Antialiasing Methods for Laser Printers

by

Christopher M Mayer

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering

and

Master of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1991

© Christopher M Mayer, MCMXCI. All rights reserved.

The author hereby grants to MIT permission to reproduce and
to distribute copies of this thesis document in whole or in part.

Author ....
Department of Electrical Engineering and Computer Science
May 20, 1991

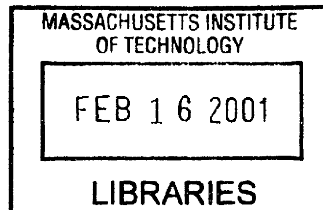
Certified by ....
Victor Michael Bove Jr.
Assistant Professor of Media Arts and Sciences
Thesis Supervisor

Certified by ....
William Light
Digital Equipment Corporation
Thesis Supervisor

Accepted by ....
Leonard A. Gould
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

LIBRARIES
ARCHIVES



Antialiasing Methods for Laser Printers

by

Christopher M Mayer

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1991, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering
and
Master of Science in Electrical Engineering

Abstract

Font and graphics antialiasing are often used methods of enhancing the output quality of video monitors. The same techniques also can enhance the quality of laser printer output. Since the laser printing process and the monitor display process differ radically, the assumptions made for antialiasing on monitors are reexamined. Raster scanning, laser modulation, xerographic transfer, information transfer, and computation are examined in the context of laser printer antialiasing. Past work on antialiasing algorithms is evaluated.

A practical method of performing anti-aliasing of text, graphics, and images is developed. Standard antialiasing scan conversion algorithms are adapted to function on two dimensional binary bitmaps of vector data. Computation issues are discussed, and a fast localized scan conversion operation based on statistical models is developed and evaluated. Finally, an architecture for the high bandwidth modulation of the laser by both power and time is developed, implemened, and evaluated.

Thesis Supervisor: Victor Michael Bove Jr.
Title: Assistant Professor of Media Arts and Sciences

Thesis Supervisor: William Light
Title: Digital Equipment Corporation

Acknowledgments

To Mom and Dad.

Thanks to Bill Light and Michael Bove for putting up with a year of cluelessness, to Praveen for a place to crash in between sections, and Robin and Ted for welcome distractions.

Contents

1 Introduction **13**

1.1 Background of Laser Xerography 14

1.2 Quality in Laser Printing 17

1.3 Statement of the Problem 18

2 Antialiasing **21**

2.1 Sampling 21

2.1.1 Point Sampling and Supersampling 22

2.1.2 Area sampling 24

2.2 Fourier Transforms 25

2.3 Aliasing and Antialiasing 29

2.3.1 High Frequency Edges 30

2.3.2 Quantization 32

2.3.3 Antialiasing 36

2.4 Reconstruction and Filtering 36

2.5 Justification of Antialiasing 43

2.6 Video Display vs. Laser Printer Antialiasing 45

2.6.1 Similarities 45

2.7 Antialiasing in Practice 47

2.7.1 Supersampling 47

2.7.2 Area Sampling 49

2.7.3 Gupta-Sproull antialiased lines 50

2.7.4 Antialiased brushes 51

2.7.5 Character look up tables 52
2.7.6 Antialiasing with bit masks and look up tables 52

3 Proposed Antialiasing Post-filter 54

3.1 Edge and Fill 55
3.1.1 Antialiasing edges 55
3.1.2 Enhanced filling 57
3.2 Antialiasing using cellular automaton 58
3.2.1 Rules for PE's 58
3.2.2 Results and problems 60
3.3 Windowed ROM based algorithm 60
3.3.1 Description of windowed ROM algorithm 61
3.3.2 Window selection 64
3.3.3 Selection of learning data 68

4 Laser diode modulation methods to support antialiasing 70

4.1 Hardware constraints 70
4.2 Subpixel patterns 71
4.3 Selected encoding scheme 73
4.4 Simulation of laser printer modulator 73
4.5 Design methodology and circuit description 81
4.6 Laser Modulation Circuit Schematics 84
4.6.1 Feedback PAL (PAL1) 88
4.6.2 Modulation PAL (PAL2) 90

5 Results, Conclusions and recommendations. 92

5.1 Cellular automaton smoothing 92
5.2 Windowed ROM antialiasing 93
5.3 Laser modulation circuit 94

A Associated Computer Programs 95

A.1 SHOW_BITMAP.C 95

A.2 FILTER.C 105
A.3 SMOOTH.C 109
A.4 CREATE_CIRC.C 114
A.5 CREATE_MAP.C 116
A.6 ANTI_ALIAS.C 131
A.7 PRINTSIM.C 136

B Experimental results **151**

B.1 Celllular automata smoothing 151
B.2 Windowed ROM antialiasing 155

List of Figures

1-1 The seven steps of the xerographic process. (a) charging the photoconductor, (b) exposing to form latent electrostatic image, (c) developing the latent image into a real image, (d) transferring the image to paper, (e) fusing the image to paper, (f) cleaning residual toner from the photoconductor, (g) erasing the electrostatic latent image. (Redrawn from [24].) 15

1-2 Example plot of a photoinduced discharge curve of a xerographic photoconductor. (Redrawn from [24].) 16

2-1 Point-sampling problems. The sample grid is represented by black dots. Objects B and D are sampled, but identical objects A and C are not. The shapes of the objects are not preserved. (Adapted from [12].) 23

2-2 Unweighted area sampling. The pixel's intensity remains constant when the object is moving with the pixel, but changes abruptly as the object traverses the border to another pixel. (Redrawn from [12].) 25

2-3 Weighted sampling without overlapping pixels. The object contributes differing amounts of intensity to the pixel depending on its position. The overall intensity of the page changes as the object moves. (Redrawn from [12].) 26

2-4 Weighted sampling with overlapping pixels. The overall intensity of the page remains roughly constant as the object moves between pixels. (Redrawn from [12].) 26

2-5 Example of a Discrete Fourier Transform (DFT). The signal is repeated at intervals of 2π. The maximum frequency able to be represented is $\frac{1}{2T}$, the Nyquist rate. The Nyquist rate is normalized to π in a DFT. 30
2-6 Aliasing. (a) Signal is sampled above the Nyquist rate, so there is no aliasing. (b) Signal is sampled exactly at the Nyquist rate. (c) Signal sampled below the Nyquist rate. Aliasing is evident; the waveform is deformed. 31
2-7 Probability density function for rounded quantization errors. 34
2-8 Quantization errors when sampling a black to white transition. (a) Sample points correspond to grayscale values, so the error is zero. (b) Sample points do not fall on grayscale values, so aliasing occurs. (c) More grayscale levels are introduced, so the aliasing is reduced. 35
2-9 Sampling and Reconstruction. (a) Original continuous signal (b) Sampled signal (c) Low pass filter (d) Reconstructed signal after passing the (sampled) impulse train through the low pass filter. 37
2-10 General laser printing model. 38
2-11 Bitonal bitmap of the letter S 39
2-12 Box filtered letter S 39
2-13 Bartlett filtered letter S 40
2-14 Gaussian filtered letter S 40
2-15 SINC function, truncated using a box filter. 41
2-16 Rectangular filter. 42
2-17 Laser printing model with compensated reconstruction filter. 43
2-18 Compensated reconstruction filter. (a) Rectangular filter. (b) Compensated reconstruction filter. 44
2-19 Prefiltering and Postfiltering. (a) Prefiltered data is antialiased before scan conversion. (b) Post-filtered data is antialiased anytime after sampling (not necessarily before bitmap storage or transmission.) . . 48

2-20 Antialiasing by supersampling. The filter is mapped across the supersampled image. The sum of product of corresponding values determines each grayscale pixel value. (Adapted from [12]) 49

2-21 Gupta-Sproull antialiased lines. (a) A conical filter. (b) Filter convolved with a line of width 1 to form a table of filter values for Gupta-Sproull antialiased line drawing. (Adapted from [14]) 51

2-22 Table lookup of polygon edge. The polygon intersects the pixel at locations 25 and 61. These two values are used as the address for a table lookup to compute the grayscale value of this pixel and also surrounding pixels. 53

3-1 Cellular Automaton Antialiasing. A bitmap is converted to higher resolution using cellular automata and then converted to a grayscale bitmap using the supersampling methods. 59

3-2 Creating the antialiasing ROM. The antialiasing ROM provides a mapping from the bitonal bitmap to the grayscale (antialiased) bitmap; the address of the ROM is computed from a window in the bitonal bitmap, and the values contained in the ROM respresent grayscale. 62

3-3 Creating the antialiasing ROM. 63

3-4 Antialiasing using ROM. Generic bitonal bitmaps (from commercial products) are antialiased by passing them through the ROM. 65

3-5 Selecting a window. A nearly vertical line is the worst case for an antialiasing ROM. This edge could go from $(0, -\infty)$ to $(-1, \infty)$ which would mean that all the intermediate pixels (i.e. the ones shown mapped in the window) would have grayscale values of about 0.5. . . 67

4-1 Example laser modulation pulse patterns. (a) A centered pulse (b) A wider centered pulse (c) An edge pulse from the left edge (d) A wider edge pulse 72

4-2 Pulse modulation patterns for laser modulator. 74

4-3 Laser printer simulator, OPC drum energy, vertical step, conventional modulation . . . 76
4-4 Laser printer simulator, toner, vertical step, conventional modulation 76
4-5 Laser printer simulator, OPC drum energy, vertical step, power modulation only. . . . 77
4-6 Laser printer simulator, toner, vertical step, power modulation only. Note the fuzziness resulting from a slow energy gradient. . . . 77
4-7 Laser printer simulator, OPC drum energy and toner, horizontal step, conventional modulation . . . 78
4-8 Laser printer simulator, OPC drum energy and toner, horizontal step, pulse edge modulation. Note that edge is still sharp. . . . 78
4-9 Laser printer simulator, OPC drum energy and toner, 1 pixel wide 45 degree line, conventional modulation . . . 79
4-10 Laser printer simulator, OPC drum energy and toner, 1 pixel wide 45 degree line, power and pulse edge modulation . . . 79
4-11 Laser printer simulator, OPC drum energy and toner, 2 pixels wide 45 degree line, conventional modulation . . . 80
4-12 Laser printer simulator, OPC drum energy and toner, 2 pixels wide 45 degree line, power and pulse edge modulation. Note that the edge is smooth, because adjacent pixels add to produce a straight line along the toner threshold. . . . 80
4-13 Laser Modulation Circuit . . . 85
4-14 Laser Modulation Circuit . . . 86
4-15 Clock Generation for Laser Modulation Circuit . . . 87

B-1 Original bitonal bitmap of the letter "S". . . . 152
B-2 Letter "S" after smoothing 1 iteration. . . . 152
B-3 Letter "S" after smoothing 2 iterations. . . . 153
B-4 Letter "S" after smoothing 4 iterations. . . . 153
B-5 Letter "S" after smoothing 8 iterations. . . . 154

B-6 Window shapes used in the windowed ROM antialiasing routine. (a) five pixel window (b) nine pixel window (c) thirteen pixel window (d) nineteen pixel window . . . 156
B-7 Bitonal learning pattern, consisting only of one pixel wide lines. . . . 157
B-8 Antialiased learning pattern, consisting only of one pixel wide lines. . 157
B-9 Overlapping lines, bitonal bitmap. . . . 158
B-10 Overlapping lines, antialiased with a window size of 5. . . . 158
B-11 Overlapping lines, antialiased with a window size of 9. . . . 159
B-12 Overlapping lines, antialiased with a window size of 13. . . . 159
B-13 Overlapping lines, antialiased with a window size of 19. . . . 160
B-14 Overlapping lines, antialiased with Gupta–Sproull method. . . . 160
B-15 Horizonal and vertical lines of various widths with one pixel steps, bitonal bitmap. This is a pathological case for windowed antialiasing. 161
B-16 Horizonal and vertical lines of various widths with one pixel steps, antialiased with a window size of 5. . . . 161
B-17 Horizonal and vertical lines of various widths with one pixel steps, antialiased with a window size of 9. . . . 162
B-18 Horizonal and vertical lines of various widths with one pixel steps, antialiased with a window size of 13. . . . 162
B-19 Horizonal and vertical lines of various widths with one pixel steps, antialiased with a window size of 19. . . . 163
B-20 The letter "S", bitonal bitmap. . . . 163
B-21 The letter "S", antialiased with a window size of 5. . . . 164
B-22 The letter "S", antialiased with a window size of 9. . . . 164
B-23 The letter "S", antialiased with a window size of 13. . . . 165
B-24 The letter "S", antialiased with a window size of 19. . . . 165
B-25 New bitonal learning pattern, consisting of areas and one pixel wide lines. . . . 166
B-26 New antialiased learning pattern, consisting of areas and one pixel wide lines. . . . 166

B-27 Overlapping lines, bitonal bitmap. 167
B-28 Overlapping lines, antialiased with a window size of 5. 167
B-29 Overlapping lines, antialiased with a window size of 9. 168
B-30 Overlapping lines, antialiased with a window size of 13. 168
B-31 Overlapping lines, antialiased with a window size of 19. Note that this learning pattern performed approximately equivalently to the previous learning pattern. 169
B-32 Overlapping lines, antialiased with Gupta–Sproull method. 169
B-33 Horizonal and vertical lines of various widths with one pixel steps, bitonal bitmap. This is a pathological case for windowed antialiasing. 170
B-34 Horizonal and vertical lines of various widths with one pixel steps, antialiased with a window size of 5. 170
B-35 Horizonal and vertical lines of various widths with one pixel steps, antialiased with a window size of 9. 171
B-36 Horizonal and vertical lines of various widths with one pixel steps, antialiased with a window size of 13. 171
B-37 Horizonal and vertical lines of various widths with one pixel steps, antialiased with a window size of 19. Note that the new learning pattern performs significantly better than the previous pattern; the two and three pixel wide lines are no longer jagged near the step. 172
B-38 The letter "S", bitonal bitmap. 172
B-39 The letter "S", antialiased with a window size of 5. 173
B-40 The letter "S", antialiased with a window size of 9. 173
B-41 The letter "S", antialiased with a window size of 13. 174
B-42 The letter "S", antialiased with a window size of 19. Note that the new learning pattern performs significantly better than the previous pattern.174

Chapter 1

Introduction

Innovations in the field of laser xerography have always been motivated by the desires for more detailed and accurate printing. Many improvements have resulted from increasing the resolution of laser printers - in fact, a major advantage of laser printers over other technologies is their relatively high resolution output at reasonable speeds. However, even as 300 dot per inch printers are becoming commonplace, their limitations are being noticed. Rather than increasing the resolution of the printer (which affects the system size and cost), the quality of the output can be increased without by using a technique called antialiasing. In antialiasing, the high frequency components are filtered out of the printout and smoother edges and curves result. The image must then be printed on a grayscale capable printer.

Most methods of antialiasing require having high resolution data (in the form of high resolution bitmaps or vector data) to work from. Subsequently, the antialiased scan conversion produces a high quality grayscale bitmap. Unfortunately, the antialiasing scan conversion must replace the conventional binary scan conversion algorithms which most software uses today in order for this scheme to work. It is possible to produce antialiased bitmaps directly from regular black and white bitmaps (at the same resolution) with a reasonable degree of accuracy.

This paper examines previous research in the field of antialiasing, proposes an improvement to allow antialiasing of bitmaps (without pre-scan conversion data), and develops the hardware necessary to implement the new antialiasing algorithm on

an existing laser printer (it converts the printer to grayscale).

1.1 Background of Laser Xerography

Laser printers evolved from two other forms of printing; xerography and dot matrix printing. Xerography, a form of electophotography, is a multistep system which transfers a image from light to paper. The seven step are shown in diagram 1-1.

The first step of the laser printing process is to pre-charge the photoconductor. A corotron, or a very thin wire that is biased at a high voltage, generates ions which drift to the surface of the photoconductor. Typically for laser printers, an organic photoconductor on a rotating drum (OPC drum) is used. After the drum has been charged, it is ready to be imaged by selectively exposing the drum to light. A modulated laser beam is swept across the drum repeatedly as the drum rotates. In every position the laser beam strikes, the photoconductor becomes temporarily conductive and discharges. The amount of discharge depends on the exposure; a typical potential vs. exposure curve is given in figure 1-2.

In the next stage, the development subsystem, the toner is applied to the photoconductor drum. "Toner" is usually a compound of two types of particles; toner particles which are 5–20 μm in diameter, and toner carrier beads which are 50-150 μm in diameter. The carrier beads are magnetic, and can be transported by spinning magnets. The toner particles are attracted to the carrier beads triboelectrically, so development occurs when the electrostatic forces from the photoconductor exceed the triboelectric forces and the toner particles stick to the OPC drum.

The toner particles are then transferred to paper. Paper is fed over the photoconductor drum, and a strong electric potential is applied over the paper. If the potential is large enough, the toner particles fly up and stick to the paper instead of the OPC drum. After the toner is on the paper, it must be fused to become permanent; heat and/or pressure are often applied with heated rollers. Finally, the system must be cleaned of excess toner and the electrostatic image must be erased. Cleaning can be accomplished with brushes and vacuums or a wiper blade. The electrostatic image is

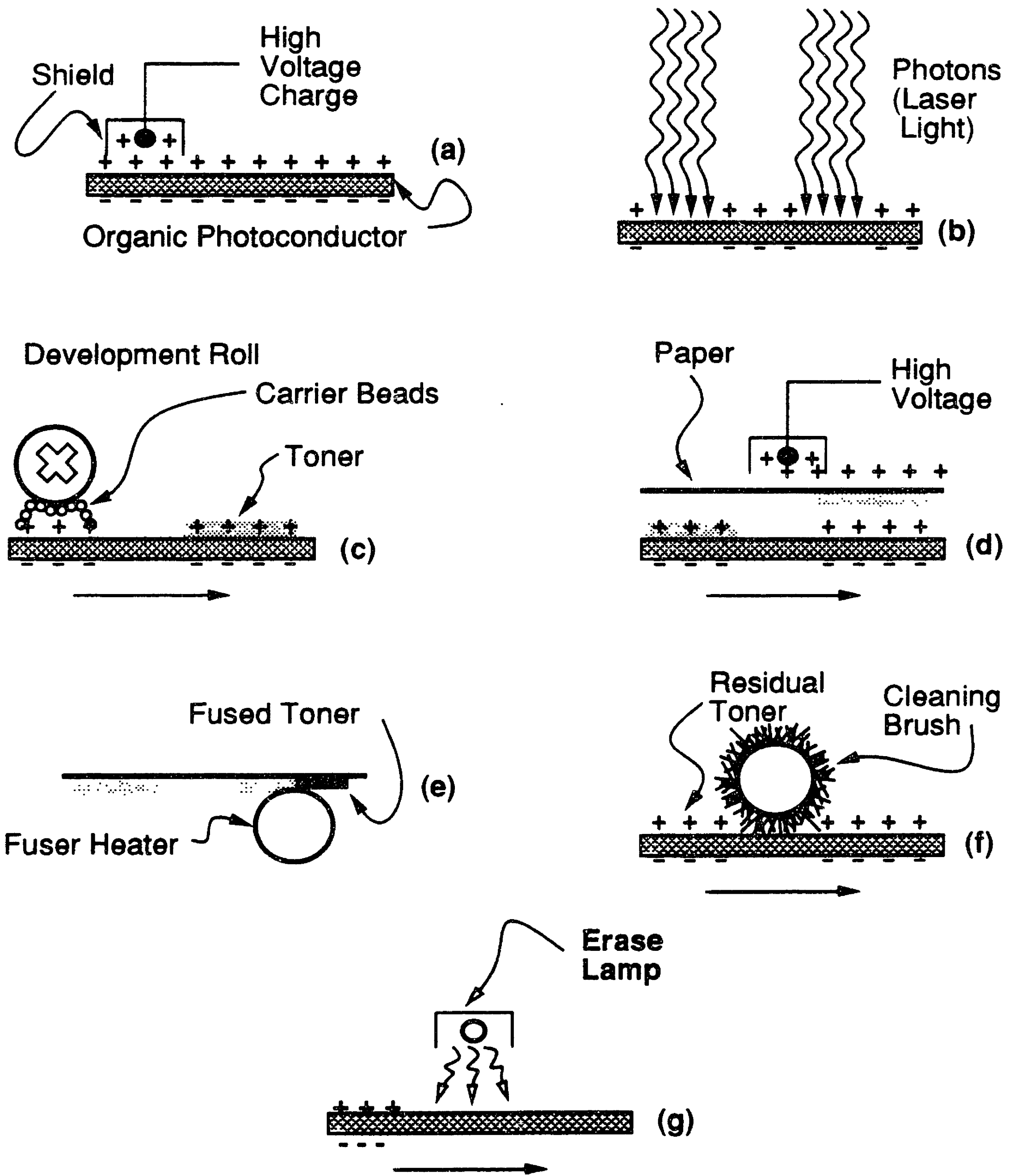


Figure 1-1: The seven steps of the xerographic process. (a) charging the photoconductor, (b) exposing to form latent electrostatic image, (c) developing the latent image into a real image, (d) transferring the image to paper, (e) fusing the image to paper, (f) cleaning residual toner from the photoconductor, (g) erasing the electrostatic latent image. (Redrawn from [24].)

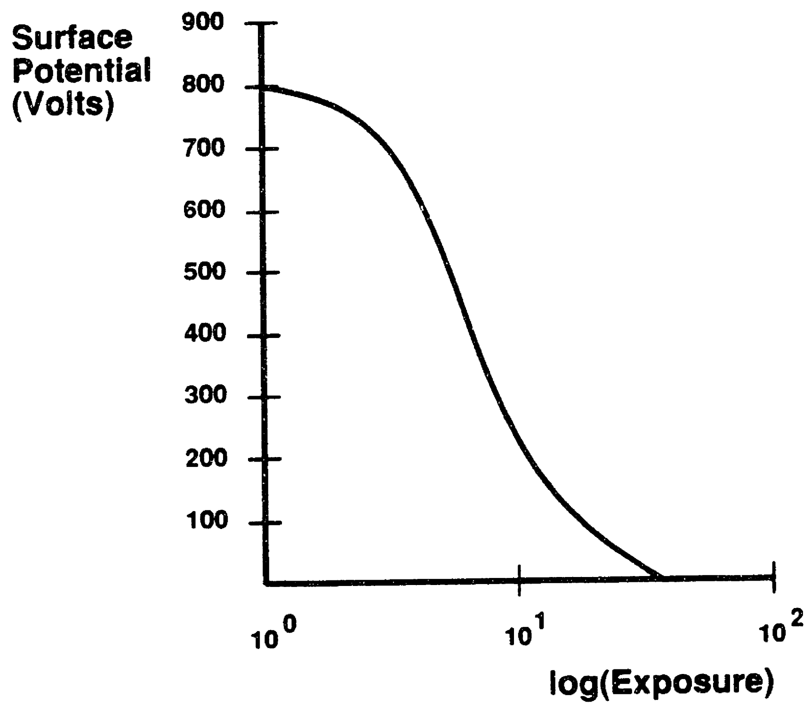


Figure 1-2: Example plot of a photoinduced discharge curve of a xerographic photoconductor. (Red-awn from [24].)

erased by illuminating the drum in the absence of a electric potential.

Today's laser printers usually accept orthogonal bitmaps of binary data. The laser beam is swept across the page as the page is rolled through the printer, modulating one row of data, or *scan line*, per pass. During each scan line, the paper is moved forward one pixel width. To obtain a quality result, it is important to produce pixels of consistent small size and shape at a correct location on the page, corresponding to the bitmap data. Many advances have been made in decreasing the pixel size and refining its shape to be more uniform and square.

Most new advances in printers today focus on increasing the resolution, since it is relatively easy to form uniform dots. Common laser printers today are capable of printing 300 pixels per inch in both directions. However, increasing resolution has its cost; the memory required to hold one page of bitmap data increased from 60,000 bytes in a typical 72 DPI dot matrix printer to more than 1,000,000 bytes in a 300 DPI laser printer. Increasing resolution has serious repercussions on the amount of memory used to store bitmap images and the bandwidth required of printers to produce images at the same speed and cost. Antialiasing provides a method of increasing the effective resolution without increasing the pixel spacing.

1.2 Quality in Laser Printing

Before methods of increasing "print quality" can be addressed, a working definition of quality must be introduced, along with a way to quantitatively measure the quality of a printout. Most current definitions base quality directly on resolution and dot shape of the printer; these parameters would certainly be specified in any type of contract for a laser printer engine. Alone, these two measures are in themselves fairly irrelevant. A more meaningful albeit subjective method of determining quality is to measure the ability to discern differences between two images or passages of text; one produced "perfectly" by photographic or other such high quality means, and the other copied xerographically. A rough measure of quality would be the distance at which any difference became discernible.

To remove the subjectivity from this measurement, a very high resolution scanner is used to sample both pages, and they can then be compared point for point. A statistical comparison yields a quality rating for the xerographic process. Another method for computing the performance of antialiased lines was proposed by J. Trueblood of the Sony Corporation [25]. In this method, a knife edge is positioned longitudinally parallel to the antialiased line. The intensity is scanned in high resolution along the knife edge only. The result is compared to a standard straight line. Both methods have the advantage of having no human intervention, but Trueblood's method can only be used for antialiased lines and not other graphics or image constructs.

1.3 Statement of the Problem

In order to characterize the types of data which are printed, I sectioned printed data into three main classes; text (fonts), graphics, and images. All representations are ultimately converted to bitmap data (matrix representation) before printing. According to J. Warnock [27], text can start from one of three classes:

Analytic: The font is encoded as a list of strokes which make up each character. The entire font can be modified by simply changing the stroke algorithms.

High Resolution Bitmap: The font is represented in higher resolution than the output will eventually be. Scan conversion includes averaging and rounding adjacent pixel values to size the font for the desired resolution.

Parametric Curved Outlines: The shape of the font is mathematically encoded. Scan conversion can occur at arbitrary resolution.

Graphics is very similar to text- it falls into the same three classes. Images are a somewhat different matter, though. Images are scanned scenes and therefore do not necessarily have a higher level representation. There is usually no way to easily convert an image to an analytic or other such form; instead, images are bitmaps.

Often images are halftoned.[1] As a result, image data are significantly harder to antialias than text or graphics data.

I looked for areas of low quality in text, graphics, and images. When I examined a page of text, problems with the edges of letters were immediately apparent; jagged edges can be found where smooth curves should exist. When fonts are digitized into discrete-time signals, this sampling naturally results in some degree of jaggedness. The jaggedness becomes especially noticeable along line edges, since the human eye is excellent at discriminating when lines are not continuous. Lines and curves drawn in graphics and image pages and images suffer from the same problem.

The problem lies with the edges of curves and lines. Mathematically, the concepts of edges can be separated from the rest of the printing process; it has been proposed by Bill Burling [6] that the two components of printing are *fill* and *edge*. Using these two operators, any text or image can be defined. "Fill" is the graphical component used to shade areas, such as the solid parts of letters and halftones in images. "Edge" is used to mark boundaries between different regions of fill. Edges can be either sharp (black and white boundaries) or gently rolled off by using shades of gray.

To fool the eye into thinking that an edge is continuous, the technique of antialiasing can be used. Rather than printing sharp boundaries between on and off regions (i.e. high frequency edges), a filtered edge (with the high frequency components removed) can be used. A Nyquist rate of edge transition can be determined for a printer, measured in distance rather than frequency. If every edge rolls off with least two pixels of transition (i.e. in a 300 DPI system, no frequencies higher than 150 DPI), then the printer is essentially "antialiased".

To implement antialiasing, the printing process must be changed to produce grayscale. One way to produce gray is to modulate the laser at less than full intensity. Due to the two dimensional Gaussian distribution of energy from the beam, the threshold region on the OPC drum will shrink in area and the dot size will decrease, leaving some white space around the black dot. The printer still operates at

[1] Note that grayscaling refers to the ability of an output device to product shades of gray, and that halftoning is the process whereby intermediate shades of gray are produced by arranging patterns of lighter and darker pixels.

the same initial resolution. Furthermore, this scheme is direction independent (it affects both the main scan and sub scan directions on the paper equally). This scheme works well in filled areas as well. Dithering is used to provide intermediate shades of gray from a darker and lighter shade. Previously black and white were the only available shades, but under the new scheme there are intermediate gray shades to choose from- as a result, halftone consistency is greatly improved.

Traditional methods of antialiasing can be used to generate grayscale pictures for the printer. However, many existing applications do not support antialiased output - in these cases, a bitonal bitmap is all there is to work from. A processor is used to filter the bitmap, edges are detected and fill zones are mapped out. The processor actually performs a reconstruction of an original analog image which has been rounded into the digital bitmap. Sampling this image and passing it the appropriate low pass filter, we can obtain most of the necessary grayscale information without supplying it from the original binary bitmap. A cellular automata method for recovering grayscale data from a bitmap is suggested. Because of computational inefficiency, an alternate method of grayscaling is suggested - a windowed look-up table.

Since laser printers are designed to print bitonally, modifications must be made to the print engine to allow grayscale data to be printed. More accurate control of the laser diode is critical. Bitonal printers operate the diodes well outside the linear zone of the development process. A laser driver circuit capable of operating the laser consistently at any power level from zero to the current level is needed. A higher bandwidth channel is required to operate the laser with grayscale. However, the video channel to the laser becomes lower bandwidth because it is now limited by the Nyquist rate for the printer; this insures that a faster laser diode device is not needed. The design parameters of such a laser diode circuit and their effect on the output of the printer are addressed, and a circuit to accomplish increased laser diode control is developed.

Chapter 2

Antialiasing

Antialiasing is a method of increasing the quality of text, graphics, and images without increasing the resolution. The method has its basis in discrete time signal processing; it is a filtering technique to remove high frequency edges from pictures. "Antialiasing" is somewhat of a misnomer - most antialiasing algorithms simply draw images without aliasing in the first place, rather than removing aliasing after scan conversion. Typically, an antialiasing filter is used during the scan conversion process. In order to examine antialiasing in detail, a good understanding of sampling theory is needed.

2.1 Sampling

To apply sampling theory to printing systems, the page data must be represented mathematically as a signal. A signal, which is defined as a function which conveys information[12], is often thought of in terms of time; however, it is equally valid to represent signals as functions of spatial coordinates. Usually, we will want to analyze the signal in only one direction at a time, so the other dimension is held constant and we observe "slices" of the image.

We normally think of the original two dimensional page data as a spatially continuous function. [1] However, it is impossible to send a two dimensional analog image

[1] This is easy to see for *analytic* and *parametric curved outline* data. *High resolution bitmap* data can be viewed as a sampled continuous signal. Although the bitmap is itself discrete, the origin of

over a wire because a wire is only one dimensional. One dimensional continuous time signals (lines) can certainly be sent over a wire, but it is undesirable to send an arbitrary analog signal because of another engineering constraint, signal noise. When analog information is transferred, any noise added to the signal results in unwanted changes to the transferred information. For information integrity, ease of transfer, and ease of storage, it is best to have both dimensions broken up into discrete values and sent digitally.

2.1.1 Point Sampling and Supersampling

The process of breaking up the continuous time signals into discrete values is called *sampling.* There are several methods of sampling. For the simplest method, *point sampling*, intensity values of the page function are recorded at regular points along both dimensions of the page. For example, a typical laser printer accepts data samples of 300 samples per inch along both dimensions of a page, horizontal and vertical. This corresponds to the final output resolution of 300 dots per inch. Usually, the sampling pattern is a regular orthogonal grid, although other patterns such as a hexagonal grids may also be used.[2] Regular orthogonal grids will be assumed in the absence of any other sampling grid specification.

There are several problems with regular point sampling. Objects on the page can be "missed" if the sampling grid is either too coarse or not aligned with the object correctly. (See figure 2-1) Objects may completely disappear when positioned off the sampling grid.

One possible fix is to sample at a higher spatial frequency. If our sample period is smaller than any object in both dimensions, then all objects are guaranteed to at least contribute to the sampled image. However, smaller objects or edges of objects are still misrepresented in the sampled version. There is still the problem that they may overlay a different number of sampling points depending on the position, and

---

the bitmap can be a continuous signal.

[2]The hexagonal grid is claimed by some authors to be the optimal sampling grid for most printing and display output processes. However, using hexagonal bitmaps adds complexity to most processing algorithms. [26]

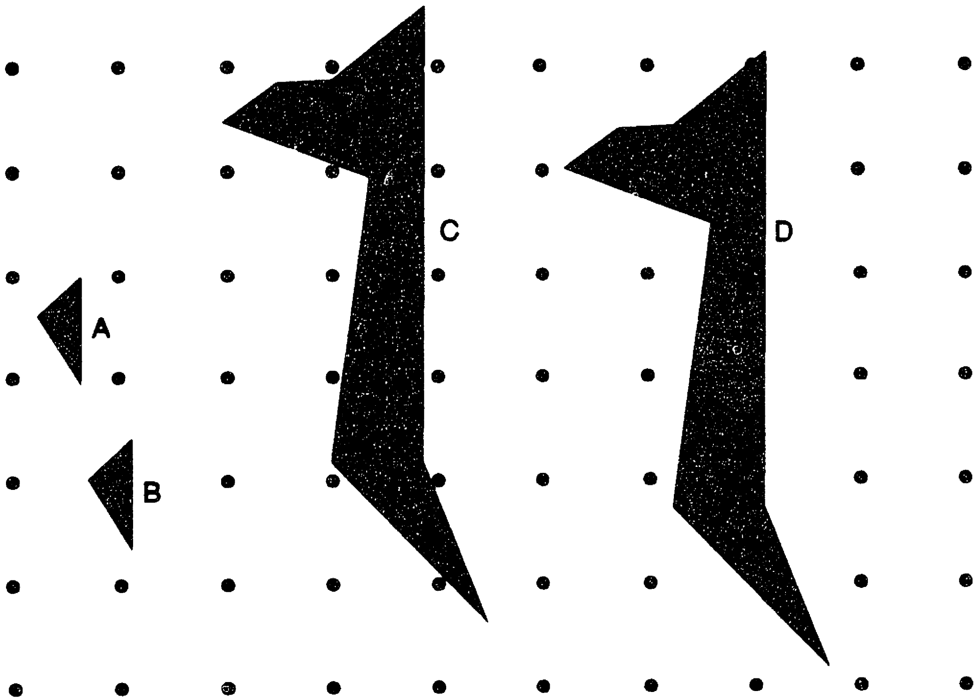


Figure 2-1: Point-sampling problems. The sample grid is represented by black dots. Objects B and D are sampled, but identical objects A and C are not. The shapes of the objects are not preserved. (Adapted from [12].)

the shapes are often lost after sampling. Effects of point sampling are even evident on objects far larger than the sampling spacing; edges are often misrepresented.

Supersampling, the technique of sampling at several points per pixel and somehow combining the results to form one sample, alleviates some of the problems of simple point sampling. The sampling rate is effectively increased to a multiple of its former frequency. Often, this produces good enough results, but the limiting case is the same. The same failure of simple point sampling can be demonstrated with supersampling; if the object is made small enough, it may be missed by the sampling entirely.

2.1.2 Area sampling

Point sampling is somewhat of a "bottom up" procedure. To point sample several objects, one must iterate through all the points of the sample grid, checking at each one if it is within any of the objects. *Area sampling*, in contrast, is a "top down" procedure; one iterates through the objects adding up their contribution to each sample. Samples are no longer thought of as values of the page function at given regular points; instead samples are proportional to the amount of area filled in around points of a regular sample grid. Typically, unweighted area samples are computed by integrating the page function over non-overlapping squares on a regular grid.

Area sampling clearly fixes the main problem of point sampling; no object, no matter how small, is missed during the sampling. Still it has a few problems to address. Coverage within a pixel is unweighted; the area in the corner of a pixel is as important as the area in the center at determining its final intensity. So small objects can be positioned arbitrarily within a pixel without affecting the final pixel intensities. But, as soon as an object starts to traverse from one pixel's area to another's, the final pixel intensities are affected disproportionately. (See figure 2-2) This dependence on the sample grid spacing is certainly a flaw; ideally, we want any change in object positioning to be reflected in the pixel intensity proportionally.

The concept of *weighted area sampling* addresses the problems of unweighted area sampling. In weighted area sampling, areas within a pixel contribute different amounts to the intensity of the sample. Typically, a conical or Gaussian weighting

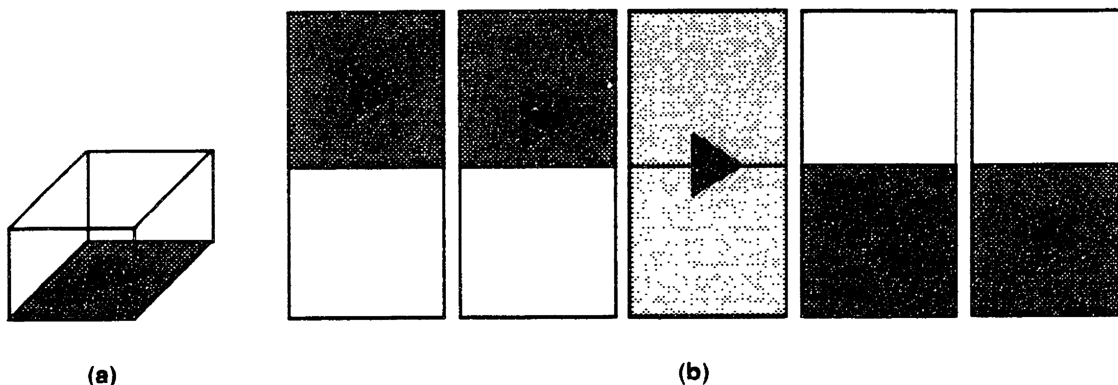


Figure 2-2: Unweighted area sampling. The pixel's intensity remains constant when the object is moving with the pixel, but changes abruptly as the object traverses the border to another pixel. (Redrawn from [12].)

function is used, where the area along the edges of the pixel contributes less than the area in the center of the pixel. This partially solves the problem of intra-pixel movement; as an object moves farther away from the center of the sampled pixel, it contributes less to the pixel's intensity. However, a given object should contribute a roughly constant intensity to the entire picture, regardless of where it is positioned in between pixels. (See figure 2-3) Therefore, as an object moves and contributes less to the intensity of a pixel, it must conversely contribute more to the intensity of the pixels it is moving toward. To solve this problem, overlapping pixels are used. (See figure 2-4) Every area contributes to more than one pixel, in proportion to the weighting function. The minimum pixel size to meet these conditions is two times the sampling area; each pixel must overlap at least halfway into all its eight neighboring pixels.

2.2 Fourier Transforms

After a page has been sampled using any one of the methods, the resulting sequence of intensity values can be stored and transferred digitally. At some point the page must be re-made from its samples in a process called *reconstruction*. The sampling process converts the signal from the continuous domain to the discrete domain, so the *reconstruction* process must accomplish the opposite, converting from discrete

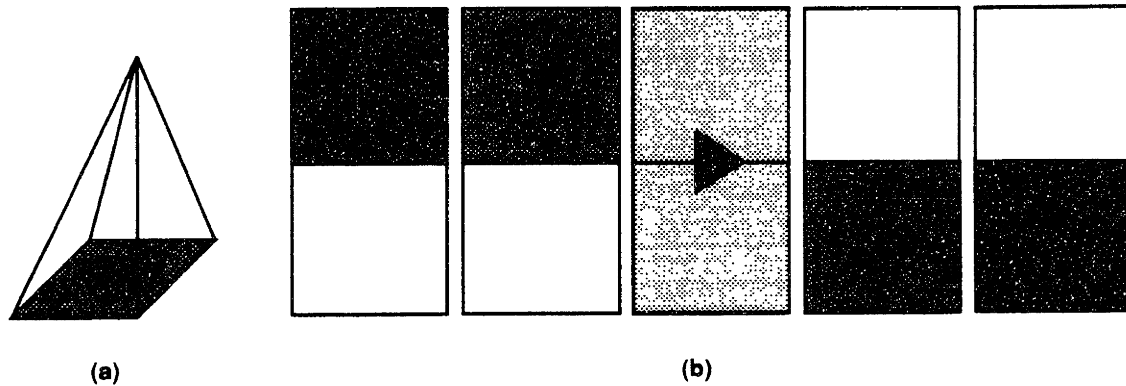


Figure 2-3: Weighted sampling without overlapping pixels. The object contributes differing amounts of intensity to the pixel depending on its position. The overall intensity of the page changes as the object moves. (Redrawn from [12].)

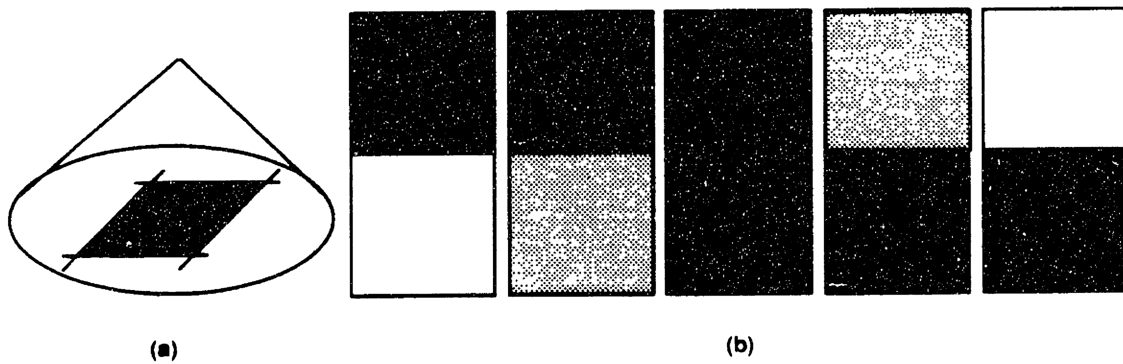


Figure 2-4: Weighted sampling with overlapping pixels. The overall intensity of the page remains roughly constant as the object moves between pixels. (Redrawn from [12].)

to continuous space. However, reconstruction is more difficult than sampling for a number of reasons. In order to see the correct process for reconstructing a sampled signal, Fourier analysis and signal process theory need to be used.

