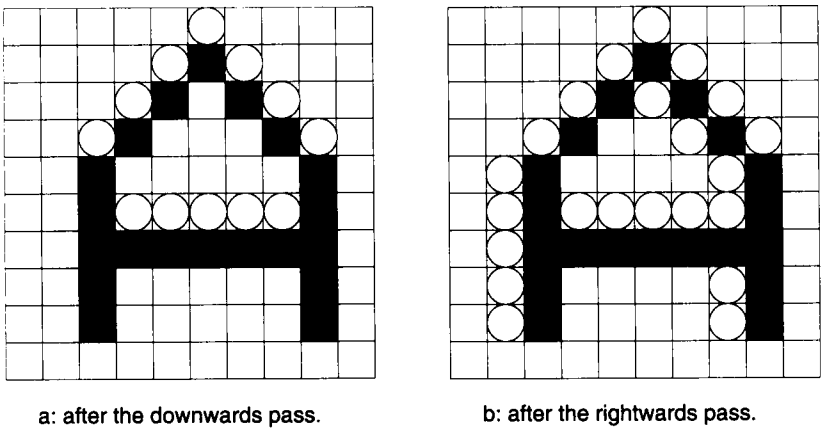


Figure 2. Generating the contour.

called "chain code." The encoding process has already been described by others (Plaziac 1991, Ballard and Brown 1992), but deserves some explanation.

The Usual Algorithm

First, the algorithm generates the boundaries. It does so in four phases, scanning the bitmap in a particular direction (leftward, rightward, upward and downward) at each phase. Each phase adds a layer of pixels to the shape any time it comes across a boundary, a bit as if it were "snowing" on the shape. Figure 2 shows how each of the first two phases adds pixels to the bitmap to progressively determine the contour of the shape. The other two passes work exactly the same way.

Finally, the algorithm finds a pixel of the contour of the shape and may now enter its main loop (see the following pseudocode).

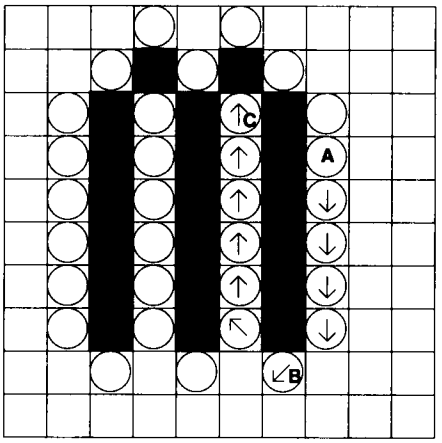


Figure 3. A dead end.

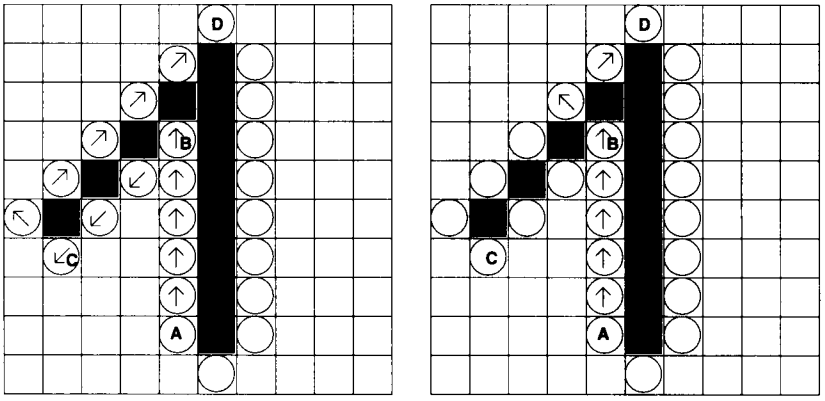
```
loop
    mark(p)                          // Mark present location.
    for idelta = 0 to 7              // Index of neigbor offsets, ccw.
    delta = ccw[idelta]              // Get unit offset vector.
    if boundary(p+delta)             // Found neighboring boundary.
        addChain(delta)              // Record motion.
        p := p + delta               // Update location (vector add).
        endif
while movesRemain()
```

Limitations

This algorithm, though it sets the ground rules of chain code generation, is not perfect. It especially has problems with thin bitmaps or bitmaps that present very narrow canyons. Following are two examples of the kind of problems that usually arise.

