

**/MAGNIFICATION OF BIT MAP IMAGES
WITH INTELLIGENT SMOOTHING OF EDGES/**

by

CHARLES ROBERT SCHAEFER

B. S. E. E., Purdue University, 1966

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

**KANSAS STATE UNIVERSITY
Manhattan, Kansas**

1986

Approved by:

William J. Hanhley
Major Professor

LD
2668
124
1986
532
c. 2

111202 664102

TABLE OF CONTENTS

Report Cover

Table of Contents i

List of Figures ii through iv

ACKNOWLEDGEMENTS v

INTRODUCTION I-1 through I-8
 The Vector-Raster Duality I-1
 Design Objectives I-3
 The Successive Contouring Procedure I-4
 Review of Related Literature I-6

METHODOLOGY M-9 through M-25
 Conventional Low-Pass Filtering M-9
 An Empirical Method would be Better M-12
 Bullseye Scanning M-16
 Discussion of Details M-20

DESCRIPTION of Program D-26 through D-31

EVALUATION of Performance E-32 through E-43

CONCLUSIONS C-44 through C-51
 Edge Smoothness C-44
 Speed and Efficiency C-46
 Shapes and proportions C-48
 Potential Applications C-49

REFERENCES and BIBLIOGRAPHY 52 through 55

APPENDIX 1 GROW Program Source

APPENDIX 2 NAPLPS Tutorial

LIST OF FIGURES AND TABLES

- Figure 1 Low-Pass Filtering Page M-10
- Printed on the laser printer at 150 dots per inch resolution from an 80 by 48 bit map magnified to 8 times its original dimensions and 64 times its original area.
- Figure 2 Surface Function Page M-11
- Printed by Daisy wheel printer as text.
- Figure 3 Elementary Contour Shapes Page M-13
- Printed on the Dot Matrix printer at 68 dots per inch from four individual 40 by 24 bit maps. Magnification was 2X.
- Figure 4 Positioning Output Pixels Page M-17
- Printed on the laser printer at 300 dots per inch from a 52 X 52 dot original. The magnification was 8X. Characters in the original image were 7 or 8 dots high.
- Figure 5 Pixel Translations Page M-19
- Printed on the dot matrix printer at 68 dots per inch from two 80 X 280 dot originals. No magnification was used.
- Figure 6 Pixel Image Transformations RND Page M-21
- Printed on the laser printer at 300 dots per inch from three originals. The first was magnified by 4X, the second by 2X, and the third by 4X. Each little circle was a 5 X 5 cell of pixels in the original.
- Figure 7 Pixel Image Transformations SQ Page M-21
- Printed on the laser printer at 300 dots per inch from three originals. The first was magnified by 4X, the second by 2X, and the third by 4X. Each little square was a 3 X 3 cell of pixels in the original.

- Figure 8 The Bullseye Page D-28
- Printed on the dot matrix printer at 68 dots per inch from an 80 X 80 dot original. No magnification was used.
- Figure 9 Successive Contouring (crude) Page E-34
- Printed on the dot matrix printer at 23 dots per inch. Each dot is produced as a small bead-shaped pattern downloaded into the printer as a text character. The magnifications shown are 2X, 4X and 8X.
- Figure 10 Successive Contouring (fancy) Page E-35
- Printed on the dot matrix printer in the same manner as Figure 9. Magnifications are 2X and 4X. The small character at the base line was added in a second pass by the laser printer. It is identical to the 4X magnified image, but printed at 300 dots per inch.
- Figure 11 Lines of various slopes Page E-36
- Printed on the dot matrix printer in the same manner as Figures 9 and 10.
- Figure 12 Arbitrary Curves Page E-37
- Printed on the laser printer at 150 dots per inch and raw text input form. The magnification is 8X.
- Figure 13 Three Kitties Page E-38
- The first and second kitty were printed on the dot matrix printer at 23 and 68 dots per inch, with a 2X magnification factor. The third kitty was printed from the same original on the laser printer at 150 dots per inch, with 8X magnification.
- Figure 14 Text Banner Enhancement Page E-39
- The input was obtained from a banner program with about 12 dots per inch. This was doubled three times to a density of 75 dots per inch, on the laser printer.

Figure 15 Music Notation Page E-40

The first two parts were done on the dot matrix printer at 23 and 68 dots per inch. The third was done on the laser printer at at 150 dots per inch, with a magnification of 8X.

Figure 16 Circles Page E-41

Illustrates both densification and magnification. Scaling is applied to illustrate both densification and magnification. Upper row pixel densities are doubled at each magnification step. Lower row pixel densities remain the same.

1X at 75dpi 2X at 150dpi 4X at 300dpi

1X 2X 4X 8X all at 150dpi

Figure 17 Anomalies and extremes Page E-42

Each input pixel pattern is diagrammed above its resulting output. All parts are printed on dot matrix printer at about 136 dots per inch. Magnification factors are 32X (part A), 16X (PartB), and 8X (Part C).

Figure 18 Grow vs Enlargement Page E-43

The first enlargement is made directly by printing the 136 dpi original at 23 dpi. Then the original goes through 3 passes of doubling to 8X and the result is printed at 68 dpi for a 16X magnification. A true 16X magnification printed at 136 dpi would look exactly like this, except in the fineness of the tiny sawtooth edges.

Figure 19 Data Flow Diagram Appendix 1 Page 1

Produced by conventional text processing tools on the laser printer.

ACKNOWLEDGEMENTS

I would like to thank my employer, AT&T, for being the major contributor to any and all accomplishments reported here. Their support and sponsorship of me in the five-year Summer-on-Campus Program has changed many impossibilities into realities. This program is a model of community effort in education. While bringing education into the company, it lends momentum to the efforts of the University, and perhaps even to the technology itself.

Special thanks goes to the AT&T Technical Training Services Organization, Dublin Ohio, where I work and its parent organization, Corporate Education and Training. In addition to time and schedule allowances, and access to useful facilities, I enjoyed the opportunity to have related real-world problems to solve on the job. The company consistently chose to keep the door open as long as I maintained the initiative to continue.

Also of great importance was the understanding and support of my wife, Dee, who made many sacrifices and provided at least half the initiative I needed to finish. And our children, Jeannie, Diane, Jim, Robert, Carol, Cathy, Arlene and Dolores could probably keep me tied up for a year if they all cashed in their IOUs for quality time.

In the subject area of this report, I was fortunate to have completed a graduate course at Ohio State University under the guidance of Dr John Gourlay. The course dealt with Computer Typography, and Dr Gourlay posed several interesting problems. One of these was the reduction of aliasing during font bitmap enlargements. Some of the techniques I devised in that course are part of the groundwork in this Master's Project. I also wish to acknowledge the work of a fellow student in that course, Greg Roe, who wrote a low-pass digital filtering program and demonstrated its behavior.

I would especially like to thank Dr Virg Wallentine and the dedicated staff of the Department of Computer Science at Kansas State for their continued interest in me, and their patience with my schedule. And the bottom line goes to my Major Professor, Dr Bill Hankley, who had the foresight and gumption to keep goal tending and probing right to the end, no matter how many times I thought it was done. The final result of this project is a tribute to you, Bill.

INTRODUCTION

The emergence of techniques to directly manipulate raster images brings out a basic duality in the nature of Computer Graphics. The view of images as collections of vector line segments must share ground with the view that they are a collection of pixels. The effort reported here attempts to further develop the raster view, improving the quality and usefulness of extreme bitmap magnifications.

The Vector - Raster Duality

The vector view rises out of the human visualization of objects. Outlines of shapes are segmented into small straight elements, which are described numerically by the coordinates of their endpoints.

The raster of pixels is an accommodation to the nature of devices that print or display the image. A raster holds the color of each point on a surface as a pixel value. The points are sequenced along straight, parallel scan lines. When the pixels are limited to two values, such as black and white, the raster may be called a bitmap to emphasize its boolean nature.

At first the raster was strictly an output mechanism, with the conversion from vector to pixel form delayed until the final stage of processing. Editing and manipulation of the image in raster form was considered impractical. But new techniques were gradually devised such as those created by developers at the Xerox PARC (Ref 1), who have vigorously pursued the processing of rasterized images.

Today there is an intermingling of vector and raster techniques in most computer graphic machines. Somewhere between input and output there is usually a conversion from vector to raster form. A few machines still use pure-vector designs, with some form of Cartesian input device and an X-Y output plotter. This report describes a pure raster magnifier, based on bitmap input from a simple text terminal and raster output to high-resolution printers.

Manipulation in raster form is discouraged by factors such as storage space, complexity of operations, and the difficulty of returning to vectors (Ref 2). But overcoming these factors yields designs that are closer to a machine's natural way of imaging. Limits to the complexity of shapes dissolve away. Cumbersome numerical processing of vectors is replaced by efficient boolean logic. Interactive work that requires mainframe power in vector designs become practical in smaller machines.

By developing a procedure for extreme magnification of bitmaps, with edges kept smooth, this report hopes to realize a small part of the overall solution. One more piece of the task, previously limited to vector processing, can now be done more efficiently, in raster form.

The realm of the binary pixel is growing fast, with the proliferation of dot matrix printers, closely linked to the video raster graphic displays on inexpensive machines. Meanwhile laser-based printing technology is making 300 dot-per-inch resolution easily attainable. A major transformation is taking place, with a quantum increase in device capabilities and storage economy. This is fertile ground for the development of bitmap processing designs.

Design Objectives:

This project will develop and implement a procedure to magnify the bitmap representation of an image. The procedure will meet the following requirements:

1. The aliasing effect which normally emphasizes the boxy shape of magnified pixels along edge lines is to be reduced by this procedure.
2. The shapes and proportions apparent in the original image are to be maintained in the magnified image, even at extremely large magnification factors.

3. The design is to utilize machine resources to best advantage, and allow fast, interactive operation on a time-shared minicomputer.

The Successive Contouring Procedure

The procedure described here is a fast, empirical scanning mechanism that transforms a binary bitmap into a magnified bitmap of twice its original dimensions. It recognizes bit patterns in a neighborhood covering 24 pixels, and correctly reproduces elementary shapes in the image. True rectangular corners are kept sharp, while the stairsteps of sloping contours are filled and rounded. Thin lines are kept intact and line endings retain their original length.

The procedure can easily be repeated in successive iterations to achieve extreme magnifications to hundreds of times the original area. The test implementation operates in fixed allocated memory with an upper limit of 640 X 640 pixels. This allows the magnification of a 40 X 40 bitmap to 256 times its area by four successive iterations.

The Successive Contouring Procedure should be used in any graphic machine that supports raster devices with different pixel densities. It can take advantage of the high resolution of graphics printers copying from relatively coarse video screens. It may be possible to reduce very large bitmaps to one fourth or even one sixteenth their

original size for storage, and reliably reconstruct them on retrieval.

The internal design of this procedure has elements in common with previous methods reported in recent years. However, it applies these in a unique combination, together with a number of original implementation techniques. The combined result is a new procedure of some value to the Computer Graphics discipline.

In magnifying a raster image, this procedure has no information about the intended result other than the coarse input raster. By definition, the input is less accurate than the output is expected to be. The procedure must create new information by some form of inference. This is a departure from processing methods which start with a vector definition of the image.

When a human looks at a coarse raster, a sense of recognition can occur with surprisingly little detail in the image. The human screens that image down through an absolutely astounding volume of patterns, recalled from previous experience. There are no words for the number of bits involved, or the scaling and translation operations that take place behind the eyes. But within milliseconds, some kind of match occurs and the recognition is realized.

The Successive Contouring Procedure incorporates that sense of recognition to an infinitesimal degree. It limits the volume of patterns to be recalled by restricting the scanning window to a small size. Still the recognition of these simple patterns provides an intelligent and useful magnification of the image. What was accomplished is a small step on the scale of human capabilities, but a structure has been established that others may wish to extend.

The third objective was the most challenging. This was to be an efficient, practical program that others may wish to incorporate into existing systems. And if they do, it would not require extensive modifications. The C Language was chosen for portability and intimate control of machine operations. The program's core of raster manipulation functions can be reapplied without modification. Sample input and output functions are provided as prototypes for typical applications.

Review of Related Literature

There is a great deal of prior research in areas related to this project. Many of the authors cited may rightfully regard parts of this work as specific applications of their own published ideas.

The question of smoothing edges in a raster image has been addressed by Gupta and Sproull (Ref 3) in the situation where the object is a collection of numerically defined vectors. They determine the intensity of grey-scale pixels as a function of nearness and thickness of all lines passing through the pixel neighborhood.

Kajiya and Ullner (Ref 4) described a synthetic image approach to grey-scale pixel evaluation suitable for the display of high quality text. Turkowski (Ref 5) applied a table lookup to the point-to-line distance for anti aliasing in his Coordinate Transformation method. Both of these works also dealt with accurately-defined objects, which is not the case with the project described here.

A closer relationship to this project can be found in previous works dealing with the derivation of an enhanced grey-scale raster from a grey-scale raster. Many of the techniques described by Lev, Zucker, and Rosenfeld (Ref 6) can be adapted to the special case of two-valued pixel processing. They suggested an iterative method of processing pixels, taking into account nearby pixels having similar grey levels. This work was later categorized as the EDLN (Edge and Line weights) method by Chin and Yeh (Ref 7) in a description of iterative picture processing methods. In the process of defining the location of an edge in a grey-scale raster, they identified 12 ways that an edge can

pass through a 3 X 3 neighborhood of pixels.

The application of a box filter to avoid Moire effects in the raster display of texture patterns was described by Norton, Rockwood and Skolmoski (Ref 8). In clamping oscillatory effects to an average value, they pointed out the importance of dealing with the periodic intervals inherent in the display device. Another method of averaging pixel values in a neighborhood to define a central pixel value was described by Lee (Ref 9) as a Sigma Filter.

Additional related sources, too numerous to mention here are detailed in the bibliography at the end of the List of References.

METHODOLOGY

In this project, the objective is arbitrary magnification of a raster of two-valued pixels to a larger two-valued raster. Edge smoothing is to be incorporated with accurate reproduction of meaningful shapes and proportions. By distilling some of the principles described in the references and simplifying from the grey scale to two-valued pixels, a method can be derived to accomplish the task.

Conventional Low-Pass Filtering

A theoretical approach to smooth-contoured raster densification might apply digital filtering in two dimensions. Rows and columns of pixels are viewed as spacial step functions which are modified by a low pass filter. The result is an array of functions with ramp-like characteristics, to which threshold decisions are applied at the closer intervals of the output raster. This is illustrated in Figure 1 below, which shows an amplitude function of a row of pixels and a filtered version of the same function.