Until now, we have only considered signals in the spatial domain, i.e. a signal's changes as it is mapped across a page. Signals can also be thought of in the frequency domain, using Fourier analysis. A new signal $X(f)$, the Fourier Transform of $x(t)$, can be generated by applying the Fourier Composition equation to $x(t)$. $X(f)$ represents the frequencies which make up $x(t)$; for every value of f, $X(f)$ tells the amount (i.e. the amplitude) of that frequency in $x(t)$. A second equation is needed to convert $X(f)$ from the frequency domain back to $x(t)$ in the time (or spatial) domain.

$$X(f) = \mathcal{F}\{x(t)\} = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt \tag{2.1}$$

$$x(t) = \mathcal{F}^{-1}\{X(f)\} = \int_{-\infty}^{\infty} X(f)e^{j2\pi ft}df \tag{2.2}$$

The continuous Fourier transform can also be computed in two spatial dimensions for a signal $c(x_1, x_2)$ and its transform $C(f_1, f_2)$,

$$C(f_1, f_2) = \mathcal{F}\{c(x_1, x_2)\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} c(x_1, x_2)e^{j2\pi(f_1x_1+f_2x_2)}dx_1dx_2 \tag{2.3}$$

$$c(x_1, x_2) = \mathcal{F}^{-1}\{C(f_1, f_2)\} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} C(f_1, f_2)e^{-j2\pi(f_1x_1+f_2x_2)}df_1df_2 \tag{2.4}$$

and also in vector form for $c(x)$ and $C(f)$.

$$C(\vec{f}) = \mathcal{F}\{c(\vec{x})\} = \int_{-\infty}^{\infty} c(\vec{x})e^{-j2\pi\vec{f}^T\vec{x}}d\vec{x} \tag{2.5}$$

$$c(\vec{x}) = \mathcal{F}^{-1}\{C(\vec{f})\} = \int_{-\infty}^{\infty} C(\vec{f})e^{j2\pi\vec{f}^T\vec{x}}d\vec{f} \tag{2.6}$$

For the remainder of this discussion, higher dimensional cases of the transforms

will be ignored, because we can always hide extra dimensions in one dimensional forms by using vector variables.

$x(t)$ and $c(x)$ represent real valued functions, but their transforms $X(f)$ and $C(f)$ are complex valued functions. The transforms are broken into two real valued functions corresponding to the magnitude and phase of the transforms. In most analysis, the magnitude will be used exclusively and the phase will be ignored.

$$|X(f)| = \sqrt{\Re\{X(f)\}^2 + \Im\{X(f)\}^2}$$

$$\phi(f) = \tan^{-1}\left[\frac{\Im\{X(f)\}}{\Re\{X(f)\}}\right]$$

Equations 2.1 through 2.6 all represent continuous signals in the time or spatial domains. For the cases of *analytic* and *parametric curved outline* data, the data is in the form of a continuous function, and the previous transforms can be used to convert to a frequency representation. However, bitmap data is not continuous, and the previous transforms cannot be used.

To transform a discrete signal, we need to modify the transform equations. When a continuous function $x(n)$ is periodically sampled, the resulting signal $x[n]$, is equivalent to a bitmap. Therefore, we can perform a continuous transform of $x(n)$ multiplied by a sampling signal, and then incorporate the sampling into the transform. Sampling signals amounts to multiplying the continuous signal with a pulse train of period T.

$$x[n] = x(t) \cdot \sum_{n=-\infty}^{\infty} \delta(t - nT)$$

Multiplication of two signals in the time or spatial domain corresponds to convolution of the signals' transforms in the frequency domain.

$$X(f)_{discrete} = \left[\mathcal{F}\{x(t)\}\right] * \left[\frac{1}{T} \sum_{n=-\infty}^{\infty} \delta(f - \frac{n}{T})\right]$$

$$= \int_{-\infty}^{\infty}\left[\int_{-\infty}^{\infty} x(t)e^{-j2\pi\nu t}dt\right]\cdot\left[\frac{1}{T}\sum_{n=-\infty}^{\infty}\delta((f-\nu)-\frac{n}{T})\right]d\nu$$

The sum can be taken out of the integral to simplify the equation. To remove the sum entirely, $X(f)_{discrete}$ is normalized to 2π and only defined over one period (i.e. one point of the sum), yielding the Discrete Fourier Transform (DFT) composition equation. Although the transform is defined over one period, it actually repeats every period, but only one period is considered when talking about the transform. Similarly, a synthesis equation can be derived.

$$X(e^{j\omega}) = \mathcal{F}\{x(t)\} = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}\,d\omega \tag{2.7}$$

$$x[n] = \mathcal{F}^{-1}\{X(f)\} = \frac{1}{2\pi}\int_{-\pi}^{\pi} X(e^{j\omega})e^{j\omega n}\,d\omega \tag{2.8}$$

2.3 Aliasing and Antialiasing

$X(e^{j\omega})$ is normalized from a sampling frequency of $\frac{1}{T}$ to a period of 2π. Since $X(e^{j\omega})$ repeats every 2π radians, this implies that there is a maximum frequency which a DFT can represent. (See figure 2-5.) Any frequency greater than $\frac{1}{2T}$ interferes with the next period of the transform. It is important to exclude any such high frequencies from the signal $x(n)$ before the sampling is performed, otherwise it will be impossible to properly reconstruct $x(n)$ from $X(e^{j\omega})$. (*Reconstruction* will be examined in more detail in section 2.4.)

Suppose that the signal $x(n)$ is not *band-limited* before sampling; some of its frequency components are greater than $\frac{1}{2T}$. High frequencies appear as low frequency components of the next period in $X(e^{j\omega})$ so when the signal $x_R(n)$ is reconstructed from $X(e^{j\omega}) = \mathcal{F}\{x(t)\}$, it differs from $x(n)$. High frequency components of x(n) appear as low frequency components in $x_R(n)$. This phenomenon is called *aliasing*. (See figure 2-6.)

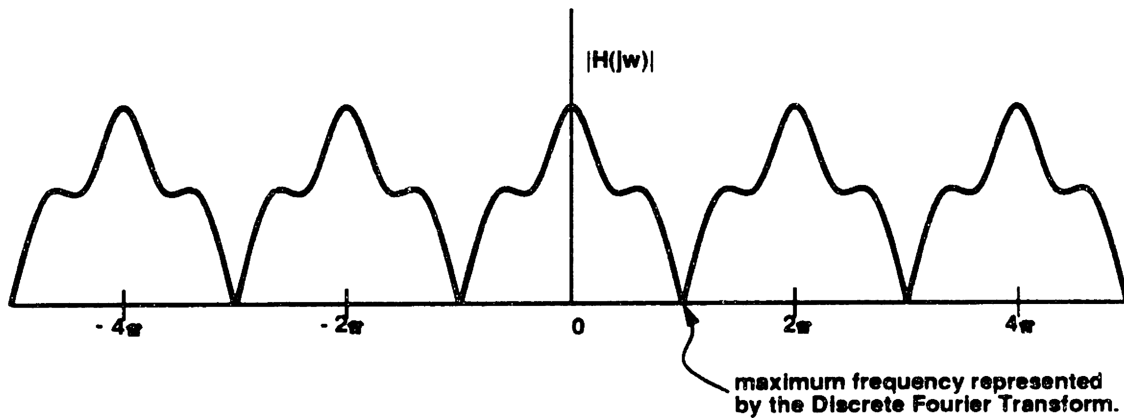


Figure 2-5: Example of a Discrete Fourier Transform (DFT). The signal is repeated at intervals of 2π. The maximum frequency able to be represented is $\frac{1}{2T}$, the Nyquist rate. The Nyquist rate is normalized to π in a DFT.

2.3.1 High Frequency Edges

Aliasing causes two major problems with computer graphics. Most seriously, we could have sampling problems such as in figure 2-1, where an object may be missed entirely or sampled several times depending on where it is positioned on the grid. Anytime "high frequency" objects are displayed, i.e. edges which switch faster than the Nyquist rate of the printer, this aliasing problem exists. For example, if a letter of a very small font is analytically defined to switch from black to white to black in less than two pixel widths, some of its frequency components are higher than the Nyquist rate. No printer can avoid aliasing this case; the image resolution is simply too high to represent.

With the resolution of today's printers, most font and graphic images have minimum feature sizes below the spatial Nyquist rate. On a 300 dpi printer, the spatial Nyquist period is $\frac{1}{150}$ in. $\frac{1}{150}$ inch is a fairly small distance on a printed page; an object with those dimensions on all sides is nearly invisible when examining the page from a typical reading distance. However, long thin objects are readily visible on the page, and they can disappear if sampled below the Nyquist rate. Fortunately, lines are usually wider than $\frac{1}{150}$ inch and only very small fonts have features which are smaller than this width. As a result, objects are rarely missed totally in modern printers. In the event that very small objects are included in the print data, a filter

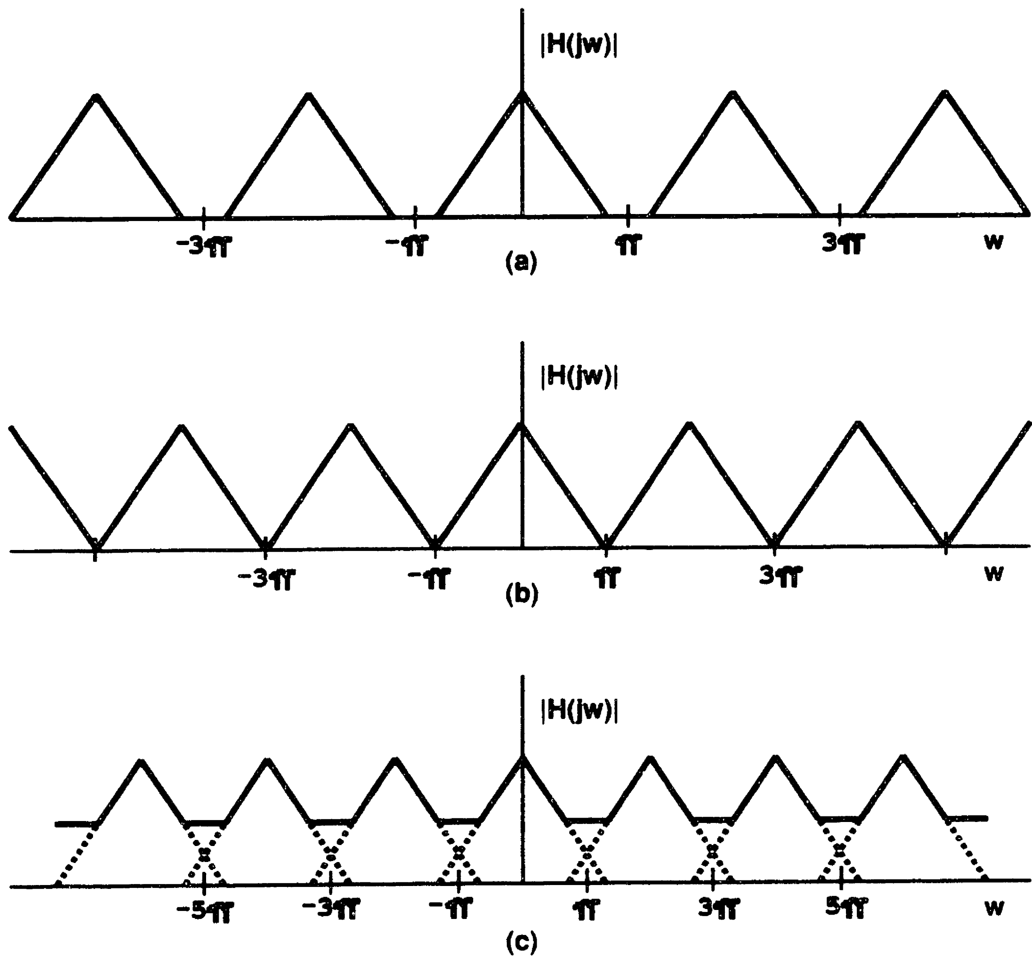


Figure 2-6: Aliasing. (a) Signal is sampled above the Nyquist rate, so there is no aliasing. (b) Signal is sampled exactly at the Nyquist rate. (c) Signal sampled below the Nyquist rate. Aliasing is evident; the waveform is deformed.

can be used to remove them.

Even though objects are not usually missed completely by sampling, they may still be sampled inaccurately by using inferior methods. In particular, when objects are point sampled, accuracy can be lost. The result of point-sampling a bitonal image to create a bitonal bitmap does not always have the same result as area-sampling the pixel and then quantizing the grayscale data to a bitonal pixel (the latter method is discussed in detail in the next section). For straight lined objects these two methods are identical, but when curves are introduced the point-sampling can yield inaccurate results. However, if the image is passed through a low pass filter and subsequently point-sampled and quantized, the resulting image matches the area-sampled and quantized result.

The key to obtaining accurate and identical results with these methods is low pass filtering; when all the high frequency components of an image are removed ($f_{picture} < \frac{1}{2}f_s$), the sampling and quantizing is valid up to a sampling frequency of f_s. Low pass filtering removes all the sharp transitions between intensities. This implies that all edges are tapered; a line is no longer thought of as a black region of fixed width, but rather a tapered region of gray which is blacker at the center and lighter near the edge. The area-sampling does not need a low pass filter to yield good results because it in fact performs a type of low-pass filtering. In general, we will always use a method of low pass filtering to insure accurate samples; either an explicit low pass filter or an area sampling method may be used.

2.3.2 Quantization

The second aliasing problem results from the quantization of data into different grayscale levels. Most laser printers have only two gray levels, black and white. If we have a discrete signal $x[n]$, then[21]

$$X(e^{j\omega}) = \mathcal{F}\{x(t)\} = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}\, d\omega \tag{2.9}$$

When $x[n]$ is quantized, a new signal $\hat{x}[n]$ results, where $\hat{x}[n] = x[n] + \Delta x[n]$. The

transform $\hat{X}(e^{j\omega})$ is

$$\hat{X}(e^{j\omega}) = \sum_{n=-\infty}^{\infty} \hat{x}[n]e^{-j\omega n}\, d\omega = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}\, d\omega + \sum_{n=-\infty}^{\infty} \Delta x[n]e^{-j\omega n}\, d\omega \quad (2.10)$$

$$= X(e^{j\omega}) + \Delta X(e^{j\omega}) \quad (2.11)$$

Assume that the signal $x[n]$ is zero outside the domain of the page; this limits the first sum. The quantization levels are constant across the entire page, so that the quantization error $\Delta x[n]$ is constant (but not necessarily zero) off the edge of the page. The off-page error is transformed into a constant value for $\Delta X(\omega = 0)$. The sum can be simplified:

$$\hat{X}(e^{j\omega}) = \sum_{n=0}^{M} x[n]e^{-j\omega n}\, d\omega + \sum_{n=0}^{M} \Delta x[n]e^{-j\omega n}\, d\omega + \delta[n] \cdot \Delta X(\omega = 0) \quad (2.12)$$

$$= X(e^{j\omega}) + \Delta X(e^{j\omega}) \quad (2.13)$$

11 The quantization error $\Delta x[n]$ depends on the page data and the quantization levels. In general, for a "random" image, the probability density function of $\Delta x[n]$ is uniform. However, typical images have a disproportionate amount of "black" and "white" compared to a "random" image. Printers are well suited to printing black and white, so $\Delta x[n]$ is nearly zero in the regions. Therefore, the typical probability density function of $\Delta x[n]$ is uniform except for a large zero error region. When the grayscale information for each pixel is represented in a B bit binary number, $2^B - 1$ uniform grayscale intervals result.[3] Furthermore, $\Delta x[n]$ is bounded, as shown in figure 2-7, and the magnitude of the frequency error can be computed.

$$|\hat{X}(e^{j\omega})| \leq (M+1) \cdot \frac{1}{2^B - 1} \quad (2.14)$$

[3] 2^B gray levels implies $2^B - 1$ grayscale intervals, since two are endpoints of the intensity spectrum. Usually, black and white are selected as endpoints; nothing can be lighter than white, and nothing can be darker than black.

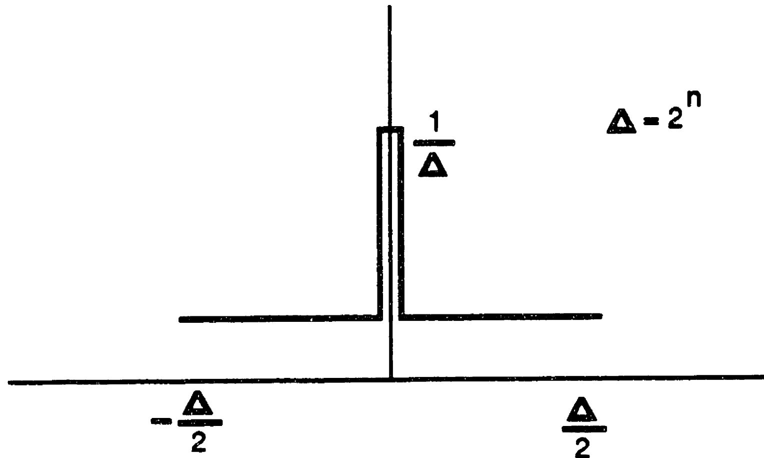
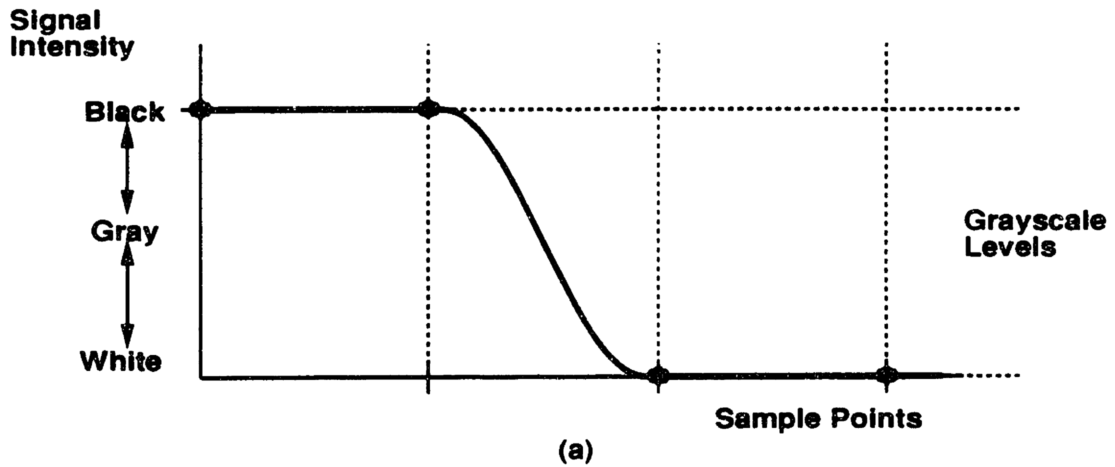


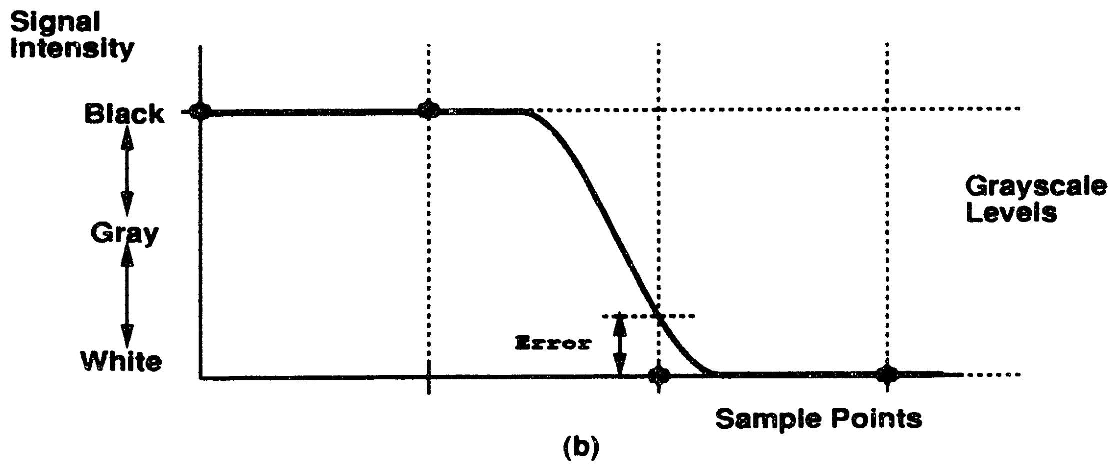
Figure 2-7: Probability density function for rounded quantization errors.

Usually only $\hat{X}(\omega = 0)$ has a large magnitude, depending on the method of rounding and exactly which grayscale values are used. Experimentally, it was found that the average magnitude of the quantization errors ($\overline{|\hat{X}(e^{j\omega})|}$) is much lower than its upper bound.

$$\overline{|\hat{X}(e^{j\omega})|} \approx \frac{1}{4} \cdot \sqrt{M+1} \cdot \frac{1}{2^B - 1} \tag{2.15}$$

In the case of a bitonal printer, a transition from black to white must occur in only one pixel. For an ideal edge which is lined up perfectly with the sampling grid, the transition between black and white occurs exactly at the spatial Nyquist rate. However, if the transition is not aligned with the sampling grid, then the resulting quantization error causes high frequency noise in the reconstructed signal. (See figure 2-8) Thus, even a "perfectly" low pass filtered and sampled object would be incorrectly printed. As gray levels are added to the printer, high frequency noise still exists, but its magnitude decreases linearly. The high frequency noise results in visual effects called "jaggies", or staircase patterns along smooth edges.





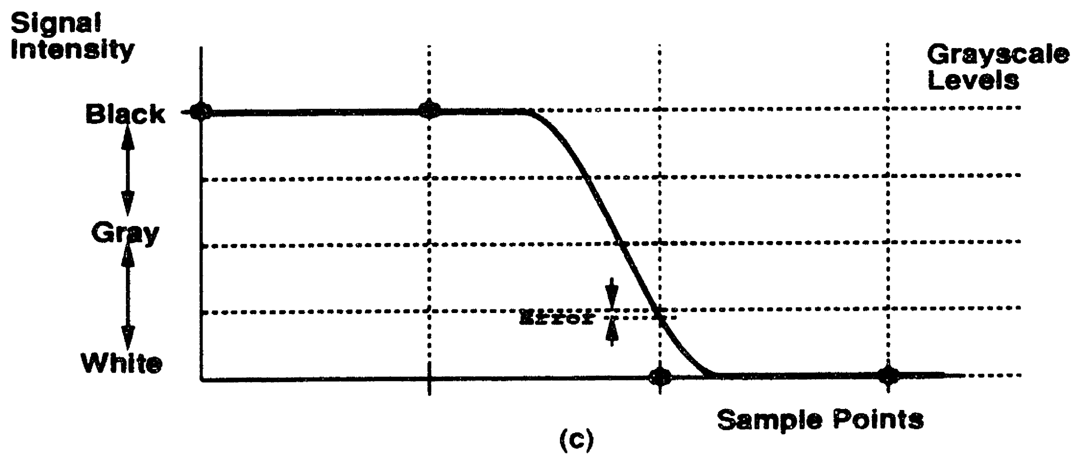


Figure 2-8: Quantization errors when sampling a black to white transition. (a) Sample points correspond to grayscale values, so the error is zero. (b) Sample points do not fall on grayscale values, so aliasing occurs. (c) More grayscale levels are introduced, so the aliasing is reduced.

2.3.3 Antialiasing

The purpose of antialiasing is to remove the high frequency components from a printer's final output. As shown, there are two causes for aliasing in images so it follows that there should be two distinct methods of fixing aliasing problems. The "high frequency data" problem can be fixed by appropriate low pass filtering to remove small objects and taper edges. The "quantization" problem can be repaired with slightly more effort; it requires additional hardware and software to be added, i.e. a printer capable of printing grayscale and antialiased scan conversion routines.

This analysis assumes several ideal components. These assumptions must be examined before any method of antialiasing can be developed. In particular, a perfect low pass filter cannot be built, and it is also difficult to build an efficient low pass filter. It is difficult to convert images from analog signals to digital information and back to analog signals. Also, an ideal printing device is also assumed in the previous analysis; this must also be challenged.

2.4 Reconstruction and Filtering

The method of reconstructing a continuous signal $x_R(t)$ from its discrete counterpart, $x[n]$ is fairly straightforward; convert $x[n]$ to an impulse train and pass the result through a low pass filter. Assuming that the original signal $x(t)$ was properly sampled above the Nyquist rate, then the reconstructed signal will be identical, i.e. ($x_R(t) = x(t)$). (See figure 2-9 and 2-10.) In the Fourier domain, the reconstruction process is represented as

$$X_R(j\Omega) = X(e^{j\omega})H_{LPF}(f) \tag{2.16}$$

where H_{LPF} is the ideal low pass filter function.

$$H_{LPF}(f) = \begin{cases} T, & \|f\| < f_s \\ 0, & \|f\| > f_s \end{cases} \tag{2.17}$$

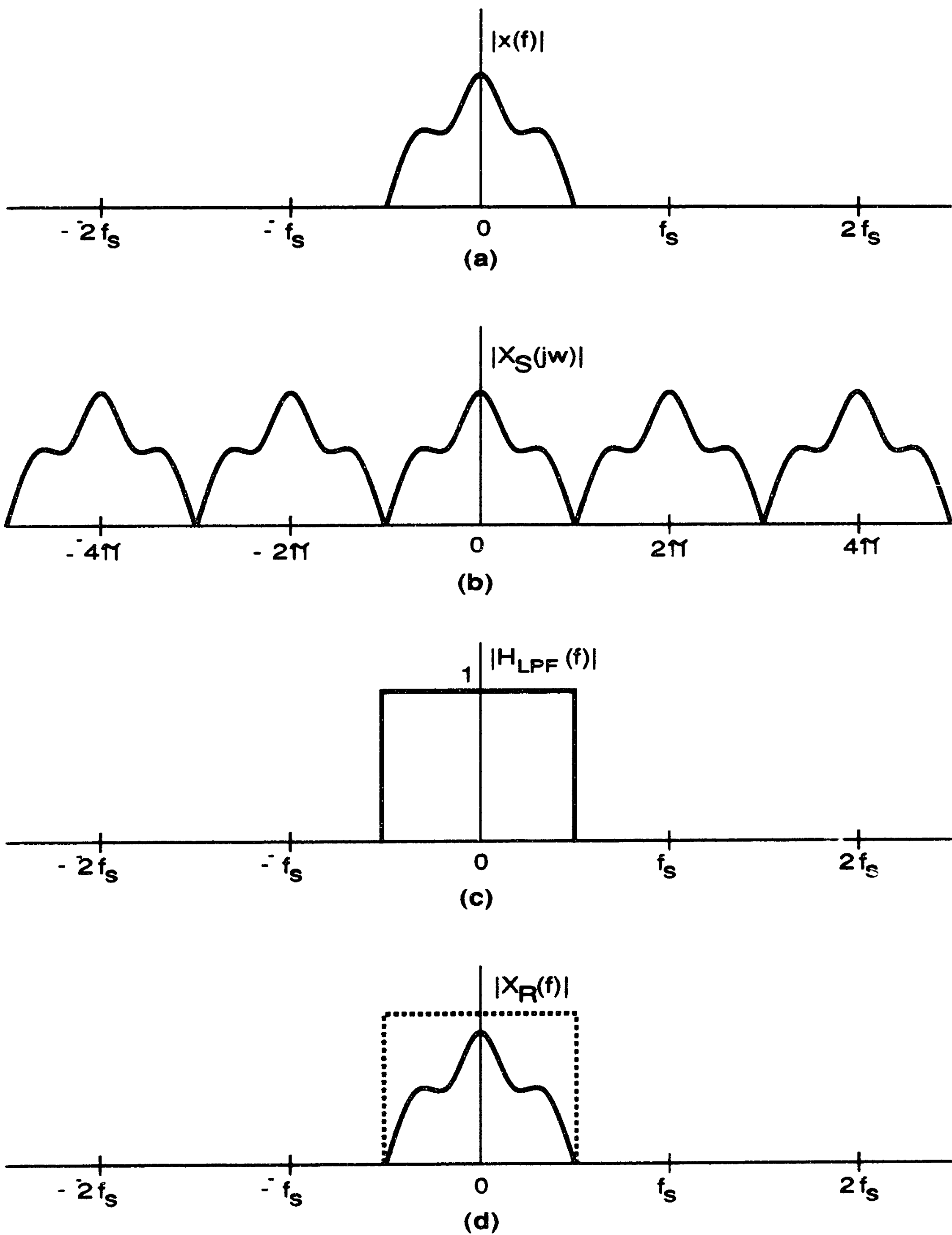


Figure 2-9: Sampling and Reconstruction. (a) Original continuous signal (b) Sampled signal (c) Low pass filter (d) Reconstructed signal after passing the (sampled) impulse train through the low pass filter.

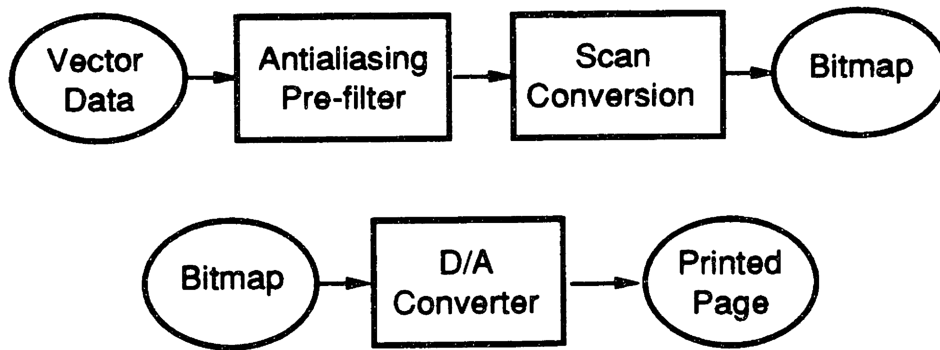


Figure 2-10: General laser printing model.

In the time (or spatial) domain, the corresponding function for the low pass filter is the sinc function, or

$$h_{LPF}(t) = \sum_{n=-\infty}^{\infty} \frac{\sin \pi t/T}{\pi t/T} \tag{2.18}$$

Unfortunately, there are two problems with making an ideal reconstruction filter; a low pass filter cannot be perfect, and it is impossible to make a perfect impulse train.

The sinc function (low pass filter) is in infinite impulse response (IIR) filter; it is non-zero over an infinite range. However, sinc approaches zero at locations far away from the origin, so it seems plausible to truncate the sinc in physical systems. A simple box filter can be applied for easy truncation, but the box filter is infinite in the frequency domain and only loosely approximates the sinc. A Bartlett filter (or triangular filter) gives a better approximation to the sinc, and a Gaussian filter is better yet. (See figures 2-11 through 2-14.)

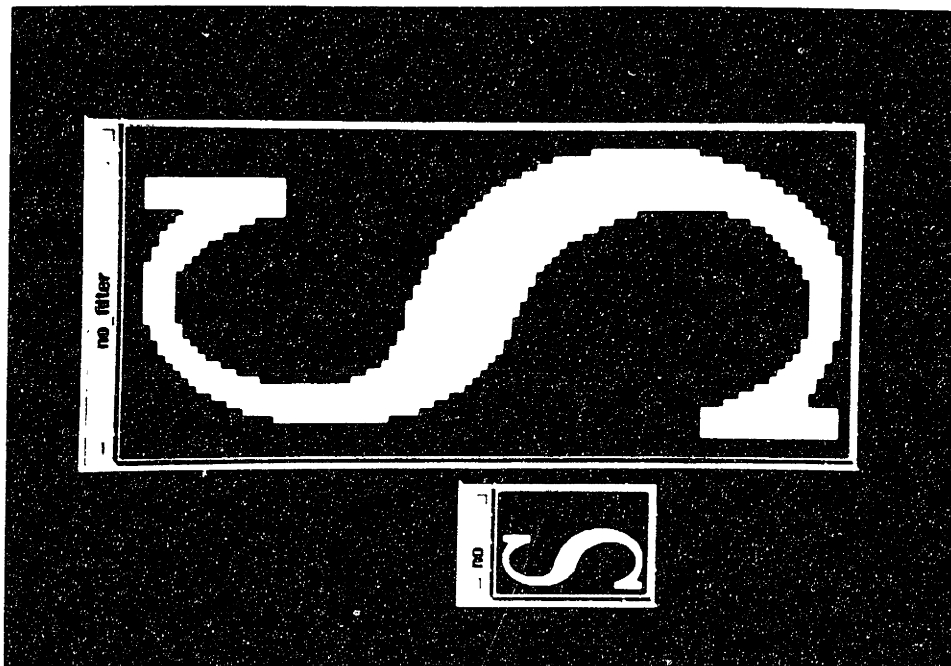


Figure 2.11 Bitonal bitmap of the letter S

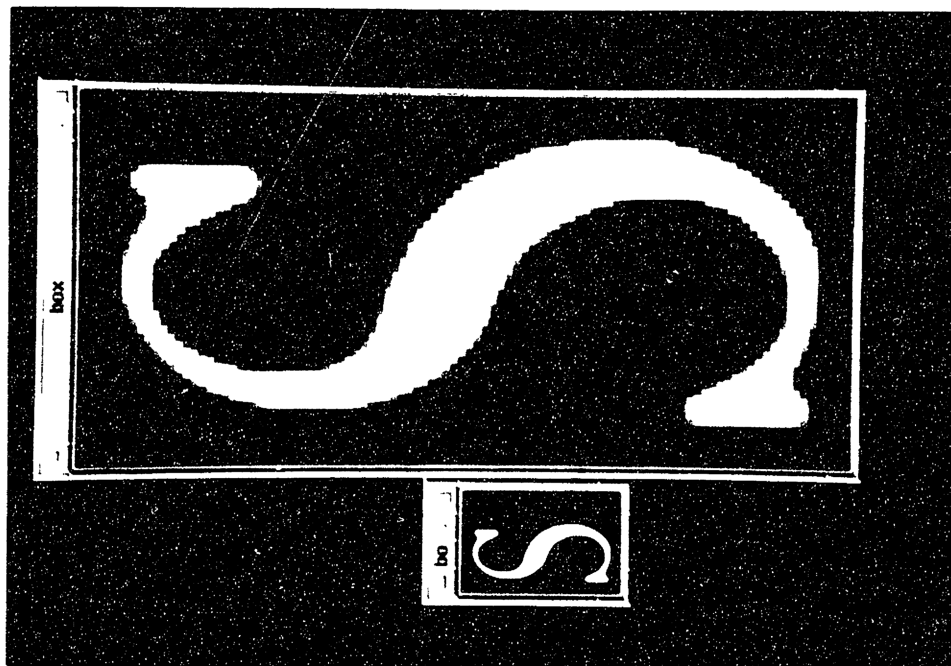


Figure 2.12 Box filtered letter S

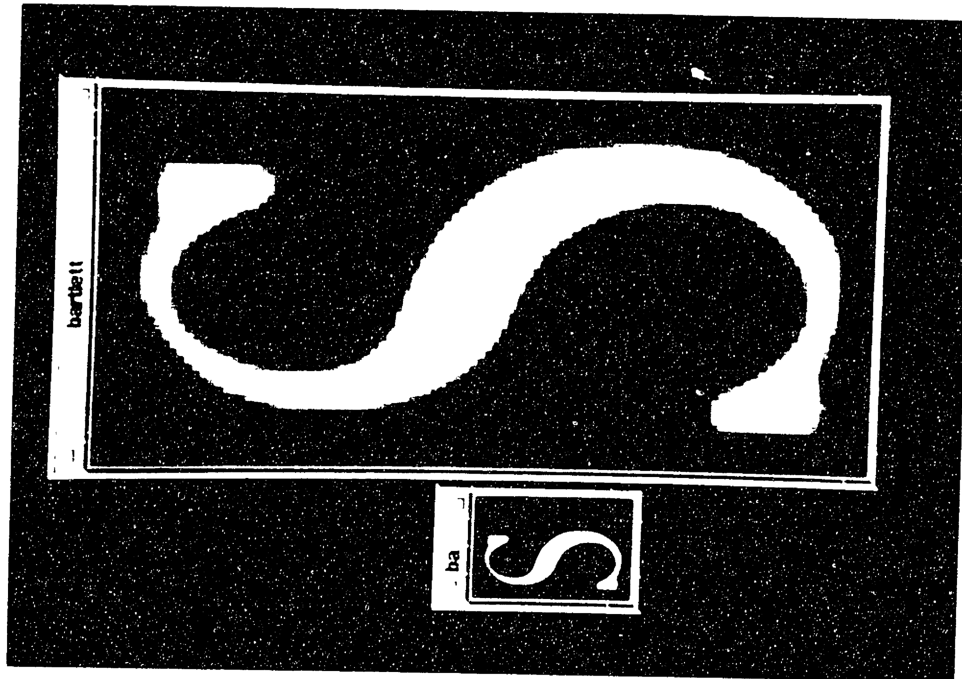


Figure 2.13 Bartlett filtered letter S

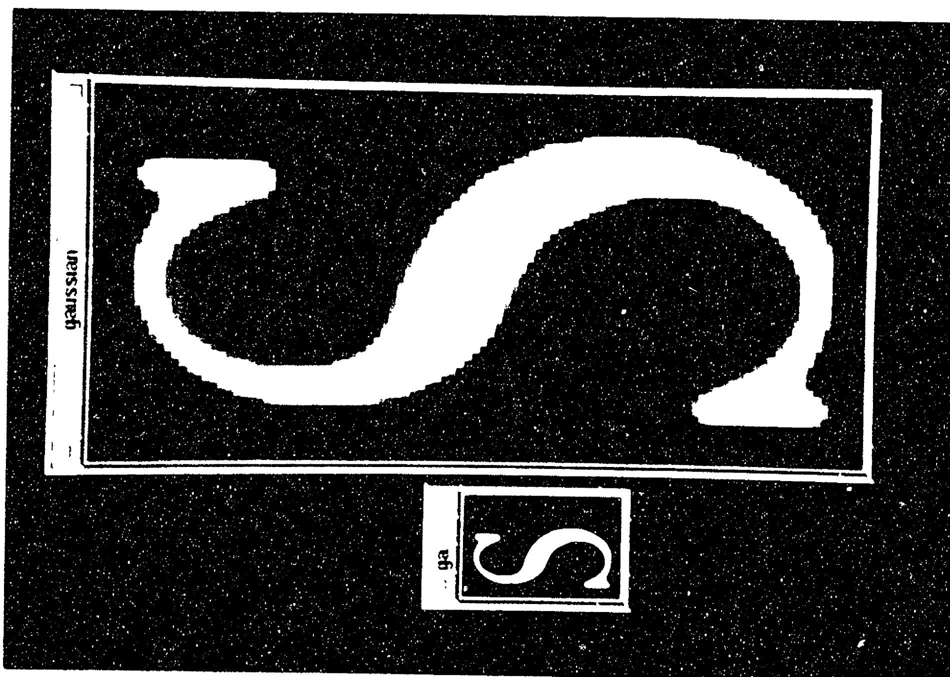


Figure 2.14 Gaussian filtered letter S

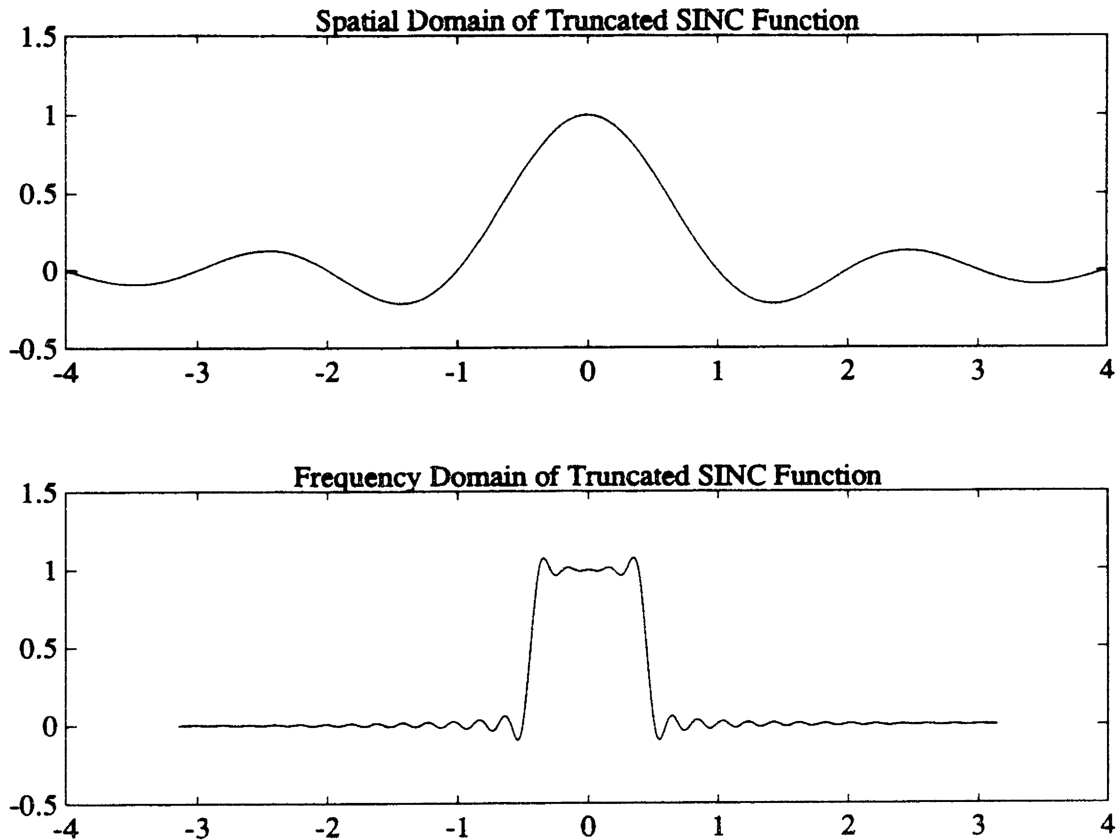


Figure 2-15: SINC function, truncated using a box filter.