Problem 1. In the first example, the algorithm is applied to a small, lower-case `m` (see Figure 3). The algorithm starts normally at point A, then comes close to a very narrow canyon (only one pixel wide), at point B. Since it is checking the directions counterclockwise, it falls into the canyon. However, as it continues deeper into the canyon, marking the pixels along its way, it does not notice that it is blocking its exit. The algorithm terminates at the bottom of the canyon (point C), leaving two-thirds of the `m` unvectorized!

Problem 2. In the second example, the algorithm is applied to a `1`, starting at point A (see Figure 4). Theoretically, a good algorithm should walk up a vertical line until



a: the right path b: a short cut

Figure 4. A leak.

point B, then take the "staircase" leftward until point C and then go back up until point D. However, as it reaches point B, the standard algorithm takes a short cut, and goes directly to point D. This comes from the fact that it checks all the directions counterclockwise: For thin 45° lines, it has no way of knowing whether route C or route D is the best one.

◊ The New Algorithm ◊

A very simple extension will solve the two problems. The trick is to subdivide every pixel of the bitmap into a 4×4 grid of "smaller" pixels, as shown in Figure 5 (anything greater than 2×2 (3×3 and beyond) would work too, requiring less memory). However, four is a more practical number, since computers can multiply and divide by four very easily using bit-shifts.

At the new scale, problem 1 simply disappears. In a 4×4 world, there are no more narrow canyons. All canyons are at least four pixels wide. There is no risk of getting stuck.

Problem 2 still persists, but becomes simpler. In the 4×4 world, the straight moves (left, right, up, down) always have priority over the oblique moves (left-up, right-up, left-down, right-down). Oblique routes should only be taken as a last resort, when no straight route is possible. Therefore, problem 2 is easily solved by changing the order in which the directions are checked. The rightward, upward, leftward, and downward directions should always be checked first, and only then the oblique directions.

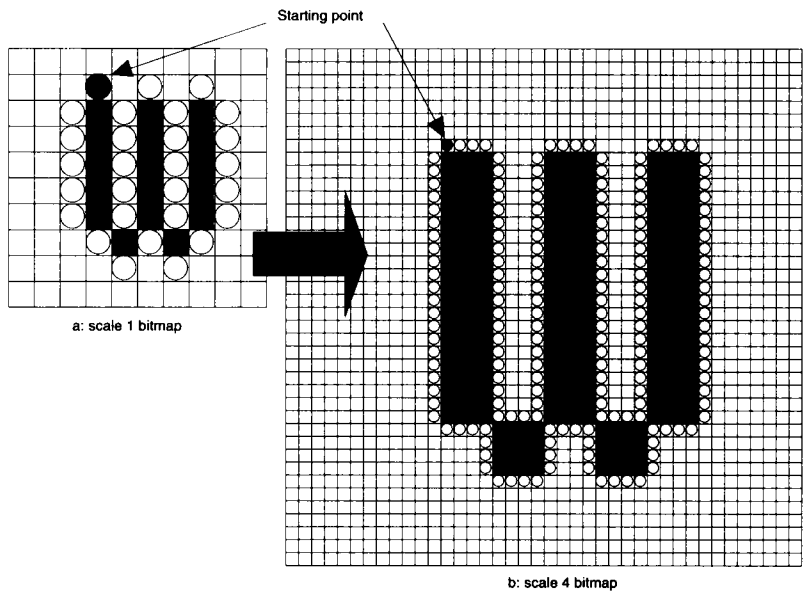


Figure 5. The new algorithm.

Shortcomings

The reader will have at least three objections regarding the new method:

1. The new algorithm generates a chain code at 4× that is much longer than the chain code at 1× and very different from it.
2. The chain code is wrong. Suppose the algorithm is applied to an oblique 45° line at 1×. Then at 4× one will end up with the chain code for a big staircase instead of the chain code for an oblique line.
3. The new algorithm will be extremely slow. Given a scale factor of 4×, there are sixteen times as many pixels to process.