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH DIAGRAMS
THAT ARE CROOKED
COMPARED TO THE
REST OF THE
INFORMATION ON
THE PAGE.**

**THIS IS AS
RECEIVED FROM
CUSTOMER.**

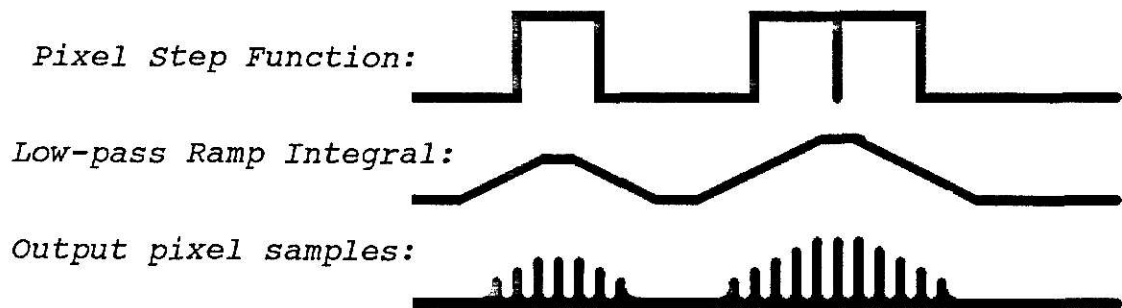
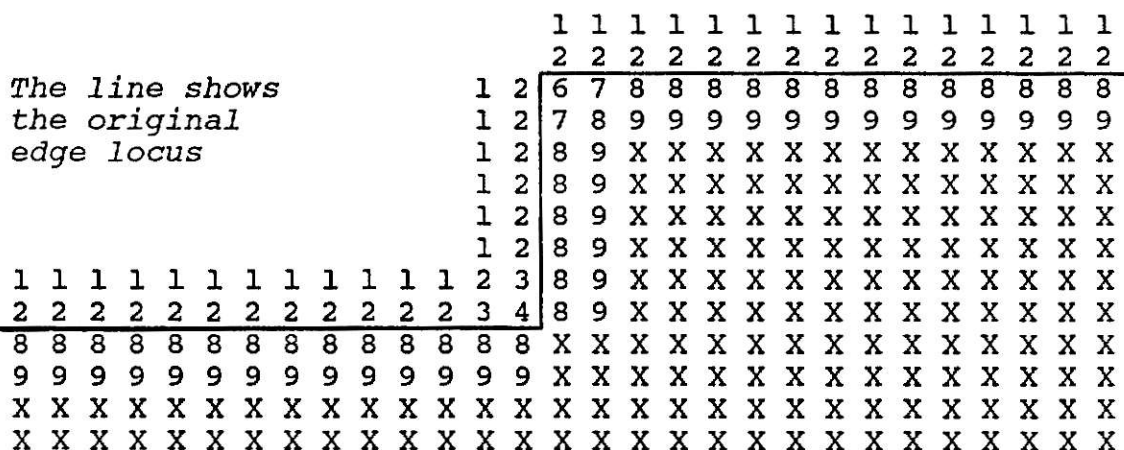


Figure 1

Low-pass filtering of a pixel row
and resultant high density output

In order to smooth out stairsteps in the edges the filtered functions have to be calculated along both horizontal and vertical scan lines, and their values combined in a surface function. Then the pixels near corner patterns can be distinguished by their values in the surface function. To illustrate this, Figure 2 shows a typical set of values for the sum of vertical and horizontal samples in the vicinity of a contour stairstep. The blank pixels have minimum value, and those marked X have the maximum value.

Intermediate values are marked in the range from 1 to 9.



An Empirical Method would be Better

The overall chance of achieving the objectives of the project with this approach was found to be very slim. Unfortunately, the math processing load would become exponentially burdensome as the magnification is increased, and the question of shape recognition had not even been approached. All we had was an unconditional rounding of all corners. This would lead to excessive rounding of parts not meant to be round. It became obvious that radical simplifications were needed before intelligent shape recognition could be added to the interpretation of the image.

Since the mathematical approach led to an arbitration of its results, it seemed justifiable to pursue a more empirical approach instead. A lot was learned from the close-up analysis of contour features in the low-pass filtering method. The idea of a lookup table to generate results seemed to have promise, but the computation of ramp functions across all scan lines seemed wasteful. An interesting discovery was that in a small neighborhood of pixels there are very few shapes that an intersecting contour line can assume. In fact there are only four that this project needs to deal with.

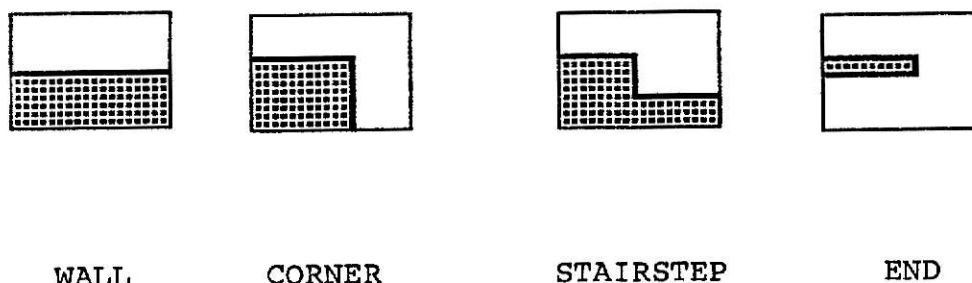


Figure 3

Elementary shapes of a Contour
Within a Small Neighborhood

Of course the four shapes described in Figure 3 can occur in any position and orientation in the input raster area being scanned. A corner is distinguished by extending straight for some distance (at least three input pixels) in both directions. The height of a stairstep is only one or two pixels of the input raster. Stairsteps do not have corners. An end is only one or two pixels thick, and is not considered to have corners either.

Now that an elementary shape vocabulary is established, decisions can be made on the disposition of each shape in the contouring process.

A wall should be unchanged, in its original position.

A corner, assumed part of a rectangle, is not filled.

A stairstep, assumed part of a diagonal, is filled.

An end is kept to its original length, rounded off.

These arbitrary definitions were the first great simplification of the problem. They seemed intuitively sound, but a working program would have to be applied to real examples to determine whether they provide intelligent shape recognition. The goal is not to recognize complex patterns in the image, but just to distinguish between the minute elements of shape that should be rounded and those that should be kept square.

The second great simplification of the problem was the decision to enforce alignment of the magnified raster with the pixels of the input raster. Various integer ratios of size were considered in an effort to allow straightforward determination of results. The choice of successive binary expansions emerged as clearly offering the best computational advantage.

Each expansion of a raster would exactly double its linear dimensions and quadruple its area in pixels. To achieve some other scale factor a raster could be doubled beyond the desired size and then linearly scaled back down. The repeated application of doubling can be more efficient than a single mathematical transformation. This is possible

because the doubling can be done without math.

The elimination of math was accomplished by the third great simplification of the problem. The small vocabulary of elementary contour shapes represented a set of patterns which could all be recognized within a small neighborhood of pixels. This led to a decision to process the raster through a small sliding window rather than a blanket matrix computation. Once all the possible patterns were predicted and their translations specified, the transformation became boolean rather than arithmetic.

With these three simplifications made, a practical Successive Contouring algorithm had been identified. But numerous details had to be worked out before an operable demonstration model could be built. First the size of the sliding window had to be determined and integrated with a workable scanning mechanism. A 4 X 4 pixel window would just barely contain the required contour shapes. But it would also contain 16 pixels, having 64 thousand possible combinations of states. This window would be translated to an 8 X 8 block of 64 output pixels. The simplest translation of a 4 X 4 window would then involve a table having 64000 entries of 64 bits each. But still, patterns could not be resolved correctly at the window boundaries.

The pursuit of radical simplification had brought the design all the way from pure math to pure mass recall of

answers. Somewhere along the way there must have been an optimal solution. It was now evident that a significant new insight was needed. In the process of evaluating the feasibility of 3 X 3 and even 2 X 2 sliding windows, I hit upon the "bullseye" method of raster scanning.

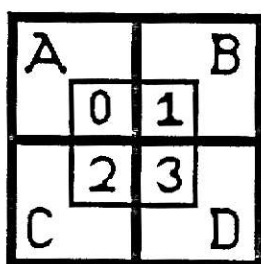
Bullseye Scanning

The bullseye is a concentric arrangement of graduated windows. The algorithm slides along, reading only the smallest window in the center. It reads pixels in the surrounding layer only if the contents of the small window do not fully identify the type of pattern in view. Then it goes to the third layer only if the inner two layers do not resolve the pattern. The design started with two layers, and later was extended to three.

The inner window is a 2 X 2 pixel neighborhood in the space of the input raster. It must be advanced in such a way that it generates a raster twice its density. It must generate a 4 X 4 output cell for the area of the inner window. But even a small 2 X 2 window has boundary problems. Shapes overlapping its borders will not be recognized. In some designs the window has a center pixel and it is advanced only one pixel at a time (Ref 9). The chosen solution for this design will also advance the input window one pixel at a time. Instead of deriving a 4 X 4

output window all at once it will create a 2 X 2 output window at each position, advancing the output window two pixels at a time.

The conceptual position of the output window on the image will be directly over the intersection of the four pixels of the input window. This is illustrated below, in Figure 4. The four output pixels cover one fourth the area of the four input pixels, but since the window covers four intersections, there will be sixteen output pixels. It is easier to think of this as a raster densification type of operation, since you can visualize both input and output rasters over the same image area. Each input pixel is simply cracked into quarters which remain in place.



The input pixels ABCD are the inner window of the bullseye. Their values in most cases can determine the values of output pixels 0123. Sometimes outer layer pixels surrounding ABCD have to be examined as well.

Figure 4

Positioning of output pixels
on the input raster.

The power of this scanning mechanism is that the inner window can resolve most of the shapes encountered without referring to the outer layers of the bullseye. Here are the sixteen possible combinations:

All white and all black - Leave as is.

Walls - top, bottom, left or right - kept as is.

Diagonals - Part of chain - becomes all black.

One black - depends on outer pixels.

One white - depends on outer pixels.

Each of these descriptions refers to more than one of the possible input patterns: All sixteen patterns are illustrated in Figure 5, with the proposed translations depending on whether corners should be rounded.

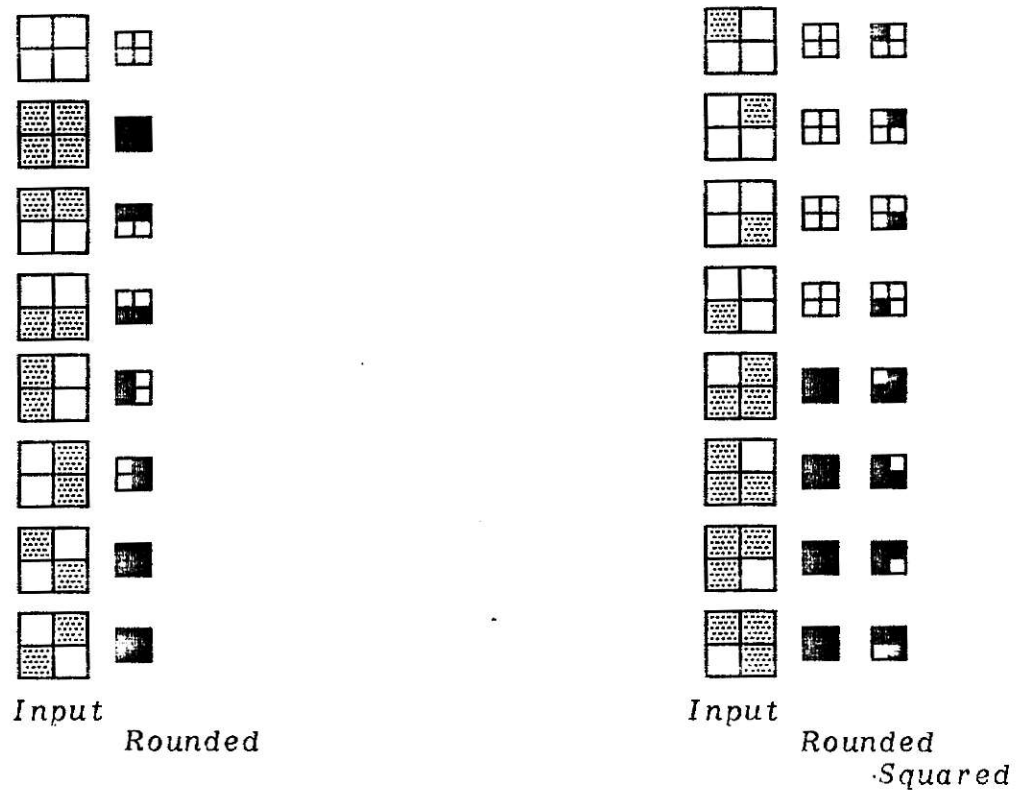


Figure 5

Examples of pixel translations

For each input pattern, the "rounded" result is that which is based only on the innermost window, without any reference to the other layers. Where "square" results are given, the outer layers are first checked to see if the rest of the pattern calls for a square corner. If not, the

result is rounded. The process of rounding a corner is nothing more than biting off a single pixel at a point. Figure 6, on the next page illustrates the operation of the algorithm in a situation where only rounded results are involved. Figure 7 includes some corners that are kept square.

Discussion of Details

Before proceeding to the actual implementation or any more examples, a little further discussion is needed on the details of Figures 3 through 7. These establish the basic logical structure on which the entire implementation will be built.

On the image examples given in Figures 6 and 7, the individual pixels are purposely enlarged, separated and drawn with characteristic shapes for close examination. However, their arrangement in the pictures illustrates the



Figure 6

Pixel Image Transformations
with Rounded Elementary Shapes

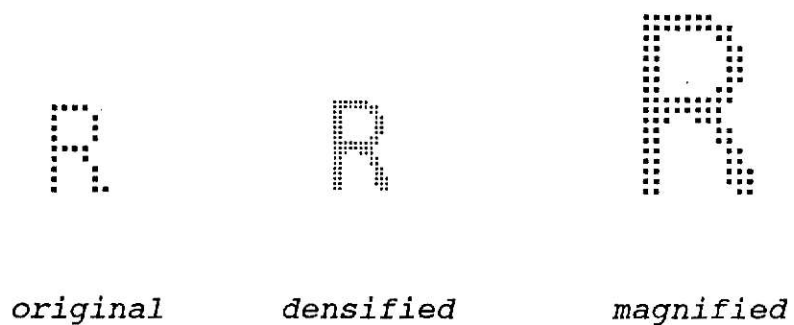


Figure 7

Pixel Image Transformations
with some Corners Kept Square

exact behavior of the translations in Figure 5.