More complex windowing methods can be used. For example, near-optimal windows can be designed using the Kaiser window filter design method, which used a window based a modified zeroth order Bessel function of the first kind. The impact in the frequency domain is easily seen; regardless of the windowing scheme, the resulting windowed sinc frequency waveform exhibits ringing (see figure 2-15). However, by making the windowed function large enough, it is possible to closely approximate a low pass filter, so it is usually possible to overcome the problem of accuracy of the sinc function.

The problem of physically converting digital samples into a pulse train is usually more bothersome; devices called digital to analog (D to A) converters usually perform the conversion. At regular intervals, a D-A converter accepts digital samples which are converted to analog and held; this "zero order hold" method is expressed

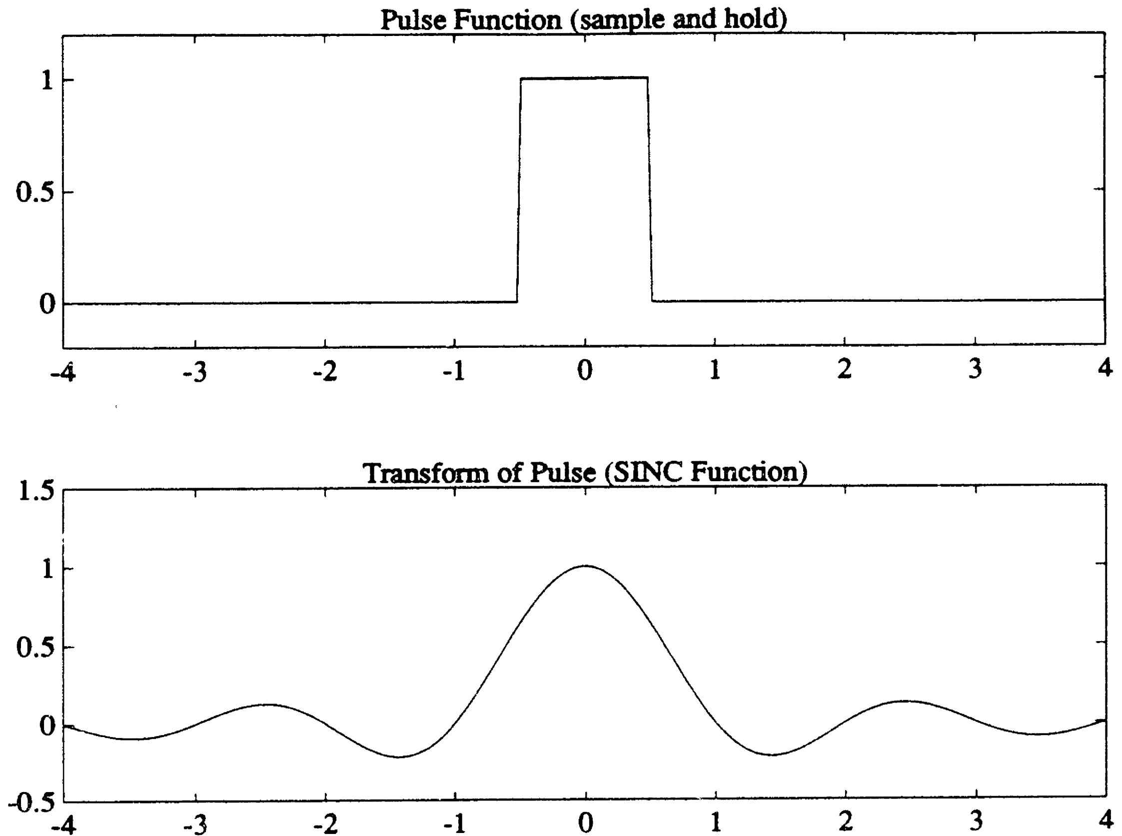


Figure 2-16: Rectangular filter.

mathematically as a pulse function. (Equations 2.19 and 2.20.) The pulse function's frequency spectrum looks little like an ideal low pass filter spectrum (see figure 2-16); the pulse function is actually a sinc function in the frequency domain. A so called "rectangular filter" still has some properties of a low pass filter, but it is far from ideal. In particular, its nonlinearity in the pass (low frequency) zone can distort the output, and there are prominent negative lobes in the frequency domain which can cause the resulting signal to fall out of the desired output intensity range.

$$h_{PULSE}(t) = \frac{1}{T} \sum_{n=-\infty}^{\infty} \frac{\sin \pi t/T}{\pi t/T} \tag{2.19}$$

$$H_{PULSE}(f) = \begin{cases} 1, & \|f\| < f_s \\ 0, & \|f\| > f_s \end{cases} \tag{2.20}$$

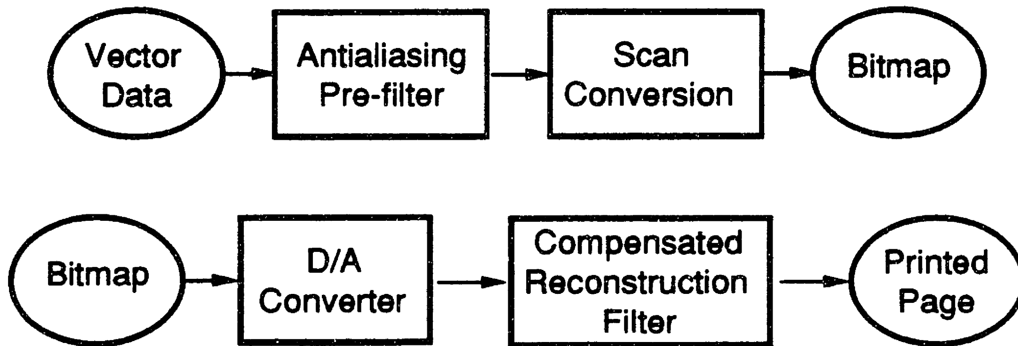


Figure 2-17: Laser printing model with compensated reconstruction filter.

As a result, A-D converters are often post-filtered to compensate for the rectangular filtering they inherently perform. The design of a compensated reconstruction filter follows from the definition of a low pass filter, the ideal reconstruction filter and the rectangular filter, which we are forced to use. (See figure 2-18 and 2-17.)

$$H_{COMP} = \frac{H_{LPF}}{H_{PULSE}} \tag{2.21}$$

Often, accurate reconstruction filters are not included in systems because of their relative complexity for the realized benefit. The most important function a compensated reconstruction fil'er performs is low pass filtering the image data; this can be accomplished with a simple low pass filter. The compensation for the frequency domain non-linearity of the zero-order hold filter is not nearly as important; at worst, the zero order hold filter attenuates the signal by −4 dB. As long as low pass filtering is done, there is little need for having a compensated reconstruction filter.

2.5 Justification of Antialiasing

It might seem that antialiasing is no more than an optical illusion - since antialiasing routines convert sharp edges to gradual (grayscale) edges, it might seem that the smoothing effect we witness is just blurring. This would clearly be a degradation of the image quality. However, from the signal processing analysis, we see that antialiasing

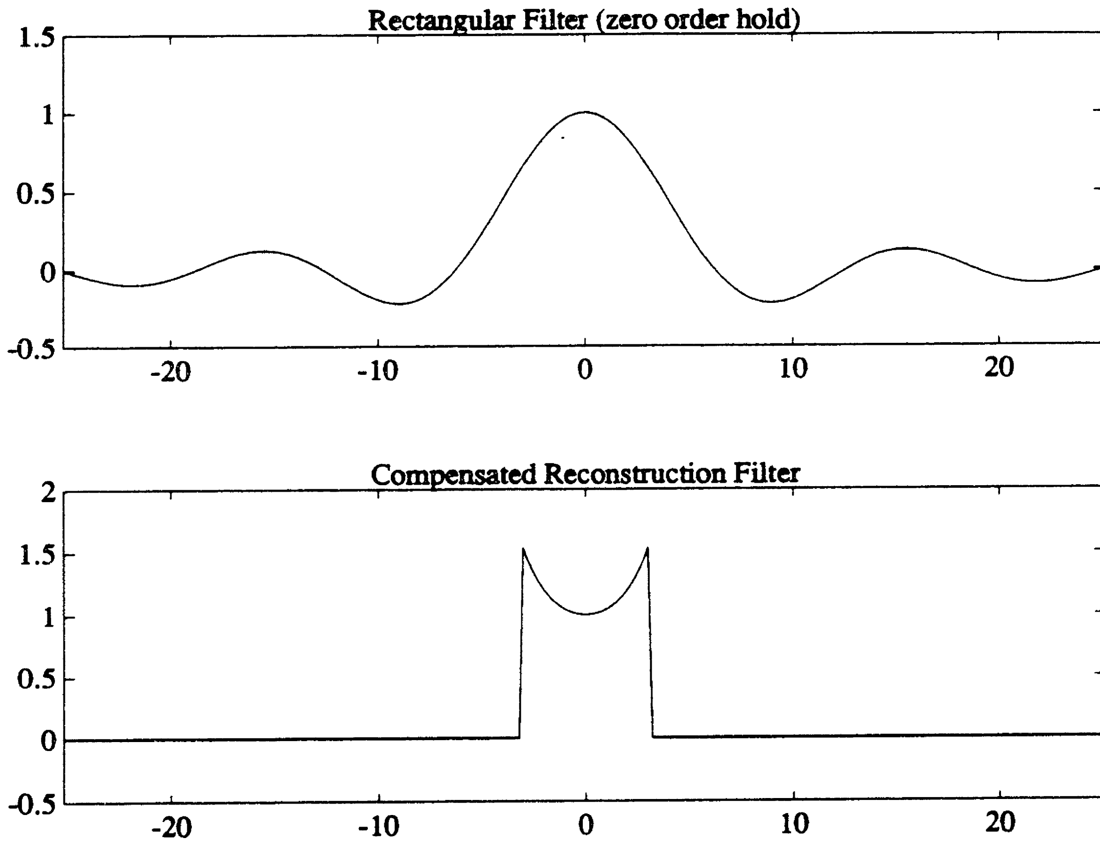


Figure 2-18: Compensated reconstruction filter. (a) Rectangular filter. (b) Compensated reconstruction filter.

only attenuates above the Nyquist frequency. The question is whether there is any useful data in the high frequency data, or whether it serves merely as a distraction to the eye. Although antialiased lines appear smoother to the eye, it was not clear that a two pixel wide line could be the same accuracy as a one pixel wide line. Since then, it has been shown that antialiasing increases the quality of the line as measured by a knife edge aperture (to measure the consistency of intensity perpendicular to the line direction)[25].

2.6 Video Display vs. Laser Printer Antialiasing

Much of the research in scan conversion algorithms has been motivated by video monitors rather than laser printers, so it's important to justify that the research devoted to video text and graphics can still apply to laser printers. Therefore, I will look at the similarities and differences between video monitors and laser printers.

2.6.1 Similarities

Similar Aliasing Problems The same types of problems due to aliasing in images are apparent in both media, such as "jaggies".

Scanning Both video monitors and laser printers write information with scanning beams. A page is displayed or imaged by writing across each scan line in succession. *Rastering*, or discontinuities between adjacent pixels as a result of scanning, is similar in both media.

Gaussian Beams Gaussian beams are used as a transfer device in both video monitors and laser printers. In a monitor, a scanning electron beam impacts the phosphor surface to create light. The distribution of the beam (in both vertical and horizontal) directions is Gaussian. Likewise, the laser beam in a laser printer which discharges regions on the OPC drum also has a Gaussian distribution. Any filtering due to the Gaussian beam would happen in both systems.

Response of Transfer Device In a video monitor, the phosphor is struck by a scanning Gaussian electron beam. The phosphor provides a capacitive effect to the beam - in effect, it integrates the beam's energy as the beam sweeps across which results in an intensity at every spot. Mathematically, the intensity of a point $I(x, y)$ is the convolution of a Gaussian with a pulse function.

$$I(x,y) = E_o \int_{-X_1}^{X_2} e^{-K_1[(x-z)^2+(K_2 y)^2]} dz \tag{2.22}$$

Laser printers employ a similar scheme. In a laser printer, the laser beam strikes a photoconductor (OPC) drum, causing parts to discharge. Although the OPC drum is not a linear capacitive device, the drum does perform some degree of integration as the laser beam is scanned. In the printing process, a particle of toner is analogous to a phosphor atom of a video screen. If the voltage left on the drum is high enough, the a toner particle will transfer to that point during the development stage. As long as the toner particles are small enough in comparison to the pixel width, then the drum can integrate the Gaussian laser beam to produce density. However, because of the non-linearity of the drum capacitance, a transfer function must be incorporated for the development process. Drawn as a density vs. log(Energy) curve (d-log-E), the transfer curve $T(Energy)$ typically maps the entire development process and not just the drum capacitive effects.[4] The density function is

$$D(x,y) = T\Big(E_o \int_{-X_1}^{X_2} e^{-K_1(x-z)^2+(K_2 y)^2} dz\Big) \tag{2.23}$$

The capacitance of the drum is somewhat dependent on the frequency of the laser pulse striking it[24], but it is guaranteed that the integration performed to find $D(x, y)$ is valid when the frequency of the modulating pulse is less than or equal to the original frequency. Fortunately, antialiasing implies that we do not need to increase the frequency of the laser beam signal; in fact, we are guaranteed that the maximum laser beam frequency will be reduced because of the low pass filtering. Also, the pixel

[4] Adding levels of gray to a laser printer directly contradicts a design goal of bitonal laser printers; for a bitonal process, it is desirable to have a very non-linear development transfer curve!

size remains unchanged even if more levels of gray are added.

2.7 Antialiasing in Practice

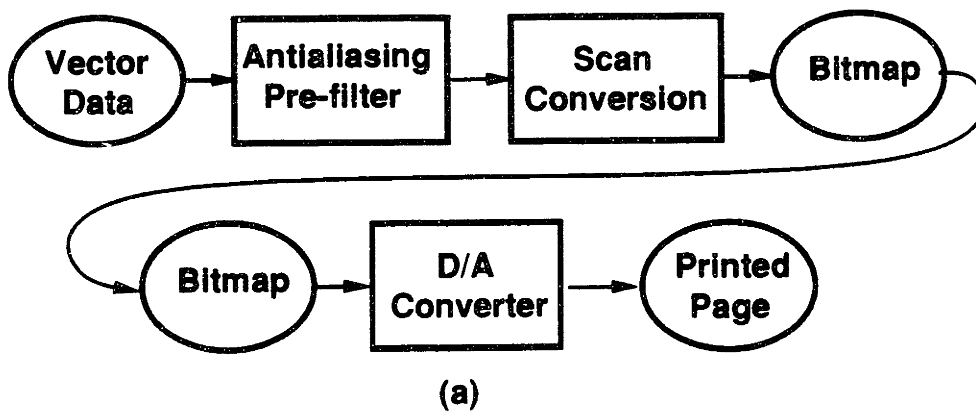
Numerous methods have been proposed to form antialiased bitmaps from vector data. The simplest form of antialiasing is simply to low pass filter the vector representation so that all frequencies above the spatial Nyquist rate are cut off. A general two dimensional low pass filter is extremely inefficient, requiring $O(M^2)$ flops per pixel, where $M \cdot M$ is the size of the filter. For a filter with $M = 20$, a page would require in the order of many hundreds of MFLOP to filter. General two dimensional filters are not used for this reason; instead, separable filters are used, which reduce the computation to $O(M)$ at the expense of some accuracy.

Most antialiasing methods break into two categories of filters, "pre-filters" and "post-filters". (See figure 2-19.) A pre-filter is employed to low pass filter data *before* it is sampled. In contrast, a post-filter is used to low pass filter data *after* it has been sampled.

2.7.1 Supersampling

Supersampling is one method of trading off memory for computation. The method is conceptually quite simple: The image is drawn bitonally on a "supersampled" grid. A grid sampled at N times the Nyquist sampling rate would require $N \cdot N$ times as much memory. After the image is scan converted, clusters of samples are combined (with a filter) to form grayscale pixels. (See figure 2-20.) Although it may seem that this idea is as computationally unreasonable as the original method (low pass filtering), only enough samples to differentiate grayscales are needed; for example, if a laser printer were capable of 16 grayscales, a $4 \cdot 4$ window would probably be sufficient. A rule of thumb is that supersampling four times in each dimension often will be satisfactory.[12]

Variations of supersampling can be used to increase efficiency. For example, **adaptive supersampling** uses different sampling rates across the image. When the sys-



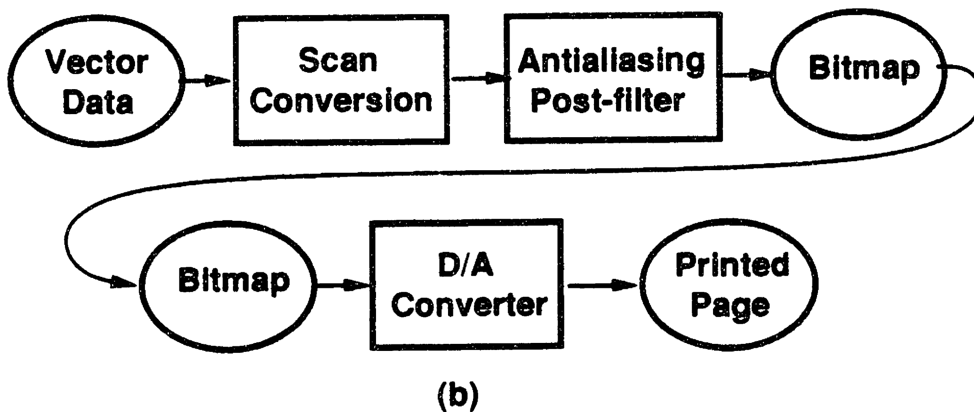


Figure 2-19: Prefiltering and Postfiltering. (a) Prefiltered data is antialiased before scan conversion. (b) Post-filtered data is antialiased anytime after sampling (not necessarily before bitmap storage or transmission.)

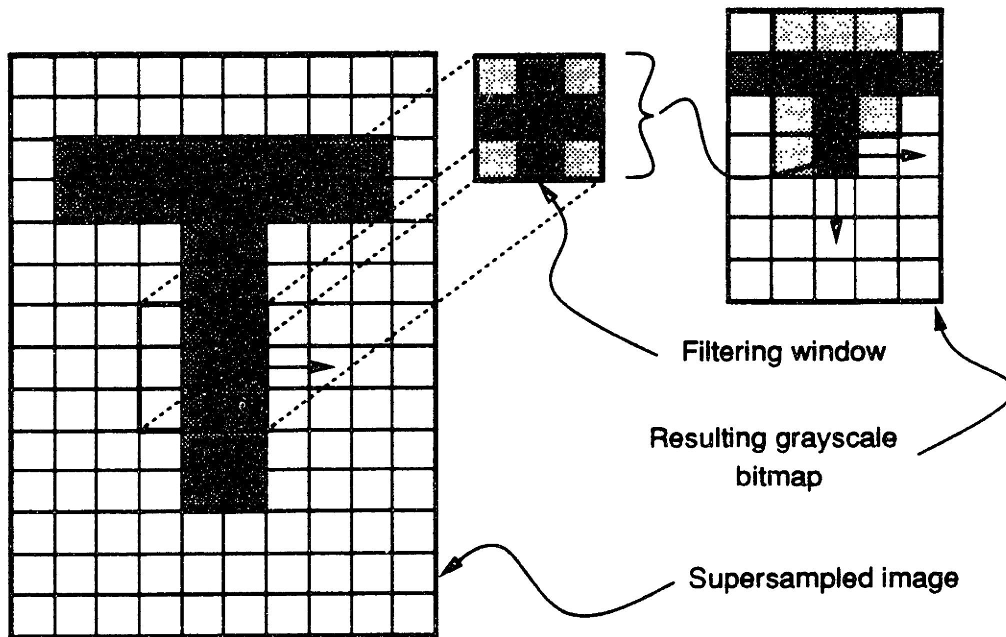


Figure 2-20: Antialiasing by supersampling. The filter is mapped across the supersampled image. The sum of product of corresponding values determines each grayscale pixel value. (Adapted from [12])

tem determines that more samples are needed, it changes the sampling rate for that region. Edges would require the maximum sampling rate, while large areas with monotonic data would need a much lower sampling rate.

2.7.2 Area Sampling

Area sampling is nothing more than a pre-filtered version of supersampling; area computations are done directly from vector data rather than converting to a high resolution bitmap first. Since the grayscale values result directly from the vector data without sampling, this method yields highly accurate results. However, weighted area sampling can be very computationally expensive; each object's area must be integrated with a weighting function to obtain the grayscale value. In addition, each object must be examined to determine the contribution to a single pixel.

Overlapping objects become a serious problem for area sampling. In supersampling, the scan conversion algorithm takes care of overlapping areas by mapping them

on to the same supersampled pixels. However, in area sampling, all objects must be considered simultaneously to resolve such conflicts. Otherwise, overlapping areas from different objects would be counted twice in computing the grayscale value of the pixel.

As in adaptive supersampling, **adaptive area sampling** can be used to increase efficiency without loss of accuracy. Edges still require an integration (or a table lookup), but large areas can easily be precomputed.

2.7.3 Gupta-Sproull antialiased lines

The previous approaches to antialiasing are extremely time consuming. Optimized methods of drawing lines and objects have been developed to overcome the inefficiency of supersampling and area sampling. One such method is the Gupta-Sproull antialiased scan conversion for lines.[14] Gupta-Sproull antialiasing uses the endpoints of a line and a pixel filter function to approximate antialiasing. A conical filter of radius 2 is suggested, and the algorithm is documented for one pixel wide lines, although this may easily be changed.

The algorithm precomputes the integration of the filter with a line, as shown in diagram 2-21. A table is made with the line distance from the pixel's center vs. the grayscale value which should appear at that pixel. The grayscale values are computed by performing an integration over the intersection of the conical filter and the line.

Scan conversion of arbitrary lines is performed using an incremental algorithm, similar to Bresenham's line drawing routine[4]. For lines in the first octant, a vertical cluster of three pixels is examined for every horizontal position. For each of the three pixels, the perpendicular distance from the center of the pixel to the center of the line is computed, and the pixel is intensified as indicated by the table.

The algorithm is easily extended to the other seven octants using symmetry. It is highly efficient because the distances are computed incrementally. The results can be made arbitrarily accurate by increasing the size and resolution of the table; for a four bit grayscale output, a table size of 24 bits sufficed[4]. The Gupta-Sproull algorithm is also easily adapted to antialias polygon edges and endpoints of a line by using different tables.

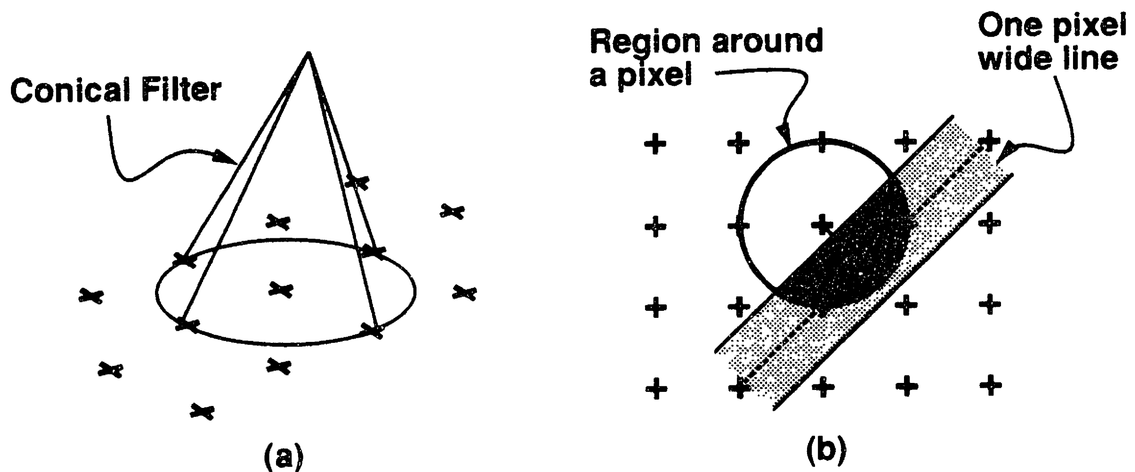


Figure 2-21: Gupta-Sproull antialiased lines. (a) A conical filter. (b) Filter convolved with a line of width 1 to form a table of filter values for Gupta-Sproull antialiased line drawing. (Adapted from [14])

2.7.4 Antialiased brushes

Few general curve antialiasing routines exist. One such method is Turner Whitted's method of using antialiased brushes to draw arbitrary curves.[29] The brush is first constructed at high resolution (i.e. a supersampled bitmap) and then digitally filtered to reduce the component frequencies to below the Nyquist rate of the drawing. Each pixel of the brush is additionally tagged with a depth (z), which prevents it from being overwritten by less important pixels as the brushes moves incrementally across tne page. The supersampled brush is then dragged across the page, but only the brush pixels which exactly coincide with the lower resolution page pixels are copied to the page. A generic z-buffer algorithm is used to determine if the page pixel should actually be updated.

There are several disadvantages to antialiased brushes. For every different type of line, a different brush is needed, so libraries of brushes must be kept. Additionally, the z-buffer algorithm is computationally expensive; the brush must have three or four bits of depth information per pixel to operate properly. Despite its drawbacks, the method does produce high quality output.

2.7.5 Character look up tables

When displaying fonts, it is time consuming to antialias each letter as it is scan converted. Fonts are especially painful because each character is both complex (consisting of possibly dozens of strokes) and interconnected. Rather than antialias during scan conversion, it is better to store grayscale antialiased copies of the font in arrays and simply copy (bit-blit) the characters to the video memory.[27]

A common problem with scan converting fonts occurs when the output resolution (pixel spacing) is too coarse to properly position the glyphs. Proper sub-pixel positioning can be achieved by storing various different positioned fonts in separate arrays, and bit-bliting from the correctly sub-positioned font array. This method has the disadvantage of using a lot of memory, but usually the fonts which have pixel spacing problems are small enough that it is irrelevant. Excellent results can be achieved using this method if enough memory is available for all the fonts.

2.7.6 Antialiasing with bit masks and look up tables

A efficient algorithm for scan-converting antialiased polygons uses a table lookup to draw antialiased edges. For each pixel, the pixel edge — polygon edge intersections are computed, and these values are used as the address to a lookup table.[1] (See figure 2-22.)

Most edges intersect at two points, creating either triangular or trapezoidal pieces. More complex intersection can be usually handled by representing area covered as a combination of simple fragments; i.e. more than one lookup is performed in the table and values are added or subtracted to yield the final grayscale value. Since an edge contained within a pixel affects surrounding pixels as well, lookup tables are computed for all the immediately surrounding pixels and grayscale values are summed.

Good results were obtained from this algorithm when processing graphics images. One serious weakness is the inability to accurately compute highly curved line segments; this algorithm would probably not be suitable for small text due to the tight curves which comprise most letters.

The two intersections shown (25 and 61) are used as addresses for a table lookup.

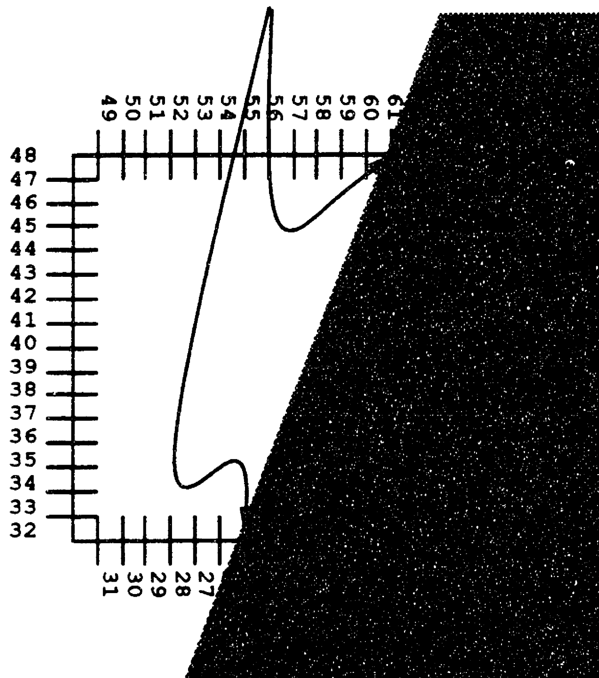


Figure 2-22: Table lookup of polygon edge. The polygon intersects the pixel at locations 25 and 61. These two values are used as the address for a table lookup to compute the grayscale value of this pixel and also surrounding pixels.

Chapter 3

Proposed Antialiasing Post-filter

There are several serious limitations with the methods of antialiasing discussed in the last chapter. All the current techniques (with the exception of supersampling) are pre-filters, so they operate solely on analytic or parametric curved outline data (not bitmaps or images). On the other hand, supersampling is computationally expensive, so it is not an attractive alternative. A more serious although perhaps less academic problem is that each method (including supersampling) requires changes to the scan conversion algorithm so that grayscale data can be generated. This is a very serious limitation, because current graphics packages cannot be easily adapted to produce antialiased bitmaps. For example, PostScript[1] would be very difficult to adapt to print grayscale bitmaps, because every scan conversion algorithm would have to be changed to an antialiased counterpart. So, a high level description of the desired antialiasing routine might be: "The desired algorithm makes use of current bitonal scan conversion routines, converting bitonal bitmaps to antialiased (grayscale) bitmaps with little computation."

In the following arguments, shades of black, white, and different grays will need to be discussed numerically. Black is assigned a value of 1.0, white is assigned a value of 0.0, and gray shades are real numbers in between black and white. For example, 0.25 denotes a shade of gray which is 25% black and 75% white.

[1] PostScript copyright Adobe Systems Inc.

3.1 Edge and Fill

Before designing such an algorithm, it is necessary to combine the different types of page data into common classes. Rather than having analytic data, parametric curved outlines, bitmaps, and images, we can classify all pictures as a combination of two components, *edge* and *fill*.[6] "Fill" is the graphical component used to shade areas, such as the solid parts of letters and halftones in images. "Edge" is used as transitions between different areas of fill.

3.1.1 Antialiasing edges

Antialiasing is simplified when using the graphical components, edge and fill. Filled regions no longer need to be considered at all when antialiasing. Only edges need to be filtered, and the antialiasing rules are fairly straightforward:

- A transition between black and white should occur at the spatial Nyquist rate, i.e. edges must must take a full pixel wide for black to white transitions. This rule is derived from the definition of the Nyquist rate; the maximum frequency allowable is equal to one half the sampling frequency. A transition from black to white and back to black corresponds to one period, so a transition from black to white corresponds to half a period. By the Nyquist criterion, a transition from black to white must occur over at least one sampling period. For less extreme transitions, such as between two similar shades of gray, the width of the edge can be scaled proportionally to the intensity difference. Analytically, the minimum width of an edge between two different shades of gray (G_A and G_B) is

$$W_{edge} = |G_B - G_A| \tag{3.1}$$

- Edges cannot be of length shorter than one pixel for a black/white transition. This is justified by the same reasoning as the previous width argument. Similarly, between shades of gray an edge can be shorter.

$$L_{edge} = |G_B - G_A| \tag{3.2}$$

- When edges cross, special methods must be used to antialias the region. Consider drawing lines i_1 and i_2, which overlap at (x, y). Many different approaches can be taken:

  - The first question is what it means to have two lines cross; is one line actually placed on top of the other, or are they considered equally? If i_1 is on top, then i_2 could be scan converted first and those pixels should be used as background values while converting the second line.

    $$I(x, y) = I_1 + I_2 + \alpha\beta i_2 \tag{3.3}$$

    where $\alpha\beta$ is a background blending function.[29] This unfortunately leads to the resulting intersection being overly bright. If instead black is used as the background when the second line is drawn, then the points around the intersection will not be bright enough.[12]

    $$I(x, y) = I_1 \tag{3.4}$$

  - When both lines are considered equally, things become more complex. Clearly, if the intensities are simply added, the result is too bright. Even worse, the result of adding bright lines may fall off the top of the intensity scale, so the result needs to be truncated.

    $$I_{combined} = trunc\left[I_1 + I_2\right] \tag{3.5}$$

  - Supersampling can be used to intersect both lines; the lines are drawn at very high resolution, and the supersampled intersection is simply converted to grayscale. This method is attractive because the intensities do not fall off the scale, and consistent results can be obtained; a drawback is the

computational expense. Supersampling only considers opaque lines.

$$I(x,y) = I_1 + I_2 - f_{overlap}(i_1, i_2) \tag{3.6}$$

- For ideal transparent line drawing, the energy is added linearly and intensities are added logarithmically. Adding intensities logarithmically might result in an intensity off the scale, so the resulting intensity must be truncated.

$$E_{combined} = trunc\Big[E_1 + E_2\Big] \tag{3.7}$$

$$I_{combined} = trunc\Big[log_B\Big[B^{I_1} + B^{I_2}\Big]\Big] \tag{3.8}$$

  This process is computationally expensive, so simpler methods of combining lines are normally used.

- A very simple but reasonably effective method is to take the maximum of the two intensities at every point when scan converting.[12]

3.1.2 Enhanced filling

Adding grayscale to a printing system can radically improve edges, and filled regions can also be enhanced by using grayscale. Gray fill is achieved on a bitonal printer through the process of halftoning; a cluster of black and white dots are duplicated over the filled region, causing the overall appearance to fall somewhere in between black and white, i.e. gray. By using grayscale pixels instead of only black and white, smaller clusters of dots are needed to achieve the same shade of gray. In a printer capable of producing n uniformly spaced shades of gray, the halftoning cluster size will be reduced by a factor of n over that of a bitonal printer.[2]

[2] This assumes that the printer is using a fixed cluster dither. In the case of a random dither such as Ulichney's blue noise dithering, there is no cluster size per se, but the consistency of the gray improves by adding more grayscale levels.

3.2 Antialiasing using cellular automaton

When considering analytic and parametric curved outline data (i.e. mathematically defined graphical data), it is readily apparent where edges and fill regions are located. Conversely, in a bitmap form, it is impossible to exactly locate edges and filled regions. Only approximations can be made, and sometimes a region will not be intelligible at all. However, if the sampling grid period is sufficiently small compared to the dimensions of the edges and filled regions, then the region can at least be recognized as an edge or fill, and the boundaries can be isolated to within one pixel.

Based on the assumption that most features on a bitmap are far larger than the sampling period, an antialiasing routine can be developed which converts a bitmap to a higher resolution format and then performs antialiasing by supersampling. The method is shown in diagram 3-1.

The drawback to this method is the huge memory requirement as the bitmap is stepped up to higher resolution. The goal is a grayscale bitmap *at the original resolution*, so it would be better if the grayscale could be added without increasing the resolution in the intermediate stages. Ultimately, antialiasing by unweighted supersampling simply counts the number of shaded subpixels within a pixel and assigns a grayscale value equal to the percentage of shaded subpixels. The positions of the pixels are not needed for unweighted supersampling, so a method can be devised which does not consider the positions of the pixel and does not step up the resolution. The rules for this procedure are given below:

3.2.1 Rules for PE's

- Only certain edge pixels are considered for modification; any pixel whose north, south, east, or west neighbors do not vary is not included in the smoothing.

- Any pixel which starts out white cannot become less than 50% white. Likewise, any pixel which starts out black cannot become less than 50% black.

- Each modified pixel assumes a grayscale value computed as the weighted aver-

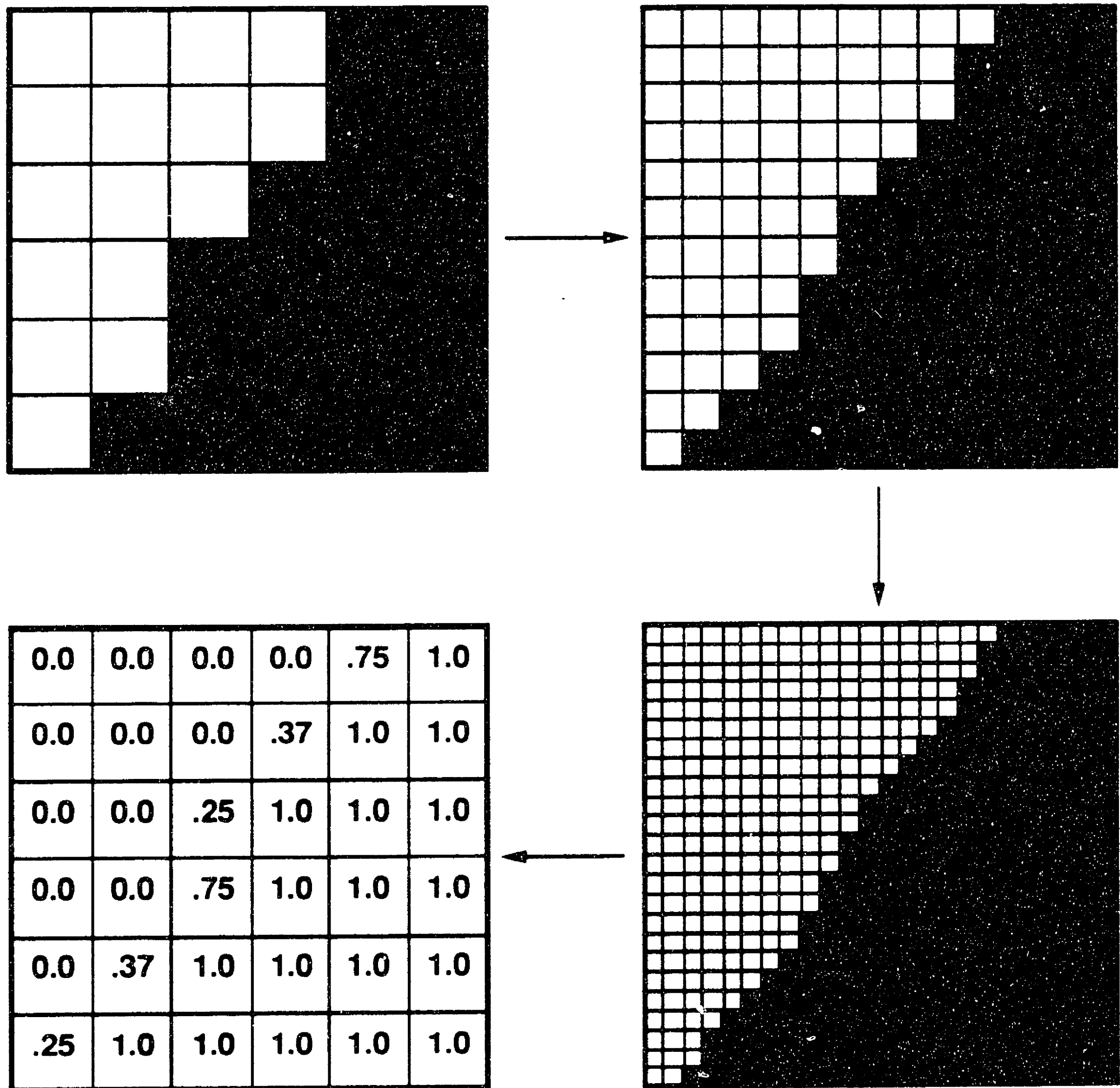


Figure 3-1: Cellular Automaton Antialiasing. A bitmap is converted to higher resolution using cellular automata and then converted to a grayscale bitmap using the supersampling methods.

age value of the eight surrounding pixels (north, south, east, west, northeast, northwest, southeast, and southwest). The pixels are weighted as the inverse of the distance to the center pixel.

- All pixels are updated with lockstep synchronization.
- Multiple iterations of this routine can be performed on a bitmap.

3.2.2 Results and problems

As shown in the photographs in Appendix B, the cellular automata method performs reasonably well at smoothing text and graphics. For a parallel processor such as the Connection Machine, the routine would be highly efficient; each pixel could be assigned to a virtual processor, and the NEWS grid could be effectively used because of the routine's data locality - each pixel must look only at the pixels surrounding it. However, for a conventional sequential processor, this method is very computationally inefficient; a processor needs to make decisions on each pixel in sequence, and worse yet do this for several iterations.