Two Additional Tricks

It turns out that two rather simple tricks will overcome most of the problems raised by these three remarks.

1. The first trick is based on an interesting property of the chain code at 4× and solves the first two objections. The chain code can be *very* easily postprocessed to obtain a 1× chain code, using three simple rules of pattern searching. Using C to indicate a code value, if the input is a code at 4× and the desired output is a code at 1×, then the reduction rules are as follows:

CCCC	→	C	reduce to one copy,
CCC	→	{}	eliminate,
CC	→	CC	(ignored),
C	→	C	identity,

in which the fourth pattern rule (identity) subsumes the third.

A simple example will prove the efficiency and the simplicity of this postprocessing technique (see Figure 5). Scanning manually from Figure 5(a), one naturally obtains the following code:

```
766662222176666222217666655353322221
```

A good algorithm should be able to achieve the same results. The original algorithm would certainly fail, since the figure presents both problems 1 and 2. The new algorithm applied to the shape yields the following results:

4× chain code [Figure 5(b)]:

```
000 7 6666 6666 6666 6666 666 000 2222 2222 2222 2222 222 1
000 7 6666 6666 6666 6666 666 000 2222 2222 2222 2222 222 1
000 7 6666 6666 6666 6666 666 5 444 666 5 444 3 222 444 666 5
444 3 222 444 3 2222 2222 2222 2222 222 1
```

After postprocessing the 4× code, one obtains the exact result.

Postprocessed 1× chain code [Figure (5a)]:

```
766662222176666222217666655353322221
```

2. The second trick does not completely solve the last objection (there is truly more processing to do), but it helps considerably. The original algorithm spent most of its time blindly testing the eight possible directions of motion, which was not very efficient. The reader will notice that the chain code at 4× presents many long segments of identical continuous values, meaning that the algorithm often keeps on moving in the same direction for at least a few steps. By keeping track of the last direction in which it was moving and using it as a first guess for the next move, the new algorithm can save many computations.

◊ Additional Remarks ◊

Computing the Derivative

It is often interesting to compute the (discrete) derivative of the chain code, since it is invariant under boundary rotations. The derivative of the chain code reflects the

change in the orientation of the tangent to the contour as one moves from one point of the contour to the next. The derivative cannot simply be the difference between two consecutive direction codes, since this would imply many discontinuities. For instance, if the current direction is 7 and the next one is 0, the change should not be recorded as $0 - 7 = -7$, but rather as -1, the shortest path on the trigonometric circle. To do this, we will use the following implementation:

$$C'_k \leftarrow (C_k - C_{k-1} + 12) \bmod 8 - 4. \tag{1}$$

The reader should be careful with the mod function, which is here meant as the genuine modulo function for relative integers (i.e., $-2 \bmod 8 \equiv 6$ and not -2, as some math packages will compute. The $a \bmod b$ function should return the smallest integer r in the $[0, b-1]$ range such that $bq + r = a$).

The Problem of the Starting Point

A difficult problem to solve is that of the starting point. Since the chain code reflects the contour of the object, it is indeed cyclic. A suitable starting point must be found. Ideally, the behavior of the algorithm should be invariant under boundary rotation. A canonical form is quite important for certain applications such as optical character recognition (OCR). The C++ code presented below chooses the contour pixel closest to the upper-left corner of the bounding box of the shape. However, this is just a convention and certainly not the perfect choice.

Extensions

Though the present method finds merely the contours of 2D objects, voids can be identified following the generation of the shape's contour. The postprocessing of the chain code using a simple pattern matcher is a powerful technique worthy of further study. Extended rules sets, particularly those useful for different scale factors, deserve attention. The interested reader is directed to standard treatments (Olson 1992, Schneider 1990, Feldman 1992, Freeman 1961) for supplementary information.