No theoretical proof is offered that the translations and the four elementary shapes in Figure 3 will correctly deal with all images. Instead, the implementation will be used on a variety of subjects to determine how closely the translations duplicate more general shapes. This may also lead to enlightened observations of the inner nature of pixel representations.

We can see already that the transformation has a generally pleasing effect on the grossly approximated bit maps of Figure 6 and 7. The 7 X 5 letter in Figure 6 is typical of inexpensive dot-matrix printers. No parts of the pattern call for the retention of square corners and the rounded result is used in every output window. If the result were again put through the same translation, further smoothing of contours would be achieved without losing the original proportions of the shape.

Figure 7 starts with a higher-grade character, typical of 9 X 7 dot matrix printers. In this case a few of the corners need to be kept square, and the algorithm must be "aware of its surroundings" while translating pixels within the window. This is done by reference to the outer layers of the bullseye when the inner window has one odd pixel.

These examples illustrate some of the more subtle requirements that will be imposed on the implementation. First of all, the same algorithm has to apply to an entire image. It must deal with square and rounded shapes as they come. In other words if the patterns of Figures 6 and 7 were side by side in the same image, the rounding decisions should still lead to the same results.

Another duality involved here is that of black and white reversal. If a white shape appears on a background of black pixels, it should be transformed in the same way that a shape in black pixels on white would be. This is illustrated by the white area inside the loop of the 'R' in Figure 7. When rounding off the contour of a black area, the corner pixel becomes white. When rounding off the contour of a white area, the corner pixel becomes black.

The question of densification versus magnification of an image is simply a matter of output scaling. Figures 6 and 7 illustrate that the two outputs have exactly the same arrangement of pixels. Both of these were produced by the same transformation. The size of the result will always be determined in part by the pixel size on the output device. Each implementation will have to account for this in the establishment of magnification factors.

Perhaps the most surprising translation in Figure 5 is that of two diagonal black pixels to all black. This is an

area for potential future extension. The current design caters to efficiency and the unbroken representation of thin black lines. This can be seen in the upper right corner of the 'R' in Figure 7, and each bend of the 'S'. In each case the input window contains two black pixels touching diagonally, as if part of a chain. The output square of black pixels creates by inference a new input pixel at the midpoint of the chain.

The upper left corner of the 'R' represents an input pattern seen as a "corner" of black pixels. Nested one pixel down and to the right is another corner pattern of white pixels. At the scan position that detects the outer corner, the window contains one black pixel. At the scan position that detects the inner corner, it contains one white pixel. In both these cases the outer layers of the bullseye are progressively scanned to determine if the given area is truly a corner pattern. If the sides do not extend for three pixels in both directions, it is assumed to be part of a stairstep or an end, and rounded off. In either case the odd pixel in the input window is known to be at the tip of the pattern.

The white area below the loop of the 'R' is seen as an end, because it is less than three pixels wide. Thus it is rounded off by turning the tip pixels black. The rules for the end pattern are dictated by the arbitrary behavior of

small integers. An end that is one pixel wide cannot be rounded because the result is only two pixels wide. An example of this is the bottom of the vertical in the 'R'. An end that is two pixels wide is rounded off, keeping its original length. An end that is three pixels wide is seen as two corners of a rectangular shape, and not rounded off.

The general chances for this set of transformations to produce useful results seemed very good. But the purely theoretical prediction of its results on large patterns is an exercise in tedium. An actual implementation of this algorithm would greatly accelerate the discovery of its true behaviors. A language was selected and the design of a crude graphic workstation was developed to test out the performance of the algorithm.

The workstation program described in the next section will operate in a wide variety of system environments on a bit map input expressed in a simple text form. Its output is either in the same plain text format or a binary file, to generate a raster graphic image on a Hewlett Packard LaserJet printer. Any terminal and text editor can create bit maps to be processed, and display output. With the magnifications possible, the input can be very small, and easily produced.

DESCRIPTION

The Successive Contouring Procedure was implemented in a C Language function named *dbl()*. A program named *grow* was built around it to deal with practical problems and test out its features. The *grow* program has an *interp()* function to gather pixel data from a text terminal or a file. It has a *paint()* function to send magnified pixel data back to a terminal, file, or a printer. Its *main()* function is a short list of calls to the other functions with appropriate arguments. *Main()* can be changed and recompiled easily to implement various test conditions.

A complete listing of the *grow* program with all its functions is in Appendix 1. This appendix should be examined in conjunction with the following description, since the amount of detail involved is significant.

There are two functions in the *grow* program, which are not called directly by *main()*. The *dbl()* function internally calls on the *fiddler()* function to perform its detailed bit manipulations. The *fiddler()* knows all the translations of the inner window and the first surrounding layer in the bullseye. If the *fiddler()* cannot resolve a shape, it calls on the *fringe()* function, which knows how to interpret the outermost layer surrounding the bullseye.

The jobs that the *dbl()* function does for itself are the interpretation of arguments and the operation of the window scanning mechanism. The arguments to *dbl()* define the desired boundaries of the input and output raster areas. The output area usually overlaps the input area to save on memory space. The *dbl()* function works from bottom to top, and works best if the upper left corners of both areas coincide.

The *dbl()* function uses a set of four 32-bit registers as a bit working area. Before each call to the *fiddler()*, all the required pixel bits are shifted into position in the working registers. The *dbl()* function copies integers from the raster into the shifting registers in such a way that the bit streams seem to flow like water.

The main raster array is a monolithic mass of unsigned short integers (16 bits each). The raster size currently defined is 40 entries wide (640 bits) by 640 rows deep, which is suitable for the printers at hand.

The *fiddler()* function uses only the 4 X 4 bit square processing zone at the low end of the registers. The upper register bits are just a staging area for *dbl()* to keep the bits flowing. The 4 X 4 processing zone contains the inner window and one more layer of the bullseye. The *fiddler()* uses shifting and bitwise logic to assemble a numeric value

indexes a switch to implement the appropriate translation. The results are written directly back to the main raster by the *fiddler()*. The main raster and its index variables are external for direct access by all functions.

When the *fiddler()* encounters a possible corner pattern it builds a five bit index number from certain parts of the surrounding layer and reads a global array *corner_rnd* or *corner_fil* for further instructions. A 0 tells *fiddler()* to zero the odd pixel (white); a 1 tells it to set the pixel (black); and a 2 tells it to call *fringe()* for help. If *fiddler()* has to call *fringe()* it assembles the arguments from the positions of the outer bits of interest, relative to the processing zone.

The BULLSEYE

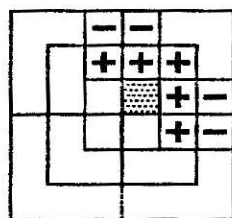


Figure 8

The diagram at the left may help illustrate second order processing by showing all three layers of the bullseye for a typical corner pixel situation.

The values of the pixels marked with a + are applied by the *fiddler()* to *corner_rnd*.

The relative positions of pixels marked - are sent by *fiddler()* to the *fringe()* function.

Fringe() returns a simple true or false result to *fiddler()*. It makes the determination by reading bits directly from the main raster. This avoids a lot of extra baggage in the working registers for infrequently used data.

The *fiddler()* never sees the contents of the outermost bits.

Another global variable named *square* is set on request from the command line by *main()*. The *fiddler()* function reads it to see if second order processing is desired. If not, it runs entirely on the inner window, rounding all corners.

This program is set up for demonstration purposes to read and write an elementary pixel definition language on any text terminal. A video character cell represents two pixels stacked vertically. On the typical terminal screen this gives 48 rows of 80 pixels with a nearly square aspect ratio. The simplest way to apply the program is to write the pixels in a file on the host system and send the result right to the screen. A 24 X 40 pixel input will fill up the screen on output.

The pixel definition language uses only four characters:

o - for a white top and black bottom

8 - for a black top and bottom

^ - for a black top and white bottom

space - for a white top and bottom

The available matrix printer is a C.Itoh Model 8510B. To print grow output, I simply download an appropriate set

of bit patterns into the four useful characters. With adjustments to the spacing modes the printer can produce a variety of bit maps in text mode directly from the screen output of the *paint()* function. The matrix printer allows about 1000 pixels per line at highest density.

The laser printer is a Hewlett Packard LaserJet, with 300 pixels per inch resolution. I wrote a special driver function named *print()* to operate its raster graphics features directly. The driver takes the pixel data directly from the main raster array in the *grow* program. It sends a stream of bytes segmented and delimited by the appropriate LaserJet escape codes.

Any additional drivers developed will tend to resemble either *paint()* or *print()* depending on whether a text mode approach is practical. Considerable flexibility is afforded for other extensions and modifications to the program. A variety of useful parameters are served by the core functions, allowing easy adaptation through rewrites of the *main()* function. Within the internal translation mechanism, the bulk of the intelligence is kept in the simple character arrays *corner_fil* and *corner_rnd*. Sweeping as well as subtle changes can be made in the operation of *dbl()* by substituting values in these arrays.

The *grow* program should compile and run without modification on systems with a standard C Compiler and the

stdio.h library. It processes text supplied to its standard input and streams the magnified result to its standard output. Files and access to devices can be handled in the normal manner by the operating system.

EVALUATION

The grow program has been debugged and operated on numerous test patterns, many of which are displayed in this section. It meets all of the objectives I required of it, in faithfulness to shapes and proportions. The discrimination of square corners is effective in the great majority of cases. Grow seems to round off where it looks good and keep square corners where they belong.

There is a clear limit to grow's intelligence when it comes to inferring the large-scale shapes of objects. It can best be described in terms of available line slopes. A crude pixel magnification could be said to have four possible line slopes: up, right, down, and left. In doing its brickwork around stairsteps, grow seems to be able to create about a dozen new slopes representable by smooth lines. The best slopes are 26.6 and 63.4 degrees above and below the horizontal. These eight slopes move two pixels in one direction for every one pixel in the other. Next are the four diagonals at 45 degrees. They tend to remain a little knobby, but uniform.

The limit of grow's intelligence, then, is about 16 line slopes. Anything not on one of its 16 favorite angles is

made from a patchwork of short lines which end up where a line of the desired angle would. The algorithm seems to behave curiously like a sailboat. It would be just as difficult to improve on this as it is to explain why it occurs. So I will attempt neither at this time.

The other side of the performance question is speed and economy of resources. I had taken serious measures in the choice of language and the program design to get the most efficient possible use of time and resources. This was one of my first concerns in the early testing of the program. I was happy to find that this program frankly moves like lightning. Let me qualify that. If I look at my watch right after typing in the command, the output will be in progress before I know what time it is. Even running up to three consecutive passes, accumulating a megapixel of output takes only about ten seconds.

The remaining pages in this section contain exhibits to illustrate the behavior of grow on sample images. Various specialized printing techniques are used to help illustrate the positioning of individual pixels in the results. Further details on the exact printing method for each figure can be found in the List of Figures at the beginning of the report.

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH THE ORIGINAL
PRINTING BEING
SKEWED
DIFFERENTLY FROM
THE TOP OF THE
PAGE TO THE
BOTTOM.**

**THIS IS AS RECEIVED
FROM THE
CUSTOMER.**


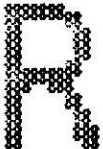
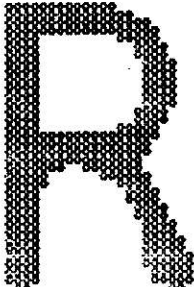
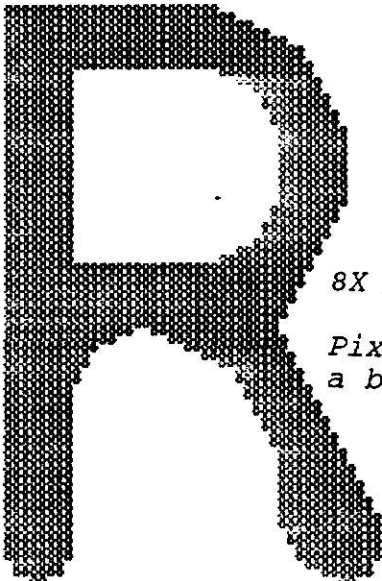
 *Original* *2X Magnification* *4X Magnification* *8X Magnification*
*Pixels represented by
a beaded pattern, 23/inch*

Figure 9

This illustrates three successive magnification steps from a simple 7X9 dot text character. There may be some question about the correctness of the shape obtained, but it has far more information than the original.

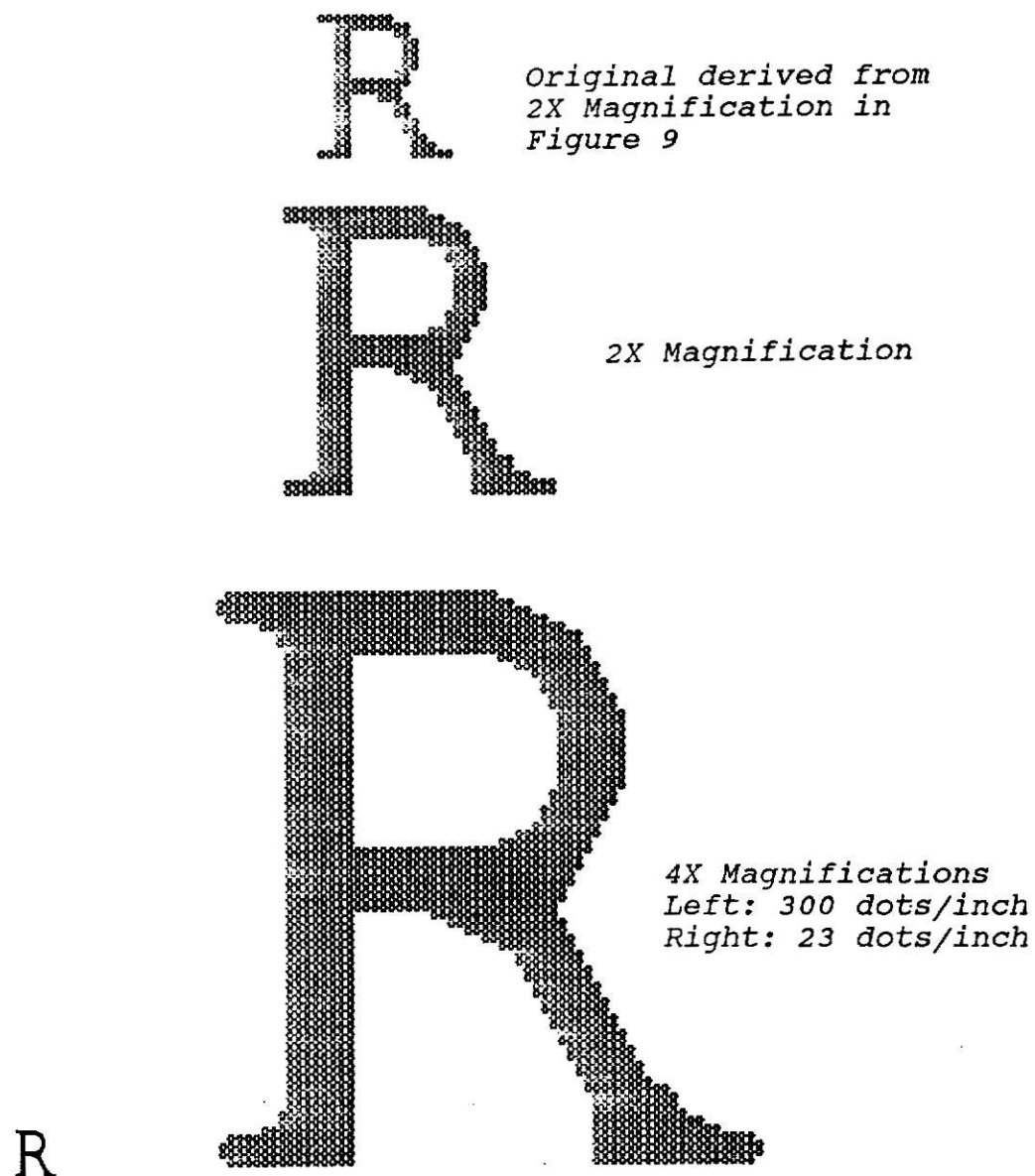


Figure 10

In this case we start with an image already magnified, and touch it up a little bit. Serifs are added to ends as well as the corner point. One pixel is erased under the bridge to allow it to be kept square. The little "R" at the bottom is identical to the largest except in the size of its dots.