3.3 Windowed ROM based algorithm

To make a more efficient method of antialiasing, it is important to not have to have a processor perform a computation on every pixel.

Using the cellular automata method of antialiasing, a processor must modify every pixel several times before the result is available. There are several problems to be addressed when designing a more efficient algorithm. Ideally, a processor should only be used in the pre-computation instead of performing elaborate computations on every pixel. The routine should be "one-pass"; results should be available immediately. Perhaps most importantly, the routine should be pipelined. Results should be computed as the bitmap is fed to the routine, rather than storing the entire bitmap and then computing results. A minimal buffer can be used to allow some storage.

A lookup table meets the efficiency requirements; if each pixel is computed with a

simple lookup table, then antialiasing becomes fast enough to include in even low cost output devices. The lookup table is trained to map from a special bitonal bitmap to its corresponding antialiased version. Ideally, after it is programmed, the lookup table can perform a mapping from arbitrary bitonal bitmaps to produce their antialiased counterparts.

This method is very similar to *vector quantization.* Vector quantization is a method of data compression where blocks of information are coded as symbols rather than individual elements.[18] Vector quantization is noisy; the result of compressing and decompressing and image is not guaranteed or even expected to be exactly the same as the original image. However, if the correct symbols, or "code vectors" are picked, the result will be similar to the original.[2]

Drawing an analogy to the windowed ROM algorithm, the code vectors correspond to the list of all possible window patterns. One major difference between the windowed ROM algorithm and vector quantization is that in vector quantization, the code vectors are independent of each other. In the windowed ROM algorithm, window mappings overlap so the window patterns are not independent in the same sense. Although the objective is different, the two classes of algorithms (vector quantization and windowed antialiasing) are somewhat similar.

3.3.1 Description of windowed ROM algorithm

- Each window pattern provides an address to the ROM. (See figures 3-2 and 3-3.)

- Each window pattern corresponds to a grayscale value, which is output from the ROM.

- If there is more than one distinct grayscale value corresponding to one window pattern, this is called a conflict. The grayscale values are averaged and the standard deviation is kept. A high overall standard deviation indicated that the window is too small or shaped incorrectly, and there are not enough bits in the window to discriminate between two fundamentally distinct antialiasing

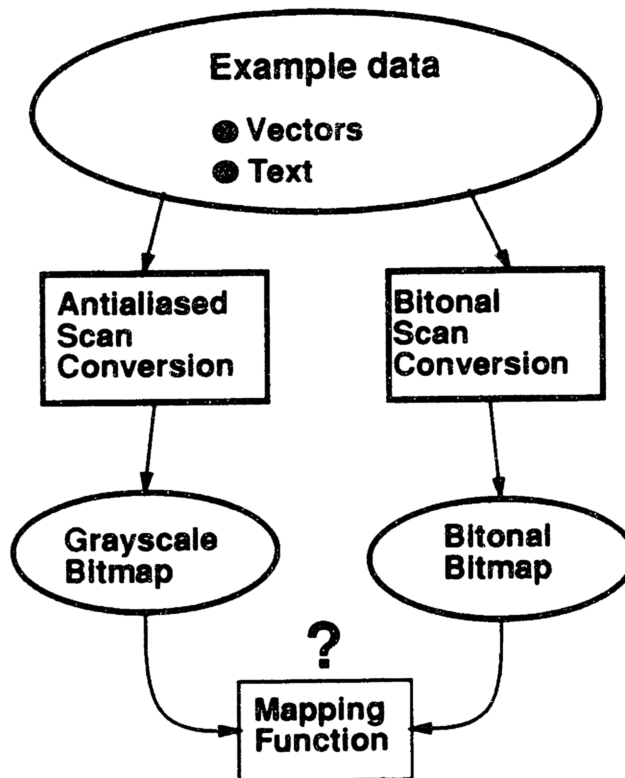


Figure 3-2: Creating the antialiasing ROM. The antialiasing ROM provides a mapping from the bitonal bitmap to the grayscale (antialiased) bitmap; the address of the ROM is computed from a window in the bitonal bitmap, and the values contained in the ROM represent grayscale.

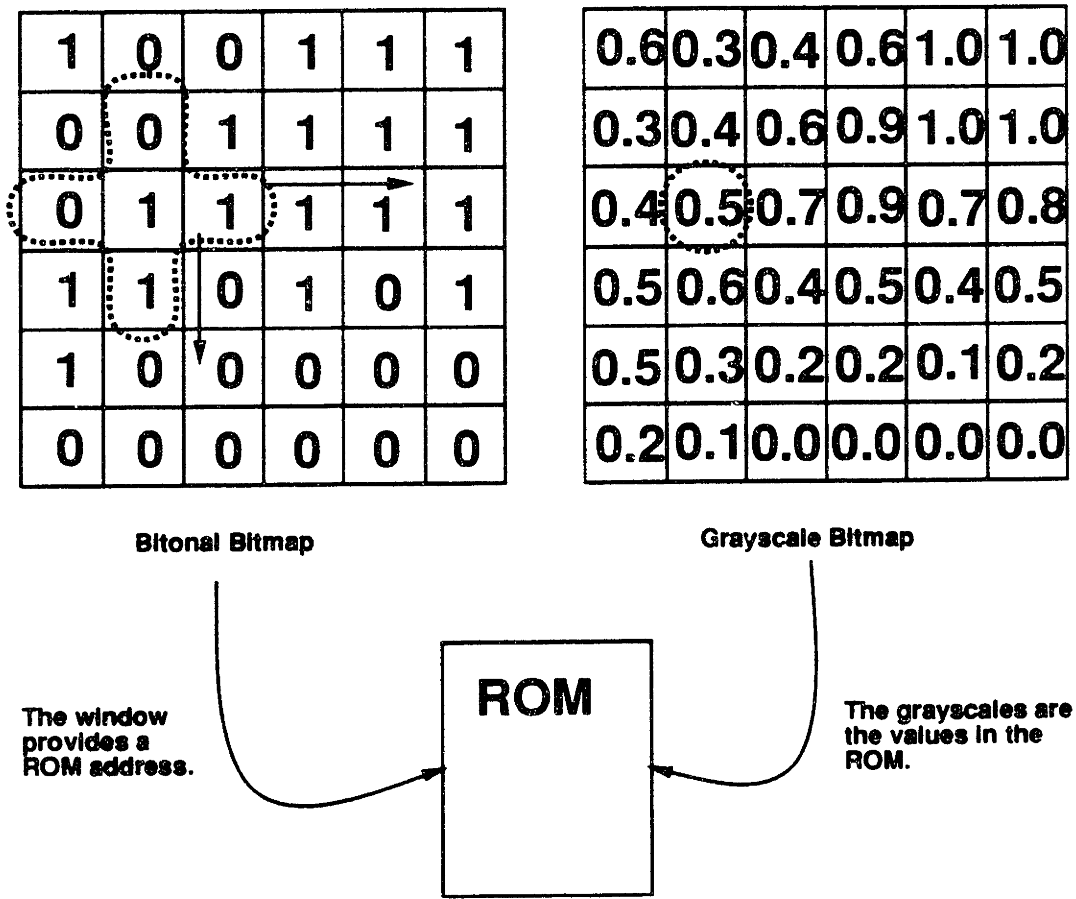


Figure 3-3: Creating the antialiasing ROM.

cases.

- A set of typical data (the *learning pattern*) is used to create the ROM.

- After performing the ROM learning process on a test suite of data, some bit patterns may not be associated with a grayscale level. Their grayscale values must be extrapolated from similar bit patterns which do have grayscale data. Each pixel in the window is weighted as the inverse of its distance from the center of the window. If a window pattern does not have a corresponding gray scale value, similar window patterns (as defined by the weighting function) are searched until a grayscale value is found and copied to the first window.

- After the entire table is complete, the ROM can be used to antialias arbitrary bitonal bitmaps. With the same window pattern as was used to create the ROM, map the window across the bitonal bitmap. At every window position, the value contained in the ROM is used as the antialiased (grayscale) pixel value. (See figure 3-4)

3.3.2 Window selection

Selecting the proper size and shape window is critical to the ROM antialiasing routine. The pixels selected in each window should be relevant to the center pixel in the antialiasing process. Note that the center pixel does not necessarily lie in the center of the window; in the case of a 2×2 window, there is no center pixel of the window, so one of the pixels must arbitrarily be selected as the centered pixel during the mapping. To avoid confusion, the centered pixel will be referred to as $(0,0)$, and other window pixels will be referred to relative to $(0,0)$, i.e. the closest northeast pixel from $(0,0)$ is $(1,1)$ and the closest northwest pixel is $(-1,1)$.

The following discussion does not consider dithered images. In a bitonal system all objects are drawn either white or black; there are no gray objects. When such an image is antialiased, grayscale is introduced only near edges, not in previously solid white or solid black regions. As in the cellular automaton method, I make the

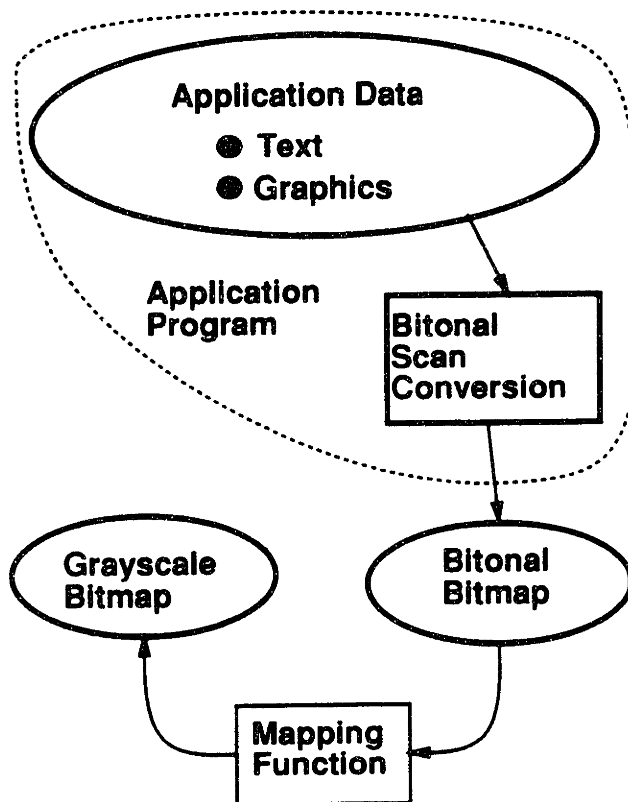


Figure 3-4: Antialiasing using ROM. Generic bitonal bitmaps (from commercial products) are antialiased by passing them through the ROM.

assumption that any pixel which is on in the bitonal bitmap should not be less than 50% in the antialiased bitmap, and any pixel which is off in the bitonal bitmap should not be more than 50% in the antialiased bitmap.

It is useful to figure out how much contribution each surrounding pixel makes to the center pixel. There are always pathological cases of nearly vertical or horizontal edges which must be guessed, unless it can be guaranteed that the window is larger than the edge length. We are concerned with windows of less than 21 positions[3], and it is obvious that edges are often much longer than 5 or 10 pixels, so we can expect to encounter many such cases. In the event of a nearly vertical or horizontal line, the maximum error is 50%. (See figure 3-5.)

To develop a metric for determining how important each pixel is, the following method can be used. To compute the importance of $pixel(n, n)$:

- Integrate over all possible infinite length edges the error which occurs from not including $pixel(n, n)$ in the window. The error is computed by comparing a perfectly antialiased bitmap to the test bitmap.
- Repeat the integration assuming that $pixel(n, n)$ *is* included in the window, and subtract.
- Since only edges affect antialiasing, and the edge width is related to the spatial Nyquist rate, then only the edges which pass within one pixel of $pixel(n, n)$ **and** $pixel(0, 0)$ should be considered.
- Certain direction edges are more frequent than others; for example, most fonts have many horizontal and vertical strokes. When computing the importance of a window pixel, this distribution should also be factored in; lines should be picked according to their probability in real text.

By now, this method a selecting a window is too complicated to implement. A better idea is to ignore the original vector data and antialiased bitmap entirely; concentrate only on the bitonal bitmap. To compute the importance of $pixel(n, n)$:

[3] 21 positions leads to a ROM size of $2^{21} = 2\ MBYTE$

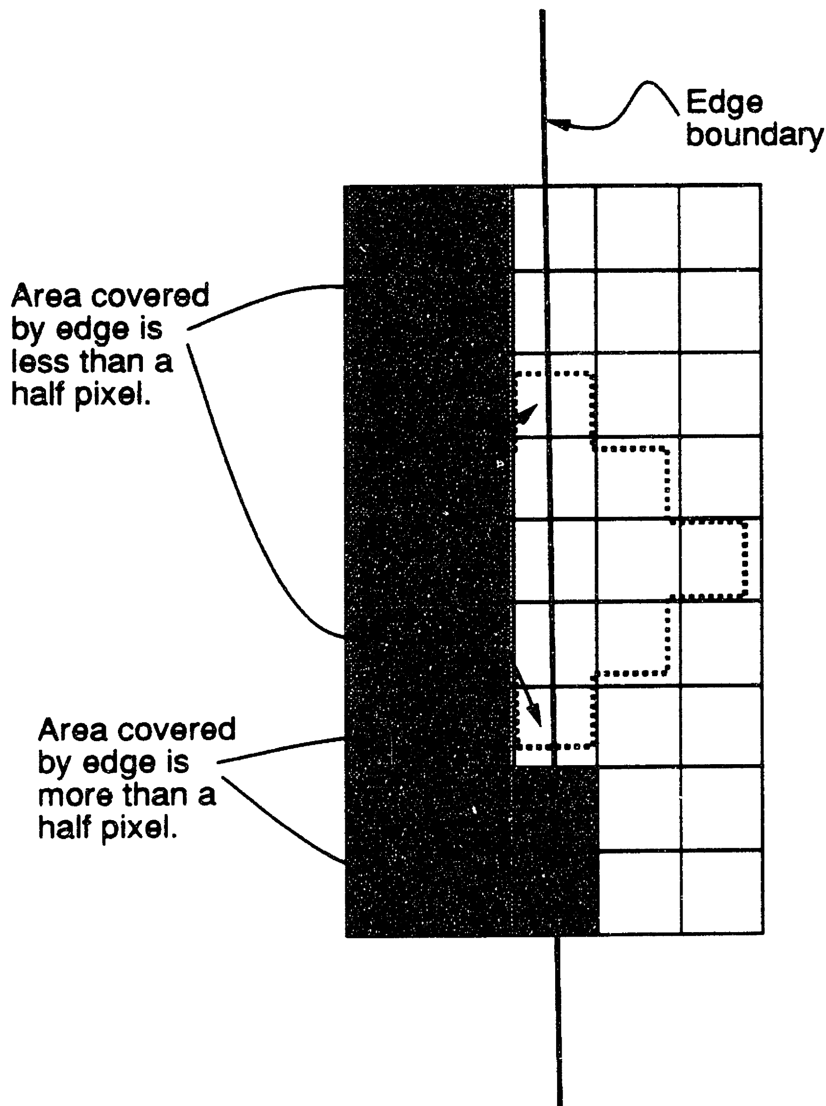


Figure 3-5: Selecting a window. A nearly vertical line is the worst case for an antialiasing ROM. This edge could go from $(0, -\infty)$ to $(-1, \infty)$ which would mean that all the intermediate pixels (i.e. the ones shown mapped in the window) would have grayscale values of about 0.5.

- Take a typical bitonal bitmap and perform the windowed antialiasing without including $pixel(n, n)$ in the window. Take a Fourier transform and compute the error as the weighted magnitude of the all frequency components over the Nyquist frequency.

- Repeat the computation assuming that $pixel(n, n)$ *is* included in the window, and subtract.

Note that this routine does not guarantee accuracy of results; the transform is not compared against the transform of the original data. As a result, the antialiasing method should *not* be optimized to reduce error with a given window; this would probably degrade the accuracy of the routine. Instead, it offers a quick way to check the importance of particular window bits in removing high frequency components.

The size of the ROM is simply a physical design tradeoff, so larger ROMs yield better antialiasing, particularly with nearly vertical or nearly horizontal lines. In experiments (see Appendix B), good results were achieved with a ROM size of 13×8 or 17×8 bits.

3.3.3 Selection of learning data

The composition of the learning data is an important part of the ROM antialiasing routine; it should be similar to the data which is later converted. Theoretically, only *edge* and *fill* need be considered as graphical features, but subtleties are introduced when they are combined and overlapping. The following complex graphical components should be considered when designing a learning pattern.

- **Lines:** Lines are combinations of two edges a fixed distance apart. For example, a one pixel wide line is two opposite facing edges spaced one pixel apart. For a large line widths (i.e. greater than several pixels) there is no interaction between the two edges, and the edges can be considered separately. For small line widths, the edges interfere and the center point will not reach full intensity. Since the edges can no longer be separated, narrow lines are pseudo-graphical components.

Lines can be drawn at arbitrary angles, and this should be reflected in the learning data by including many lines at various angles. The length of each line should be several times the window width to allow the ROM to learn different sub-pixel positionings of the line.

- **Edges:** Edges should be included in the learning data at various angles. The length of each edge should be several times the window width to allow the ROM to learn sub-pixel positioning.

- **Curves:** Curves can be added to the set of learning data, but usually only small radius curves are valuable because large radius curves appear as lines to the window. A useful way to implement curves in learning data is to include several small solid circles.

- **Intersections:** Intersections are important to consider. It is beneficial to include a region of random overlap of lines, edges and curves in the training data. Care must be taken that the overlapped regions are scan-converted correctly as discussed in section 3.1.1.

- **Text:** Small fonts of text should be included in the learning data. Large fonts are comprised of edges, curves, and intersections and do not need to be mapped again, although there is nothing wrong with including them as well.

- **Dithering:** Regular dithering can be converted to grayscale if the window is large enough. By including sections of dithering in the learning bitmap and providing a sophisticated scan-conversion routine, the ROM can recognize dithered areas and convert them to grayscaled areas.

Chapter 4

Laser diode modulation methods to support antialiasing

The **laser modulation circuit** provides an digital interface to the diode laser in the printer. The motivation for designing this circuit stems from the difficulty of controlling the laser diode under a varying environment - the laser diode is electrically a very sensitive device, which is easily destroyed by voltage spikes and overheating. In addition, its transfer curve changes radically depending on the junction temperature of the device, which means that constant re-adjustment is necessary.

4.1 Hardware constraints

Ideally, the grayscale bitmap data originating from an application program would pass through the laser modulation circuitry and be transformed into uniform pixels of varying shades of gray on the page. There are several difficulties which must be addressed with the laser modulation circuitry.

- **Gaussian laser beam:** The laser beam which strikes the OPC (photoconductive) drum is Gaussian in distribution, and does not necessarily fall totally within one pixel. This presents two problems; the center of the pixel receives more energy than the sides, and the sides overlap with neighboring pixels. The net result is that it is impossible to make a square pixel so pixels overlap.

- **Shades of gray:** Toner is black, so the only way to generate shades of gray is to partially fill a pixel with toner. For example, a gray value of 0.50 would require half of the pixel area to be filled with toner.

- **Information bandwidth:** Although it is necessary to modulate the laser beam within a pixel, it is important to consider the information bandwidth going to the laser. For example, to arbitrarily modulate eight subpixels within each pixel, eight bits per pixel are needed. It is better to figure out only the subpixel patterns which are needed, and encode each pixel with a pattern number.

4.2 Subpixel patterns

Two different classes of pixel patterns should be implemented: *centered pulses*, and *edge pulses*. Centered pulses (shown in figure 4-1) are used for general grayscaling. A centered pulse will produce a scan-direction symmetric energy distribution within the pixel. If the proper power is applied to the laser during the on time of the pulse, the sub-scan energy distribution can be matched to the scan-direction energy distribution, and an x and y symmetric subpixel dot can be produced (i.e. a rounded square dot). By using different energy levels and different pulse widths, various size dots can be constructed which correspond to various shades of gray.

One other type of subpixel pattern should be implemented, the edge pulses. Antialiasing increases the effective resolution, but does not actually change the spacing of the pixels. However, by using edge pulses, the real resolution can actually be increased in the scan direction. Another way of stating this is that antialiased grayscaling increases the information placed on the page, but does not change the positioning of pixels. By using edge pulses, sub pixel positioning is possible.

Although the antialiasing routines in chapter 3 do not necessarily make use of this feature, it is fairly easy to adapt the ROM based routine to recognize scan-direction edges and use pulse positioning (as well as antialiasing) to improve the quality.

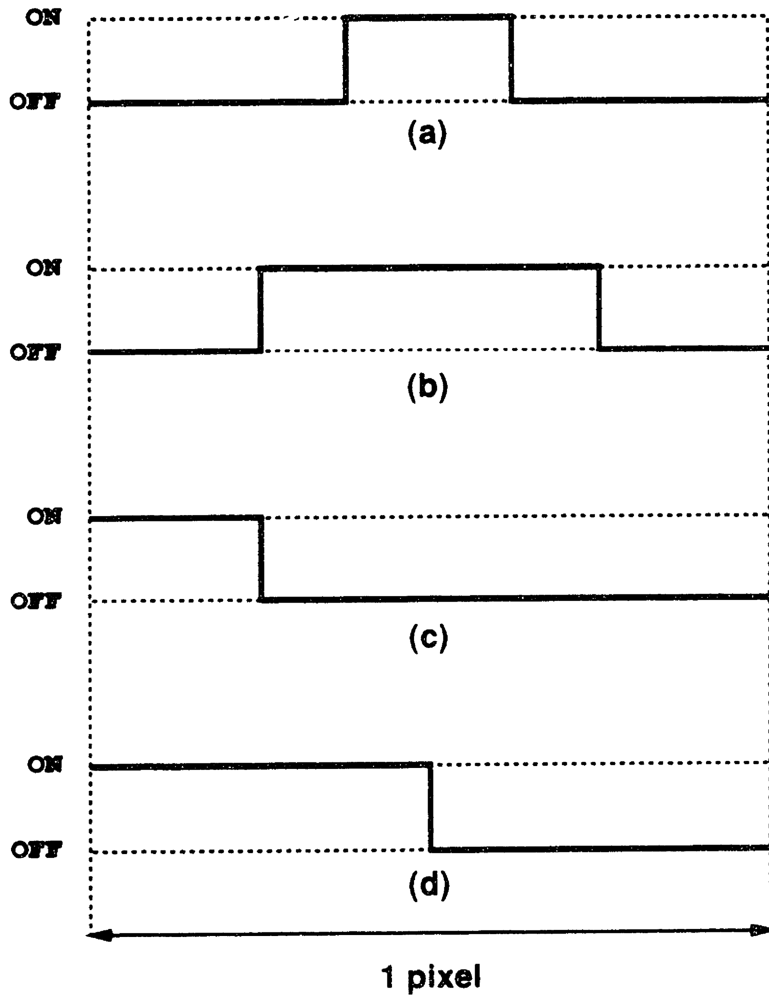


Figure 4-1: Example laser modulation pulse patterns. (a) A centered pulse (b) A wider centered pulse (c) An edge pulse from the left edge (d) A wider edge pulse

4.3 Selected encoding scheme

To experiment with the effects of different modulation schemes on a printer, the laser modulator was designed with power modulation, pulse width modulation, and pulse position modulation. For each bit, an eight bit encoded control word is passed to the modulator board, consisting of the following fields.

- $E[2:0]$ is laser power for the pixel. $E = 0b111$ corresponds to full power, and $E = 0b000$ corresponds to one eighth of full power.[1]
- $P[2:0]$ is the encoded pulse pattern for the pixel. The pulse patterns are shown in figure 4-2.
- I is the inverse bit for the pixel. When $I = 1$, then the pulse pattern is flipped, i.e. at every sub-pixel the laser is turned off instead of on, or on instead of off.
- M is the mode bit for the pixel, which indicated if the pulse pattern is a centered pulse or an edge pulse. $M = 1$ corresponds to centered pulses, and $M = 0$ corresponds to edge pulses.

4.4 Simulation of laser printer modulator

Before implementing the laser printer modulator in hardware, the modulation encoding was tested with software to insure that the encoding was robust enough to smooth jagged pixel edges. A laser printer simulator was developed to graphically display the results of various modulation techniques. OPC drum energy is displayed in color, with lighter colors representing higher drum energies. Toner distribution is displayed in black and white, and is statistically approximated at transition zones.

Equations 2.22 and 2.23 are used to compute the OPC drum energy distribution and the toner distribution. Values for E_o, X_1 and X_2 are derived from decoded

[1]The laser is fully turned off by using the correct pulse modulation signal, not by the power signal.

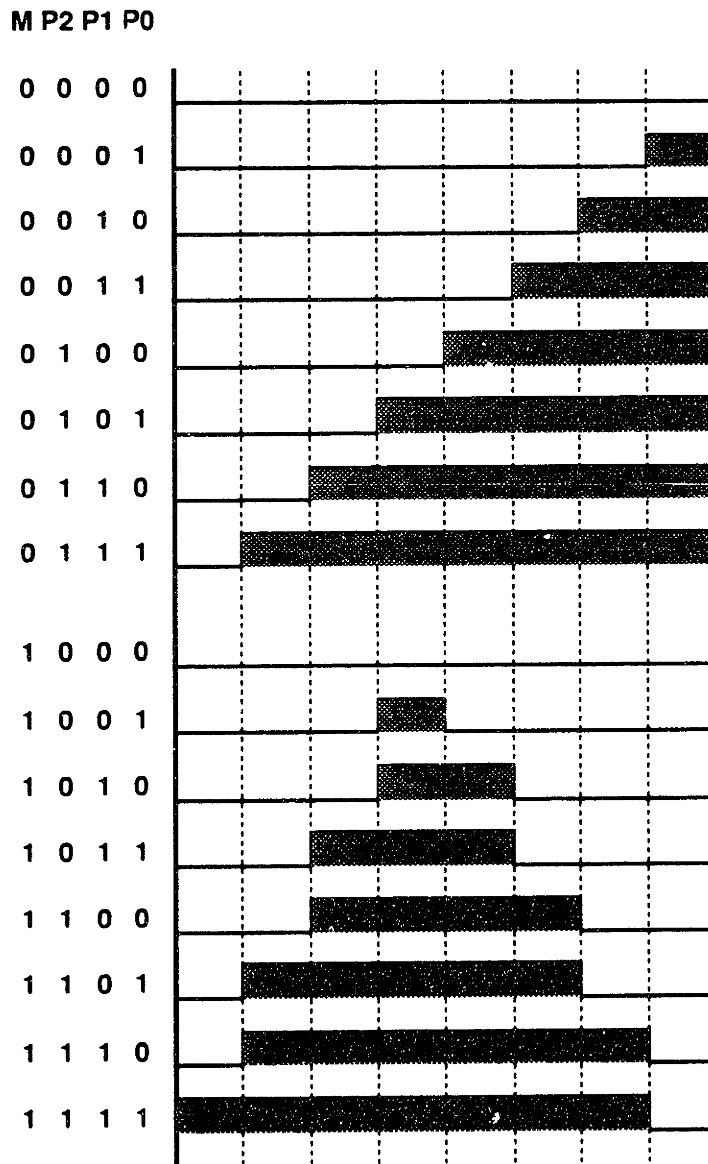
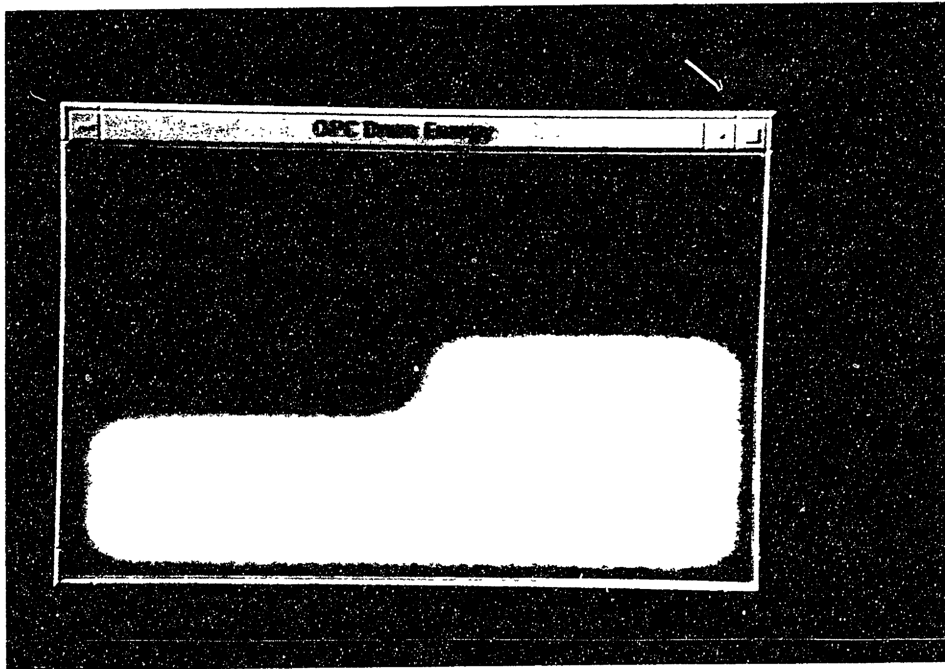
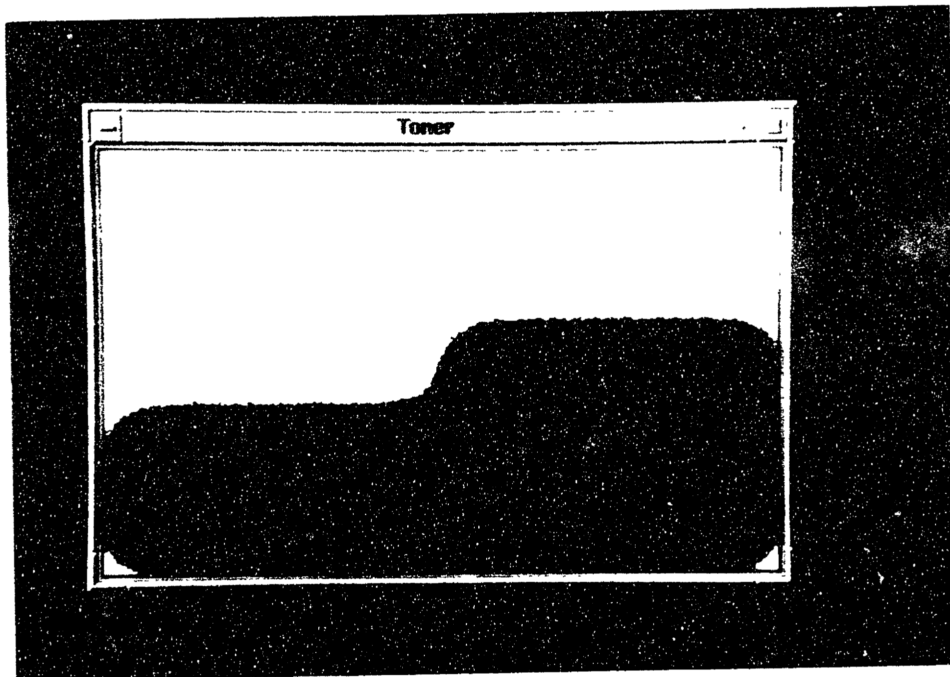
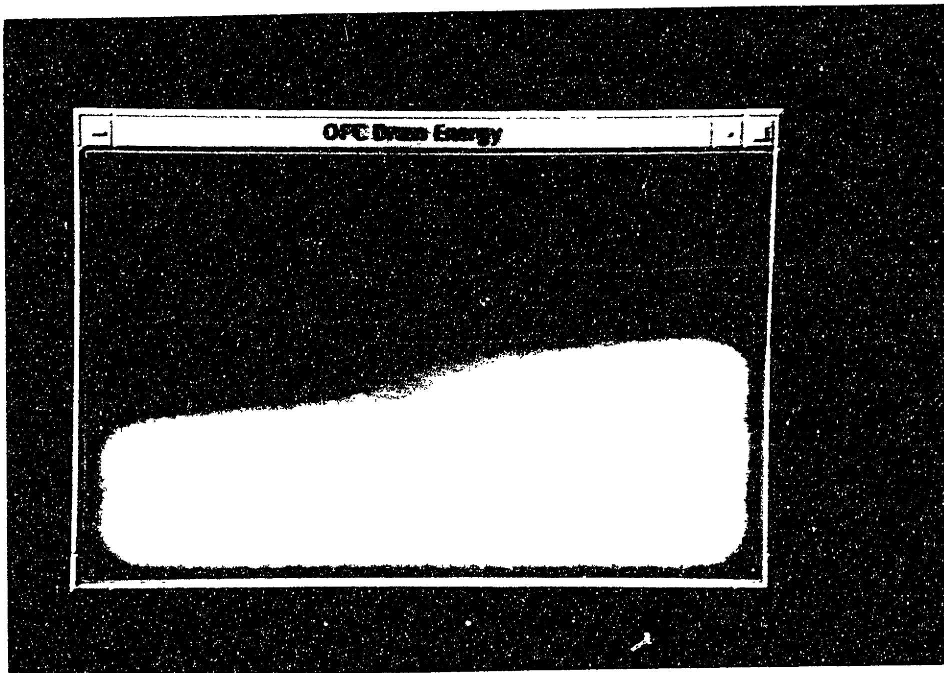


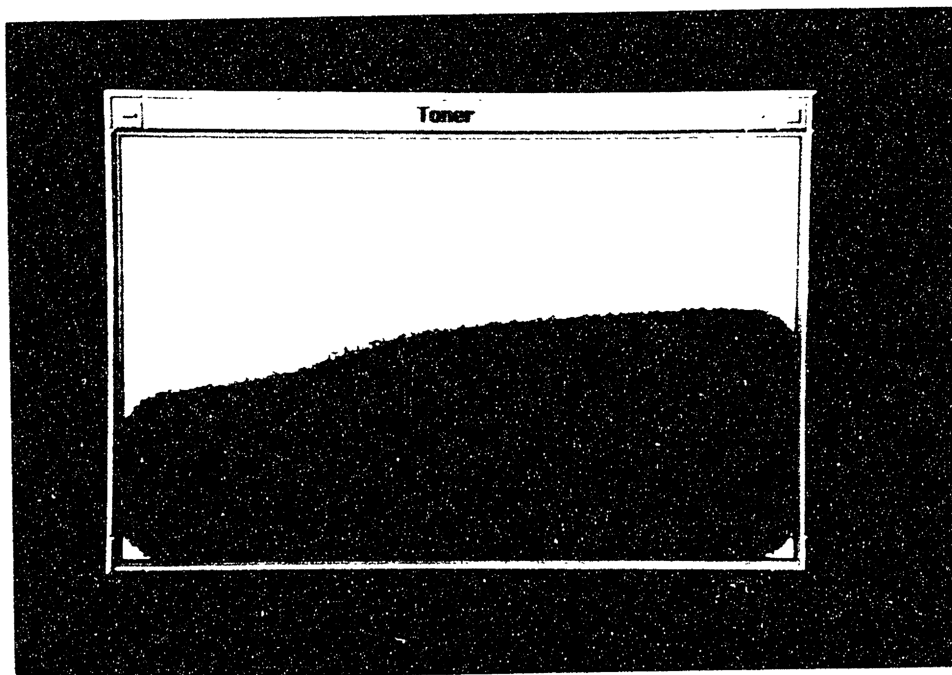
Figure 4-2: Pulse modulation patterns for laser modulator.

modulation data. K_1 and K_2 are approximated from the laser energy distribution curves. The final unknown variable, the density function $D[e]$ is approximated by comparing a standard bitonal modulation simulation with actual printed results.


OPC Drum Energy


Toner


OPC Drum Energy


Toner

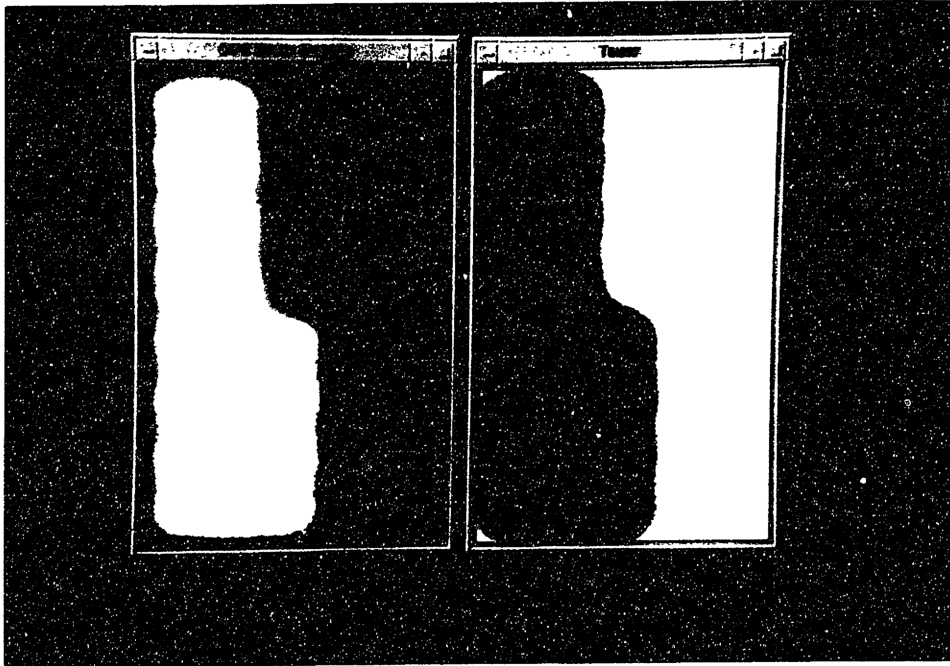


Figure 4-7: Laser printer simulator, OPC drum energy and toner, horizontal step, conventional modulation

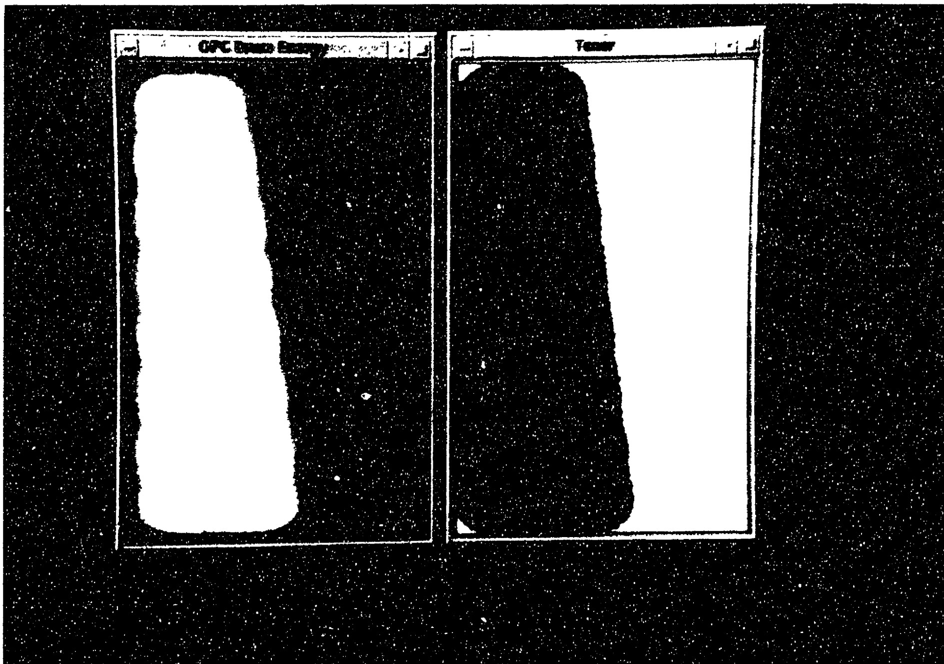


Figure 4-8: Laser printer simulator, OPC drum energy and toner, horizontal step, pulse edge modulation. Note that edge is still sharp.