◊ C Code ◊

pt2.H

```
typedef struct pt2Struct{
  int x,y;
} pt2;

extern pt2* addPt2(pt2 *a, pt2 *b, pt2 *c);
extern pt2* subPt2(pt2 *a, pt2 *b, pt2 *c);
```

pt2.C

```
#include "pt2.h"

/*****************************************************************/
/*                                                               */
/* Two utility functions to add and subtract 2D integer points.*/
/*                                                               */
/*****************************************************************/

pt2* addPt2(pt2 *a, pt2 *b, pt2 *c)
{ c->x = a->x + b->x; c->y = a->y + b->y; return c; }

pt2* subPt2(pt2 *a, pt2 *b, pt2 *c)
{ c->x = a->x - b->x; c->y = a->y - b->y; return c; }
```

chainCode.H

```
#define DEFAULT_CODE_LENGTH 512
#define SCALE 4

class chainCode{
    public:
        char* code;
        int   length;

        chainCode();
        ~chainCode();
        void add(char c);
        chainCode* postProcess();
        void printSelf();
};
```

```
\paragraph{chainCode.C}
\begin{codingeightpt}
#include <stdlib.h>
#include <stdio.h>
#include "chainCode.h"

/*****************************************************************/
/*                                                               */
/* Class constructor.                                            */
/*                                                               */
/*****************************************************************/

chainCode::chainCode()
{
code = malloc(DEFAULT_CODE_LENGTH * sizeof(char));
```

```
code[0] = '\0';
length = DEFAULT_CODE_LENGTH;
}

/*************************************************************/
/*                                                           */
/* Class destructor.                                         */
/*                                                           */
/*************************************************************/

chainCode::~chainCode()
{ free(code); }


/*************************************************************/
/*                                                           */
/* This method appends a new code to the chain. If there     */
/* is not enough memory left, the function doubles the size  */
/* of the chain code.                                        */
/* It receives as a parameter the new code to be added (c).  */
/*                                                           */
/*************************************************************/

void chainCode::add(char c)
{
int l = strlen(code);

if (l >= length-1){
    length *= 2;
    code = realloc(code, length);
    }
code[l] = c;
code[l+1] = '\0';
}


/*****************************************************************/
/*                                                               */
/* This method post-processes a 4x chain code to generate a 1x   */
/* chain code. A pointer to the 1x code is returned. The method  */
/* uses the 4 following rules:                                   */
/* CCCC ->  C :  reduce to one copy                              */
/*  CCC -> {} :  eliminate                                       */
/*   CC -> CC :  (ignored)                                       */
/*    C ->  C :  identity                                        */
/*                                                               */
/*****************************************************************/

chainCode* chainCode::postProcess()
{
int         i = 0, j;
chainCode   *filtCode;
```

```
filtCode = new chainCode();
while (i<length){
    if (i+SCALE-1 < length){
        for (j=0; j<SCALE-1; j++)
            if (code[i+j] != code[i+j+1])
                break;
        if (j == SCALE-1){
            filtCode->add(code[i]);
            i += SCALE;
            continue;
            }
        }

    if (i+SCALE-2 < length){
        for (j=0; j<SCALE-2; j++)
            if (code[i+j] != code[i+j+1])
                break;
        if (j == SCALE-2){
            i += SCALE-1;
            continue;
            }
        }
    filtCode->add(code[i]);
    i++;
    }
return filtCode;
}

/*********************************************************************/
/*                                                                   */
/* A utility method to display the chain code                        */
/*                                                                   */
/*********************************************************************/

void chainCode::printSelf()
{ printf("\n%s", code); }
```