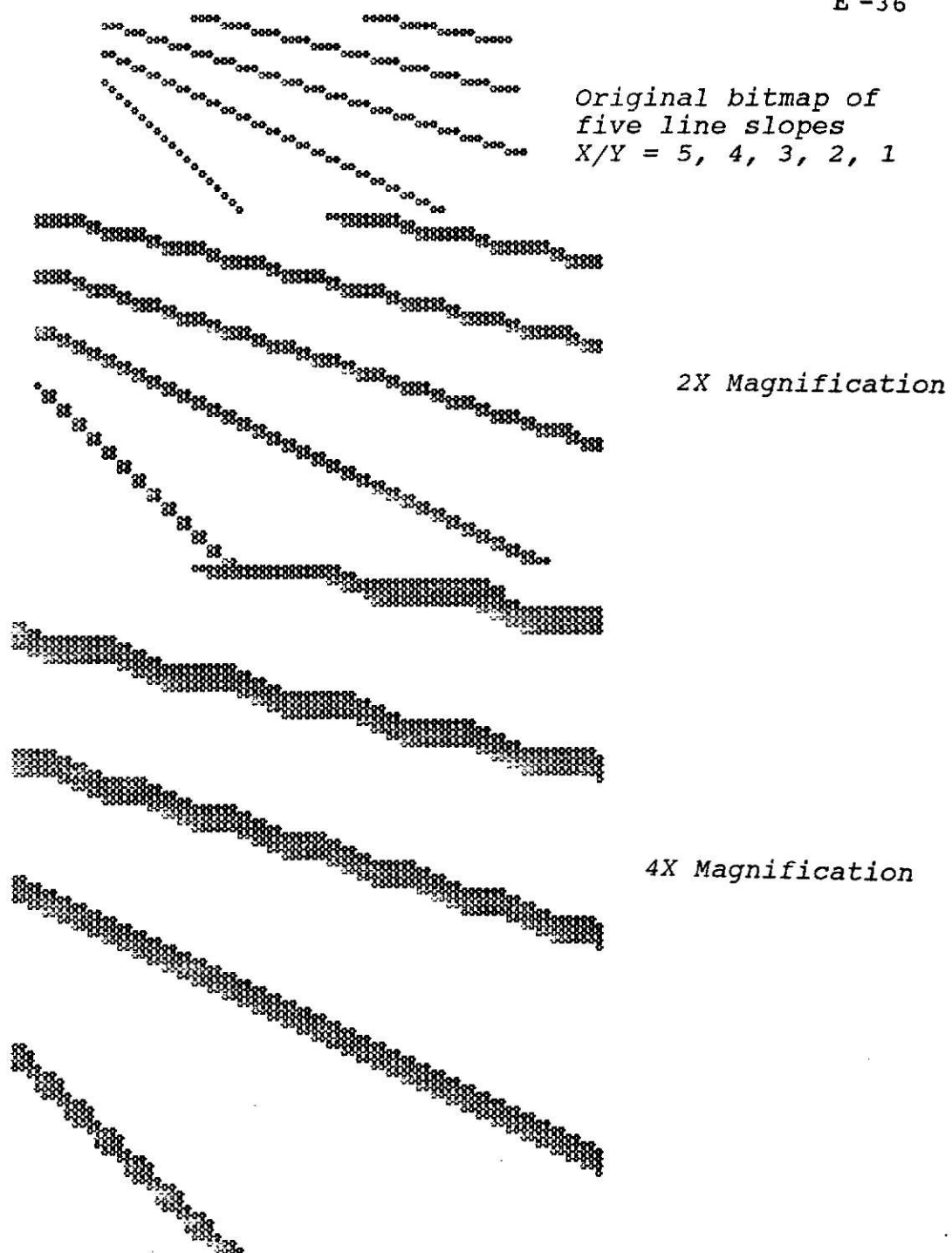
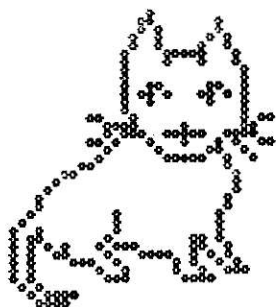


Figure 11

A few lines of assorted slopes go through two stages of magnification. Again the pixel size is grossly enlarged for examination of the dot patterns involved.

Original in
bead pattern
23 dots/inch



2X magnificatiion
68 dots/inch



8X Magnification
150 dots/inch

Figure 13

Given a picture with a meaningful shape, the algorithm certainly doesn't lose the meaning. It also keeps lines unbroken, although they get a little knobby in places. The last kitty has 64 times as many pixel as the first.

```

      8      8
      8      8      8
      8      8      8
      8888888      8
      8      8      8
      8      8      8
      8      8      8

```

Original from "banner" program
 Printed in text format
 12 dots/inch

```

      8888888
      8      8      8      8      8
      8      8      8      8      8
      8888888      8      8      8
      8      8      8      8      8
      8      8      8      8      8
      8888888      8      8888888      8888888

```

H I
 B I L L

8X magnification
 75 dots/inch

Figure 14

Most systems have a banner printing program. Successive Contouring makes an improvement in this one's appearance.

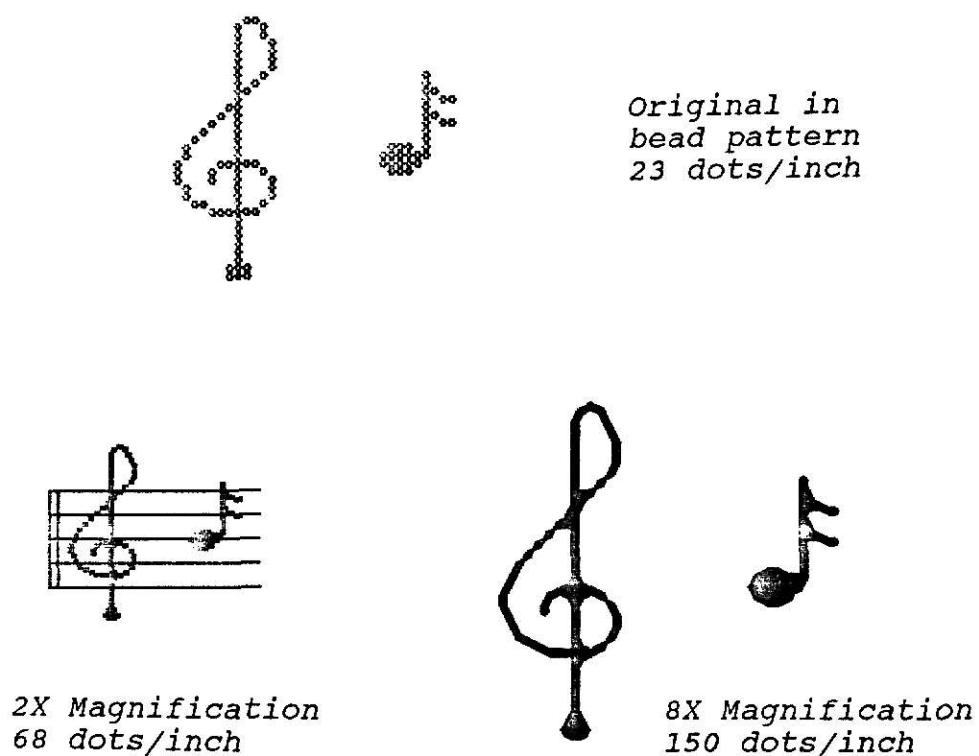
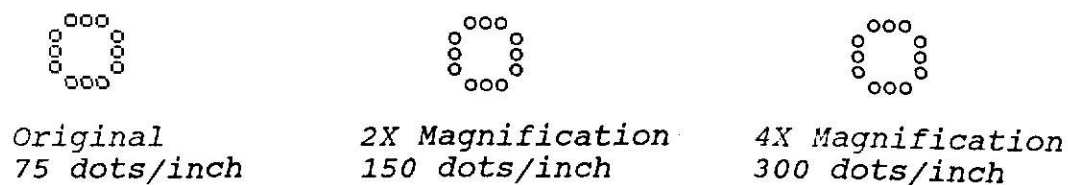


Figure 15

A few pieces of music notation pose a difficult challenge. The algorithm cannot make the classic shapes we are used to seeing from a crude dot pattern.



All below printed at 150 dots/inch

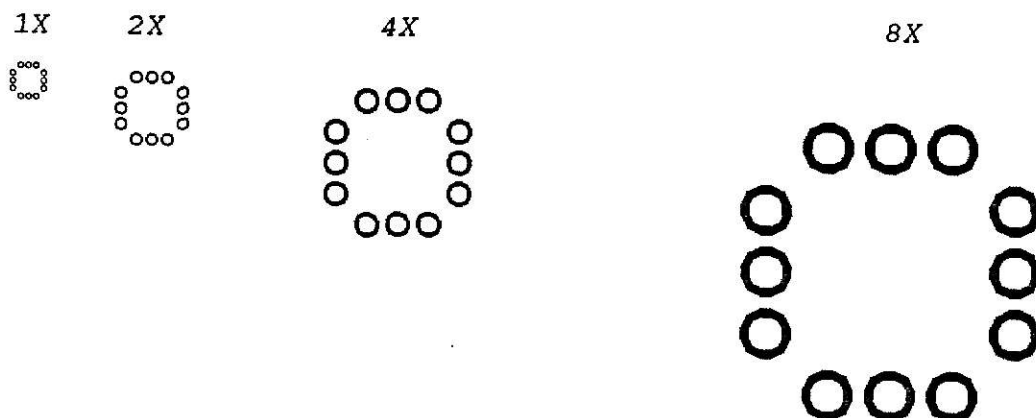


Figure 16

Circles seem to be handled very well. Each pattern here is a circle of circles which duplicates the basic pixel pattern from which all the circles were built. A dozen little dots go a long way. The top row illustrates densification, in which the size of pixels decreases as much as their numbers multiply. In the bottom row, all pixels are the same size.

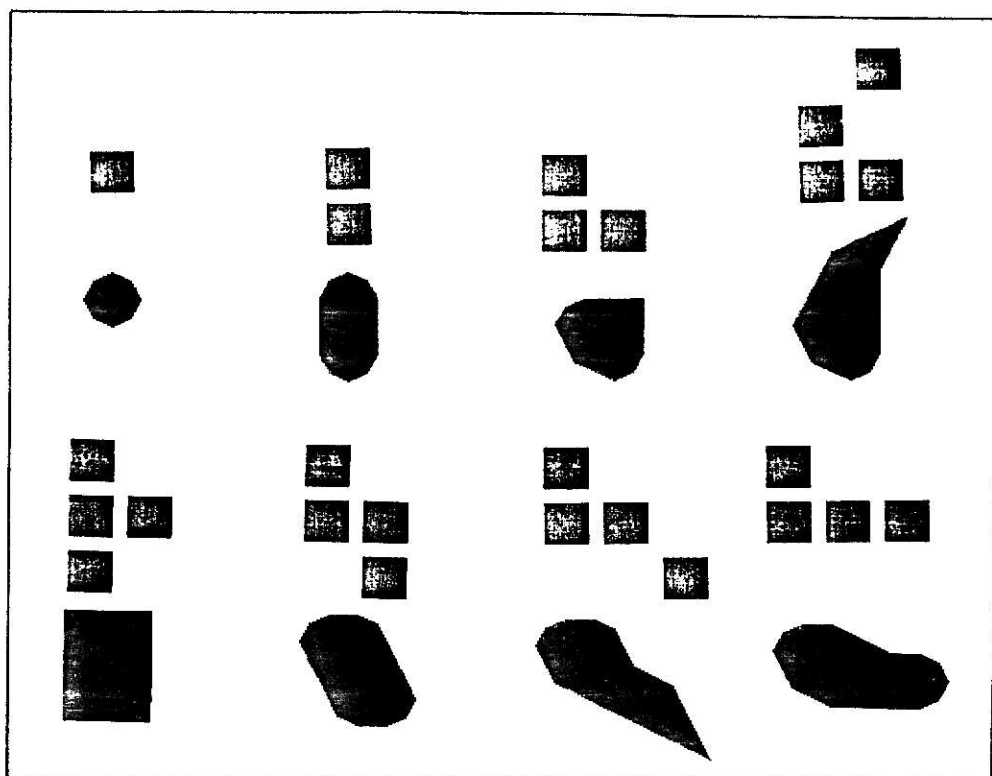


Figure 17A - Anomalous Subwindow Objects Magnified 1024 times in area (32X)