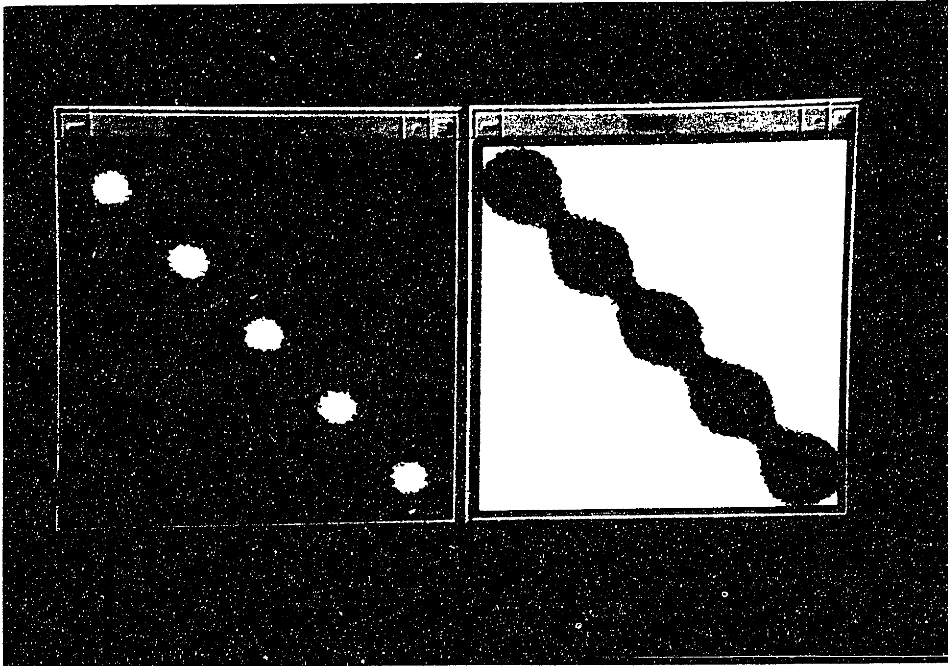


Figure 4-9: Laser printer simulator, OPC drum energy and toner, 1 pixel wide 45 degree line, conventional modulation

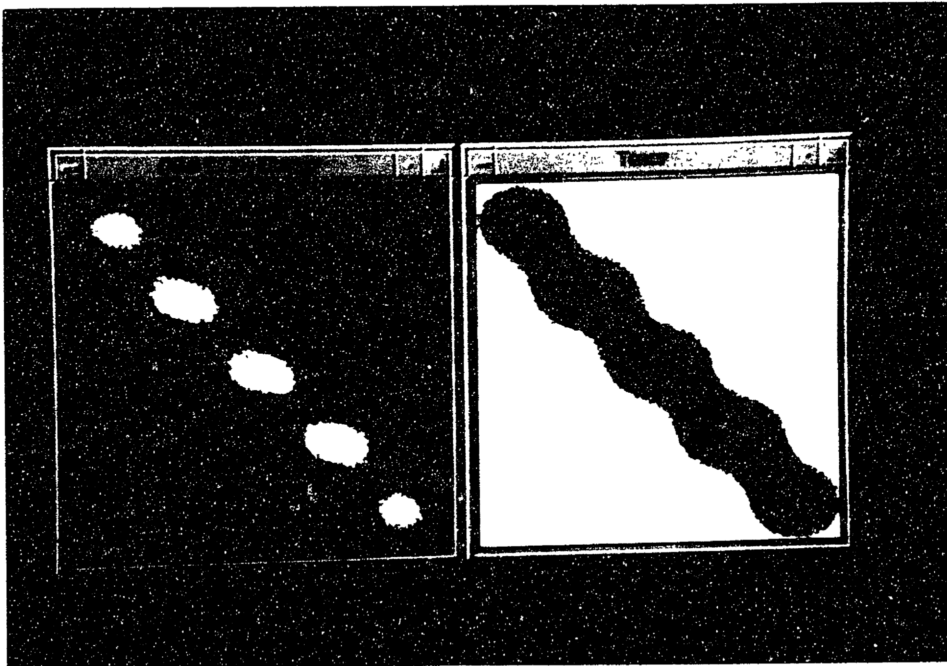


Figure 4-10: Laser printer simulator, OPC drum energy and toner, 1 pixel wide 45 degree line, power and pulse edge modulation

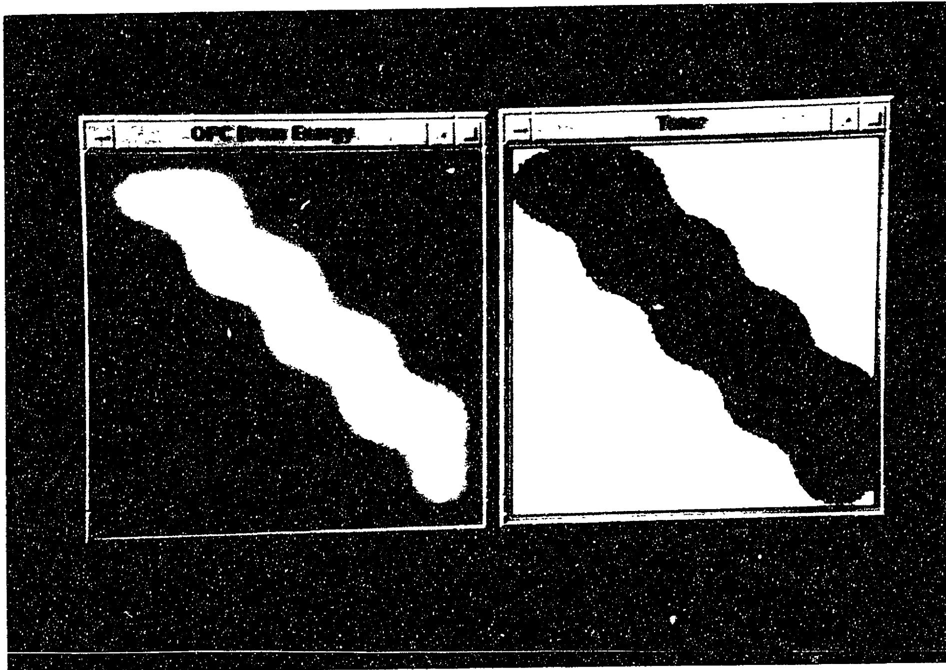


Figure 4.11 Laser printer simulator, OPC drum energy and toner, 2 pixels wide 45 degree line, conventional modulation

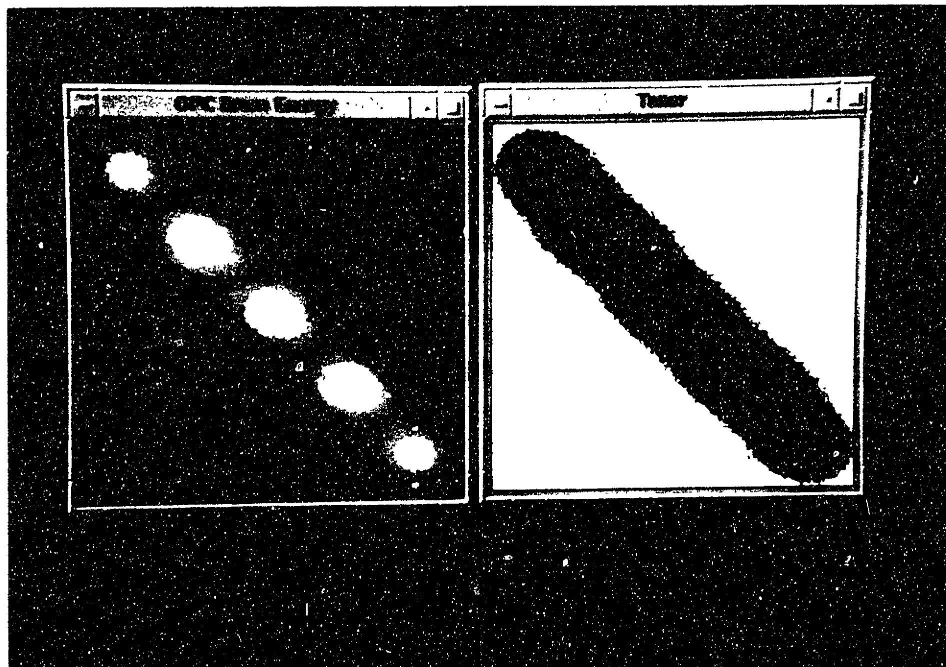


Figure 4.12 Laser printer simulator, OPC drum energy and toner, 2 pixels wide 45 degree line, power and pulse edge modulation. Note that the edge is smooth, because adjacent pixels add to produce a straight line along the toner threshold

4.5 Design methodology and circuit description

The laser diode is a current driven device; it turns on when it is forward biased at between 1.2 and 1.8 Volts. For 5 mW operation approximately 80 mA of forward current is needed, and the output is roughly proportional to the current. All of these parameters can vary significantly depending on the temperature of the laser diode junction, which in turn changes during normal operation. A monitoring photodiode is included in the package, which is used in reverse bias configuration. The photodiode operates by monitoring the laser light output and letting through a proportional amount of back-current, roughly 1 mA for 3 mW of laser output power.

One can easily construct an analog feedback loop to turn the laser diode on and maintain a steady output power. However, the laser printer application places some additional constraints on the laser modulation subsystem. When using the laser in a printer, the laser must be able to be switched at a high frequency (to produce arbitrary patterns on the page), not just turned on for long periods of time. To accomplish this modulation, the feedback loop must be interrupted at the same frequency to prevent it from compensating when the laser is off. Also, since the laser diode is electrically fragile, it is a necessary characteristic of the feedback loop that it be overdamped; otherwise, the overshoot could damage the laser diode.

The laser is digitally modulated at speeds of 20 MHz or more. The laser should be capable of being switched at least three times faster than this to insure sharp edges; this implies a control loop bandwidth of 60 MHz. Keeping in mind that a slightly overdamped system is needed, the analog voltage must be sampled at a rate of over 120 MHz. Even if accurate amplification were available at that bandwidth, the sample and hold device necessary to switch on and off the feedback loop is not available in current technology at this speed. In addition, it is difficult to modulate the laser at lower power levels without the feedback causing problems. A different solution is needed.

The laser modulation signal can be broken into three basic signals of various speeds. The three signals are combined to make the modulation signal at the last

stage. There needs to be a bias signal which is derived from the feedback loop. This signal sets the operation point of the laser - it indicates how much power to send to the laser when it is fully on. Second, there is a power signal which determines the percentage of power at which the laser is operating, i.e. a grayscale level. With these two signals alone grayscaling could be accomplished, but a third signal - a modulation signal - adds generality and improves the capability of the circuit. The modulation signal is the fastest of the three control signals, and simply turns on or off the laser at the specified power level and bias.

Now that the component signals are identified, bandwidths must be considered. The pixel rate of the modulator is 5 MHz, and the sub-pixel rate is 20 MHz.[2] To generate the sub-pixel clock, a phase lock loop is used to multiply the basic 5 MHz signal. (See figure 4-15.)

The modulation signal clearly must be capable of switching at the highest rate, 20 MHz. Since the modulation signal is a one bit digital signal, there is no difficulty. The power level signal can be slower if resolution is sacrificed, i.e. if one power level is used over all the sub-pixels of a pixel. In this implementation the power signal changes at 5 MHz. It is important to turn off the feedback loop when the laser is off and also when the laser is operating at low power levels because the feedback loop is only calibrated to a 5 mW output (laser fully on). Even though the feedback loop must be turned on and off relatively quickly, it does not need to respond quickly when it is on; its purpose is to respond to thermal fluctuations, which are very low bandwidth changes. Therefore, the choice for the feedback loop is a limited slew rate digital feedback loop.

The principle behind the limited slew rate digital feedback loop is that the "operating point" of the feedback loop is held in a digital register. Depending on the feedback, the operating point is either increased or decreased a fixed amount, after which it is converted to an analog signal through a digital to analog converter. Most of the components of this loop (such as the D-A converter) are allowed to be slow

[2] As shown, the modulation PAL (PAL2) generates 8 sub-pixels per pixel, which implies a 40 MHz modulation rate. However, the circuit was built and tested at 20 MHz before trying to increase to 40 MHz.

because the amount of change to the operating point is small, and instability of the feedback loop due to slow components is bounded by the magnitude of the changes to the operating point. In other words, the feedback loop will be unstable, but the amount of instability will be relatively small and it will occur only within a valid output range.

The feedback circuit is implemented with three cascaded LS169 up/down counters (E3, E4, E5) to hold the operating point of the feedback loop. (See figure 4-13.) Under the control of a PAL, they can be enabled to add one or subtract one from the operating point every cycle. Together, there are 12 bits of resolution in the operating point. The output of the counters is fed into a 12 bit D-A converter (E6). The output of the D-A converter is fed into an op-amp follower (E7) to provide isolation. The resulting signal, **LaserBias**, is a voltage which indicates the bias level of the laser diode.

The feedback portion of the loop starts with the photodiode. The voltage at the base of the photodiode is compared to a variable set voltage using an open loop op-amp (E1). (The set voltage determines the operating point of the laser. The operating point can be adjusted by a variable resistor.) The op-amp saturates positively or negatively depending on whether the photodiode voltage is higher or lower than the set voltage. The output of the op-amp is fed into a line receiver (E2) to convert it to a TTL signal. The TTL signal simply indicates "high" or "low", and is fed into the control PAL (PAL1) for the feedback loop. PAL1 generates enable and direction signals for the LS169's (E3, E4, E5) based on the value of the feedback signal. The PAL must also be able to switch off the feedback loop (i.e. signal the LS169's to hold the current value without changing) depending on the video data being sent to the laser. Therefore, the **I, Power[2:0]**, and *Pattern[2:0]* signals are gated together and the resulting signal is sent to PAL1 as well to tell it when the laser is at full power.

After the laser diode bias is determined, it is dropped across a voltage divider of eight resistors to scale it for the eight different power levels. Each of these power levels has the laser bias incorporated within it; when the bias changes, so do all the power levels. The eight power levels are sent to an analog multiplexor (E8), which can

select one of the voltages based on the signal Power[2:0], and pass it to the output. Since the analog MUX has a finite resistance and limited current capacity, an op-amp follower (E9) is used for isolation. The op-amp follower has a diode forward biased in its feedback path, which adds an offset 0.7 Volts to the output. (This is of some use for driving the transistor (Q1) linearly, since the base emitter voltage drop is approximately 0.7 Volts).

The modulation transistor (Q1) is connected in a common emitter fashion to drive the laser (in the collector). (See figure 4-14.) It acts as roughly linear voltage controlled current source. The gain is controlled by the 6K resistor in series at the base. The transistor must be capable of drawing a collector current of about 150 mA, switching smoothly at speeds of more than 20 MHz, and sinking at least 500 mW of power.

Finally, the modulation circuitry (which must be very fast) is added. The analog MUX and the op-amp are too slow to support the bandwidth of the modulation signal, so the modulation signal is added after the power signal is generated in E9. Transistor Q2 is used in a common emitter setup, with the base being driven by the modulation signal. When Q2 is switched on, the variable resistor in the base of Q2 should be adjusted so that Q2 does not saturate but does draw enough current to switch off Q1, thereby shutting off the laser. When Q2 is in shut off, Q1 is unaffected and the laser operates as usual. The modulation signal used to drive Q2 is generated from the encoded bit data by the modulation PAL (PAL2).

The result is a circuit which is capable of digitally modulating a diode laser at speeds of up to 20 MHz while simultaneously power modulation the diode laser at speeds of up to 5 MHz. The circuit uses the feedback of the laser diode to correctly set the operating point of the laser and modulate only in the correct range.

4.6 Laser Modulation Circuit Schematics

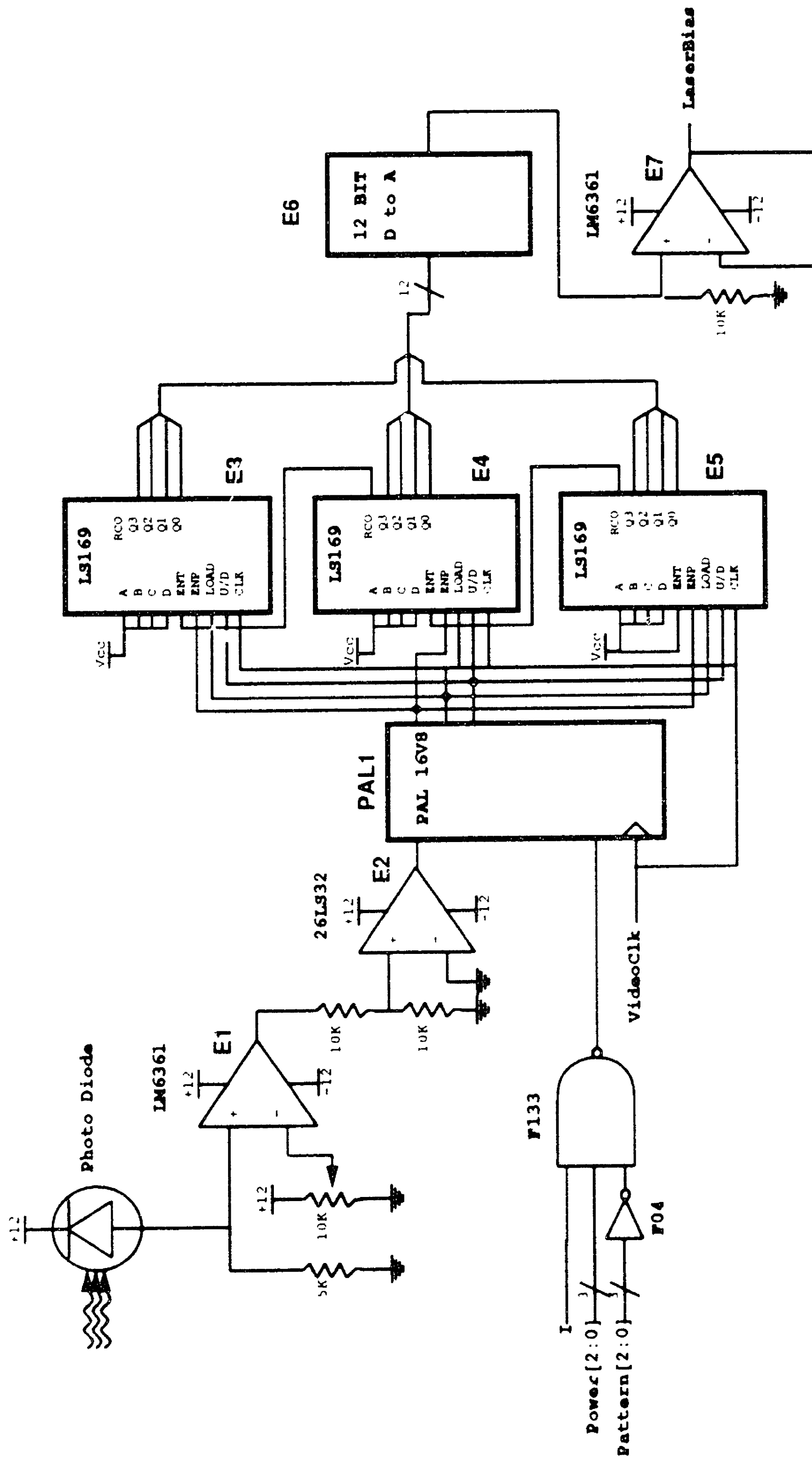


Figure 4-13: Laser Modulation Circuit

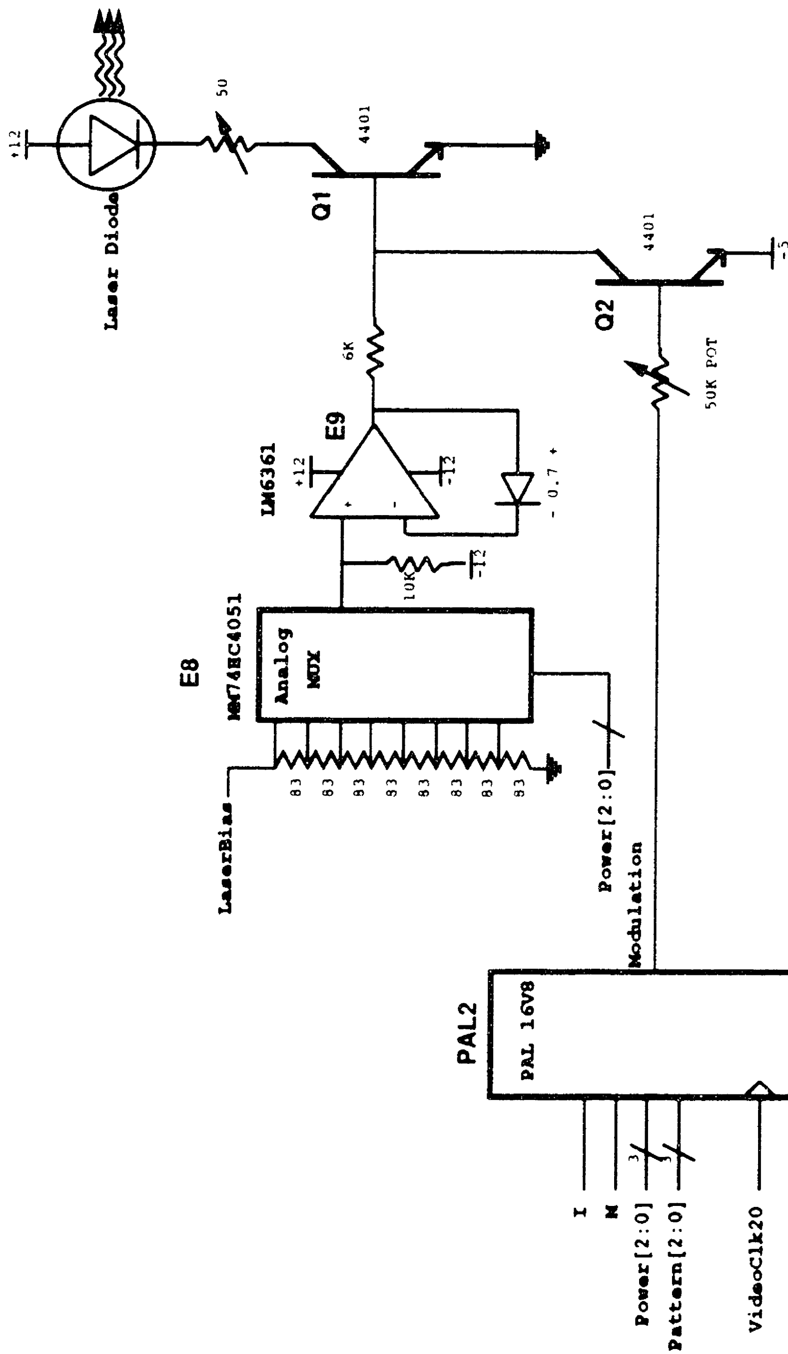


Figure 4-14: Laser Modulation Circuit

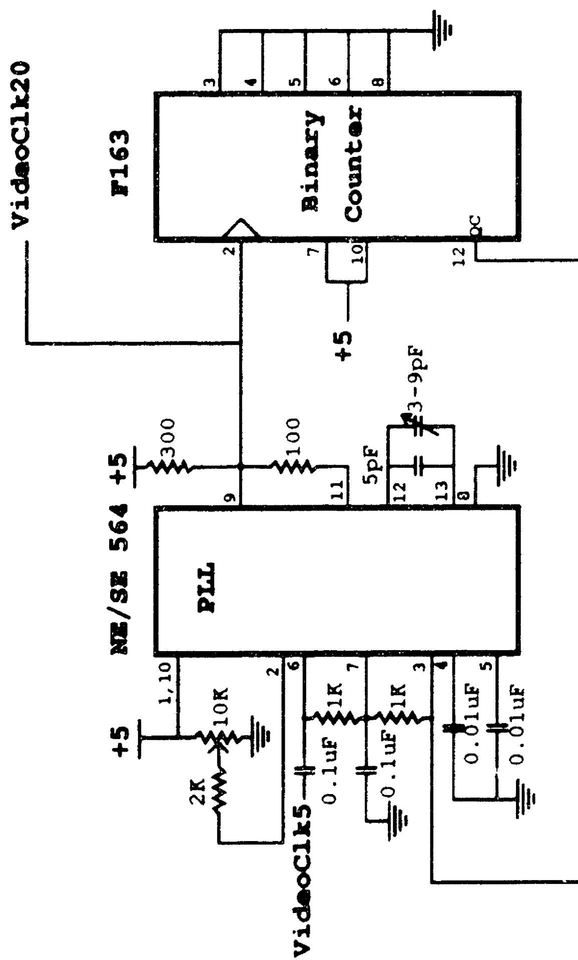


Figure 4-15: Clock Generation for Laser Modulation Circuit

4.6.1 Feedback PAL (PAL1)

```
Name            Christopher Mayer;
Partno          XX;
Date            {Date};
Revision        {0};
Designer        Christopher Mayer;
Company         Digital Equipment Corporation;
Assembly        Laser Feedback Control GAL;
Location        ;
Device          g16v8;
Format          j;

/****************************************************************************
** This pal generates control signals for up/down counters used to set the
** operating point of a laser diode.
****************************************************************************/

/********************************************************************************
**          Inputs (Pins 2-9)
********************************************************************************/

Pin 2   =   feedback; /* Feedback from laser      */
Pin 3 =   !laser_on;  /* Indicates laser is full power      */
Pin 4   =   reset; /* spare       */
Pin 5   =   spare1; /* spare       */
Pin 6   =   spare2; /* spare       */
Pin 7   =   spare3; /* spare       */
Pin 8   =   spare4; /* spare       */
Pin 9   =   spare5; /* spare       */

/********************************************************************************
**          Outputs (Pins 12-19)                                              **
********************************************************************************/

Pin 12  =   !enable_count; /* enables up/down counters      */
Pin 13  =   !clear_count; /* Reset counters, turn off laser      */
```

```
Pin 14  =    up_down; /* counter direction       */
Pin 15  =    delay1; /* registered delay line       */
Pin 16  =    delay2; /* registered delay line       */
Pin 17  =    delay3; /* registered delay line       */
Pin 18  =    delay4; /* registered delay line       */
Pin 19  =    delay5; /* registered delay line       */


$DEFINE TRUE    'b'1
$DEFINE FALSE   'b'0

enable_count.d = laser_on & !enable_count &
  delay5 & delay4 & delay3 & delay2 & delay1;
/* Enables counters if laser  */
/* was on for several cycles  */

clear_count.d = reset;

up_down.d = !feedback; /* Control term       */


delay1.d = !enable_count; /* delay line.                 */
delay2.d = delay1;
delay3.d = delay2;
delay4.d = delay3;
delay5.d = delay4;
```

4.6.2 Modulation PAL (PAL2)

```
Name            Christopher Mayer;
Partno          XX;
Date            {Date};
Revision        {0};
Designer        Christopher Mayer;
Company         Digital Equipment Corporation;
Assembly        Laser Modulation Control GAL;
Location        ;
Device          g16v8;
Format          j;

/*************************************************************************
** This pal generates control signals for pulse width modulating a laser.
*************************************************************************/
/*****************************************************************************
**          Inputs (Pins 2-9)
*****************************************************************************/
Pin 2  =    i; /* Inverse       */
Pin 3 =    m; /* Mode (0 = Edge, 1 = Fill)       */
Pin 4  =    p2; /* Pulse pattern, bit 2       */
Pin 5  =    p1; /* Pulse pattern, bit 1       */
Pin 6  =    p0; /* Pulse pattern, bit 0       */
Pin 7  =    c2; /* Count 2        */
Pin 8  =    c1; /* Count 1        */
Pin 9  =    c0; /* Count 0        */
/*****************************************************************************
**          Outputs (Pins 12-19)                                            **
*****************************************************************************/
Pin 12  =    o3; /* First stage of output.  c2=1, c1=1       */
Pin 13  =    o2; /* First stage of output.  c2=1, c1=0       */
Pin 14  =    o1; /* First stage of output.  c2=0, c1=1       */
Pin 15  =    o0; /* First stage of output.  c2=0, c1=0       */
Pin 16  =    mod; /* Second stage of output       */
Pin 17 =    modout; /* Third (and last) stage of output       */
Pin 18 =    i2; /* Delay line for i       */
```

```
Pin 19 =    spare2; /* Spare        */

$DEFINE TRUE    'b'1
$DEFINE FALSE   'b'0

o0 = !c0 & ( m & p2 & p1 & p0 ) #
  c0 & ( !m & p2 & p1 & p0 #
         m & p2 & ( p1 # p0 ) );

o1 = !c0 & ( !m & p2 & p1 #
          m & ( p2 #
                p1 & p0 ) ) #
  c0 & ( !m & p2 & ( p1 # p0 ) #
         m & ( p2 # p1 # p0 ) );

o2 = !c0 & ( !m & p2 #
          m & ( p2 # p1 ) ) #
  c0 & ( !m & ( p2 #
                p1 & p0 ) #
         m & p2 );

o3 = !c0 & ( !m & ( p2 # p1 ) #
                  m & p2 & p1 ) #
  c0 & !m & ( p2 # p1 # p0 );

i2.d = i;

mod.d = ( c2 &  c1 & o3 #
    c2 & !c1 & o2 #
   !c2 &  c1 & o1 #
   !c2 & !c1 & o0 );

modout.d  = mod $ i2;
```

Chapter 5

Results, Conclusions and recommendations.

The results of the algorithms presented in chapter 3 are quite promising. (See Appendix B for pictures.) A brief analysis of the results with suggestions for future research follows:

5.1 Cellular automaton smoothing

The cellular automaton smoothing algorithm provided adequate results within a few iterations; often the difference between two and four passes was impossible to distinguish. Many edge pixels attain half gray value and can progress no further, i.e. edge pixels either turn from white (1.0) to mid-gray (0.50) or from black (0.0) to mid-gray (0.50). This boundary condition is encountered along entire edges, as shown in figure B-4 in Appendix B. This indicates a problem with the algorithm; far too many pixels are stopped from crossing gray level 0.50 by the boundary condition, which indicates that the smoothing is unstable with the current rules.

A statistical comparison of antialiased images and smoothed images would probably indicate gross inaccuracies in the current smoothing algorithm for output with more accuracy than two bit grayscale; this comparison should be done. The smoothing rules (and formula) should be changed to more closely approximate antialiasing

and avoid the instability problem discussed above.

5.2 Windowed ROM antialiasing

The windowed ROM algorithm performed quite admirably. A thirteen or seventeen bit ROM mapping yields results practically indistinguishable from "perfect" methods of antialiasing such as supersampling or weighted area sampling, with practically no run-time computation. The first work which should be done to validate the windowed ROM algorithm is a statistical study comparing the algorithm to standard methods of antialiasing, devoting particular attention to conflict patterns. Although the algorithm seems to work extremely well, a better set of learning data is probably needed for general antialiasing work.

In the current set of learning data, lines and areas are well represented. This set of data performs quite good antialiasing on large fonts. At 300dpi, the letter "S" shown in Appendix B is printed at 18 points. However, for smaller fonts, the tight curves demand specialized learning patterns to properly antialias. It would be useful to study how well the current mapping performs on 6 point text, and to possible expand the learning data set to include some smaller fonts.

A halftone to grayscale mapping would be challenging to implement using the windowed ROM antialiasing. Probably the easiest way to account for halftone is simply to include halftone areas in the learning data, but this challenges the window shape selection; halftone clusters may work better with different (i.e. more round) window shapes rather than the hyperbolic star which is currently used. Halftone patterns might be introduced into the mapping by hand which would result in very consistent gray fields, but this would prove very tedious to do near edges. Perhaps a combination of an augmented learning pattern and some hand improvements to the mapping would produce better results.

Implementing a routine to convert blue noise dithering to grayscale would also be possible, although this would probably best be accomplished by including blue noise dithered areas in the learning pattern.

The windowed ROM algorithm should be adapted to provide not only grayscale values but also edge position information. This can be used by the laser printer driver to increase the resolution in the scan direction, while still maintaining antialiasing in the sub-scan direction. This modification could be made most easily by adding another few bits of output to the ROM and encoding the position.

Finally, the windowed ROM algorithm can be customized to the particular laser modulator being used. This could be accomplished two ways; the laser modulation codes could be programmed directly into the ROM instead of grayscale values, or another level of circuitry could be added to accomplish a mapping between the grayscale value and position to the laser modulation codes.

5.3 Laser modulation circuit

The laser modulation circuit should be installed in a laser printer engine to test the simulation results. After the simulation model is validated, there is work to be done developing a mapping between grayscale values and modulation codes to achieve the proper balance of pulse width and power for different gray levels.

Appendix A

Associated Computer Programs

A.1 SHOW_BITMAP.C

SHOW_BITMAP displays a grayscale bitmap using X windows.

```
/* show_bitmap.c windows grayscale bitmap previewer */
/* C. Mayer, 10-15-90, Digital Equipment Corp. */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>
#include <math.h>
#include <strings.h>

#define fontWidth(f)  ((f)->max_bounds.width)                                  10
#define fontHeight(f) ((f)->max_bounds.ascent + (f)->max_bounds.descent)

#define X_GRID 1000
#define Y_GRID 1000

#define TRUE 1
#define FALSE 0

int grayscale_image[Y_GRID][X_GRID];
```

```
int x_grid, y_grid;
char data_file_name[128], name[128];
int gray_levels;
int drawn;
int pixel_size;
int max_level;
int autolevel;

#define COLORLEVELS 100
char *colornames[COLORLEVELS+1];
unsigned long pixelvalue[COLORLEVELS+1];

Display      *dpy;
Window       window_grayscale;
GC           gc_grayscale, gc_grayscale_text;
XFontStruct *font_text;

/* ************************************************************ */
/* ************************************************************ */

void GetArgs(argc, argv)
    char **argv;
    int argc;

{
  char s[80];
  int flag;

  strcpy(data_file_name, "bitmap.dat");
  gray_levels = 100;
  autolevel = FALSE;
  max_level = 255;
  pixel_size = 10;
  argv++;
  argc--;
  while(argc > 0){
```

```
argc--;
flag = 0;
strcpy(s, *argv++);

if(strcasecmp(s, "-GRAYLEVELS") == 0){                                  60
  gray_levels = atoi(*argv++);
  argc--;
  if((gray_levels > COLORLEVELS) || (gray_levels < 0)){
      printf("Error:  GRAYLEVELS out of range.\n");
      exit(0);
  }
  flag = 1;
}

if(strcasecmp(s, "-MAXLEVEL") == 0){                                    70
  if (strcasecmp(*argv, "AUTO") == 0){
      autolevel = TRUE;
      max_level = 0;
      *argv++;
      argc--;
  } else {
      max_level = atoi(*argv++);
      argc--;
      if(max_level < 0){
        printf("Error:  MAXLEVEL out of range.\n");                    80
        exit(0);
      }
  }
  flag = 1;
}

if(strcasecmp(s, "-PIXELSIZE") == 0){
  pixel_size = atoi(*argv++);
  argc--;
  if((pixel_size > 100) || (pixel_size < 0)){                           90
      printf("Error:  PIXELSIZE out of range.\n");
```

```
        exit(0);
      }
      flag = 1;
    }

    if(strcasecmp(s, "-FILENAME") == 0){
      argc--;
      strcpy(data_file_name, *argv++);
      flag = 1;                                                     100
    }

    if(strcasecmp(s, "-NAME") == 0){
      argc--;
      strcpy(name, *argv++);
      flag = 1;
    }

    if(flag == 0){
      printf("Illegal argument:  %s\n", s);                        110
      printf("show_bitmap -filename <fn> -pixelsize # -maxlevel #
-graylevels #\n");
      exit(0);
    }
  }
}

/* ************************************************************** */

void UpdateScreen()                                                 120

{
  XEvent event;
  int i, j, c, lastc;

  while (XEventsQueued(dpy, QueuedAlready))
    {
```

```
  XNextEvent(dpy, &event);
  switch(event.type)
      {                                                              130
      case Expose :
        if (event.xexpose.count == 0)
          { XClearWindow(dpy, window_grayscale);
            drawn = 0;
          }
        break;
      case ButtonPress:
        XCloseDisplay(dpy);
        exit(0);
        break;                                                       140
      case MappingNotify:
        XRefreshKeyboardMapping((XMappingEvent *)&event);
        break;
      }
  }
if (!drawn) {
  lastc = -1;
  for(i = 0; i < x_grid; i++){
    if (XEventsQueued(dpy, QueuedAlready))
        drawn = FALSE;                                               150
    else {
        for(j = 0; j < y_grid; j++) {
          c = Map_grayscale_process(grayscale_image[j][i]);
          if (c != lastc) {
            lastc = c;
            XSetForeground(dpy, gc_grayscale, pixelvalue[c]);
          }
          XFillRectangle(dpy, window_grayscale, gc_grayscale,
                         i*pixel_size, j*pixel_size,
                         pixel_size, pixel_size);                    160
        }
    }
  }
```

```
    XSync(dpy, 0);
    drawn = TRUE;
  }

  XSync(dpy, 0);
}                                                                     170

/* ****************************************************************** */

int Map_grayscale_process(x)
    int x;
{
  int temp;
  double r;

  if (x < 0) x = 0;                                                   180
  r = (double) x / (double) max_level;
  r = (double) floor(r * (double) gray_levels);
  r = r / (double) (gray_levels - 1) * (double) COLORLEVELS;
  temp = floor(r + 0.5);
  return(temp);
}

/* ****************************************************************** */

void Compute()                                                        190
{
  int i, j;
  int n;
  char s[80];
  FILE *fp;

  fp = fopen(data_file_name, "r");
  fscanf(fp, "%s", s);
  x_grid = atoi(s);
```

```
  fscanf(fp, "%s", s);                                              200
  y_grid = atoi(s);
  for(j = 0; j < y_grid; j++){
    for(i = 0; i < x_grid; i++){
      fscanf(fp, "%s", s);
      n = atoi(s);
      grayscale_image[j][i] = n;
      if (autolevel)
            if (n > max_level) max_level = n;
    }
  }                                                                 210
  fclose(fp);
  max_level++;
}

/* ******************************************************************* */

void InitScreen()
{
  XEvent          event;
  unsigned long   foreground, background, pixvalues[8], status;     220
  Colormap        cmap;
  XColor          cdef;
  int             screen;
  XSizeHints      hint;
  int             ii, i;
  char            c_ten,c_one, s[80];

  dpy = XOpenDisplay("");

  screen = DefaultScreen(dpy);                                      230
  cmap = DefaultColormap(dpy, screen);
  foreground = WhitePixel(dpy, screen);
  background = BlackPixel(dpy, screen);
  hint.x      = (DisplayWidth(dpy, screen) - (x_grid * pixel_size)) / 2;
  hint.y      = (DisplayHeight(dpy, screen) - (y_grid * pixel_size)) / 2;
```

```
hint.width  = (x_grid * pixel_size);
hint.height = (y_grid * pixel_size);
hint.flags  = PPosition | PSize;
window_grayscale =
  XCreateWindow(dpy, DefaultRootWindow(dpy), hint.x, hint.y,                240
                    hint.width, hint.height, 5, DefaultDepth(dpy, screen),
                    InputOutput, (Visual *)CopyFromParent, 0L, NULL);

strcpy(s, "*");
for(c_ten='0'; c_ten <= '9'; c_ten++) {
  for(c_one='0'; c_one <= '9'; c_one++){
    i = (c_ten - '0') * 10 + (c_one - '0');
    colornames[i] = (char *) malloc(10);
    strcpy(colornames[i], "gray");
    if (c_ten != '0'){                                                      250
          s[0] = c_ten;
          strcat(colornames[i], s);
    }
    s[0] = c_one;
    strcat(colornames[i], s);
  }
}
colornames[100] = (char *) malloc(10);
strcpy(colornames[100], "gray100");
                                                                            260
for(i=0; i < (COLORLEVELS+1); i++){
  status = XParseColor(dpy, cmap, colornames[i], &cdef);
  if (status == 0) {
    printf("%d %s Color Parse Error.\n", i, colornames[i]);
  }
  status = XAllocColor(dpy, cmap, &cdef);
  if (status == 0) {
    printf("%s Color Allocation Error.\n", colornames[i]);
  }
  pixelvalue[i] = cdef.pixel;                                               270
}
```

```
status = XParseColor(dpy, cmap, "WHITE", &cdef);
status = XAllocColor(dpy, cmap, &cdef);
foreground = cdef.pixel;

status = XParseColor(dpy, cmap, "BLACK", &cdef);
status = XAllocColor(dpy, cmap, &cdef);
background = cdef.pixel;
                                                                                    280
XSetWindowBackground(dpy, window_grayscale, background);
XSetWindowBorder(dpy, window_grayscale, foreground);
XSetStandardProperties(dpy, window_grayscale, name, name,
                       None, NULL, 0, &hint);

gc_grayscale = XCreateGC(dpy, window_grayscale, 0, 0);
XSetForeground(dpy, gc_grayscale, foreground);
XSetBackground(dpy, gc_grayscale, background);

gc_grayscale_text = XCreateGC(dpy, window_grayscale, 0, 0);                         290
XSetForeground(dpy, gc_grayscale_text, foreground);
XSetBackground(dpy, gc_grayscale_text, background);

XSelectInput(dpy, window_grayscale,
             ButtonPressMask|ExposureMask|VisibilityChangeMask);

XMapRaised(dpy, window_grayscale);

font_text = XLoadQueryFont(dpy,
 "-adobe-new century schoolbook-bold-i-normal--12-120-75-75-p-76-iso8859-1");      300

XSetFont(dpy, gc_grayscale_text, font_text->fid);

do
  XNextEvent(dpy, &event);
while (event.type != VisibilityNotify);
```

```
  XClearWindow(dpy, window_grayscale);
  XSync(dpy, 0);
                                                                              310
  drawn = 0;
}


main(argc, argv)
char **argv;
int argc;
{
  GetArgs(argc, argv);
  Compute();                                                                  320
  InitScreen();
  while (1) {
    UpdateScreen();
  }
}
```