vectorize.C

```
#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include "chainCode.h"
#include "pt2.h"

/* DEFINITION OF THE CONSTANTS */

#define CONTOUR 'c'
#define VISITED 'v'
#define BLACK '1'
#define WHITE '0'
```

```
/* DEFINITION OF THE MACROS */

#define PIX(a,b) ((b) * f_size.x + (a))
#define PIX2(a,b) ((b) * size->x + (a))
#define MIN(x,y) ((x)<(y) ? (x) : (y))
#define MAX(x,y) ((x)>(y) ? (x) : (y))


/*************************************************************/
/*                                                           */
/* This is the main function. It receives as a parameter a   */
/* bitmap image of size 'size' and outputs a chain code.     */
/* The following constraints are placed on the bitmap:       */
/* + Each pixel is encoded as a char.                        */
/* + Only white (0) and black (1) pixels are taken into      */
/*   account.                                                */
/* + The shape to encode should have no holes and should be  */
/*   in a single piece.                                      */
/*                                                           */
/*************************************************************/

chainCode* encode(pt2 *size, char *bitmap)
{
static pt2    contour_dir[8] = {{ 1, 0},
                                { 0,-1},
                                {-1, 0},
                                { 0, 1},
                                { 1,-1},
                                {-1,-1},
                                {-1, 1},
                                { 1, 1}};
chainCode   *code1,
            code4;
char        *fatmap,
            direction_code[8] = {'0','2','4','6','1','3','5','7'};
int         i,j,u,v,
            flag,
            d, distance,
            last_dir;
pt2         pixel,
            test_pixel,
            start_pixel,
            f_size,
            bbox[2] = {{INT_MAX, INT_MAX},
                       {-INT_MAX, -INT_MAX}};

/* CREATE AN EMPTY CHAIN CODE TO RETURN THE RESULT */
code1 = new chainCode();

/* RESCAN THE BITMAP AT A GREATER RESOLUTION (4x4 GREATER) */
/* ADD TWO BLANK LINES TO THE LEFT, RIGHT, TOP AND BOTTOM  */
/* OF THE FATMAP. THESE COULD BE NECESSARY TO AVOID THE    */
/* CONTOUR TO BE DRAWN OUTSIDE OF THE BOUNDS OF THE MATRIX */
```

```
f_size.x = 2 + SCALE*size->x + 2;
f_size.y = 2 + SCALE*size->y + 2;
fatmap = malloc(f_size.x * f_size.y * sizeof(char));
for (i=0; i<f_size.x * f_size.y; i++)
    fatmap[i] = WHITE;
for (j=0; j<size->y; j++)
    for (i=0; i<size->x; i++)
        if (bitmap[PIX2(i,j)] == BLACK)
            for(v=0; v<SCALE; v++)
                for(u=0; u<SCALE; u++)
                    fatmap[PIX(2+4*i+u, 2+4*j+v)] = BLACK;

/* GENERATE THE CONTOUR OF THE BITMAP USING 4 SUCCESSIVE */
/* PASSES: FOR EACH DIRECTION, WE SCAN EACH LINE UNTIL   */
/* WE REACH A BLACK PIXEL: THE PIXEL JUST BEFORE IT IS A */
/* CONTOUR PIXEL                                         */

/* PASS 1: LEFTWARDS */
for (j=0; j<f_size.y; j++)
    for(i=1; i<f_size.x; i++)
            if (fatmap[PIX(i,j)] == BLACK){
                if (flag == 0) {
                    fatmap[PIX(i-1, j)] = CONTOUR;
                    flag = 1;
                    }
                }
            else
                flag = 0;

/* PASS 2: RIGHTWARDS */
for (j=0; j<f_size.y; j++)
    for (i=f_size.x - 1; i>=0; i--)
            if (fatmap[PIX(i,j)] == BLACK){
                if (flag == 0) {
                    fatmap[PIX(i+1, j)] = CONTOUR;
                    flag = 1;
                    }
                }
            else
                flag = 0;

/* PASS 3: DOWNWARDS */
flag = 0;
for (i=0; i<f_size.x; i++)
    for (j=0; j<f_size.y; j++)
            if (fatmap[PIX(i,j)] == BLACK){
                if (flag == 0) {
                    fatmap[PIX(i, j-1)] = CONTOUR;