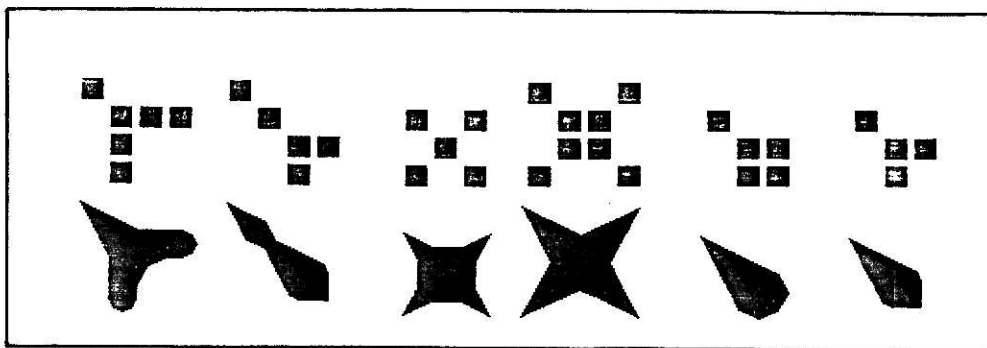


Figure 17B - Very Small Objects Magnified 256 times in area (16X)

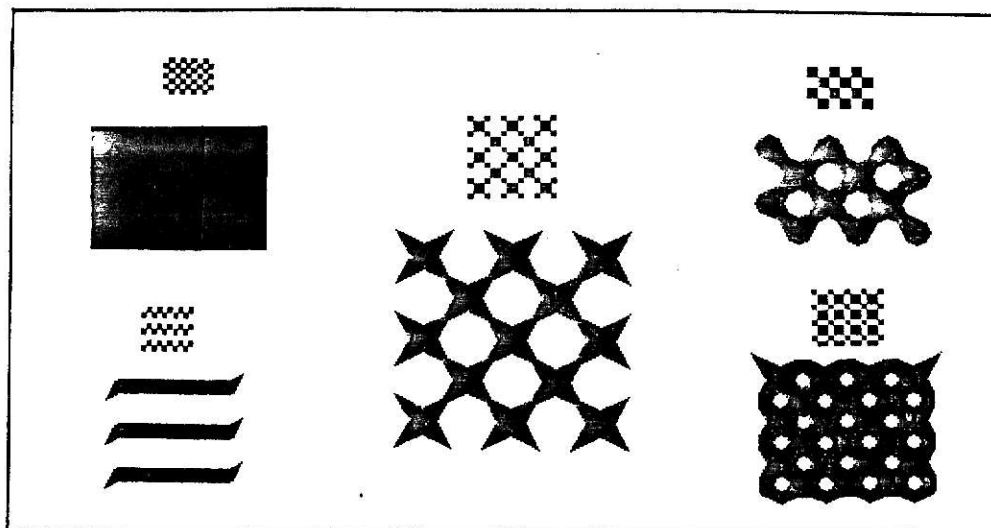


Figure 17C - Repeating Subwindow Patterns Magnified 64 times in area (8X)

Extreme magnifications:

The middle iris is a direct enlargement without using the grow program, to show the effect of expanding pixel areas. The iris on the right shows the result of four passes through the `dbl()` function.

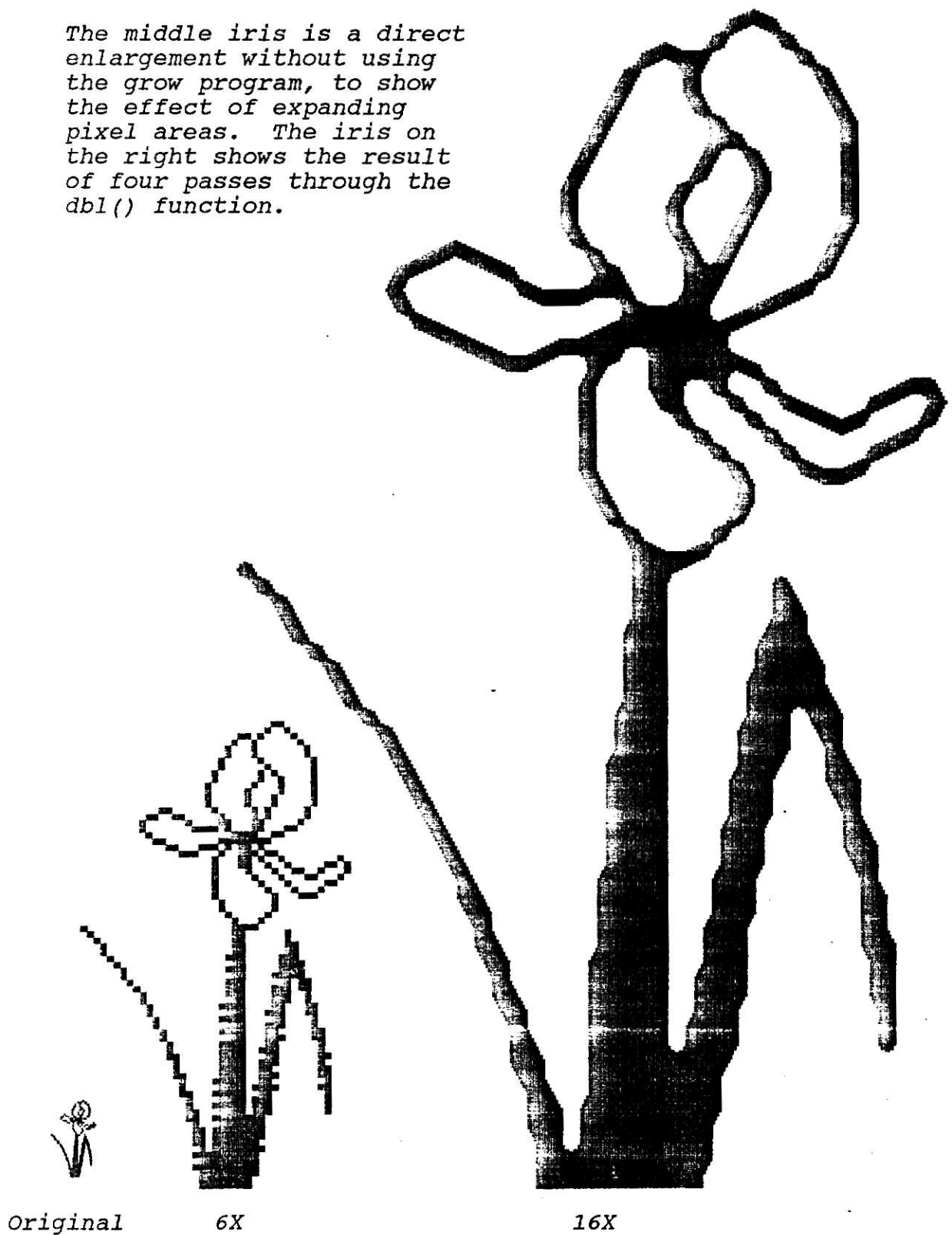


Figure 18

CONCLUSIONS

Based on the results achieved, this report concludes that bitmaps can be magnified efficiently with improved edge smoothness, and correctly proportioned shapes. These conclusions will be discussed along with some interesting observations about the manipulation of bitmaps.

Edge Smoothness

Every bitmap image is rippled along sloping edges, and perfect smoothness is never achieved. When an ordinary enlargement is made of a bitmap, the edge smoothness stays the same in proportion to the size of the object. But the area occupied by each original pixel takes on a boxy, objectionable appearance when enlarged to visible size. Since the pixels themselves are not enlarged, the edge smoothness of the enlarged image is not as good as it potentially could be.

For pleasing results, the enlargement should be accompanied by an improvement in the degree of edge smoothness. As the image grows, the relative size of the pixels becomes smaller and able to better conform to the edges. The locus of the "intended" edge through the area of the original pixel must be estimated on the basis of

surrounding pixel values. Then the fragments of each original pixel area can be redistributed along that edge.

The edge locus estimation becomes complex and burdensome at extreme magnifications. But it was found that an exact doubling of the bitmap dimensions greatly simplifies the problem. Each pixel area is just split into four quarters, with only sixteen possible results. It was further determined that the major decisions were limited to corner junctures along the edge. Each decision was simply whether a corner pixel should be broken off for a rounding effect or kept square. The doubling procedure was found to be accurate and efficient, and extreme magnifications could be made by successive applications of doubling.

Successive doubling also avoided the need for conventional low-pass filtering techniques in the rounding of corners. A method was briefly examined to sample a ramp integral of the pixel step functions along rows and columns. Although it would have worked, the compounding of mathematical steps was too extreme at large magnifications. In effect, all the considerations of edge locus estimation and corner rounding were applied empirically in the translations that generate each 2X2 pixel output cell.

The degree of accuracy achieved was good, but limited. It was found that there is some residual ripple along certain edge slopes. For example, an angular diamond carving effect can be observed along some of the edges in Figures

11, 15 and 18. Each residual protrusion is the rounded remnant of an original pixel that was poorly aligned with the intended edge. Improvements can be made in these areas by building additional patterns into the existing program structure. The cost of increasing smoothness would be some loss of operating speed.

Speed and Efficiency

This implementation has shown that intelligent magnification of a bitmap can be accomplished with minimal pattern recognition. The inference of intended edge lines from coarse pixel patterns can be done efficiently and accurately. The procedure developed makes judgements of elementary shapes in the image, based on the scanning of a quite limited neighborhood. The experience has brought to light a few interesting characteristics of bitmaps.

Just as the increase in size of a scanning window exponentially compounds the number of possible patterns contained, an extremely small window can surprisingly decrease the number of possible patterns. This effect was used in the selection of a basic scanning window only two pixels square. When an edge happens to pass through a window this size, the portion viewed is so microscopic that normally-visible shapes don't even need to be considered. It was learned that every possible contour shape on this scale can be described as one of four elementary types:

wall, staircase, corner, end.

With only sixteen possible input patterns in the basic window, the program could be built to switch to a different case of actions for each one. Half of these cases lead directly to fixed output patterns with no further processing needed. The more interesting cases are handled individually by scanning pixels in two concentric surrounding layers. The first layer is carried along in machine registers, together with the basic 2X2 window. Repeated references to memory are kept to a minimum. In a small number of scan positions on the image (typically less than 1%), it is necessary to examine pixels in the second surrounding layer. These pixels are scanned directly from memory. The conditional scanning of concentric layers around a small window has been dubbed "bullseye scanning". It has achieved high speed, combined with a flexible sensitivity to neighboring pixel patterns.

The C Programming Language was found to have several advantages over Pascal for this application. It allowed pixel values to be logically manipulated individually, and also moved around in bulk as unsigned integers. The scanning mechanism benefits from C's ability to configure machine registers and use their fast bitwise shifting capability. It was possible to implement one of the most heavily-used translation functions as a simple index into an array of characters, because the language could assemble the

index by reading boolean pixels as binary numerics. There were many other instances in the procedure where types had to be cast as other types, and the corresponding Pascal code would have been cumbersome.

Shapes and Proportions

The results obtained indicate that the shape of the magnified image is an exact duplicate of the large-scale appearance of the original. Some magnification procedures, such as low-pass filtering, would have a tendency to bloat or shrink the image, by adding or stripping layers of pixels. By exactly doubling the size at each step, the Successive Contouring Procedure is able to lock the output grid to the input grid, and never have fractional output pixels.

On the other hand, the small-scale appearance of the image is sometimes radically changed by successive doublings. As illustrated in Figure 17, a tiny cluster of pixels, on the order of the scanning window size can grow into unexpected shapes under extreme magnification. It seems interesting that a small window is faithful to large shapes and devastating to the tiny scraps. It would also be interesting to compare Figure 17 with the way these patterns are perceived by the human visual system.

Potential Applications

The Successive Contouring Procedure can be profitably applied in numerous existing raster-based graphics systems. One example would be in machines that print pictures both on a coarse video screen and on a high-resolution matrix printer. Typically, the system is optimized for the screen, and the printer is served as an afterthought. Successive Contouring can be added to such a system to exploit the full capability of the printer.

Systems which transmit graphic images in serial data streams could enjoy drastic reductions in transmission time if the picture is sent in low resolution form and densified at the receiving end. Further development in this area could lead to a reduction procedure that is exactly reversible by the Successive Contouring Procedure. In combination, the two procedures would provide a general purpose method of compacting raster images by extreme ratios.

The North American Presentation Level Protocol Syntax (NAPLPS) is an image communications language supporting both raster and vector representations of pictures. One of its formats is a mosaic representation of a bitmap in which each character cell is two pixels wide and three pixels high. The cell value is represented as one of 64 plain text ASCII characters. The Mosaic character set receives little attention today because of the coarse, checkered appearance

of its displays. If the Successive Contouring Procedure were implemented in NAPLPS workstations to densify Mosaic displays, the Mosaic code would be rediscovered as a slick vehicle for any bitmap image. Since NAPLPS is a recent development in Computer Graphics, a brief tutorial description is provided in Appendix 2 of this report. The complete NAPLPS Language is defined by the ANSI Standard X3.110-1983 (Ref 10).

Another NAPLPS mechanism is a pure bit map definition feature, which happens to be similar to the protocol used by the print() function to put a raster on the HP LaserJet. With some modifications to its drivers, the grow program could become a general purpose enlarging server to a NAPLPS workstation. It would specialize in processing images that are already rasterized.

The grow program is prepared for easy extension to interactive use. It runs fast enough to be applied effectively as part of a graphics editor. A few minor changes to main() would allow it to recurrently process input commands while keeping the main raster active. In fact the interp() function already has parameters to OR, exclusive-OR, and erase data into the raster as well as overwrite it.

The stored font files for typesetter programs take up huge blocks of space for every size of every text character in a variety of styles. With contoured densification, it

would be practical to store them in one fourth or even one sixteenth the space they now require. They could be magnified to full size on first use within each print job.

Readers of this report will probably think of many potential applications of this procedure not mentioned or even thought of here. Applications of raster and bitmap processing seem to be multiplying rapidly. Perhaps the *grow* program will contribute to the trend and begin to appear in various forms in many places.

REFERENCES AND BIBLIOGRAPHY

Entries marked "Ref" are sources related to the subject of this report, and specifically referenced in the main body.