A.2 FILTER.C

FILTER reads a bitmap, filters it using the specified digital filter, and writes the resulting bitmap.

```
/* filter.c - program to digitally filter one bitmap with another */
/* C. Mayer, 10/16/90, Digital Equipment Corp. */

#include <stdio.h>
#include <strings.h>

#define TRUE 1
#define FALSE 0

char bitmapname[132], filtername[132];                                   10
int *bitmap, *filter;
int bitmap_x, bitmap_y, filter_x, filter_y, output_x, output_y;

/* ************************************************************* */

void GetArgs(argc, argv)
    char **argv;
    int argc;

{                                                                        20
  char s[132];
  int flag;

  if (argc != 3) {
    printf("filter:  Wrong number of arguments.\n");
    exit(0);
  }

  *argv++;
  strcpy(bitmapname, *argv++);                                           30
  strcpy(filtername, *argv++);
```

```
}

/* ************************************************************ */

Initialize()

{
  int i,j;
  FILE *fp;                                                           40
  char s[80];

  fp = fopen(bitmapname, "r");
  fscanf(fp, "%s", s);
  bitmap_x = atoi(s);
  fscanf(fp, "%s", s);
  bitmap_y = atoi(s);
  bitmap = (int *) malloc(sizeof(int) * bitmap_x * bitmap_y);

  for (j=0; j < bitmap_y; j++) {                                      50
    for (i=0; i < bitmap_x; i++) {
      fscanf(fp, "%s", s);
      *(bitmap + j*bitmap_x + i) = atoi(s);
    }
  }
  fclose(fp);

  fp = fopen(filtername, "r");
  fscanf(fp, "%s", s);
  filter_x = atoi(s);                                                 60
  fscanf(fp, "%s", s);
  filter_y = atoi(s);
  filter = (int *) malloc(sizeof(int) * filter_x * filter_y);

  for (j=0; j < filter_y; j++) {
    for (i=0; i < filter_x; i++) {
      fscanf(fp, "%s", s);
```

```
      *(filter + j*filter_x + i) = atoi(s);
    }
  }                                                                    70
  fclose(fp);

  output_x = bitmap_x + filter_x - 1;
  output_y = bitmap_y + filter_y - 1;
}

/* ************************************************************** */

Compute()
                                                                       80
{
  int x, y, i, j;
  int total;

  printf("%d %d\n", output_x, output_y);

  for (y = 0; y < output_y; y++){
    for (x = 0; x < output_x; x++){
      total = 0;
      for (j = 0; j < filter_y; j++){                                  90
          for (i = 0; i < filter_x; i++){
            if ((((y-j) >= 0) &&
                 ((y-j) < bitmap_y)) &&
                (((x-i) >= 0) &&
                 ((x-j) < bitmap_x))) {
              total+= *(filter + (filter_y - j - 1)*filter_x + (filter_x - i - 1)) *
                *(bitmap + (y - j)*bitmap_x + (x - i));
            }
          }
      }                                                                100
      printf("%d ", total);
    }
    printf("\n");
```

```
  }
}

/* ************************************************************ */
/* ************************************************************ */

main(argc, argv)                                                    110
char **argv;
int argc;
{
  GetArgs(argc, argv);
  Initialize();
  Compute();
}
```

A.3 SMOOTH.C

SMOOTH performs a cellular atomaton antialiasing of an bitmap. An argument specifies how many iterations of the routine are computed.

```
/* filter.c – program to digitally filter one bitmap with another */
/* C. Mayer, 10/16/90, Digital Equipment Corp. */

#include <stdio.h>
#include <strings.h>
#include <math.h>

#define TRUE 1
#define FALSE 0
#define BITMAP1(x,y) *(bitmap1 + (y) * bitmap_x + x)                    10
#define BITMAP2(x,y) *(bitmap2 + (y) * bitmap_x + x)
#define WHITE 1000
#define BLACK –WHITE
#define NEWS_MULT 7
#define DIAG_MULT 5
#define CENTER_MULT 0
#define DIVISOR 30.0

char bitmapname[132];
int *bitmap1, *bitmap2, *temp;                                          20
int bitmap_x, bitmap_y;
int iterations;
/* ************************************************************ */

void GetArgs(argc, argv)
    char **argv;
    int argc;

{
  char s[132];                                                          30
  int flag;
```

```
  if (argc != 2) {
    printf("smooth:  Wrong number of arguments.\n");
    exit(0);
  }
  *argv++;
  iterations = atoi(*argv++);
}
                                                                              40
/* ************************************************************ */

Initialize()

{
  int i,j;
  char s[80];

  scanf("%s", s);
  bitmap_x = atoi(s);                                                         50
  scanf("%s", s);
  bitmap_y = atoi(s);
  bitmap1 = (int *) malloc(sizeof(int) * bitmap_x * bitmap_y);
  bitmap2 = (int *) malloc(sizeof(int) * bitmap_x * bitmap_y);

  for (j=0; j < bitmap_y; j++) {
    for (i=0; i < bitmap_x; i++) {
      scanf("%s", s);
      BITMAP1(i,j) = atoi(s) * 2 * WHITE - WHITE;
    }                                                                         60
  }
}

/* ************************************************************ */

Compute()
```

```
{
  int i, j, result;
  int c, n, s, e, w, ne, se, nw, sw, reverse;                                              70
  double r;

  for (j = 1; j < (bitmap_y − 1); j++) {
    BITMAP2(0,j) = BITMAP1(0,j);
    BITMAP2(bitmap_x−1,j) = BITMAP1(bitmap_x−1,j);
  }
  for (i = 0; i < bitmap_x; i++) {
    BITMAP2(i,0) = BITMAP1(i,0);
    BITMAP2(i,bitmap_y−1) = BITMAP1(i,bitmap_y−1);
  }                                                                                        80

  for (j = 1; j < (bitmap_y − 1); j++){
    for (i = 1; i < (bitmap_x − 1); i++){
      c = BITMAP1(i, j);
      n = BITMAP1(i,j−1);
      s = BITMAP1(i,j+1);
      e = BITMAP1(i+1,j);
      w = BITMAP1(i−1,j);
      BITMAP2(i, j) = c;
      if (!((n == s) && (s == e) && (e == w))){                                            90
          if (((c == WHITE) &&
               ((n == BLACK) || (s == BLACK) || (e = BLACK) || (w = BLACK))) ||
              ((c == BLACK) &&
               ((n == WHITE) || (s == WHITE) || (e = WHITE) || (w = WHITE)))) {
            ne = BITMAP1(i+1,j−1);
            se = BITMAP1(i+1,j+1);
            nw = BITMAP1(i−1,j−1);
            sw = BITMAP1(i−1,j+1);
            if (c < 0) {
              reverse = −1;                                                                100
              c *= −1;
              n *= −1;
              s *= −1;
```

```
        e *= -1;
        w *= -1;
        ne *= -1;
        se *= -1;
        nw *= -1;
        sw *= -1;
      }                                                                   110
      else reverse = 1;

      result = NEWS_MULT * (n + s + e + w) + DIAG_MULT *
        (nw + sw + ne + se) + CENTER_MULT * c;
      r = (double) result;
      result = floor(r / DIVISOR);
      if (result < 1) result = 1;
      if (result > WHITE) result = WHITE;
      BITMAP2(i, j) = result * reverse;
     }                                                                    120
    }
   }
  }
}

/* ************************************************************** */

Output()

{                                                                         130
  int i,j;

  printf("%d %d\n", bitmap_x, bitmap_y);
  for (j=0; j < bitmap_y; j++) {
    for (i=0; i < bitmap_x; i++) {
      printf("%3d ", BITMAP1(i,j) + WHITE);
    }
    printf("\n");
  }
```

```
}                                                                  140

/* ************************************************************** */
/* ************************************************************** */

main(argc, argv)
char **argv;
int argc;
{
  int count;
                                                                   150
  GetArgs(argc, argv);
  Initialize();
  for (count = 0; count < iterations; count++){
    Compute();
    temp = bitmap2;
    bitmap2 = bitmap1;
    bitmap1 = temp;
  }
  Output();
}                                                                  160
```

A.4 CREATE_CIRC.C

CREATE_CIRC creates a pattern of vectors which is used during the learning process of building antialiasing ROMS.

```
#include <stdio.h>
#include <math.h>

#define S 10
#define PRECISION 10000
#define VECTORS 25

main()

{                                                                    10
  double sin(), cos();
  int i;
  int t;
  double r;
  FILE *fp;

  fp = fopen("vectors.dat", "w");

  for(t=0; t < 360; t += S) {
    for(r=290; r > 40.0; r -= 10.12345) {                            20
      fprintf(fp, "%lf ", r * cos((double) t / 180.0 * 3.1415) + 250.0);
      fprintf(fp, "%lf ", r * sin((double) t / 180.0 * 3.1415) + 250.0);
      fprintf(fp, "%lf ", r * cos(((double) (t + S) - 300.0 / r) /
180.0 * 3.1415) + 250.0);
      fprintf(fp, "%lf ", r * sin(((double) (t + S) - 300.0 / r) /
180.0 * 3.1415) + 250.0);
      fprintf(fp, "\n");
    }
  }
                                                                     30
  for (i = 0; i < VECTORS; i++) {
```

```
   fprintf(fp, "%lf ", (double) (random() % (100 * PRECISION)) /
PRECISION + 200.0);
   fprintf(fp, "%lf ", (double) (random() % (100 * PRECISION)) /
PRECISION + 200.0);
   fprintf(fp, "%lf ", (double) (random() % (100 * PRECISION)) /
PRECISION + 200.0);
   fprintf(fp, "%lf ", (double) (random() % (100 * PRECISION)) /
PRECISION + 200.0);
   fprintf(fp, "\n");                                                    40
  }
  fclose(fp);
}
```

A.5 CREATE_MAP.C

CREATE_MAP creates an antialiasing ROM from a set of vectors.

```
/* create_map2.c adaptive anti-aliasing program */
/* This program creates aliased and anti-aliased bitmaps from vectors
   and then outputs a mapping between the two bitmaps. */
/* C. Mayer, 11-14-90, Digital Equipment Corp. */

#include <stdio.h>
#include <math.h>
#include <strings.h>

#define TRUE 1                                                              10
#define FALSE 0
#define MAXDOUBLE 1.0e20

#define THRESHOLD 0.5000
#define PRECISION 1000

struct WINDOW_TYPE {
  int x;
  int y;
  double weight;                                                            20
};

struct MAPPING_TYPE {
  int     pattern;
  double  value, sumsqr;
  int     count;
  struct MAPPING_TYPE *left, *right;
};

char   data_file_name[80], filter_name[80], rom_name[80], window_name[80];  30
int    *bitmap, *bitmap_a, *bitmap_n;
int    *filter;
```

```
struct WINDOW_TYPE *window;
struct MAPPING_TYPE *mapping_head;
int     xpix, ypix;
int     filter_length, window_size;
int     mode;
int     output_mode;
int     pow2[32];
double  error;                                                          40
FILE    *fp;

double rint();
double floor();
double sqrt();
double pow();

/* ********************************************************** */
/* ********************************************************** */
                                                                        50
void GetArgs(argc, argv)
    char **argv;
    int argc;

{
  char s[80];
  int flag;

  if (output_mode == 2) printf("Initializing.\n");
  strcpy(data_file_name, "vectors.dat");                                60
  strcpy(filter_name, "filter.dat");
  strcpy(rom_name, "mapping.dat");
  strcpy(window_name, "window.dat");
  xpix = 50;
  ypix = xpix;
  output_mode = 2;
  argv++;
  argc--;
```

```
while(argc > 0){
  argc--;                                            70
  flag = 0;
  strcpy(s, *argv++);

  if(strcasecmp(s, "-VECTORS") == 0){
    argc--;
    strcpy(data_file_name, *argv++);
    flag = 1;
  }

  if(strcasecmp(s, "-FILTER") == 0){                 80
    argc--;
    strcpy(filter_name, *argv++);
    flag = 1;
  }

  if(strcasecmp(s, "-MAPPING") == 0){
    argc--;
    strcpy(rom_name, *argv++);
    flag = 1;
  }                                                  90

  if(strcasecmp(s, "-WINDOW") == 0){
    argc--;
    strcpy(window_name, *argv++);
    flag = 1;
  }

  if(strcasecmp(s, "-XPIX") == 0){
    argc--;
    xpix = atoi(*argv++);                            100
    if ((xpix < 1) || (xpix > 1000)) {
        printf("Error:  xpix out of range.\n");
        exit(0);
    }
```

```
    flag = 1;
  }

  if(strcasecmp(s, "-YPIX") == 0){
    argc--;
    ypix = atoi(*argv++);                                                    110
    if ((ypix < 1) || (ypix > 1000)) {
        printf("Error:  ypix out of range.\n");
        exit(0);
    }
    flag = 1;
  }

  if(strcasecmp(s, "-NORMAL") == 0){
    output_mode = 0;
    flag = 1;                                                                120
  }

  if(strcasecmp(s, "-ANTIALIAS") == 0){
    output_mode = 1;
    flag = 1;
  }

  if(flag == 0){
    printf("Illegal argument:  %s\n", s);
    printf("create_map [-parameter ...]\n\nValid parameters are:\n           130
  [-vectors <fn>]\n   [-filter <fn>]\n   [-mapping <fn>]\n   [-window <fn>]\n
  [-xpix <int>]\n   [-ypix <int>]\n   [-normal | -antialias]\n");
    exit(0);
  }
 }
}

/* ********************************************************* */

void initialize()                                                            140
```

```
{
  int i, j;
  double x;

  for (i=0; i<32; i++)
    pow2[i] = (int) rint(pow(2.0, (double) i));

  fp = fopen(filter_name, "r");
  fscanf(fp, "%d", &filter_length);                                          150
  filter = (int *) calloc(filter_length, sizeof(int));
  for (i = 0; i < filter_length; i++) {
    fscanf(fp, "%lf", &x);
    *(filter+i) = (int) rint(x * PRECISION);
  }
  fclose(fp);

  fp = fopen(window_name, "r");
  fscanf(fp, "%d", &window_size);
  window = (struct WINDOW_TYPE *) calloc(window_size, sizeof(struct WINDOW_TYPE));160
  for (i = 0; i < window_size; i++) {
    fscanf(fp, "%d", &j);
    (window+i)->x = j;
    fscanf(fp, "%d", &j);
    (window+i)->y = j;
    fscanf(fp, "%lf", &x);
    (window+i)->weight = x;
  }
  fclose(fp);
}                                                                            170

/* ************************************************************ */

int map(oldval, distance)
    int oldval;
    double distance;
```

```
{
  int newval;
  int i;
                                                                              180
  i = (int) floor(distance / 2.0 * (double) filter_length);
  if (i >= filter_length) i = filter_length - 1;
  if (mode == 0) {
    if (distance < THRESHOLD) i = 1; else i = oldval;
  } else {
    newval = *(filter + i);
    if (newval > oldval) {
      i = newval;
    } else {
      i = oldval;                                                             190
    }
  }
  return(i);
}

/* ********************************************************** */

scan_convert_line(x1, y1, x2, y2)
    double x1, y1, x2, y2;
                                                                              200
{
  double i, j, x, y, b, xx, yy, d, m;
  int k;

  if (fabs(y2 - y1) > fabs(x2 - x1)) {
    if ((y2 - y1) < 0.0) {
      y = y1; y1 = y2; y2 = y;
      x = x1; x1 = x2; x2 = x;
    }
    m = (x2 - x1) / (y2 - y1);                                                210
    b = x1 - m * y1;
```

```
  for (i = rint(y1 - 1.0); i <= rint(y2 + 1.0); i = i + 1.0) {
    y = i;
    x = m * y + b;
    for (j = rint(x - 1.0); j <= rint(x + 1.0); j = j + 1.0) {
         if ((i >= 0.0) && (j >= 0.0) && (i < ypix) && (j < xpix)) {
           yy = (m * (j - b) + i) / (m * m + 1);
           xx = m * yy + b;
           d = sqrt((y - i) * (y - i) + (x - j) * (x - j));                     220
           k = map(*(bitmap + ((int) i) * xpix + ((int) j)), d);
           *(bitmap + ((int) i) * xpix + ((int) j)) = k;
         }
    }
  }
} else {

  if ((x2 - x1) < 0.0) {
    x = x1; x1 = x2; x2 = x;
    y = y1; y1 = y2; y2 = y;                                                    230
  }
  m = (y2 - y1) / (x2 - x1);
  b = y1 - m * x1;

  for (i = rint(x1 - 1.0); i <= rint(x2 + 1.0); i = i + 1.0) {
    x = i;
    y = m * x + b;
    for (j = rint(y - 1.0); j <= rint(y + 1.0); j = j + 1.0) {
         if ((i >= 0.0) && (j >= 0.0) && (i < xpix) && (j < ypix)) {
           xx = (m * (j - b) + i) / (m * m + 1);                                240
           yy = m * xx + b;
           d = sqrt((x - i) * (x - i) + (y - j) * (y - j));
           k = map(*(bitmap + ((int) j) * xpix + ((int) i)), d);
           *(bitmap + ((int) j) * xpix + ((int) i)) = k;
         }
    }
  }
}
```

```
}
                                                                              250
/* ************************************************************ */

void make_image()

{
  double x1, x2, y1, y2;

  if (output_mode == 2) printf("Making image.\n");
  bitmap = (int *) calloc(xpix * ypix, sizeof(int));
                                                                              260
  fp = fopen(data_file_name, "r");
  while (!feof(fp)){
    fscanf(fp, "%lf %lf %lf %lf", &x1, &y1, &x2, &y2);
    scan_convert_line(x1, y1, x2, y2);
  }
  fclose(fp);
}

/* ************************************************************ */
                                                                              270
void write_image()

{
  int i, j;

  if (output_mode == mode) {
    printf("%d %d\n", xpix, ypix);
    for (j = 0; j < ypix; j++) {
      for (i = 0; i < xpix; i++) {
            printf("%d ", *(bitmap + j * xpix + i));                          280
      }
      printf("\n");
    }
  }
```

```
}

/* ************************************************************ */

struct MAPPING_TYPE *create_node()
                                                                290
{
 int i;
 struct MAPPING_TYPE *m;

 m = (struct MAPPING_TYPE *) malloc(sizeof(struct MAPPING_TYPE));
 m->pattern = 0;
 m->value = 0.0;
 m->sumsqr = 0.0;
 m->count = 0;
 m->left = NULL;                                                300
 m->right = NULL;
 return(m);
}

/* ************************************************************ */

generate_mapping()

{                                                               310
 int    i, j, k, y2, x2, p;
 char   s[80];
 struct MAPPING_TYPE *m;

 if (output_mode == 2) printf("Generating mapping.\n");
 mapping_head = create_node();
 for (j = 0; j < ypix; j++) {
   for (i = 0; i < xpix; i++) {
     p = 0;
     for (k = 0; k < window_size; k++) {
          x2 = i + (window+k)->x;                               320
```

```
        y2 = j + (window+k)->y;
        if (!((x2 < 0) || (x2 >= xpix) || (y2 < 0) || (y2 >= ypix)))
          if (*(bitmap_n + y2 * xpix + x2) == 1)
            p = p + pow2[window_size - k - 1];
    }
    m = mapping_head;
    while (m->pattern != p) {
        if (p > m->pattern) {
          if (m->right == NULL) {
            m->right = create_node();                                          330
            m->right->pattern = p;
          }
          m = m->right;
        } else {
          if (m->left == NULL) {
            m->left = create_node();
            m->left->pattern = p;
          }
          m = m->left;
        }                                                                      340
    }
    k = *(bitmap_a + j * xpix + i);
    m->count++;
    m->sumsqr += pow((double) k, 2.0);
    m->value = (((m->count - 1) * m->value) + ((double) k)) / m->count;
   }
  }
}

/* ************************************************************ */               350

print_tree(m)
   struct MAPPING_TYPE *m;
{
  if (m == NULL) return(0);
  else {
```

```
    printf("%d  %5.3lf  %5.3lf  %5d\n", m->pattern, m->value, m->sumsqr,
              m->count);
    print_tree(m->right);
    print_tree(m->left);                                                    360
  }
}

/* ******************************************************** */

double weight_diff(p1, p2)

    int p1, p2;
{
  int i, j;                                                                 370
  double total;

  i = 0;
  total = 0.0;
  j = p1 ^ p2;
  for (i = 0; i < window_size; i++) {
    if (j & pow2[window_size - i - 1])
      total += (window+i)->weight;
  }
  return(total);                                                            380
}

/* ******************************************************** */

struct MAPPING_TYPE *find_closest_node(m, p)
    struct MAPPING_TYPE *m;
    int p;

{
  struct MAPPING_TYPE *ml, *mr;                                             390
  double m_error;
  int i, j, k, done;
```

```
if (m == NULL) return(NULL);

if (p == m->pattern) {
  error = 0.0;
  return(m);
}
                                                                         400
if (p > m->pattern) {                    /* Take a right */
  mr = find_closest_node(m->right, p);
  if (error == 0.0)                      /* Quick return for a zero */
    return(mr);
  m_error = weight_diff(m->pattern, p);      /* Incorporate self */
  if (m_error < error) {
    error = m_error;
    mr = m;
  }
  /* Now figure out the closest that we can come in the other side of the tree. */   410
  /* Start with the highest order bit (also highest weighted) and work until
     one of the bits is not able to match.  Then match all the rest. */
  k = 0;
  done = FALSE;
  for (i = 0; (i < window_size) && (!done); i++){
    k += p & pow2[window_size - i - 1];
    if (k > m->pattern) {
        k -= p & pow2[window_size - i - 1];
        k += p & (pow2[window_size - i - 1] - 1);
        done = TRUE;                                                     420
    }
  }
  m_error = weight_diff(k, p);        /* Find how small the error could be */
  if (m_error < error) {            /* If it is smaller, branch */
    m_error = error;                /* Save error for test later */
    ml = find_closest_node(m->left, p);
    if (error == 0.0)                    /* Quick return for a zero */
        return(ml);
```

```
    if (error < m_error)
        mr = ml;                                                           430
  }
  return(mr);

} else {                        /* Take a left */

  ml = find_closest_node(m->left, p);
  if (error == 0.0)                         /* Quick return for a zero */
    return(ml);
  m_error = weight_diff(m->pattern, p);       /* Incorporate self */
  if (m_error < error) {                                                   440
    error = m_error;
    ml = m;
  }
  /* Now figure out the closest that we can come in the other side of the tree. */
  /* Start with the highest order bit (also highest weighted) and work until
     one of the bits is not able to match.   Then match all the rest. */
  k = pow2[window_size] - 1;
  done = FALSE;
  j = p ^ (pow2[window_size] - 1);
  for (i = 0; (i < window_size) && (!done); i++){                           450
    k -= j & pow2[window_size - i - 1];
    if (k > m->pattern) {
        k += j & pow2[window_size - i - 1];
        k -= j & (pow2[window_size - i - 1] - 1);
        done = TRUE;
    }
  }
  m_error = weight_diff(k, p);       /* Find how small the error could be */
  if (m_error < error) {            /* If it is smaller than what we have, branch */
    m_error = error;               /* Save error for test later */           460
    mr = find_closest_node(m->right, p);
    if (error == 0.0)                        /* Quick return for a zero */
        return(mr);
    if (error < m_error)
```

```
      ml = mr;
    }
    return(ml);
  }
}

/* ********************************************************** */

expand_mapping()

{
  int i, j, k, num_nodes;
  struct MAPPING_TYPE *m;

  num_nodes = pow2[window_size];
  if (output_mode == 2) printf("Expanding mapping to %d nodes.\n", num_nodes);
  fp = fopen(rom_name, "w");
  for(i = 0; i < num_nodes; i++){
    if (output_mode == 2)
      if ((i & 0x00000fff) == 0)
          printf("    Expanding...    node %d\n", i);
    error = MAXDOUBLE;      /* This MUST precede every call to find_closest_node */
    m = find_closest_node(mapping_head, i);
    putw((int) rint(m->value), fp);
  }
  fclose(fp);
}

/* ********************************************************** */
/* ********************************************************** */

main(argc, argv)
char **argv;
int argc;
{
  GetArgs(argc, argv);
```

```
  initialise();
  for (mode = 0; mode < 2; mode++) {
    make_image();
    if (mode == 0) {
      bitmap_n = bitmap;
    } else {
      bitmap_a = bitmap;
    }
    write_image();
  }                                                              510
  if (output_mode == 2) {
    generate_mapping();
    expand_mapping();
  }
}
```

A.6 ANTI_ALIAS.C

ANTIALIAS uses an antialiasing ROM to antialias a bitonal bitmap.

```
/* anti-alias.c adaptive anti-aliasing program */
/* This program anti-aliases bitmaps using a mapping */
/* C. Mayer, 11-26-90, Digital Equipment Corp. */

#include <stdio.h>
#include <math.h>
#include <strings.h>

#define TRUE 1
#define FALSE 0                                                        10

struct WINDOW_TYPE {
  int x;
  int y;
  double weight;
};

struct MAPPING_TYPE {
  char    *pattern;
  double  value, sumsqr;                                               20
  int     count;
  double     error;
  struct MAPPING_TYPE *left, *right;
};

char    rom_name[80], window_name[80];
int     *bitmap;
struct WINDOW_TYPE *window;
struct MAPPING_TYPE *mapping_head;
int     xpix, ypix;                                                    30
int     window_size;
int     pow2[32];
```

```
FILE    *fp;

double rint();
double floor();
double sqrt();
double pow();

/* ****************************************************************** */    40
/* ****************************************************************** */

void GetArgs(argc, argv)
    char **argv;
    int argc;

{
  char s[80];
  int flag;
                                                                              50
  strcpy(rom_name, "mapping.dat");
  strcpy(window_name, "window.dat");
  xpix = 50;
  ypix = xpix;
  argv++;
  argc--;
  while(argc > 0){
    argc--;
    flag = 0;
    strcpy(s, *argv++);                                                       60

    if(strcasecmp(s, "-MAPPING") == 0){
      argc--;
      strcpy(rom_name, *argv++);
      flag = 1;
    }

    if(strcasecmp(s, "-WINDOW") == 0){
```

```
      argc--;
      strcpy(window_name, *argv++);                                   70
      flag = 1;
    }

    if(flag == 0){
      printf("Illegal argument:  %s\n", s);
      printf("anti_alias -mapping <fn> -window <fn>\n");
      exit(0);
    }
  }
}                                                                     80

/* **************************************************************** */

void initialize()

{
  int i, j, k;
  double x;

  for (i=0; i<32; i++)                                                90
    pow2[i] = (int) rint(pow(2.0, (double) i));

  fp = fopen(window_name, "r");
  fscanf(fp, "%d", &window_size);
  window = (struct WINDOW_TYPE *) calloc(window_size, sizeof(struct WINDOW_TYPE));
  for (i = 0; i < window_size; i++) {
    fscanf(fp, "%d", &j);
    (window+i)->x = j;
    fscanf(fp, "%d", &j);
    (window+i)->y = j;                                                100
    fscanf(fp, "%lf", &x);
    (window+i)->weight = x;
  }
  fclose(fp);
```

```
  scanf("%d %d", &xpix, &ypix);
  bitmap = (int *) malloc(xpix * ypix * siseof(int));
  for (j=0; j < ypix; j++)
    for (i=0; i < xpix; i++) {
      scanf("%d", &k);                                                    110
      *(bitmap + j*xpix + i) = k;
    }

  fp = fopen(rom_name, "r");
}

/* *************************************************************** */

make_bitmap()
                                                                          120
{
  int i, j, k, p, x2, y2, v;
  char *s[80];

  printf("%d %d\n", xpix, ypix);
  for (j = 0; j < ypix; j++) {
    for (i = 0; i < xpix; i++) {
      p = 0;
      for (k = 0; k < window_size; k++) {
            x2 = i + (window+k)->x;                                       130
            y2 = j + (window+k)->y;
            if ((x2 >= 0) && (x2 < xpix) && (y2 >= 0) && (y2 < ypix)) {
              if (*(bitmap + y2 * xpix + x2) == 1) {
                p = p + pow2[window_size - k - 1];
              }
            }
      }
      fseek(fp, p * siseof(int), SEEK_SET);
      v = getw(fp);
      if (v < 0) {                                                        140
```

```
        printf("ERROR: Read past EOF.\n");
        exit(0);
      }
      printf("%d ", v);
    }
    printf("\n");
  }
}

/* ****************************************************************** */                    150
/* ****************************************************************** */

main(argc, argv)
char **argv;
int argc;
{
  GetArgs(argc, argv);
  initialize();
  make_bitmap();
                                                                                            160
}
```

A.7 PRINTSIM.C

PRINTSIM simpulates the printing process of a laser printer. The OPC drum energy level and the resulting toner distribution are displayed for a specified bitmap and modulation scheme.

```
/* printsim.c laser printer simulation */
/* C. Mayer, 6-19-90, Digital Equipment Corp. */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>
#include <math.h>
#include <strings.h>

#define fontWidth(f)  ((f)->max_bounds.width)                                          10
#define fontHeight(f) ((f)->max_bounds.ascent + (f)->max_bounds.descent)

#define X_PIX_DIVS_MAX 100
#define Y_PIX_DIVS_MAX 100

#include "grid_size.h"

#define INTEGRATION_ACCURACY 200.0

#define TRUE 1                                                                         20
#define FALSE 0

#define K1 1.2        /* power */
#define K2 1.0        /* quality or drop off */
#define K3 0.8        /* y/x scale factor */
#define ENERGY_SCALE 4.2
#define TONER_THRESHOLD 1.0

#define PWM_MODELS 32
/* Caution: if PWM_MODELS is changed, change the masking below */                     30
```

```
#define X_SIZE x_pix_divs * X_GRID
#define Y_SIZE y_pix_divs * Y_GRID
#define INT_STEP (2.0 / INTEGRATION_ACCURACY)

int digital_image[Y_GRID][X_GRID];
double gaussian_x[PWM_MODELS][3 * X_PIX_DIVS_MAX];
double gaussian_y[3 * Y_PIX_DIVS_MAX];
int x_pix_divs, y_pix_divs;
double xp1[PWM_MODELS], xp2[PWM_MODELS], xp3[PWM_MODELS], xp4[PWM_MODELS];
char data_file_name[128];                                                    40
char pwm_data_file_name[128];



int      drawn, energy, toner;
struct PAGE_TYPE {
  double   data[Y_PIX_DIVS_MAX * Y_GRID][X_PIX_DIVS_MAX * X_GRID];
} page_map;

typedef char *COLORNAMES;
#define COLORLEVELS 8                                                        50
COLORNAMES colornames[] = { "BLACK",
                     "DARKGREEN",
                       "MIDNIGHTBLUE",
                       "BLUE",
                 "RED",
                       "GREEN",
                       "YELLOW",
                       "WHITE" };
unsigned long pixelvalue[COLORLEVELS];
                                                                             60
Display    *dpy;
Window      window_toner, window_energy;
GC        gc_energy, gc_energy_text, gc_toner, gc_toner_text;
XFontStruct *font_text;

/* ****************************************************************** */
```

```
/* ****************************************************************** */

void GetArgs(argc, argv)
    char **argv;                                                          70
    int argc;

{
  char s[80];
  int flag;

  strcpy(data_file_name, "printsim.dat");
  strcpy(pwm_data_file_name, "modulation.dat");
  x_pix_divs = 50;
  y_pix_divs = 50;                                                        80
  toner = FALSE;
  energy = FALSE;
  argv++;
  argc--;
  while(argc > 0){
    argc--;
    flag = 0;
    strcpy(s, *argv++);
    if(strcasecmp(s, "-XPIXEL") == 0){
      x_pix_divs = atoi(*argv++);                                         90
      argc--;
      if((x_pix_divs > X_PIX_DIVS_MAX) || (x_pix_divs < 0)){
           printf("Error:  XPIXEL out of range.\n");
           exit(0);
      }
      flag = 1;
    }
    if(strcasecmp(s, "-YPIXEL") == 0){
      y_pix_divs = atoi(*argv++);
      argc--;                                                             100
      if((y_pix_divs > Y_PIX_DIVS_MAX) || (y_pix_divs < 0)){
           printf("Error:  YPIXEL out of range.\n");
```

```
            exit(0);
        }
        flag = 1;
      }
      if(strcasecmp(s, "-PULSEMOD") == 0){
        argc--;
        strcpy(pwm_data_file_name, *argv++);
        flag = 1;                                                    110
      }
      if(strcasecmp(s, "-FILENAME") == 0){
        argc--;
        strcpy(data_file_name, *argv++);
        flag = 1;
      }
      if(strcasecmp(s, "-ENERGY") == 0){
        energy = TRUE;
        flag = 1;
      }                                                              120
      if(strcasecmp(s, "-TONER") == 0){
        toner = TRUE;
        flag = 1;
      }
      if(flag == 0){
        printf("Illegal argument:  %s\n", s);
        exit(0);
      }
    }
    if ((energy == FALSE) && (toner == FALSE))                       130
      toner = TRUE;
  }

/* ****************************************************************** */

void UpdateScreen()

{
```

```
XEvent event;
int i, j, c_energy, c_toner, lastc_energy, lastc_toner;                    140

while (XEventsQueued(dpy, QueuedAlready))
 {
  XNextEvent(dpy, &event);
  switch(event.type)
       {
       case Expose :
        if (event.xexpose.count == 0)
          { if (energy == TRUE) XClearWindow(dpy, window_energy);
            if (toner == TRUE) XClearWindow(dpy, window_toner);            150
            drawn = 0;
          }
        break;
       case ButtonPress:
        printf("Exiting...\n");
        XCloseDisplay(dpy);
        exit(0);
        break;
       case MappingNotify:
        XRefreshKeyboardMapping((XMappingEvent *)&event);                  160
        break;
       }
 }
if (!drawn) {
  printf("Developing...\n");

  c_energy = -1;
  if (energy == TRUE){
   for(i = 0; i < X_SIZE; i++){
       if (XEventsQueued(dpy, QueuedAlready))                              170
         drawn = 0;
       else {
         for(j = 0; j < Y_SIZE; j++){
           lastc_energy= Map_energy_process(page_map.data[j][i]);
```

```
            if (c_energy != lastc_energy) {
              c_energy = lastc_energy;
              XSetForeground(dpy, gc_energy, pixelvalue[c_energy]);
            }
            XDrawPoint(dpy, window_energy, gc_energy, i, j);
          }                                                                   180
        }
      }
    }

    c_toner = -1;
    if (toner == TRUE){
      for(i = 0; i < X_SIZE; i++){
        if (XEventsQueued(dpy, QueuedAlready))
          drawn = 0;
        else {                                                                190
          for(j = 0; j < Y_SIZE; j++){
            lastc_toner= Map_toner_process(page_map.data[j][i]);
            if (c_toner != lastc_toner) {
              c_toner = lastc_toner;
              XSetForeground(dpy, gc_toner, pixelvalue[c_toner]);
            }
            XDrawPoint(dpy, window_toner, gc_toner, i, j);
          }
        }
      }                                                                       200
    }

    XSync(dpy, 0);
    drawn = 1;
  }

  XSync(dpy, 0);
}

/* ****************************************************************** */  210
```

```
double PowerLevel(c)
    int c;
{
  return(pow(((double) c / 7.0),0.75));
}

/* ************************************************************ */

int Map_energy_process(x)                                              220
    double x;
{
  int temp;
  double r;

  r = (double) (random()&511) - 256;
  r = r * 0.4 / 256;
  temp = (int) floor(x * ENERGY_SCALE + r);
  if (temp > 7) temp = 7;
  if (temp < 0) temp = 0;                                              230
  return(temp);
}

/* ************************************************************ */

int Map_toner_process(x)
    double x;
{
  int temp;
  double r;                                                            240

  r = (double) (random()&511) - 256;
  r = r * 0.1 / 256;
  r = x / TONER_THRESHOLD + r;
  if (r >= 1)
    temp = 0;
```

```
  else
    temp = 7;
  return(temp);
}                                                                    250

/* ******************************************************* */

void MakeTables()
{
  double x, y;
  int i, pwm;
  char s[80];
  FILE *fp;
  double strtod();                                                   260

  printf("Reading pulse width modulation schemes from %s...\n", pwm_data_file_name);
  fp = fopen(pwm_data_file_name, "r");
  for (i=0; i < PWM_MODELS; i++){
    fscanf(fp, "%s", s); xp1[i] = strtod(s, (char **) NULL) / 8.0;
    fscanf(fp, "%s", s); xp2[i] = strtod(s, (char **) NULL) / 8.0;
    fscanf(fp, "%s", s); xp3[i] = strtod(s, (char **) NULL) / 8.0;
    fscanf(fp, "%s", s); xp4[i] = strtod(s, (char **) NULL) / 8.0;
  }
  fclose(fp);                                                        270
  printf("Computing gaussian tables...\n");
  for(pwm = 0; pwm < PWM_MODELS; pwm ++){
    for(i = 0; i < (3 * x_pix_divs); i++){
      x = (double) (i - (x_pix_divs * 3 / 2)) / (x_pix_divs / 2);
                                  /* x ranges from -3 to 3 */
      gaussian_x[pwm][i] = 0.0;
      for(y = xp1[pwm]; y <= xp2[pwm]; y+= INT_STEP){
          gaussian_x[pwm][i] += K1 * exp(-1 * K2 * pow(x - y, 2.0)) * INT_STEP;
      }
      if (xp3[pwm] != xp4[pwm]) {                                    280
          for(y = xp3[pwm]; y <= xp4[pwm]; y+= INT_STEP){
            gaussian_x[pwm][i] += K1 * exp(-1 * K2 * pow(x - y, 2.0)) * INT_STEP;
```

```
          }
        }
      }
    }
    for(i = 0; i < (3 * y_pix_divs); i++){
      y = (double) (i - (y_pix_divs * 3 / 2)) / (y_pix_divs / 2);
                                          /* y ranges from -3 to 3 */
      gaussian_y[i] = exp(-1 * K2 * K3 * K3 * y * y);                          290
    }
}

/* ************************************************************ */

double Intensity(pl, pwm, x, y)
     int x, y, pl, pwm;
{
  double res;
                                                                               300
  res = PowerLevel(pl) * gaussian_x[pwm][x] * gaussian_y[y];

  return(res);
}

/* ************************************************************ */

void Compute()
{
  int i, j, x, y, power_level, pulse_width_modulation;                        310
  int n;
  char s[80];
  FILE *fp;

  printf("Making Image...\n");
  for(i = 0; i < X_SIZE; i++){
    for(j = 0; j < Y_SIZE; j++){
      page_map.data[j][i] = (double) 0.0;
```

```
  }
}                                                               320
fp = fopen(data_file_name, "r");
for(j = 0; j < Y_GRID; j++){
  for(i = 0; i < X_GRID; i++){
    fscanf(fp, "%s", s);
    n = atoi(s) * 8;
    fscanf(fp, "%s", s);
    n += atoi(s);
    printf("%3d ", n);
    digital_image[j][i] = n;
  }                                                             330
  /*    while(fgetc(fp) != '\n');     */
  putchar('\n');
}
fclose(fp);
for(i = 0; i < X_GRID; i++){
  for(j = 0; j < Y_GRID; j++){
    power_level = digital_image[j][i] & 0x07;
                                    /* mask power bits */
    pulse_width_modulation = digital_image[j][i] >> 3;
                          /* Shift (and mask) pwm bits */        340
    for(x = -x_pix_divs; x < (2*x_pix_divs); x++){
         for(y = -y_pix_divs; y < (2*y_pix_divs); y++){
           if((digital_image[j][i]) &&
              ((j * y_pix_divs + y) >= 0) &&
              ((i * x_pix_divs + x) >= 0) &&
              ((j * y_pix_divs + y) < Y_SIZE) &&
              ((i * x_pix_divs + x) < X_SIZE))
             page_map.data[j * y_pix_divs + y][i * x_pix_divs + x]+=
              Intensity(power_level, pulse_width_modulation,
                         x + x_pix_divs, y + y_pix_divs);        350
         }
    }
  }
}
```

```
}

/* ****************************************************************** */

void InitScreen()
{                                                                          360
  XEvent        event;
  unsigned long  foreground, background, pixvalues[8], status;
  Colormap      cmap;
  XColor        cdef;
  int           screen;
  XSizeHints    hint;
  int           i;

  printf("Creating window(s)...\n");
  dpy = XOpenDisplay("");                                                  370

  if (energy == TRUE) {
    screen = DefaultScreen(dpy);
    cmap = DefaultColormap(dpy, screen);
    foreground = WhitePixel(dpy, screen);
    background = BlackPixel(dpy, screen);
    hint.x      = (DisplayWidth(dpy, screen)  - 2 * X_SIZE - 10) / 2;
    hint.y      = (DisplayHeight(dpy, screen) - Y_SIZE) / 2;
    hint.width  = X_SIZE;
    hint.height = Y_SIZE;                                                  380
    hint.flags  = PPosition | PSize;
    window_energy = XCreateWindow(dpy, DefaultRootWindow(dpy), hint.x, hint.y,
                        hint.width, hint.height, 5, DefaultDepth(dpy, screen),
                        InputOutput, (Visual *)CopyFromParent, 0L, NULL);

    for(i=0; i<COLORLEVELS; i++){
      status = XParseColor(dpy, cmap, colornames[i], &cdef);
      status = XAllocColor(dpy, cmap, &cdef);
      pixelvalue[i] = cdef.pixel;
    }                                                                      390
```

```
status = XParseColor(dpy, cmap, "WHITE", &cdef);
status = XAllocColor(dpy, cmap, &cdef);
foreground = cdef.pixel;

status = XParseColor(dpy, cmap, "BLACK", &cdef);
status = XAllocColor(dpy, cmap, &cdef);
background = cdef.pixel;

XSetWindowBackground(dpy, window_energy, background);                    400
XSetWindowBorder(dpy, window_energy, foreground);
XSetStandardProperties(dpy, window_energy,
                        "OPC Drum Energy",
                        "OPC Drum Energy", None,
                        NULL, 0, &hint);

gc_energy = XCreateGC(dpy, window_energy, 0, 0);
XSetForeground(dpy, gc_energy, foreground);
XSetBackground(dpy, gc_energy, background);
                                                                         410
gc_energy_text = XCreateGC(dpy, window_energy, 0, 0);
XSetForeground(dpy, gc_energy_text, foreground);
XSetBackground(dpy, gc_energy_text, background);

XSelectInput(dpy, window_energy,
             ButtonPressMask|ExposureMask|VisibilityChangeMask);

XMapRaised(dpy, window_energy);

font_text    = XLoadQueryFont(dpy,                                       420
 "-adobe-new century schoolbook-bold-i-normal--12-120-75-75-p-76-iso8859-1");

XSetFont(dpy, gc_energy_text, font_text->fid);

do
 XNextEvent(dpy, &event);
```

```
  while (event.type != VisibilityNotify);

  XClearWindow(dpy, window_energy);
  XSync(dpy, 0);                                                         430
}

if (toner == TRUE) {
  screen = DefaultScreen(dpy);
  cmap = DefaultColormap(dpy, screen);
  foreground = WhitePixel(dpy, screen);
  background = BlackPixel(dpy, screen);
  hint.x      = (DisplayWidth(dpy, screen) + 10) / 2;
  hint.y      = (DisplayHeight(dpy, screen) - Y_SIZE) / 2;
  hint.width  = X_SIZE;                                                  440
  hint.height = Y_SIZE;
  hint.flags  = PPosition | PSize;
  window_toner = XCreateWindow(dpy, DefaultRootWindow(dpy), hint.x, hint.y,
                               hint.width, hint.height, 5, DefaultDepth(dpy, screen),
                               InputOutput, (Visual *)CopyFromParent, 0L, NULL);

  for(i=0; i<COLORLEVELS; i++){
    status = XParseColor(dpy, cmap, colornames[i], &cdef);
    status = XAllocColor(dpy, cmap, &cdef);
    pixelvalue[i] = cdef.pixel;                                          450
  }

  status = XParseColor(dpy, cmap, "WHITE", &cdef);
  status = XAllocColor(dpy, cmap, &cdef);
  foreground = cdef.pixel;

  status = XParseColor(dpy, cmap, "BLACK", &cdef);
  status = XAllocColor(dpy, cmap, &cdef);
  background = cdef.pixel;
                                                                         460
  XSetWindowBackground(dpy, window_toner, background);
  XSetWindowBorder(dpy, window_toner, foreground);
```

```
XSetStandardProperties(dpy, window_toner, "Toner", "Toner", None, NULL, 0,
                       &hint);

gc_toner = XCreateGC(dpy, window_toner, 0, 0);
XSetForeground(dpy, gc_toner, foreground);
XSetBackground(dpy, gc_toner, background);

gc_toner_text = XCreateGC(dpy, window_toner, 0, 0);                    470
XSetForeground(dpy, gc_toner_text, foreground);
XSetBackground(dpy, gc_toner_text, background);

XSelectInput(dpy, window_toner,
             ButtonPressMask|ExposureMask|VisibilityChangeMask);

XMapRaised(dpy, window_toner);

font_text    = XLoadQueryFont(dpy,
  "-adobe-new century schoolbook-bold-i-normal--12-120-75-75-p-76-iso8859-1");480

XSetFont(dpy, gc_toner_text, font_text->fid);

do
  XNextEvent(dpy, &event);
while (event.type != VisibilityNotify);

XClearWindow(dpy, window_toner);
XSync(dpy, 0);
}                                                                        490
drawn = 0;
}


main(argc, argv)
char **argv;
int argc;
{
```

```
  GetArgs(argc, argv);
  InitScreen();                                                    500
  MakeTables();
  Compute();
  while (1) {
    UpdateScreen();
  }
}
```

Appendix B

Experimental results

B.1 Celllular automata smoothing

The cellular automata smoothing technique described in chapter 3 is demonstrated here. Smoothing is performed on the letter "S" in a bitmap of approximately 72×36 pixels.