                    flag = 1;
                    }
                }
            else
```

```
                flag = 0;

/* PASS 4: UPWARDS */
flag = 0;
for (i=0; i<f_size.x; i++)
    for (j=f_size.y - 1; j>=0; j--)
            if (fatmap[PIX(i,j)] == BLACK){
                if (flag == 0) {
                    fatmap[PIX(i, j+1)] = CONTOUR;
                    flag = 1;
                    }
                }
            else
                flag = 0;


/* COMPUTE THE BOUNDING BOX OF THE CHARACTER (L,T,R,B) */
for (j=0; j<f_size.y; j++)
    for(i=1; i<f_size.x; i++)
        if (fatmap[PIX(i,j)]==CONTOUR){
            bbox[0].x = MIN(i, bbox[0].x);
            bbox[0].y = MIN(j, bbox[0].y);
            bbox[1].x = MAX(i, bbox[1].x);
            bbox[1].y = MAX(j, bbox[1].y);
            }

/* DETERMINE THE CONTOUR PIXEL CLOSEST TO THE UPPER LEFT CORNER */
/* OF THE BOUNDING BOX                                          */

distance = INT_MAX;
for (j=0; j<f_size.y; j++)
    for(i=1; i<f_size.x; i++)
        if (fatmap[PIX(i,j)]==CONTOUR){
            d = (i-bbox[0].x) * (i-bbox[0].x) + (j-bbox[0].y) * (j-bbox[0].y);
            if (d < distance) {
                distance = d;
                start_pixel.x = i;
                start_pixel.y = j;
                }
            }


/* BEGIN THE ENCODING PROCEDURE */
pixel.x = start_pixel.x;
pixel.y = start_pixel.y;
fatmap[PIX(pixel.x, pixel.y)] = VISITED;
last_dir = 4;
while(0 < 1) {
    /* AT FIRST, CHECK THE PIXEL IN THE LAST KNOWN DIRECTION */
    addPt2(&pixel, &contour_dir[last_dir], &test_pixel);
    if (fatmap[PIX(test_pixel.x, test_pixel.y)] == CONTOUR){
        pixel.x = test_pixel.x;
        pixel.y = test_pixel.y;
```

```
        fatmap[PIX(pixel.x, pixel.y)] = VISITED;
        code4.add(direction_code[last_dir]);
        }
    /* CHECK ALL THE POSSIBLE DIRECTIONS, CLOCKWISE */
    for (i=0;i<8;i++) {
        addPt2(&pixel, &contour_dir[i], &test_pixel);
        if (fatmap[PIX(test_pixel.x, test_pixel.y)] == CONTOUR){
            pixel.x = test_pixel.x;
            pixel.y = test_pixel.y;
            fatmap[PIX(pixel.x, pixel.y)] = VISITED;
            code4.add(direction_code[i]);
            last_dir = i;
            break;
            }
        }
    if (i == 8)
        break;
    }

/* WRITE THE LAST MOVE TO THE OUTPUT VECTOR */
for (i=0; i<8; i++) {
    subPt2(&start_pixel, &pixel, &test_pixel);
    if (test_pixel.x==contour_dir[i].x && test_pixel.y==contour_dir[i].y){
        code4.add(direction_code[i]);
        break;
        }
    }

/* POST-PROCESSING LOOP:                                    */
/* GO BACK TO A LOWER RESOLUTION BY FILTERING THE 4x CODE */

code1 = code4.postProcess();
return code1;
}
```

◇ Bibliography ◇

(Ballard and Brown 1992) Dana H. Ballard and Christopher M. Brown. *Computer Vision.* Prentice Hall, 1992.

(Feldman 1992) Tim Feldman. Generating isovalue contours from a pixmap. In David Kirk, editor, *Graphics Gems III.* AP Professional, Boston, 1992.

(Freeman 1961) H. Freeman. On the encoding of arbitrary geometric configurations. *IRE Transactions on Electronic Computers*, EC-10(2), 1961.

(Michaud *et al.* 1993) François Michaud, Ruben Gonzalez Rubio, and Alain Berkane. Etude de l'interdépendance des paramètres d'apprentissage avec la rétropropagation standard. To appear, 1993.

(Olson 1992) John Olson. Smoothing enlarged monochrome images. In Andrew Glassner, editor, *Graphics Gems.* AP Professional, Boston, 1992.

(Plaziac 1991) Nathalie Plaziac. Reconnaissance de caractères par les réseaux neuronaux. Master's thesis, Institut National de la Recherche Scientifique, Montréal, Canada, 1991.

(Schneider 1990) Philip J. Schneider. An algorithm for automatically fitting digitized curves. In David Kirk, editor, *Graphics Gems III.* AP Professional, Boston, 1990.