- Ref 1 A Language for Bitmap Manipulation
 Leo J. Guibas, Xerox Palo Alto Research Center
 Jorge Stolfi, Stanford University
 ACM Trans on Graphics, Vol 1, #3, July 82, page 191
- Explains the Bit Boundary Block Transfer, based on the RasterOp described earlier by Newmann and Sproull. Introduces the Mumble language for bitmap operations. Also describes methods of rotating bitmaps by shearing and Floyd Masks.
- Ref 2 Raster-to-Polygon Conversion of Images
 Seymour Shlien, Communications Research Ctr, Ottawa
 Computers and Graphics, Vol 7, #3-4, page 327
- Explains methods of tracing border elements along the contours of uniform regions and expressing them as polygonal curves or Freeman Chain Codes. Describes segmentation, topological connection, shape descriptors and spacial relationships. Recommends a "half polygon" approach, with right sides extended indefinitely and left sides fitted to the contours in the picture.
- Ref 3 Filtering Edges for Grey-Scale Displays
 Satish Gupta, Carnegie Mellon University
 Robert F. Sproull, Carnegie Mellon University
 Computer Graphics, Vol 15, #3, Aug 81, page 1
- Explains a refinement to Bresenham's Point Plotting Algorithm. Pixel intensity is based on the volume of a unit cone intersected by all lines passing near the pixel center. End points are treated as a special case and math complexity is somewhat increased. Results are visibly improved.

- Ref 4 Filtering High Quality Text for Display on Raster
J. Kajiya, California Institute of Technology
M. Ullner, California Institute of Technology
Computer Graphics Vol 15, #3, Aug 81, page 7

Compares various earlier methods of anti-aliasing, including the $\sin(x)/x$ filter, triangular filter and interpolation of intensity between pixels. Proposes a "synthetic image method" suitable for the stringent demands of text display.

- Ref 5 Anti-Aliasing through Coordinate Transformations
Kenneth Turkowski, CADLINC, Inc. Palo Alto
Computer Graphics Vol 16, #3, July 82

Explains the smooth raster display of lines and polygons through a Point Spread Function implemented as a table lookup. The method is described as a continuous analytic convolution of object space, which is rendered into image space.

- Ref 6 Iterative Enhancement of Noisy Images
A. Lev, S. W. Zucker, A. Rosenfeld
IEEE Transactions SMC-7, 1977, page 435

Examines the correlations of grey levels among neighboring pixels, and methods of determining if nearby pixels are part of the same region on the image.

- Ref 7 Edge Preserving Smoothing
Roland T. Chin, Chia-Lung Yeh
Computer Vision Graph Image Proc Vol 23, 1983, page 67

Describes a class of iterative nonlinear noise cleaning filters, based on local weighted averaging of pixel values to approximate the ideal image. Contrasts ordinary averaging with "K Nearest Neighbor" averaging and "Edge and Line Weights" method. Relates EDLN method to previous work described in Ref 6.

- Ref 8 Clamping: Method of Antialiasing Textured Surfaces
Alan Norton, IBM Research Center
Alyn P. Rockwood, Evans & Sutherland Computing
Philip T. Skolmoski, Evans & Sutherland Computing
Computer Graphics Vol 16, #3 July 82, page 1

Illustrates the use of the physical resolution of the display in determining the upper frequency bound of the filtering process. Frequencies above the limit are selectively damped by forcing them to their local average values. Shows the elimination of Moire pattern effects in the display of periodic textures.

- Ref 9 Digital Smoothing by the Sigma Filter
Jong-Sen Lee, US Naval Research Laboratory
Comp Vision Grph Image Proc Vol 21 #3, Mar 83, pg 255

Explains the use of averaging the values of neighboring pixels in the raster-to-raster transformation of an image.

- Ref 10 North American Presentation Level Protocol Syntax
ANSI Standard X3.110-1983
CSA Standard T500-1983

This is the definitive reference on NAPLPS communication of graphic images. Specific reference is made to Section 5.3.3.6 on Incremental commands and Section 5.4 on the Mosaic Character Set.

The entries below, marked "Bib" are sources related to the subject, but not specifically mentioned in the report.

- Bib 11 Contour Filling in Raster Graphics
Theo Pavlidis, Bell Laboratories at Murray Hill, NJ
Computer Graphics Vol 15, #3, Aug 81, page 29

Describes methods of filling areas in raster images, including parity schemes and Line Adjacency Graphs. The LAG segments the horizontal scan line into nodes by changes in color. Branches are defined at intervals where adjacent scan lines have matching color. Nodes are ordered by their vertical coordinates to produce a directed graph, which is traversed during the area filling process.

- Bib 12 Parallel Processing Image Synthesis & Anti Aliasing
Richard Weinberg, Univ of Minnesota, Cray Research
Computer Graphics Vol 15 #3, Aug 81, page 55

Describes antialiasing in the synthesis of images based on calculated polygons. Several methods of partitioning are included, especially the partition of image space with a distributed z buffer.

- Bib 13 Image Enhancement Using HSHTF
Hassan J. Eghbali, Shiraz University, Iran
Computer Graphics Vol 5, 1980, page 23

Describes several specialized methods of processing raster images to enhance their visual impact, including High Sequency Ordered Hadamard Transform filtering. Methods involve the mapping of grey levels to alternate values to sharpen the appearance of edges and "eye modeling", which preconditions the image for the human visual system.

- Bib 14 A Frame Buffer System with Enhanced Functionality
F. C. Crow, Ohio State University
M. W. Howard, Ohio State University
Computer Graphics Vol 12 #2, July 78
Computer Graphics Vol 15 #3, Aug 81, page 63
- Describes improvements in the raster display of vectors and text characters through the use of grey scale processing.
- Bib 15 Waveform segmentation and Edge Preserving Smoothing
K. Prazdny, Fairchild Inc.
Computer Vision Image Proc Vol 23 #3, Sep 83, page 327
- Describes the segmentation of waveform patterns and methods of smoothing edges in a grey scale raster representation of an image.
- Bib 16 Subpixel Edge Estimation
Peter D. Hyde, University of Maryland
Pattern Recognition Vol 16 #4, 1983, page 413
- Compares several methods of estimating the locus of an edge passing through a pixel. Points out the simplicity of the Least Squares Estimation method.
- Bib 17 Characterizing Textures by Eigenfilters
F. Ade, Swiss Federal Institute of Technology
Signal Processing Vol 5 #5, Sep 83, page 451
- Describes the application of a 3 X 3 window of pixels in the scanning and processing of a raster containing texture patterns. The grey values of all 9 pixels are combined in a new value assigned to the center pixel.
- Bib 18 False Contour Removal by Random Blurring
Seiichi Nishihara, University of Tsukuba
Computer Graphics IP Vol 20 #4, Dec 82, page 391
- Describes the elimination of false contours and spurious brightness discontinuities for more natural appearance of video pictures.
- Bib 19 Image Segmentation
Minsoo Suk, University of Korea
Pattern Recognition Vol 16 #5, 1983, page 469
- Describes the application of a 2 X 2 pixel window in the scanning of a raster image.
- Bib 20 Azriel Rosenfeld's List
Azriel Rosenfeld, Ctr for Automation Res. U of Md
Computer Vision Graph, Image Proc Vol 26 #3, Jun 84
- A survey of picture processing research papers through 1983. Organized by related areas of interest, such as Filtering and Smoothing.

APPENDIX 1 - Description and Text of the Grow Program

The GROW Program operates the `dbl()` function in the data environment described in Figure 19, below. The `dbl()` function performs the main doubling of the bitmap in the Raster Array, calling on the `fiddler()` and `fringe()` functions for detailed work. External interface to the devices and files is provided by the `interp()`, `paint()`, and `print()` functions, under control of the `main()` function (dashed lines).

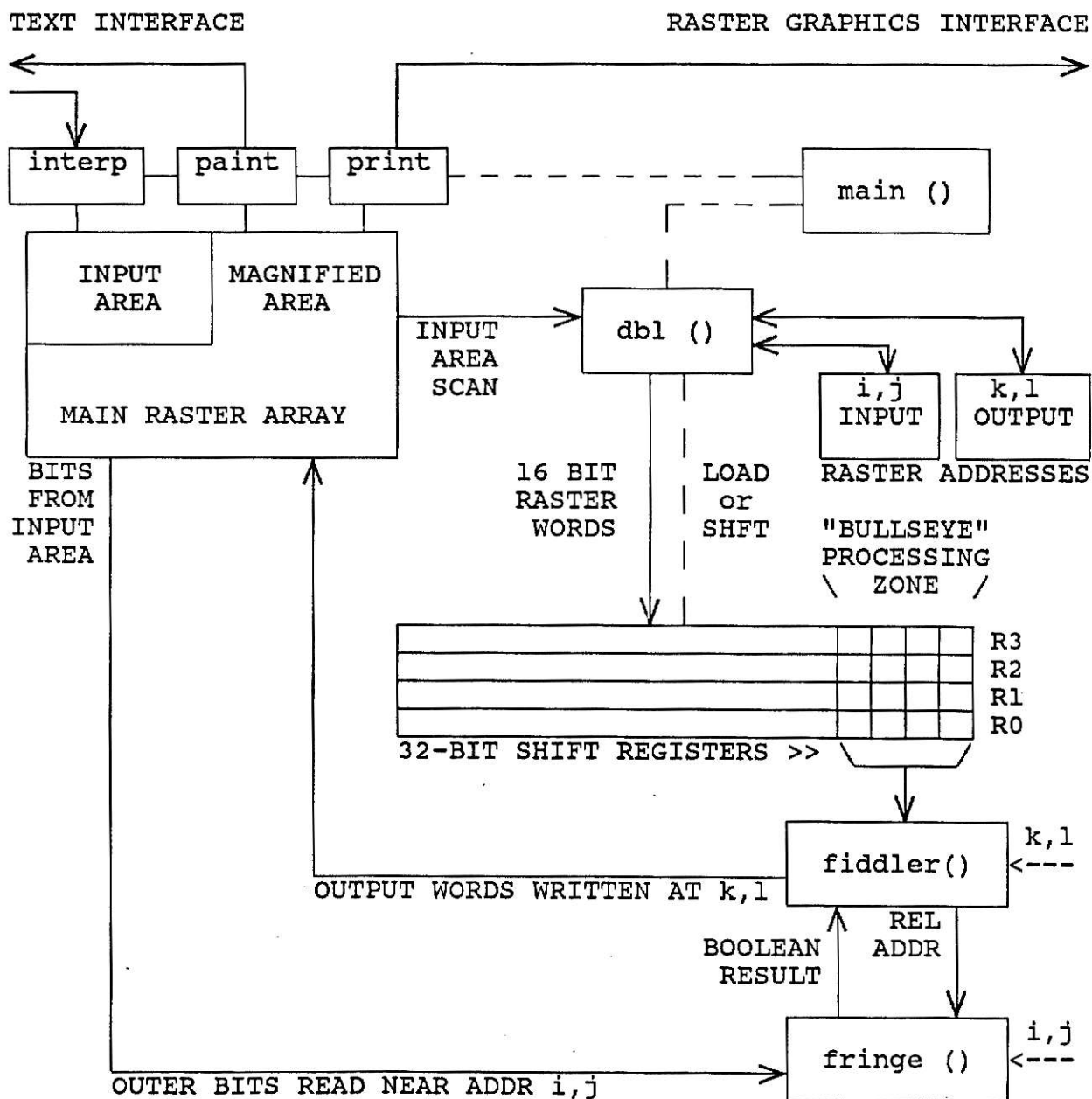


Figure 19
Data Flow Diagram of the GROW Program

```

/* This program is designed to test and demonstrate a Successive
** Contouring Algorithm to eliminate aliasing (jaggies) in a raster
** image when its resolution is multiplied. A very rudimentary
** mechanism is used to produce pixel arrays on a text terminal
** and apply the contouring algorithm to them. A simple output
** driver produces the demo results on a laser printer with raster
** graphics capability or in the text form used for input.
*/

#include <stdio.h>
unsigned short ras[641][41]; /* Main raster array for image enlargement */
/* The main raster array stores 641 rows of 656 bits, representing
** a rectangle about 2.16 inches square at the maximum
** output resolution of 300 dots per inch. The array size is limited
** in this case by the capacity of the printer used.
*/
int i, j, k, l; /* i,j - source point indices; k,l - destination */
int height, width; /* Dynamic size of original image */
int h_mag, w_mag; /* Dynamic size of magnified image */
int n; /* counting variable for bit-shift position */
int mag = 1; /* Magnification factor - set to 1, 2, 4, or 8 */
int corner_rnd[32] = {1,0,1,0,1,0,0,0,1,0,2,0,0,0,2,0,
                      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int corner_fil[32] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                      0,2,0,0,0,2,0,1,0,0,0,1,0,1,0,1};
/* Corner arrays are used in the manner of functions to return
** boolean-interpreted results. They are second order functions
** which are applied only where a corner pattern appears in the
** pixel processing zone. The corner_rnd result indicates
** whether to bite off an outside corner. The corner_fil result
** indicates whether to fill in an inside corner.
*/

```

```

char square; /* Used as boolean global flag to indicate whether
** the second-order cornering decision should be applied. When it
** is zero the inside corners are filled in and outside corners are
** bitten off, for an overall rounding effect. When it is nonzero
** then the square shape of corners is determined by the corner arrays,
** based on the states of selected surrounding pixels. Square is
** controlled by user keyboard input for each doubling of a raster.
*/

dbl(vs, hs, ve, he, Ve, He) /* double a bit array */
int vs, hs, ve, he, Ve, He;
    /* vs, hs, ve and he are the v & h start & end points */
    /* Ve and He are the end points of the enlarged array */
/* This function applies the Successive Contouring Algorithm, which
** doubles the density or the size of a raster array without magnifying
** the shape of individual pixels within the original raster. The source
** and destination points may be the same, in which case the transformation
** is done "in place". This is possible because the arrays are read
** and written from the end back to the beginning. The source should be
** positioned in the upper left corner (start) of the destination array.
*/

{
register unsigned long R0, R1, R2, R3;
/* Four 32-bit unsigned registers are used as a working area for
** bitwise operations. Four horizontal pixel rows are shifted through
** these registers in parallel, while the transformation is applied
** to the low four bits of the registers. Each bit in the array is
** examined sixteen times - once in each position of the 4 X 4 bit
** processing zone, covering the low 4 bits of the registers.
** Each time a new 16-bit row segment is loaded in a register, the
** high four bits of the previous segment remain in the processing
** zone. This provides bitwise continuity from one raster entry to
** the next.