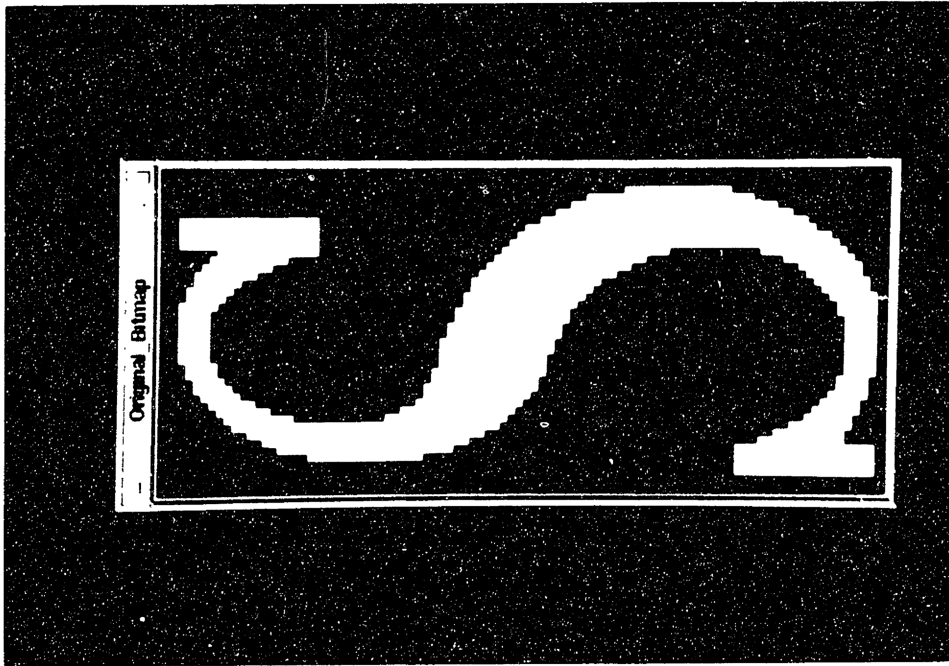


Figure B.1 Original bitonal bitmap of the letter "S"

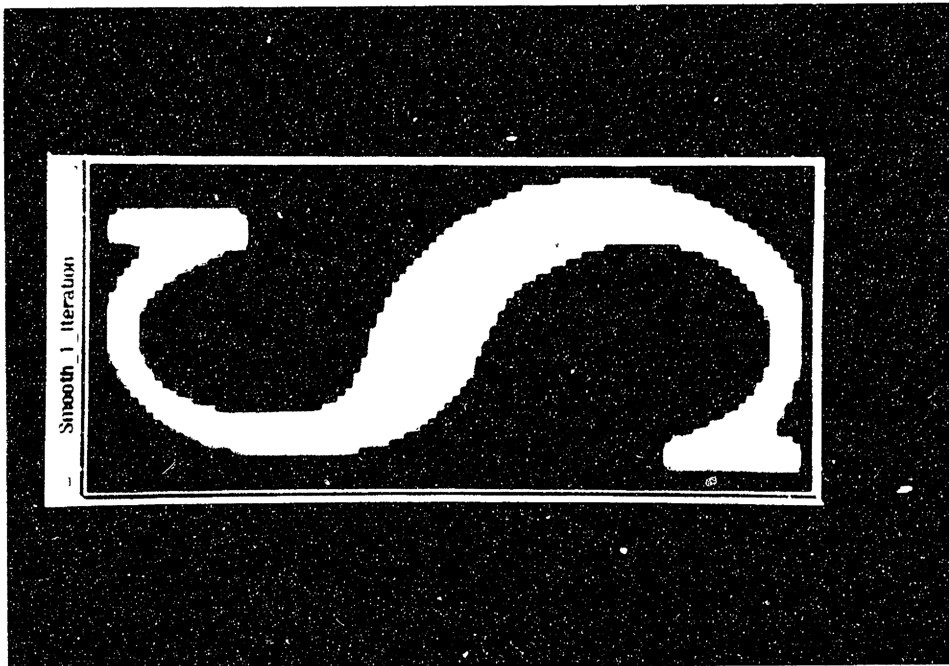


Figure B.2 Letter "S" after smoothing 1 iteration

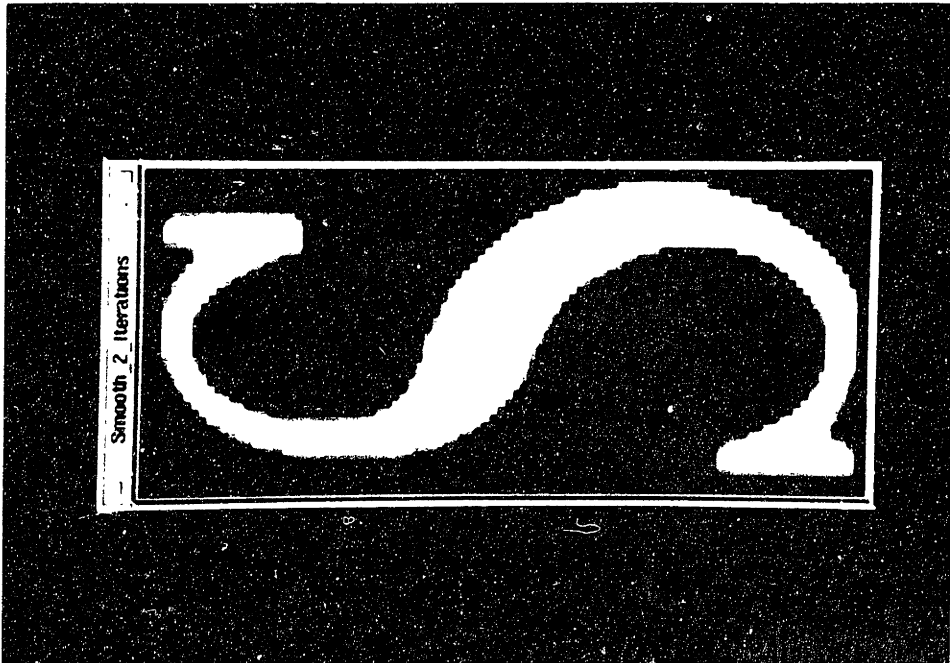


Figure B.3. Letter "S" after smoothing 2 iterations

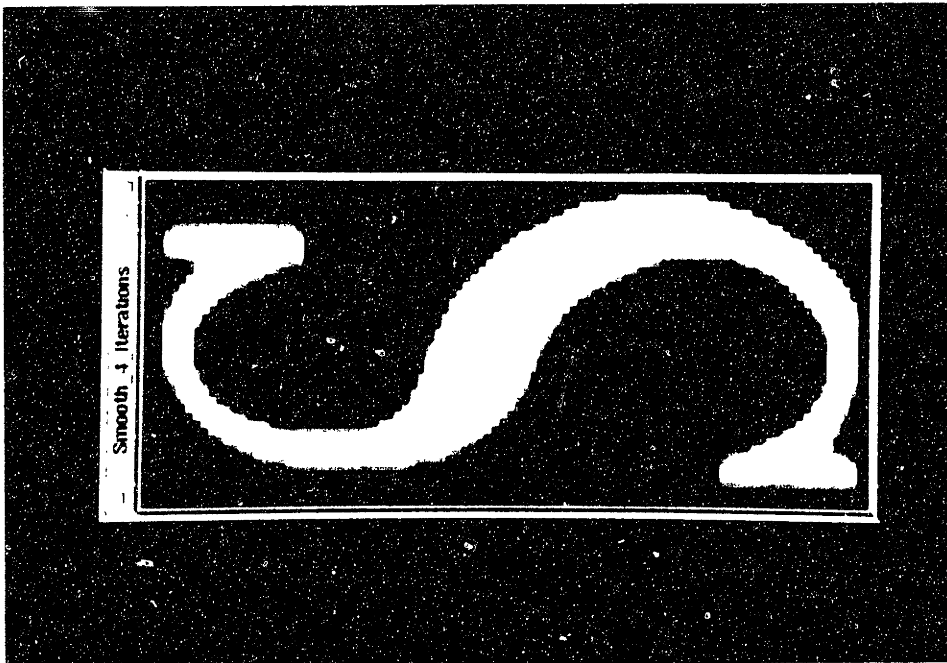


Figure B.4. Letter "S" after smoothing 4 iterations

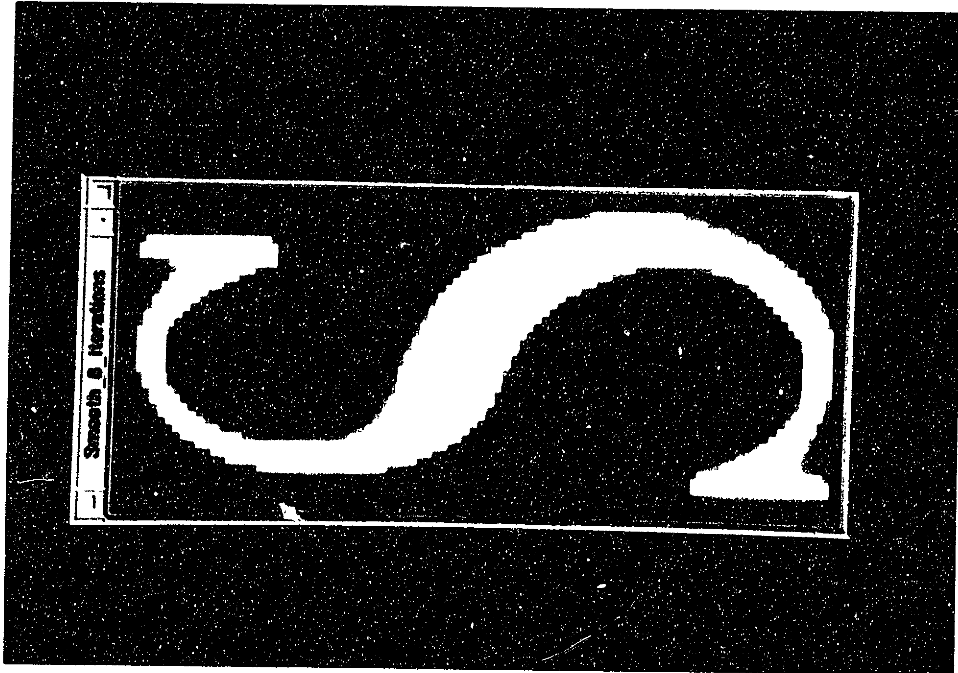


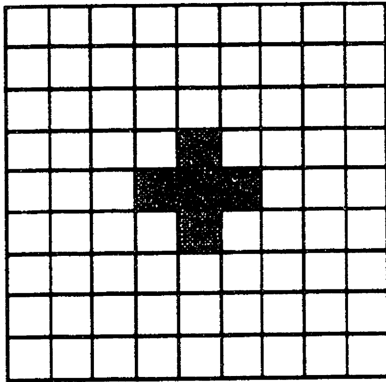
Figure B-5: Letter "S" after smoothing 8 iterations.

B.2 Windowed ROM antialiasing

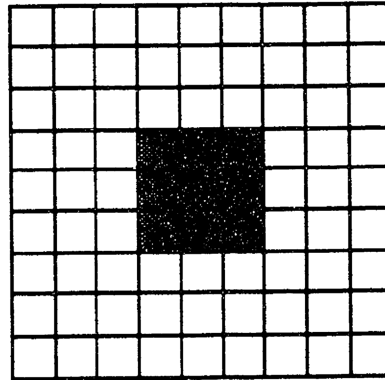
The windowed ROM antialiasing algorithm described in chapter 3 is demonstrated here. Two different learning patterns are used. First, a learning pattern consisting solely of one pixel wide lines is programmed. The window shapes are shown in diagram B-6. The pattern is composed of a circular arrangement of straight lines, spaced ten degrees apart. In the center, several lines are overlapped randomly to allow the mapping to include overlapping lines.

The algorithm was applied to test cases of randomly arranged overlapping lines, various thickness lines with one pixel steps, and the letter "S". The results are quite satisfactory for one pixel wide lines, but results from edges bordering on solid areas are inadequate.

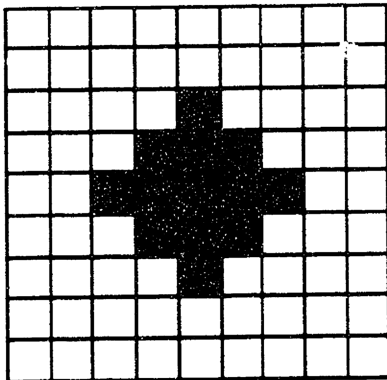
The learning pattern is changed to include areas; several regions between concentric circles are shaded. The algorithm is again applies to the test cases. The results are markedly better, showing excellent edges on all thickness lines and particularly on the letter "S".



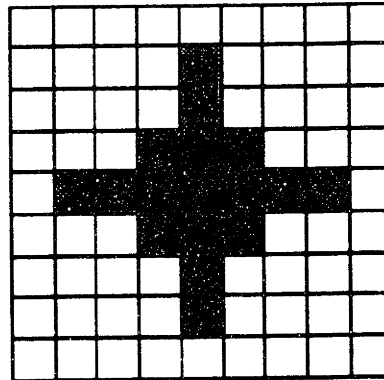
(a)



(b)



(c)



(d)

Figure B-6: Window shapes used in the windowed ROM antialiasing routine. (a) five pixel window (b) nine pixel window (c) thirteen pixel window (d) nineteen pixel window

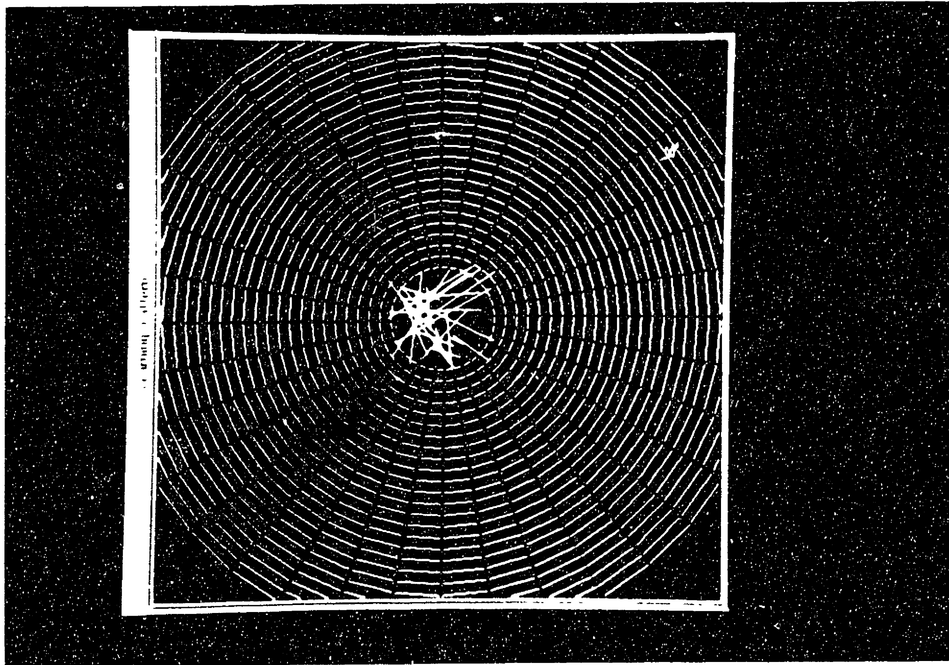


Figure B.7 [illegible] learning pattern consisting only of one pixel wide lines

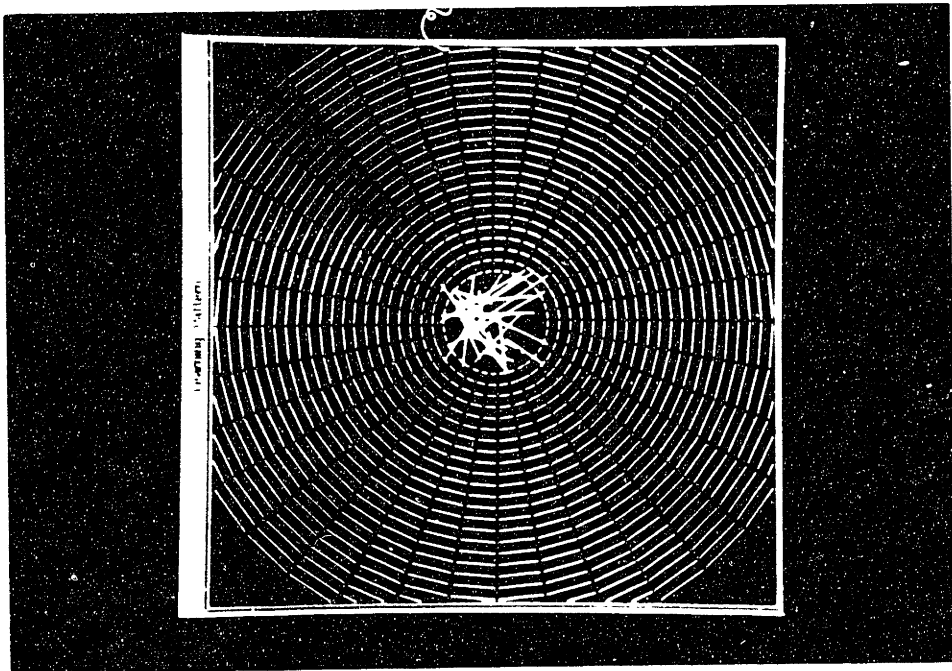


Figure B.8 [illegible] learning pattern consisting of [illegible] of one pixel wide lines

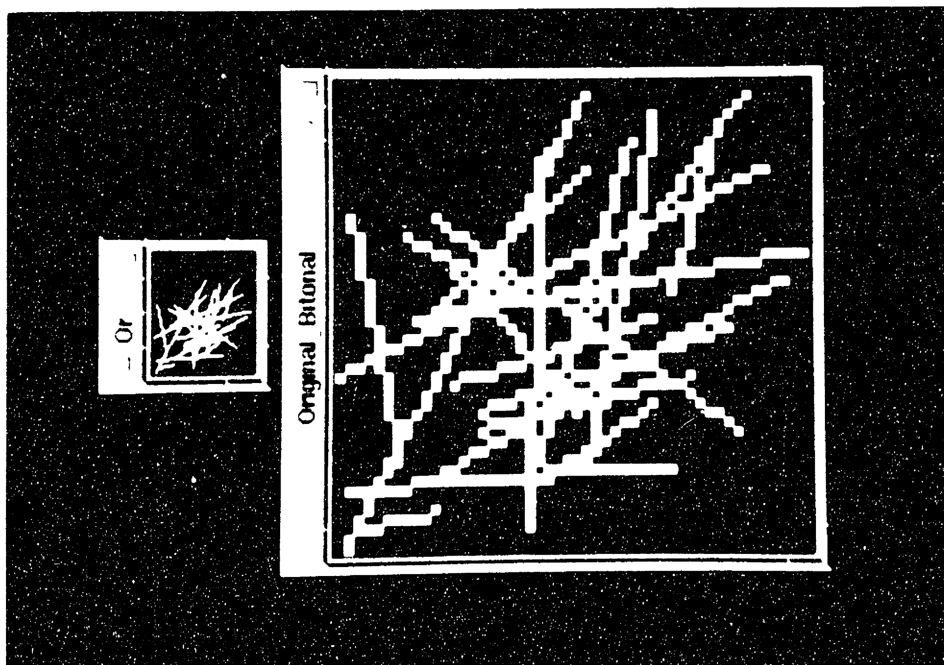


Figure B.[illegible] Overlapping [illegible] bitmap

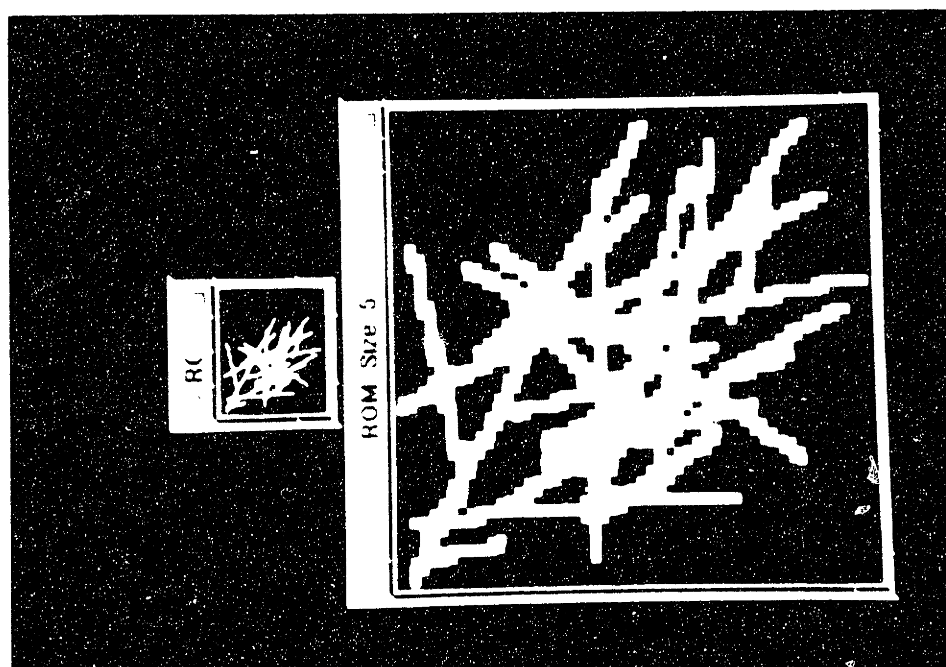


Figure B.[illegible] Overlapping lines [illegible] with a window size of [illegible]

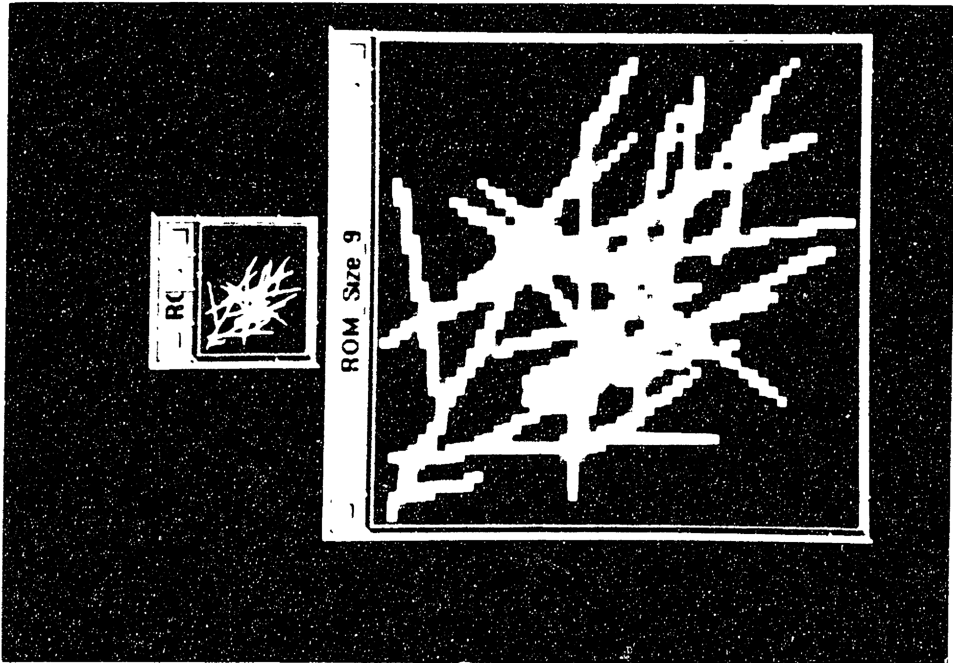


Figure B.[illegible] Overlapping [illegible] window size [illegible]

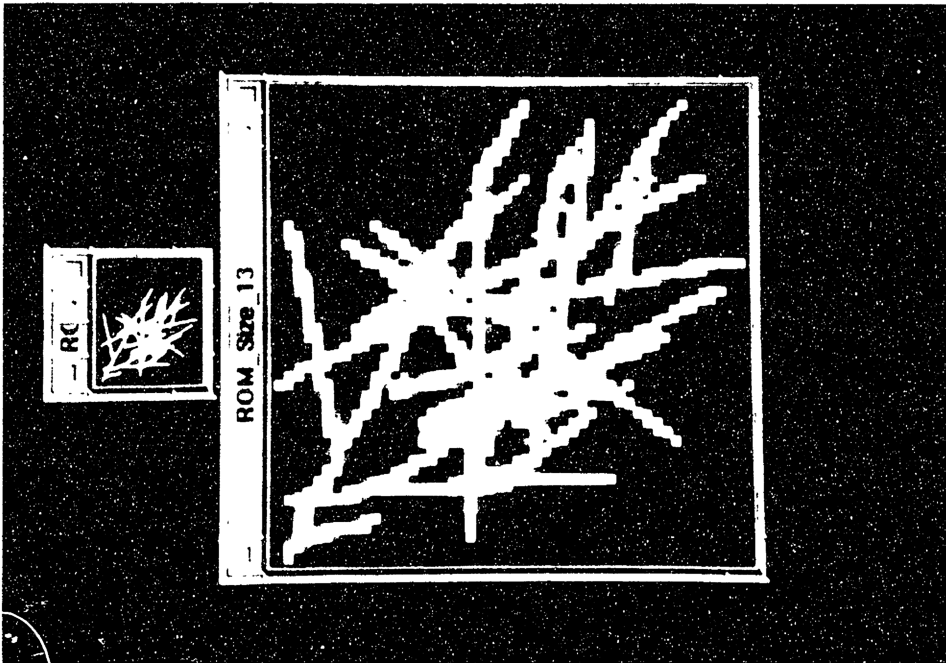


Figure B.[illegible] Overlapping [illegible] window size [illegible]

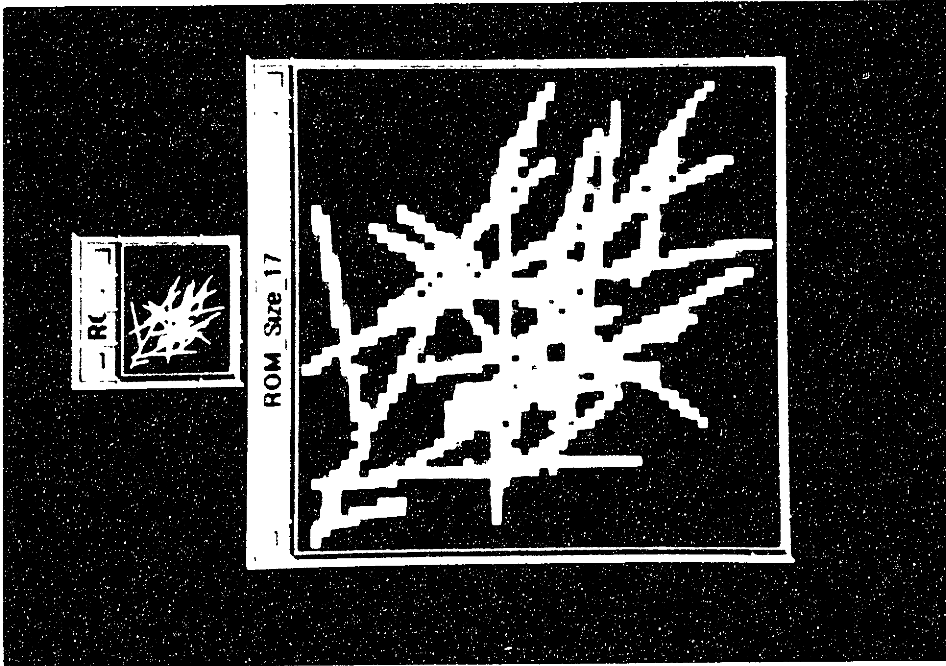


Figure B.3 Overlapping lines antialiased with [illegible]

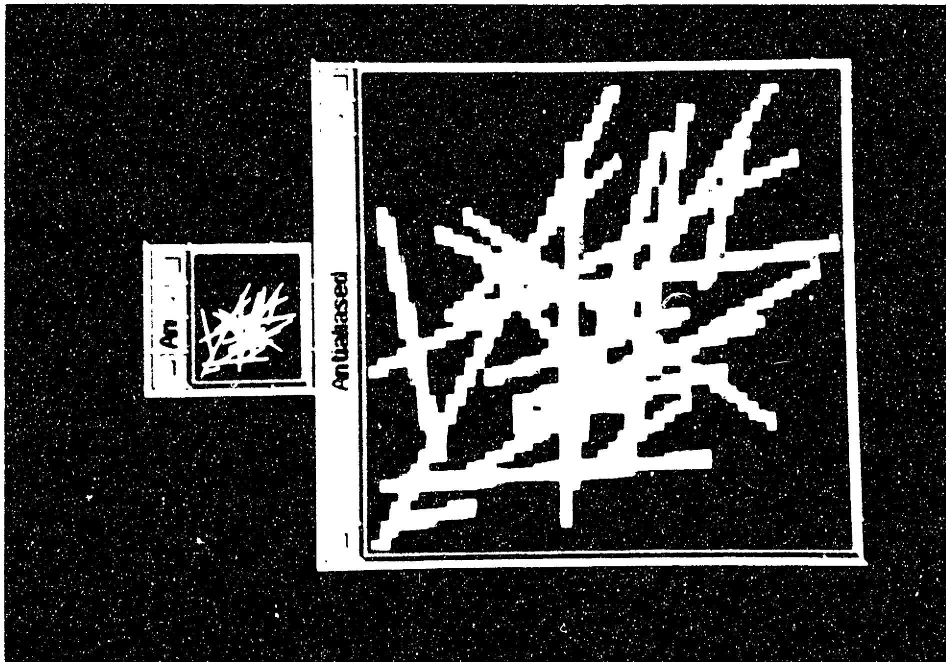


Figure B.4 Overlapping lines antialiased with [illegible] method

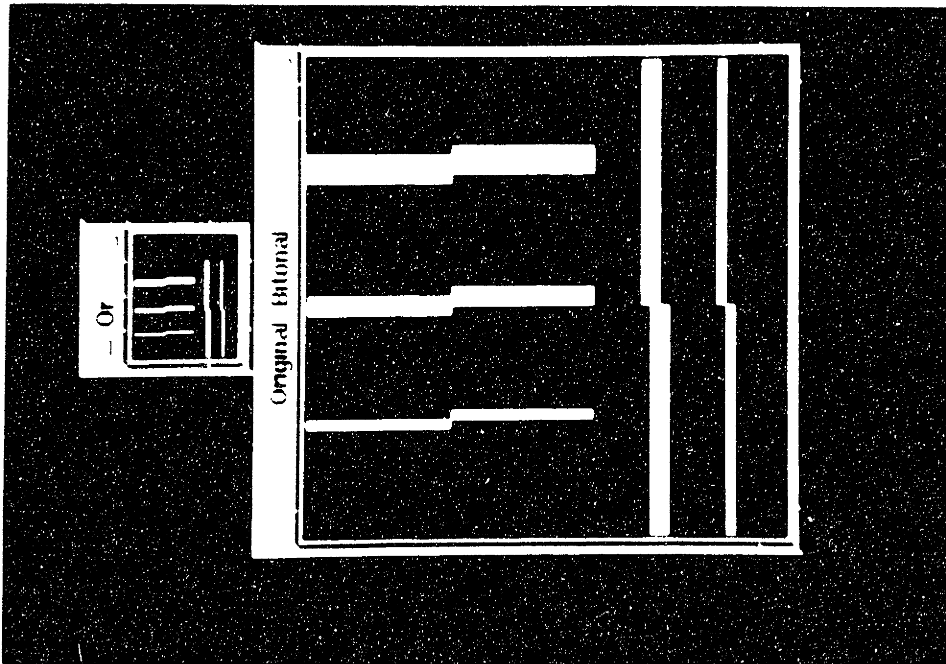


Figure B[illegible]

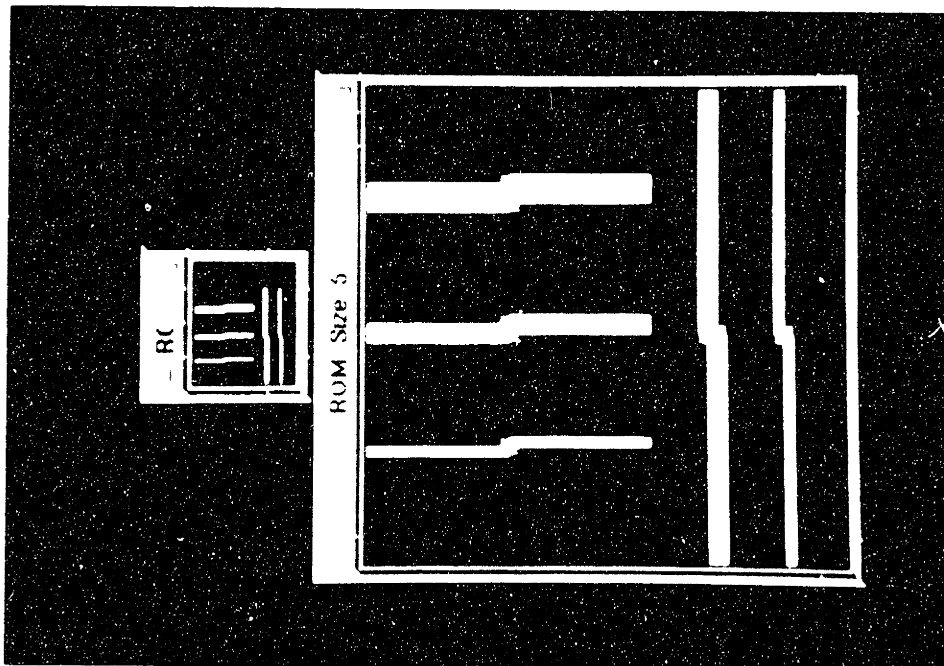
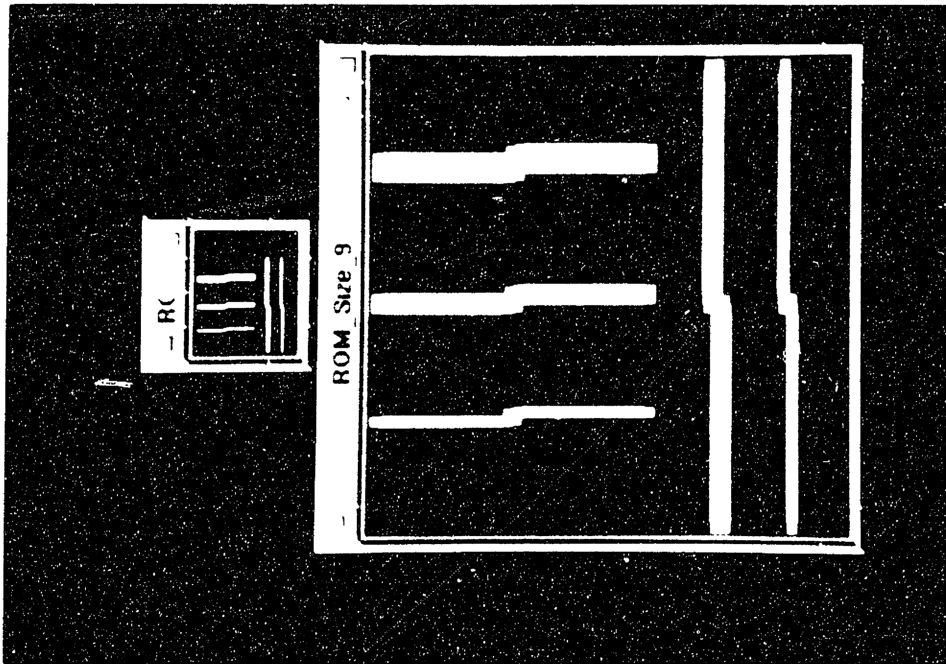
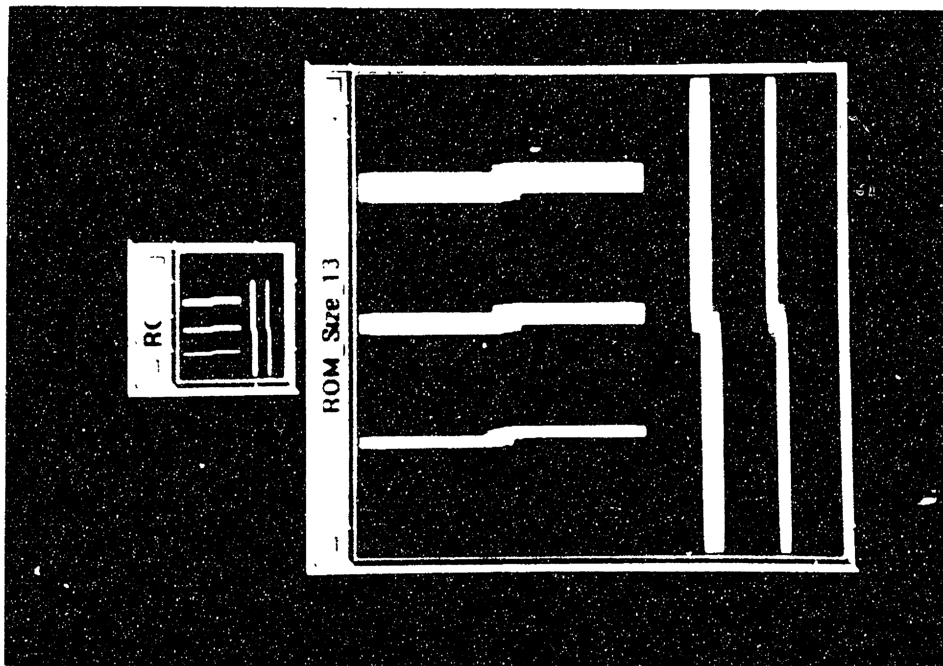