```

```

*/
for(i=(ve+1), k=Ve; i>=(vs+3); i--, k-=2) /* Convert rows from bottom to top */
/* This outer loop processes one row of input pixels and generates
** two double-length rows of output pixels on each pass. During
** each pass it also reads neighboring rows of the input array.
*/
{
    R0 = R1 = R2 = R3 = 0; /* clear registers at beginning of each row */
    for (j=he, l=He; j>=hs; j--) /* scan right to left across one row */
    /* This loop processes one 16-bit input raster entry in each pass,
    ** conceptually loaded in R1, while three vertically-aligned
    ** neighboring entries are carried in R0, R2 and R3.
    */
    {
        R3|=((unsigned long)ras[i-3][j])<<4; /* Load top border row */
        R2|=((unsigned long)ras[i-2][j])<<4; /* Load top main row */
        R1|=((unsigned long)ras[i-1][j])<<4; /* Load bottom main row */
        R0|=((unsigned long)ras[i][j])<<4; /* Load bottom border row */
        /* When loading 16-bit entries into the 32-bit registers, they
        ** are converted by the cast mechanism to unsigned long values
        ** before being shifted left four bits. Thus, no high bits
        ** are lost in the shift.
        ** The result is ORed into the registers alongside the remaining
        ** four bits left by the previous pass.
        */
        if (R1 || R2) /* Process only if main rows have painted pixels */
        {
            for (n=0; n<16; n++) /* produce a 2 X 16-bit output segment */
            {
                fiddler(R0, R1, R2, R3); /* Transform 4X4 processing zone */
                R0 >>= 1; /* shift bottom border row to next bit position */
                R1 >>= 1; /* shift bottom main row to next bit position */
                R2 >>= 1; /* shift top main row to next bit position */
                R3 >>= 1; /* shift top border row to next bit position */
                if (n==7) l--; /* index next output raster column */
            }
            l--; /* index next output raster column */
        }
    }
}

```



```

        /* produce a 2 X 16-bit output segment */
    }
    else /* both main registers are blank: zero the output segments */
    {
        ras[k][1] = ras[k][1-1] = ras[k-1][1] = ras[k-1][1-1] = 0;
        l-=2;
    }
}
}

fiddler(r0, r1, r2, r3) /* Generate a 2X2 quad of output raster bits */
register unsigned long r0, r1, r2, r3;
/* This function examines the 4X4 processing zone of the registers and shifts
** two output bits into the high end of two 16-bit entries of the output
** raster array. In most cases, the full result is obtained directly from
** the central 2X2 area of the processing zone, without using the bordering
** 12 pixels.
*/

{
    int cor; /* to hold result of indexing corner tables */
    ras[k][1] >>= 2;
    ras[k-1][1] >>= 2;
    /* Two output raster entries are shifted down by two bits, leaving
    ** the high two bits in each entry zeroed. After eight calls to the
    ** fiddler, these two output entries will be completely replaced.
    */
    switch ((r1<<1&12)|(r2>>1&3))
    /* The switch value is assembled from the central 2X2 square of
    ** the register processing zone.
    */
    {
        case 0:
            break; /* Do nothing - the output quad is already zero. */

        case 15: /* 1 1

```



```

                                1 1  */
case 9:                        /* 0 1
                                1 0  */

case 6:                        /* 1 0
                                0 1  */
    ras[k][1]  |= (unsigned short)0140000;
    ras[k-1][1] |= (unsigned short)0140000;
/* The output is      1 1
                        1 1  */
break;

case 3:                        /* 1 1
                                0 0  */
    ras[k-1][1] |= (unsigned short)0140000;
/* The output is      1 1
                        0 0  */
break;

case 12:                       /* 0 0
                                1 1  */
    ras[k][1]  |= (unsigned short)0140000;
/* The output is      0 0
                        1 1  */
break;

case 5:                        /* 0 1
                                0 1  */
    ras[k][1]  |= (unsigned short)0040000;
    ras[k-1][1] |= (unsigned short)0040000;

/* The output is      0 1
                        0 1  */

```

```

break;

case 10:      /*  1 0
                1 0 */
    ras[k][1]  |= (unsigned short)01000000;
    ras[k-1][1] |= (unsigned short)01000000;

/* The output is      1 0
                      1 0 */
break;

case 4:      /*  0 0
                0 1 */
if (square && ((cor = corner_rnd[(r0<<2&28) | (r1<<1&2) | r2&1]) == 1))
    /* The index to the corner array is constructed from the
    ** five border bits which are adjacent to the set bit
    ** in the main square. In this case, the configuration
    ** of the examined border bits is:  - - - -
    **           where X is examined      - 0 0 X
    **           and - is ignored          - 0 1 X
    **                                   - X X X
    */
    ras[k][1] |= (unsigned short)00400000;
    /* The output is      0 0
                        0 1 if square and corner are == 1 */
else
    if ((cor == 2) && square && fringe(1, -3, 1, -1, -5, -2, -5))
        ras[k][1] |= (unsigned short)00400000;
    /* If pr zone has a corner pattern and fringe bits are
    ** aligned with the sides, the corner bit is set
    ** Otherwise the quad output is zero
    */
break;

```



```

else
    if ((cor == 2) && square && fringe(-4, -3, 2, -2, 0, -1, 0))
        ras[k-1][1] |= (unsigned short)0100000;
break;

case 13:      /* 0 1
                1 1 */
    ras[k][1] |= (unsigned short)0140000;
    if (square && ((cor = corner_fil[(r3>>1&7) | (r2&8) | (r1<<1&16)]) == 1))
        /* The index to the corner array is constructed from the
        ** five border bits adjacent to the zero in the main square.
        ** Configuration of examined bits:  X X X -
        **                                  X 0 1 -
        **                                  X 1 1 -
        **                                  - - - -
        */
        ras[k-1][1] |= (unsigned short)0040000;
        /* The output is      0 1
                                1 1 if square and corner are == 1 */
    else
        if ((cor == 2) && square && fringe(-4, -3, 1, -1, 0, -2, 0))
            ras[k-1][1] |= (unsigned short)0040000;
        else
            ras[k-1][1] |= (unsigned short)0140000;
        /* Otherwise      1 1
                            1 1 */
break;

case 14:      /* 1 0
                1 1 */
    ras[k][1] |= (unsigned short)0140000;
    if (square && ((cor = corner_fil[(r3<<2&28) | (r2<<1&2) | (r1&1)]) == 1))
        ras[k-1][1] |= (unsigned short)0100000;
        /* The output is      1 0

```

```

                                1 1  if square and corner are == 1  */
else
    if ((cor == 2) && square && fringe(-4, -3, 2, -1, -5, -2, -5))
        ras[k-1][1] |= (unsigned short)0100000;
    else
        ras[k-1][1] |= (unsigned short)0140000;
    /* Otherwise      1 1
                     1 1  */
break;

case 7:      /*  1 1
                0 1  */
ras[k-1][1] |= (unsigned short)0140000;
if (square && ((cor = corner_fil[(r0>>1&7) | (r1&8) | (r2<<1&16)]) == 1))
    ras[k][1] |= (unsigned short)0040000;
/* The output is  1 1
                  0 1  if square and corner are true  */
else
    if ((cor == 2) && square && fringe(1, -3, 1, -2, 0, -1, 0))
        ras[k][1] |= (unsigned short)0040000;
    else
        ras[k][1] |= (unsigned short)0140000;
    /* Otherwise      1 1
                     1 1  */
break;

case 11:     /*  1 1
                1 0  */
ras[k-1][1] |= (unsigned short)0140000;
if (square && ((cor = corner_fil[(r0<<2&28) | (r1<<1&2) | (r2&1)]) == 1))
    ras[k][1] |= (unsigned short)0100000;
/* The output is  1 1
                  1 0  if square and corner are true  */
else

```

```

        if ((cor == 2) && square && fringe(1, -3, 2, -2, -5, -1, -5))
            ras[k][1] |= (unsigned short)0100000;
        else
            ras[k][1] |= (unsigned short)0140000;
        /* Otherwise      1 1
                           1 1 */

        break;
    }
}

```

```

fringe(vp, hp, val, vs, hs, vz, hz) /* find square border on the fringe */
int vp, hp, val, vs, hs, vz, hz; /* relative coordinate info */
/* vp and hp locate a horiz pair of bits either above or below the zone.
** val gives the two-bit expected value of bits starting at vp, hp.
** vs and hs give coords of a bit on left or right expected to be set.
** vz and hz give coords of a bit on left or right expected to be zero.
** This function is called from fiddler only if sharp corners are requested
** and the current processing zone has an actual corner pattern in it.
** The fringe function reads the outlying bits directly from the input
** source raster locations, so they do not have to be carried in registers.
** Fringe also uses the global values n, i and j in effect when fiddler()
** was called, to locate the bits to be read.
** The basic function of fringe is to return a true result if a corner
** pattern extends a total distance of three bits in both directions.
*/
{
    int result = 0;
    int hl, incr;
    register unsigned long buff;
    /* hl is the effective horiz location of a bit, based on n and args.
    ** The incr variable is true if bits have to be read from the previous
    ** horiz raster word, as indicated when hl is initially negative.
    */
    hl = n + 13;

```

```

    buff = ras[i+vp][j];
    buff <= 16;
    buff |= ras[i+vp][j+1];
/* The raster is indexed with two relative adjustments to the coordinates:
**     The row i is modified by the vp argument
**     The selected bits are shifted down by hl
** Then the two-bit result is compared with the val argument for the result
*/
    result = ((buff>>hl&3) == val);
    if (result) /* quit if first test failed */
    {
        if (incr = (int)((hl = n + hs) < 0))
            hl += 16;
        result = (ras[i+vs][j+incr]>>hl&1);
    }
/* Row selection is modified by argument vs, which locates the side bit
** expected to be set.
*/
    if (result) /* quit if any previous tests failed */
        result = !(ras[i+vz][j+incr]>>hl&1);
/* Row selection is modified by argument vz, which locates the side
** bit expected to be zero.
*/
    return (result);
}

```

```

interp(v, h, op) /* Interpret an input file for graphic pixels */
int v, h; /* Block coordinates of upper left corner */
char op;
/* This function reads the standard input a line at a time and converts
** the characters 8, o, ^, and other into vertical pairs of pixel values
** and loads them into the input raster array two rows at a time.
** The ints v and h give the upper left raster starting coordinates.

```

```

** The op character is:   w to write over the existing raster.
**                       o to OR into the existing raster.
**                       x to X OR into the existing raster.
**                       e to Erase the existing raster where marked.
** The o, x, and e choices are intended for future applications in which
** recurrent operations are performed on the raster interactively.
*/

```

```

{   char buf[656], *cp, ch; /* Buffer to collect input characters */
    register unsigned short r1, r2; /* working buffers */
    int i, j, k, go, ll;
    ll = 0;
    j = v+1;
    while ( (gets(buf)) && (j<640) )
    {   cp = buf; i = 16; k = h;
        go=1;
        while ( (go==1) && (k<41) )
        {   ch = *(cp++);
            r1 <<= 1;
            r2 <<= 1;
            switch (ch)
            {   case '8':
                r1 |= (unsigned short)1;
                r2 |= (unsigned short)1;
                break;

                case 'o':
                r2 |= (unsigned short)1;
                break;

                case '^':
                r1 |= (unsigned short)1;
                break;
            }
        }
    }
}

```



```

        case '\0':
            r1 <=<= (i-1);
            r2 <=<= (i-1);
            if (l1 < k)
                l1 = k;
            go=0;
            i=1;
            break;
    }
    i--;
    if (i==0)
    {
        switch (op)
        {
            case 'w':
            case 'W':
                ras[j][k] = r1;
                ras[j+1][k] = r2;
                break;

            case 'o':
            case 'O':
                ras[j][k] |= r1;
                ras[j+1][k] |= r2;
                break;

            case 'x':
            case 'X':
                ras[j][k] ^= r1;
                ras[j+1][k] ^= r2;
                break;

            case 'e':
            case 'E':
                ras[j][k] &= r1;
                ras[j+1][k] &= r2;

```

```

        break;
    }
    k++;
    i = 16;
}
j += 2;
}
height = j;
width = ll+1;
}

```

```

paint () /* Paint the raster on screen */
{
    char buf[656], *cp; /* Buffer to collect output characters */
    register unsigned short r1, r2; /* working buffers */
    static char pix[] = {' ', 'o', '^', '8'};
    int i, j, k, l, c;
    for(j = 0; j<h_mag; j+=2)
    {
        c = 16 * (w_mag + 1);
        /* buf[c--] = '0'; */
        for(k=w_mag; k>=0; k--)
        {
            r1 = ras[j][k];
            r2 = ras[j+1][k];
            if (r1 || r2)
            {
                for (i=0; i<16; i++)
                {
                    buf[--c] = pix[(int)((2 * (r1&1)) + (r2&1))];
                    r1 >>= 1;
                    r2 >>= 1;
                }
            }
        }
        else
    }
}

```

```

        for (i=0; i<16; i++)
            buf[--c] = ' ';
    }
    c = 16 * (w_mag + 1) - 1;
    while (buf[c] == '\0' && c > 0)
        c -= 1;
    buf[c+2] = ' ';
    puts (buf);
/*    for (i=0; i<=(w_mag * 16); i++)
        if (buf[i]) putchar(buf[i]);
        else putchar('X');
*/
}

print (siz)
char siz;
{    char buf[87]; /* Buffer to collect output characters */
    int j, k;
    buf[0]=' 33'; buf[1]='*'; buf[2]='b';
    buf[5]='W';
    buf[3] = (char)('0' + ((2 * w_mag) / 10));
    buf[4] = (char)('0' + ((2 * w_mag) % 10));
    switch (siz)
    {    case '1':
        printf("%c*t300R%c*r1A", '\033', '\033');
        break;

        case '2':
        printf("%c*t150R%c*r1A", '\033', '\033');
        break;
    }
}

```

```

        case '3':
        printf("%c*t100R%c*r1A", '\033', '\033');
        break;

        case '4':
        printf("%c*t75R%c*r1A", '\033', '\033');
        break;
    }
    for(j = 0; j<640, j<=(h_mag); j++)
    {
        for(k=0; k<=(w_mag); k++)
        {
            buf[2 * k + 7] = (char)(ras[j][k]&255);
            buf[2 * k + 6] = (char)(ras[j][k]>>8&255);
        }
        for(k=0; k<=(w_mag * 2 + 6); k++)
            putchar(buf[k]);
    }
    printf ("%c*rB0", '\033');
}

```

```

/* The main() Function operates all the other functions in the program.
** This function could be as simple as the following:
**
**     main()
**     {   interp();
**         dbl (0, 0, 320, 19, 640, 39);
**         paint();
**     }
**
** In this form it would need no arguments, and would simply double

```

```

** its standard input once, producing a 2X magnified standard output.
**
** However, the effort of testing and refining the procedure has led
** to a more flexible and convenient version of main(), which is included here.
** This main() function allows control of the square variable and full
** control of output devices, magnification factor and pixel size, with
** built-in size management to serve the casual user.
**
** The operation of this version is controled by three simple arguments from
** the command line:
**
**      argv[1]      's'      Enter the letter 's' for square corners
**                          Enter any other letter for all rounded shapes
**
**      argv[2]      1-9      Enter a digit for mag factor in dbl()
**                          Gravitates to values 1, 2, 4 and 8
**
**      argv[3]      0-4      Enter 0 for text mode output by paint()
**                          Enter 1-4 for bitmap output for laser printer
**                          The print() function uses value 1-4 for pixel size.
**
** Actual sizes of input are measured by interp and reported internally to
** other functions. If the input size times the requested mag factor
** exceeds memory size, the result is clipped without attempting to
** use unallocated memory. If memory size itself is to be changed for different
** applications, main() should be rewritten with the limits 40, 80, 320 and 640
** defined as constants.
**/

main(argc, argv)
int argc;
char *argv[];
{   interp(0, 0, 'w');
    if (*argv[1] == 's')

```

```

    square = 1;
else square = 0;
switch (*argv[2])
{
    case '1':
        h_mag = height;
        w_mag = width;
        break;

    case '2':
    case '3':
        mag = 2;
        if ((h_mag = height * 2) > 640)
        {
            h_mag = 640;
            height = 320;
        }
        if ((w_mag = width * 2) > 40)
        {
            w_mag = 40;
            width = 20;
        }
        dbl ( 0, 0, height, width-1, h_mag, w_mag-1);
        break;

    case '4':
    case '5':
        mag = 4;
        if ((h_mag = height * 4) > 640)
        {
            h_mag = 640;
            height = 160;
        }
        if ((w_mag = width * 4) > 40)
        {
            w_mag = 40;
            width = 10;
        }
        dbl ( 0, 0, height, width-1, (2*height), (2*width-1));

```

```

    dbl ( 0, 0, (2*height), (2*width-1), h_mag, w_mag-1);
    break;

    case '6':
    case '7':
    case '8':
    case '9':
    mag = 8;
    if ((h_mag = height * 8) > 640)
    {
        h_mag = 640;
        height = 80;
    }
    if ((w_mag = width * 8) > 40)
    {
        w_mag = 40;
        width = 5;
    }
    dbl ( 0, 0, height, (width-1), (2*height), (2*width-1));
    dbl ( 0, 0, (2*height), (2*width-1), (4*height), (4*width-1));
    dbl ( 0, 0, (4*height), (4*width-1), h_mag, w_mag-1);
    break;
}
if (*argv[3] > '0')
    print (*argv[3]);
else paint();
}

```

INTRODUCTION TO NAPLPS

When computer technology moved beyond the realm of numbers, and into the processing and production of printed text, a new alphabet swept over the industrialized world. Streams of transmitted data became peppered with "control codes" having mysterious names, and which were not seen by the eye in the printed message.

The 128-character ASCII set established a new universal principle of communication. The control codes can be freely mixed with the normal characters in a message. The display device screens them out and modifies its operating states according to the instructions they convey. At the same time, it displays the normal characters as received. In effect, the sender can simultaneously converse with the human reading the message and with the machine displaying it.

Now some fundamental extensions have been applied to the ASCII character set, producing a new language to converse in images as well as text. The potential foreseen by the designers of the ASCII character set is being realized in the North American Presentation Level Protocol Syntax.

1. NAPLPS as a Language

The first step in learning to deal with NAPLPS is simply to recognize that it is a language. Unlike physical devices or systems, it can materialize within a variety of existing systems. Most computers can be programmed to converse in graphic images via NAPLPS. Designers can use any internal means of generating and interpreting NAPLPS with the comfort of knowing that other designers are building toward the same connecting language.

The 128 codes of the ASCII character set can accomplish everything in the NAPLPS language. Certain of the codes are able to substitute alternate meanings into whole groups of codes. In effect NAPLPS has the ability to dynamically swap its alphabets during a conversation. The various alphabets have a wide range of meanings, spanning all the elements of graphic image definition.

A NAPLPS data stream may contain a code that we recognize as the letter "j" by its ASCII definition. But in a European alphabet it would represent the character "OE" joined in a ligature. As a Mosaic cell it would represent three white squares stacked vertically alongside three black squares. As an operand to a Picture Description Instruction (PDI), it could give part of the coordinates of a polygon. Then there are redefinable characters and even macro definitions to invoke an arbitrary sequence of stored instructions.

Beyond all these capabilities, however, NAPLPS is not a language in the sense that Pascal is a language. It would not be applied in solving a general logical or mathematical problem. It will not evaluate data and dynamically branch to procedures appropriate to the situation. Instead it is simply a descriptive language for graphic images to be written on a display. A NAPLPS byte stream is a linear sequence of instructions used as a communications line.

2. Code Extension within NAPLPS

The Code Extension technique that makes NAPLPS work uses a collection of different character sets in addition to the old ASCII definitions. At least four sets are kept in Random Access Memory for instant selection. However, there is no increase in the size or complexity of the individual characters. The selection of character set is done by a specific control code, after which all following characters are interpreted from the selected set of meanings.

Character set selection is usually built into an index variable in the operating state of the display device. In certain character sets, such as PDI, each "character" leads to another index into a collection of primitive graphic drawing functions. We customarily think of ASCII codes in a two-dimensional array with eight columns and sixteen rows. That array now becomes one shelf in the three-dimensional structure of NAPLPS codes.

NAPLPS compares to ASCII as chess compares to checkers.

Like the basic game board, the 128 numeric codes remain the same. But the squares hold different kinds of players, who take on a variety of roles. Complexity takes a quantum leap, and the interpretation of each move is immensely more dependent on the "state" of the game.

3. Commonly-used Terminology

The alphabets I have described as "shelves" in the 3-D structure of NAPLPS are usually called "tables" in the literature. A table is divided between the low 32 values of control codes and the upper 96 values of graphic codes. In NAPLPS everything that makes a mark, such as text, mosaics, PDIs and foreign alphabets is called a "graphic" code. The collection of 32 control codes in a table is called a C set, and the 96 graphic codes are called a G set.

There are two C sets currently defined in NAPLPS.

The first, designated C0, is based on the original ASCII set of control codes.

Within this set, the transmission control codes (STX, EOT, SYN, etc) and the device control codes (DC1, DC2, etc) are invisible to the NAPLPS Presentation Layer (That just means NAPLPS interpreters should disregard them). They are used for messaging functions and physical reconfigurations in the OSI Session Layer.

Other control codes, namely backspace, linefeed, tab, return, formfeed, cancel, null, bell and escape have similar functions in NAPLPS, but new English names in some cases.

The characters from the C0 set used in the designation and selection of other character sets include escape, shift-in, shift-out, EM (renamed single-shift-2), and GS (renamed single-shift-3).

The second C set, is designated C1.

The escape character followed by any character in column 4 or 5 ("@" thru " " including all capital letters) makes a one-time selection from the second C set (C1). This is the familiar "escape" mechanism used in most text terminals. It comes from the general code extension structure recommended by the International Standards Organization in ISO 2022.2 - the basis of most communication protocols including NAPLPS.

The functions performed by codes in the C1 set include macro and DRC definitions, character size and attributes, field scrolling and protection, and cursor style.

The diagram on the next page (Table 1) defines the NAPLPS C sets and G sets on a detailed numeric basis. It combines the definitions of a half dozen tables from the NAPLPS Standard into one ready reference. These details are well-documented by the ANSI Standard X3.110-1983, (Ref 10).

The entries in Table 1 each consist of a box containing most of the definitions for a specific numeric character value. The Primary (ASCII) definition is always in the upper left area of the box. The definition from the Supplementary set comes next in the top center area. If a Mosaic is defined it is diagrammed in a nested box at the upper right corner. The PDI op codes of columns 2 and 3 are printed in the lower right area of the box. The letters S, R, A, and F above PDI op codes mean Set, Relative, Absolute and Filled.

Characters involved in escape sequences to select character sets have additional notations in nested boxes at the lower left. Control codes of columns 0 and 1 that have new names under NAPLPS have their old names in the lower right corner.

4. Sets of Graphic Characters

The four sets of graphic codes maintained in memory are designated G0, G1, G2, and G3. One of these sets is identified as "in use" by the value of an index variable, which points to G0 by default. Each G set in memory can also be reloaded from time to time with table values from some other storage area.

NAPLPS provides control code sequences to select an external table from a "repertory" and copy its contents into a chosen "G set." Up to 64 different graphic tables could be stored and accessed by this mechanism, but so far only a half dozen have been universally established.

The repertory at this time consists of:

Selected by escape * B

The Primary character set, which is the same as the ASCII set.

Selected by escape * |

The Supplementary character set, consisting of miscellaneous symbols, fractions, phonetics, line fragments, and European alphabets.


















































































































































 CR SCHAEFER		C0 CONTROL SET		PDI OP CODES		C1 ESCAPE CODES		MOSAIC (also 2&3)							
000	0	001	1	010	2	011	3	100	4	101	5	110	6	111	7
0000	0	NUL	DLE	SPACE											
0001	1	SOH	DC1	! i											
0010	2	STX	DC2	" ¢											
0011	3	ETX	DC3	# £											
0100	4	EOT	DC4	\$ \$											
0101	5	ENQ	NAK	% ¥											
0110	6	ACK	SYN	& #											
0111	7	BEL	ETB	' §											
1000	8	APB	CAN	(¤											
1001	9	APF	SS2) ' ¢											
1010	10	APD	SDC	* " ¢											
1011	11	APU	ESC	+ « ¢											
1100	12	CS	APS	, ← ¢											
1101	13	APR	SS3	- ↑ ¢											
1110	14	SO	APH	. → ¢											
1111	15	SI	NSR	/ ↓ ¢											

TABLE 1

Summary of NAPLPS Codes

Selected by escape * W

The Picture Description Instruction (PDI) set. The first 32 PDI values, from "space" to "?" are op codes to trigger calls to graphic functions such as plotting points, lines, arcs, rectangles, polygons and irregular shapes. The remaining 64 values provide numeric operands to follow the op codes and specify locations, distances and various other data. The operand characters are easily distinguished by the receiving device, since their seventh bit is always set.

Selected by escape * }

The Mosaic character set, in which each character represents a block of six binary pixels in a raster array. The block is three pixels high and two pixels wide. The values of the pixels are determined by six of the bit values in the character code.

Selected by escape * z

The Macro set is initially empty, but its members can be defined during a session by a control code sequence, including the NAPLPS instructions to be capsulized by the given character.

Selected by escape * {

The Dynamically Redefinable Character Set takes the idea of macros a step further. While a macro is executed exactly as it is originally defined, a DRC is executed subject to current state variables controlling the rotation and scaling and display attributes of graphic characters.

The '*' character after the escape in the above definitions is also a variable. If it is an asterisk as shown, the selected character set will be loaded into table G2. However, any set can be loaded into any table, so here are the four characters that select the four tables:

G0 is loaded by the character: (

G1 is loaded by the character:)

G2 is loaded by the character: *

G3 is loaded by the character: +

The 3-character escape sequence that loads a table might not have an immediate effect on the display in progress. The selection of the currently "active" G-set is a separate process, using a different group of codes.

The Si (shift in) control code activates the character set loaded in G0. This is typically the primary text character set, although any set can be loaded there.

The So (shift out) control code activates the character set loaded in G1. In some cases this would be the PDI set, but in a raster-intensive system, the Mosaic set would be useful in this position.

The "escape n" sequence activates whatever character set is loaded in G2. This G-set is often used for the supplementary set of European characters and graphic symbols. Since it is commonplace to access single characters from it and immediately return to another set, a single-shot mechanism is also provided. Any time the SS2 (single shift 2) control code is received, the very next character will be interpreted from the G2 set.

The "escape o" sequence activates the character set loaded in G3. The SS3 control character provides the one-shot interpretation of the next character from G3.

This description takes liberties in referring to positions in a table by their familiar ASCII character names. Other authors have tended to avoid this, perhaps to emphasize the abstract numeric connotation of their values under NAPLPS. For instance in avoiding reference to the character 'j' most authors would call it 6/10. For the benefit of readers who are unaccustomed to thinking in columns and rows, I will risk being told some day that when a 'j' is in the PDI set, it isn't a 'j'.

Another area where a bit of confusion can be avoided is in the eight-bit implementation of NAPLPS. This description has avoided the confusion by ignoring the eight bit case altogether. However, everything stated here remains true in an eight-bit implementation. The only difference is that two different G-sets and C-sets are active at once and their selection is made by the eighth bit in every character. In most system environments, this turns out to be more trouble than the resource savings can justify. The language is already sufficiently complex and flexible in its seven-bit form.

MAGNIFICATION OF BIT MAP IMAGES
WITH INTELLIGENT SMOOTHING OF EDGES

by

CHARLES ROBERT SCHAEFER

B. S. E. E., Purdue University, 1966

ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

ABSTRACT

While images defined by vector linestrokes are easily manipulated in all the usual ways, there are some systems which perform manipulations directly on bitmaps. These can process images of extreme detail and complexity with no penalty in speed. They can also process images which start out in bitmap form, such as photo scanner output. But one type of manipulation that bitmap-based machines still find difficult is magnification. Magnified bitmaps are usually characterized by enlarged pixel shapes, producing jagged, checkered edges.

This report introduces a procedure for extreme magnification of bitmaps, which adheres to significant shapes and improves the smoothness of contour lines. The procedure magnifies by densification, cracking each pixel into quarters. The scan mechanism involves a 2 X 2 input window and a 2 X 2 output cell which exactly doubles the linear dimensions in pixels. Extreme magnification uses successive exact doubling of the bitmap. Shapes in the window are discerned by conditionally scanning pixels in concentric surrounding layers. This technique, dubbed "bullseye" scanning, achieves both high speed and effective pattern recognition.

The Successive Contouring Procedure has been implemented by the C program documented in Appendix 1. It is arranged for testing on time-shared minicomputers from any type of text terminal. A sample driver is provided for high-quality output to a laser printer. In actual tests it doubled an 80 by 48 bitmap in a half second. Successive doubling to 16X (a million output pixels) required about ten seconds. The procedure was found to create new information by inference, in order to reconstruct contours with the higher-resolution pixels. Its adherence to significant shapes and proportions in the image is excellent.

Applications for this procedure can be found in the transfer of video displays to higher-resolution printers, including laser printers. It can match the highest device densities with economical, low-resolution bitmaps. Zoom features on raster displays can be improved in range and readability. Extensions of this procedure could add the processing of other graphic languages, including typesetter font files or NAPLPS Mosaic codes. Many additional applications and extensions are possible.