Figure B[illegible]


ROM_Size_9


ROM_Size_13

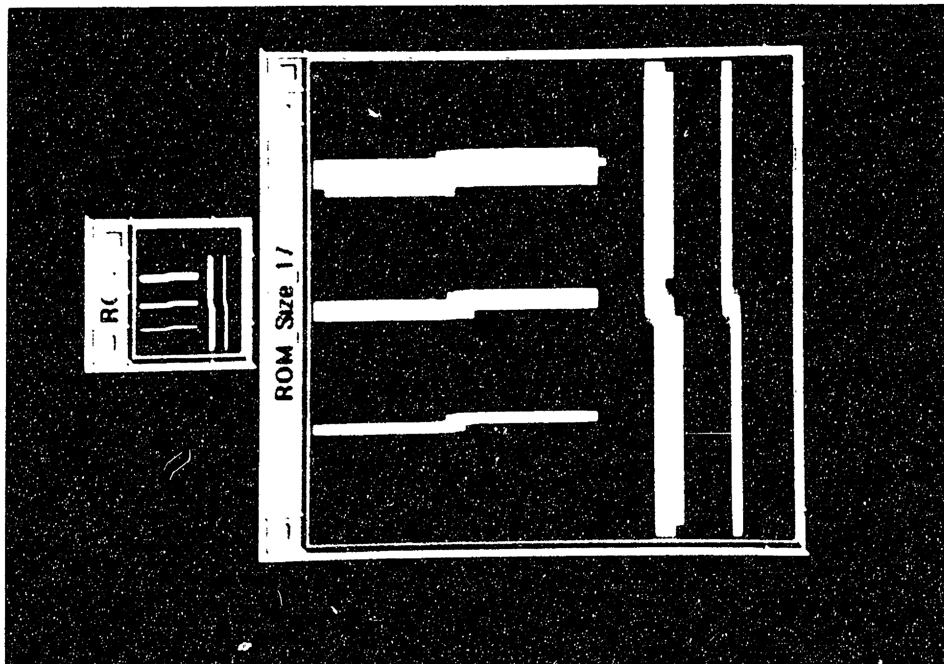


Figure B.[illegible] Horizontal and vertical [illegible]

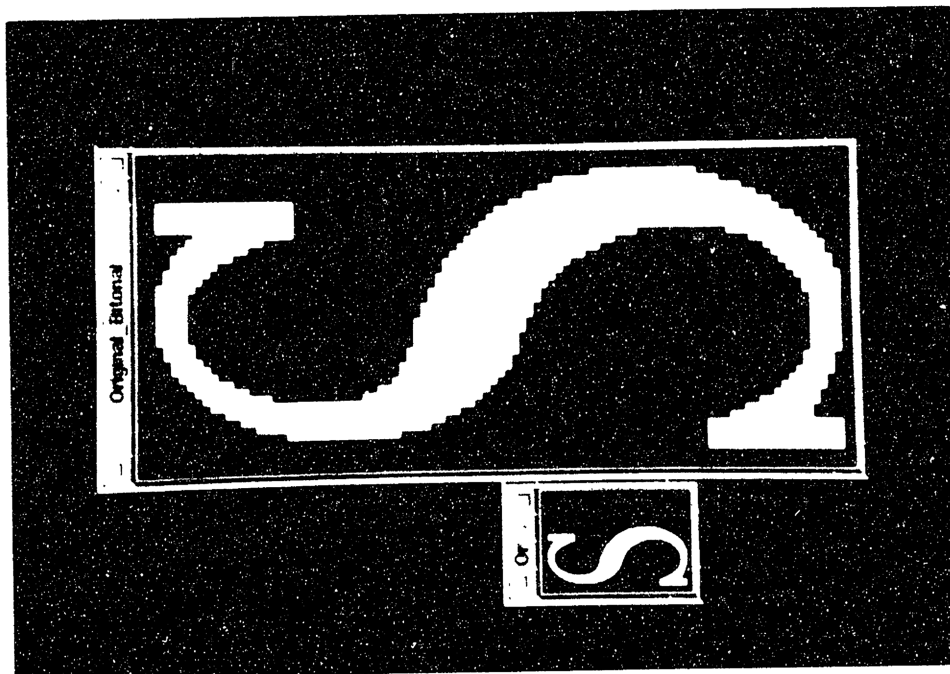


Figure B.20 The [illegible]

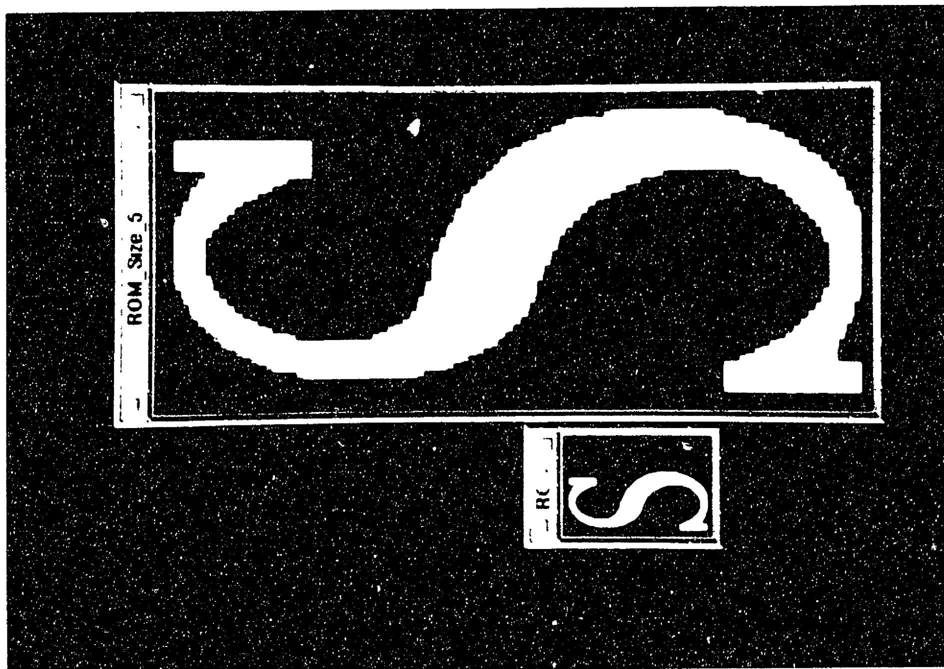


Figure B.21 The letter "S" antialiased with a window size of [illegible]

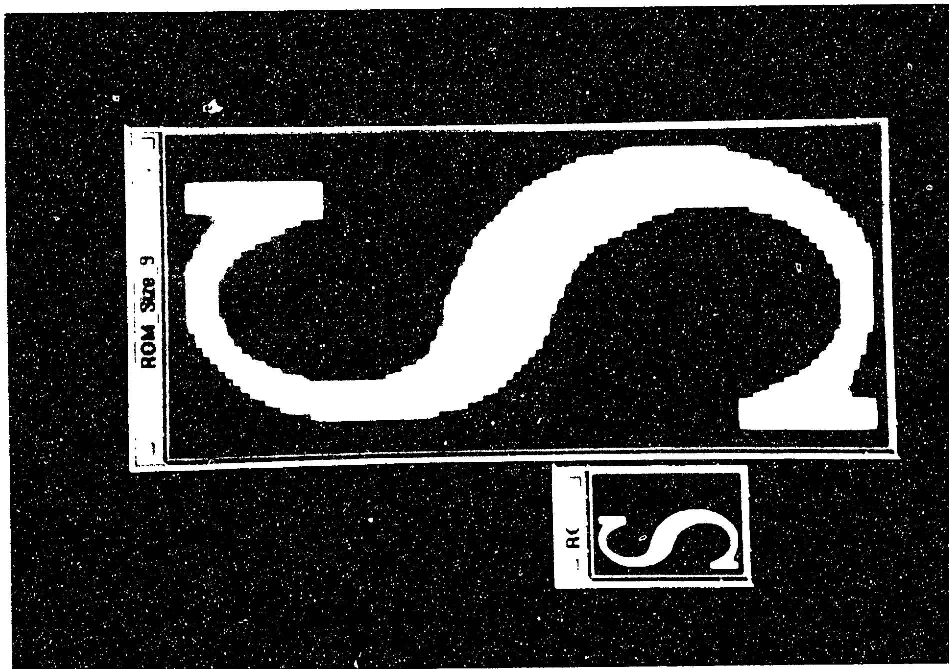


Figure B.22 The letter "S" antialiased with a window size of [illegible]

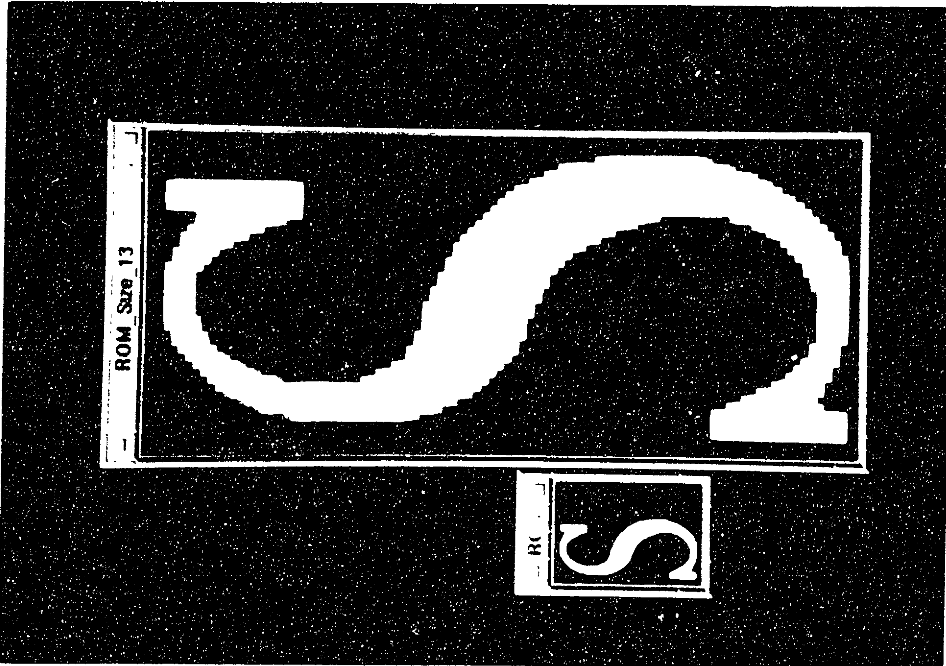


Figure B.23 The letter "S" [illegible]

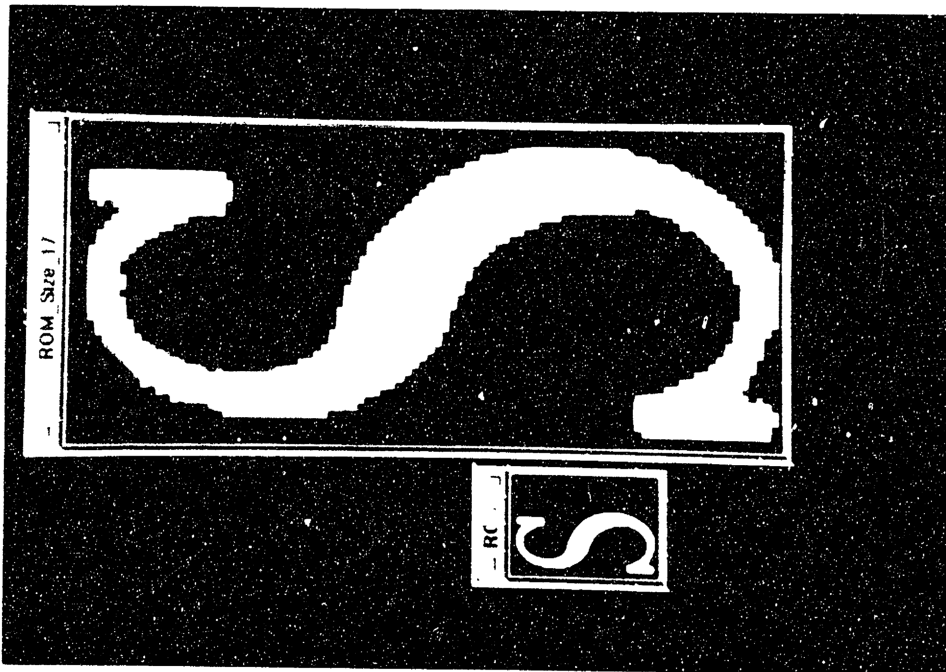


Figure B.24 The letter "S" [illegible]

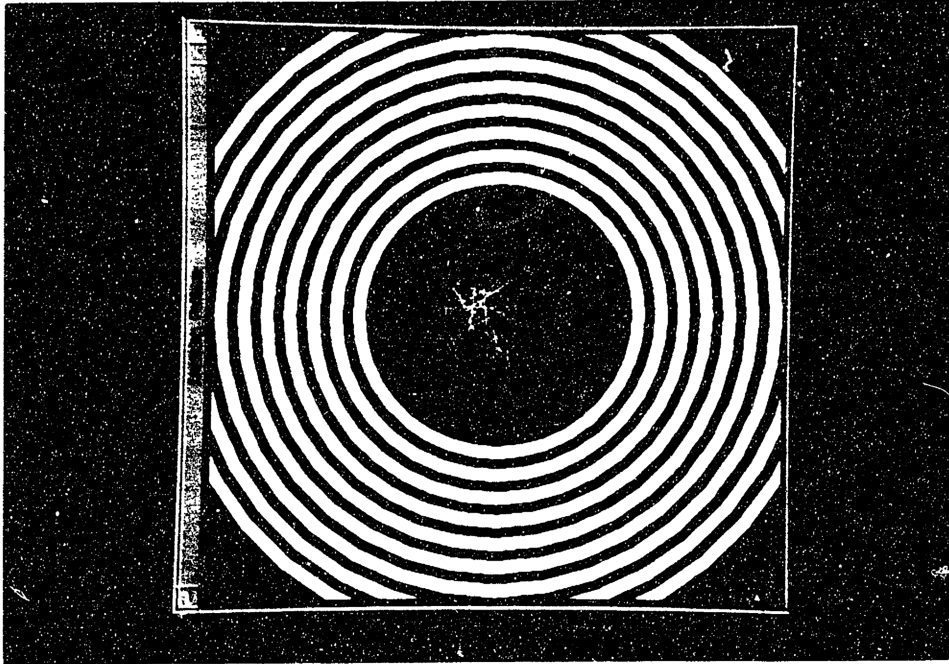


Figure B.25: New original learning pattern, consisting of areas and one pixel wide lines.

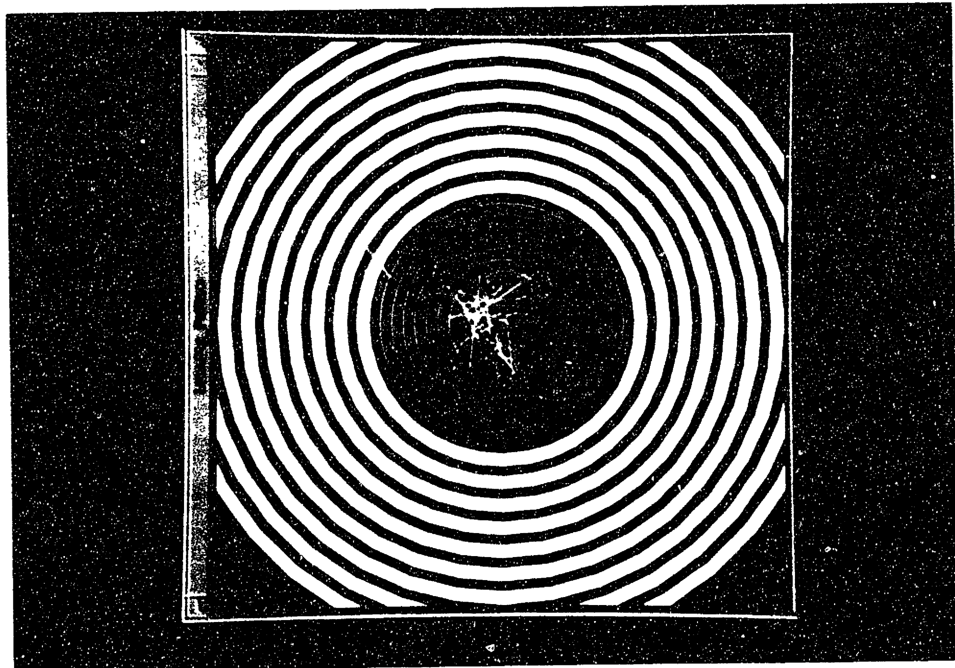


Figure B.26: New antialiased learning pattern, consisting of areas and one pixel wide lines.

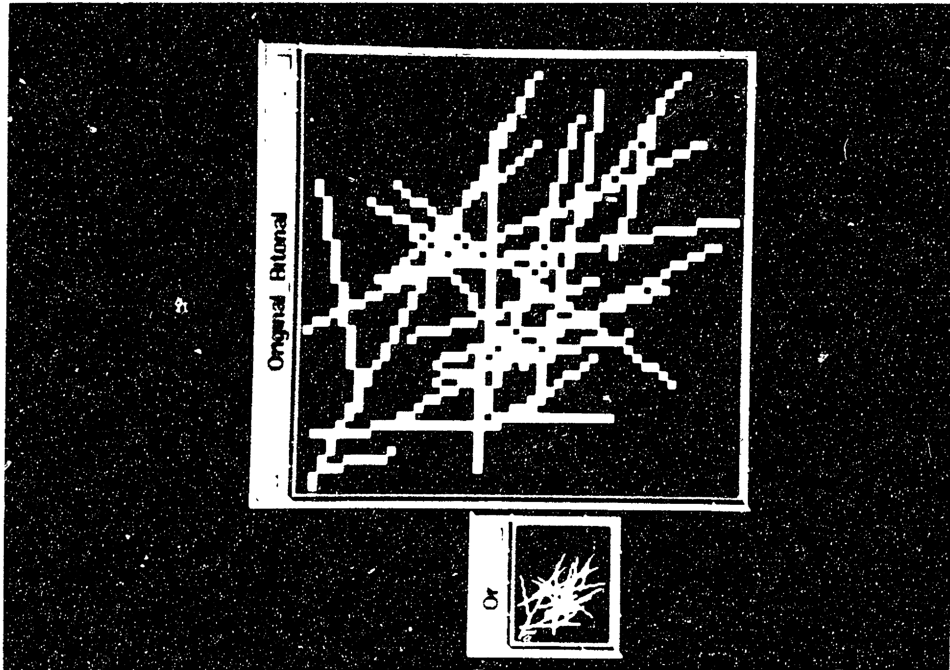


Figure B.27 [illegible]

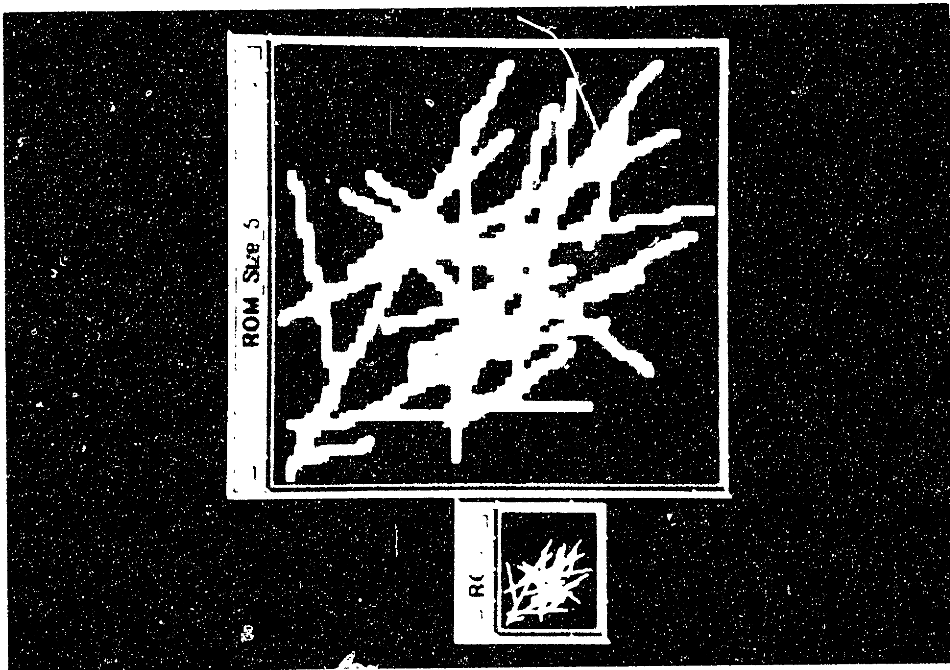


Figure B.28: Overlapping [illegible] with a window size of 5

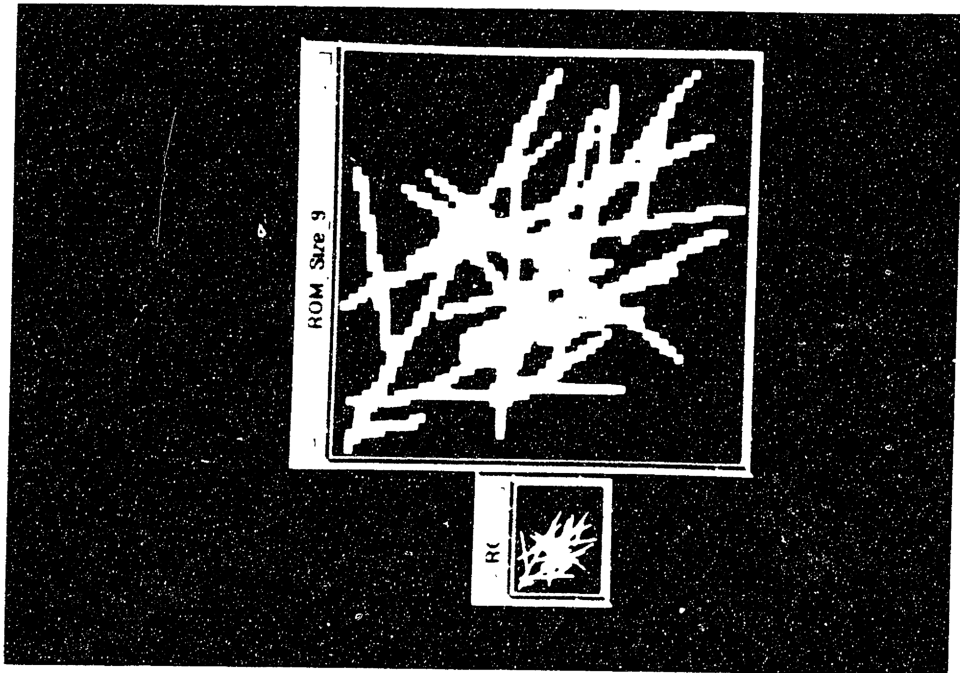


Figure B.[illegible] Overlapping lines antialiased [illegible]

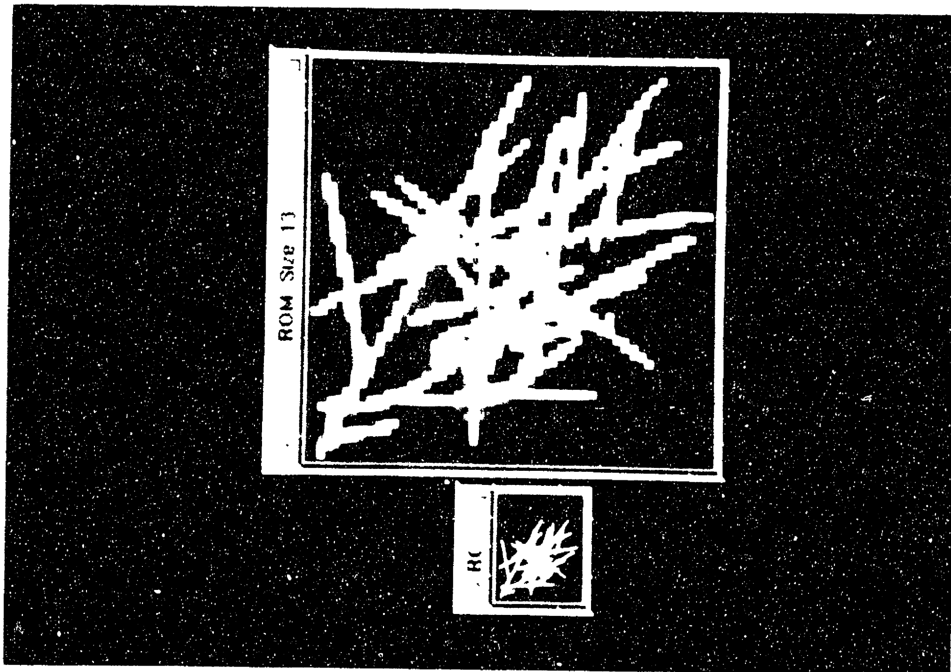
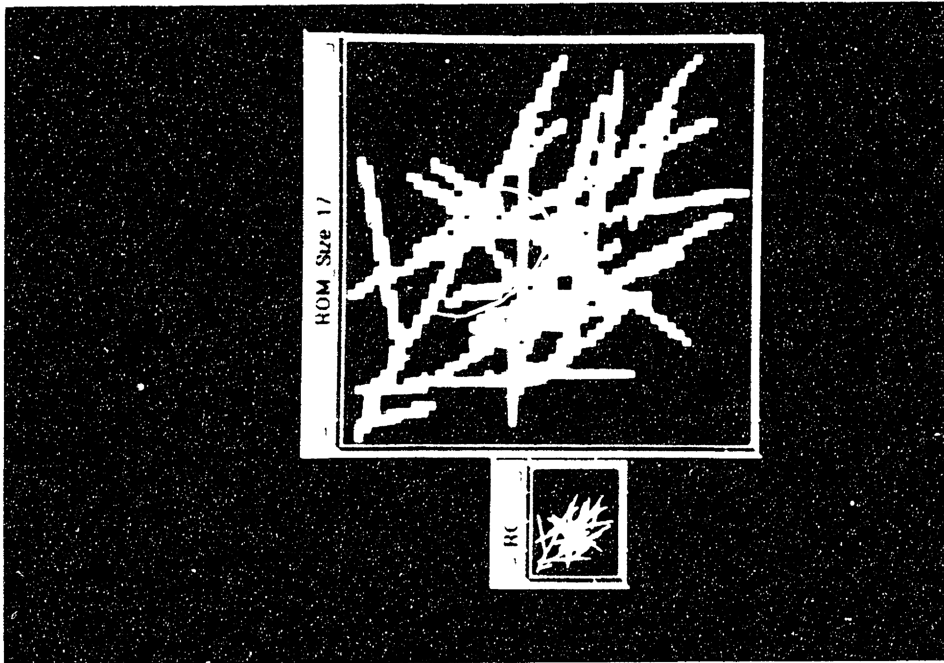
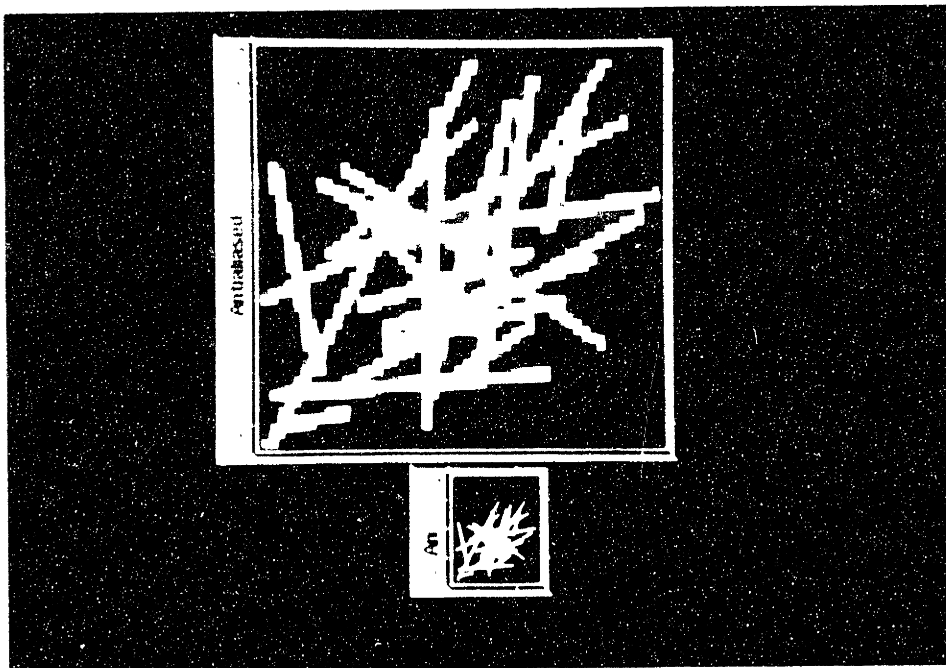
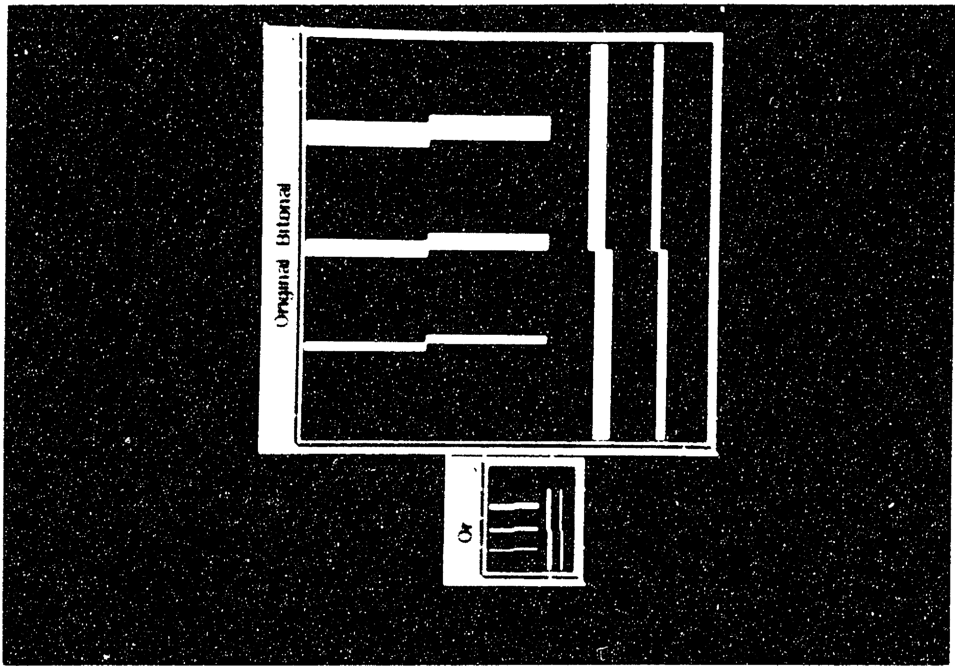
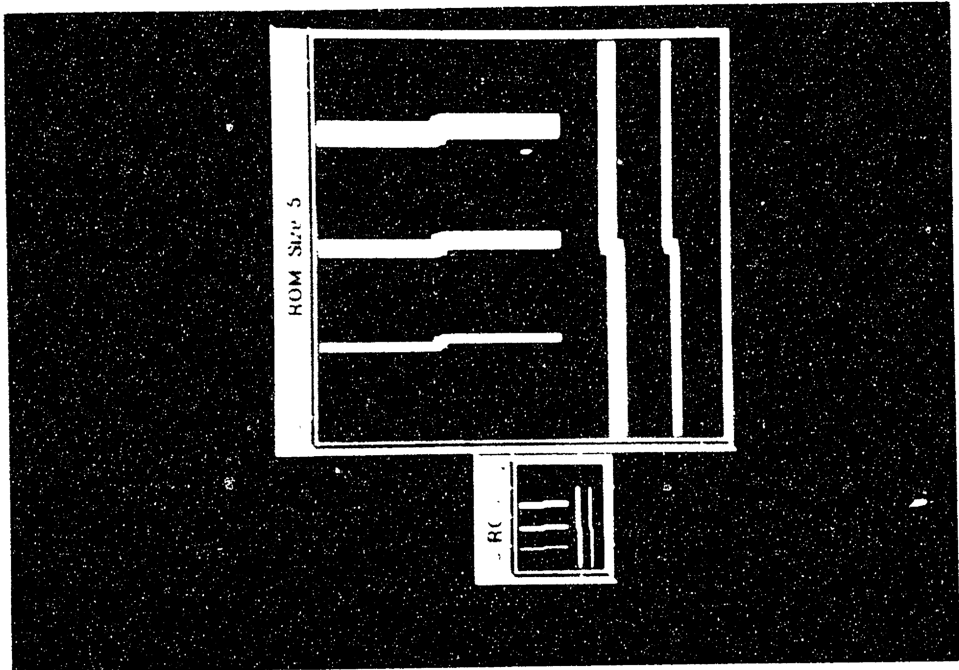


Figure B.30 Overlapping lines antialiased with a window size of 13


ROM Size 17


Antialiased


Original Bitonal


ROM Size 5

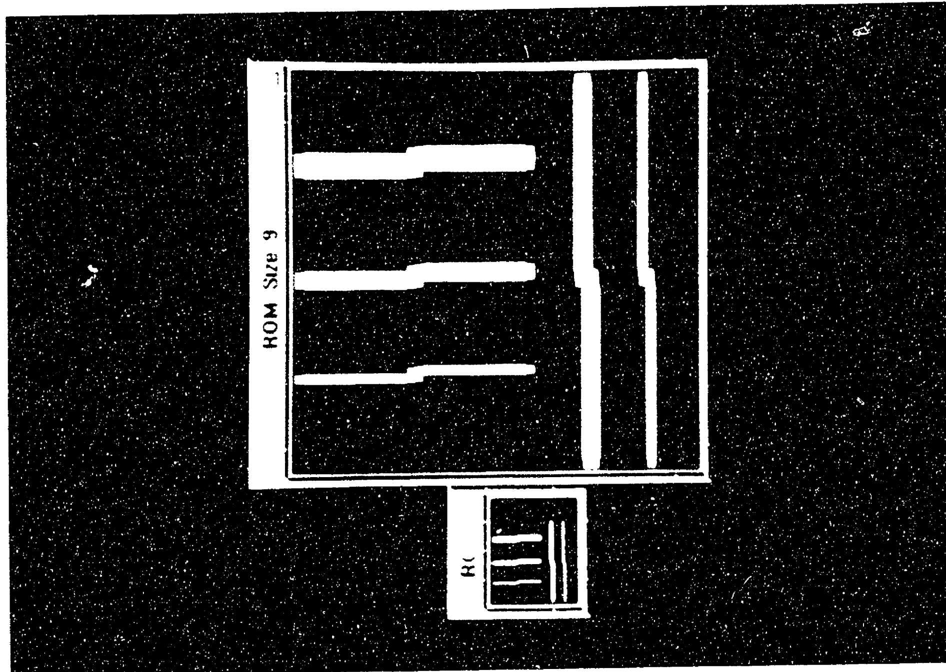


Figure R.[illegible] [illegible]

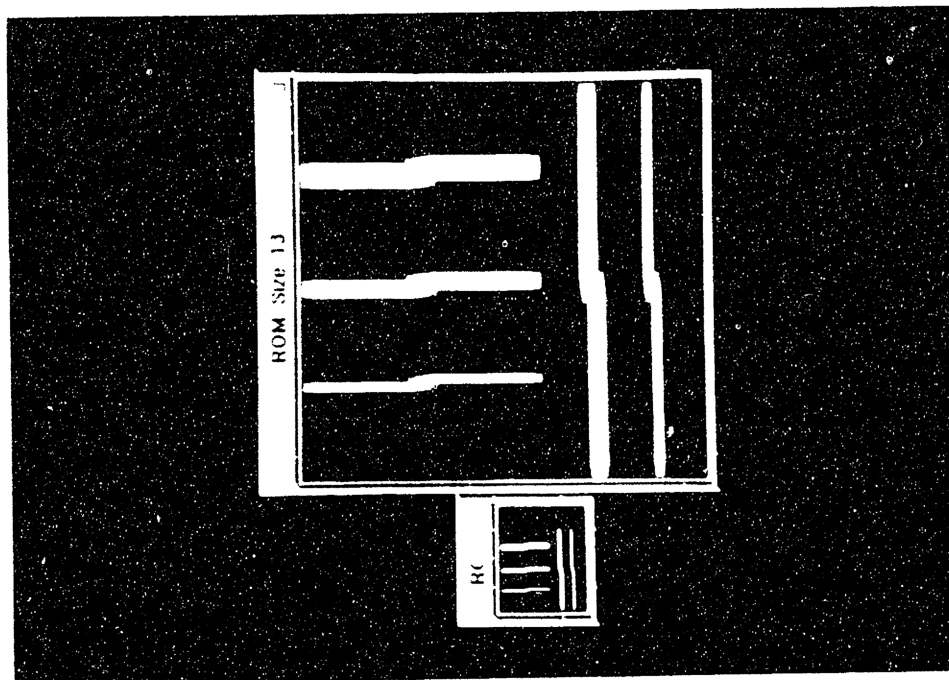
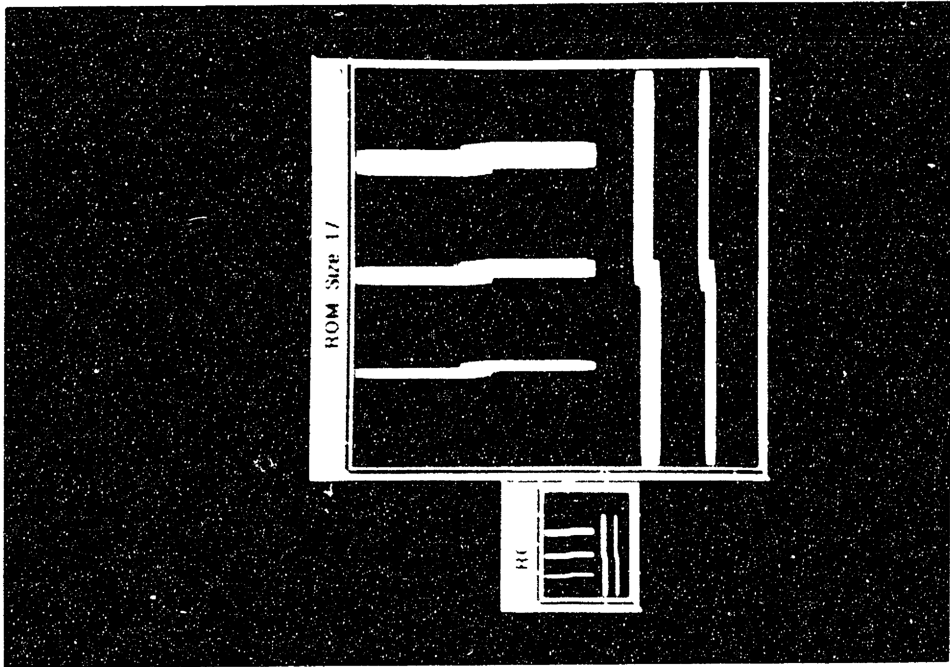
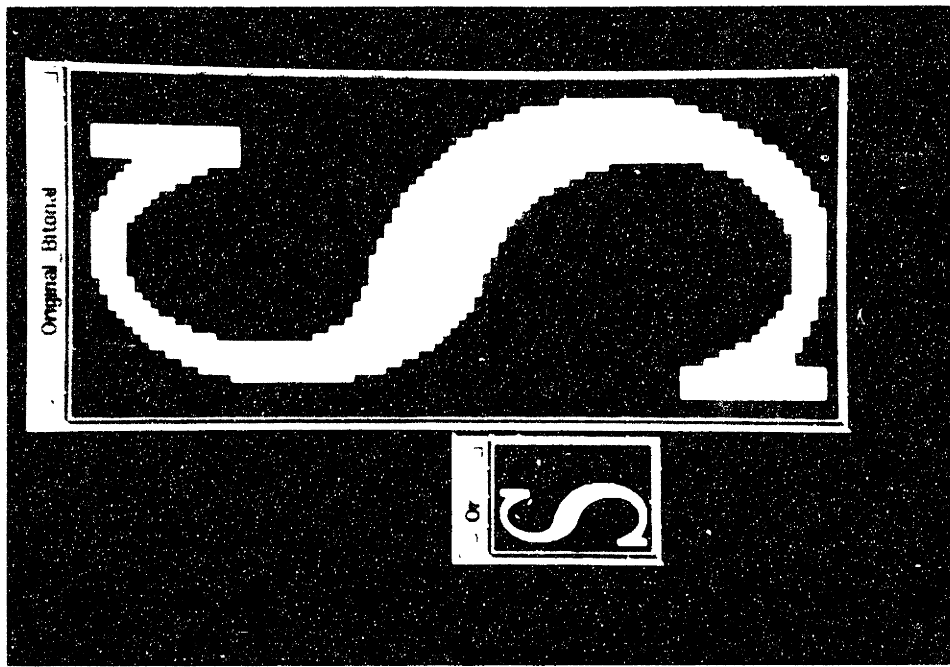


Figure R.8 Horizontal and vertical line [illegible] with one pixel step [illegible] size of 13



ROM Size 17



Original Bitonal

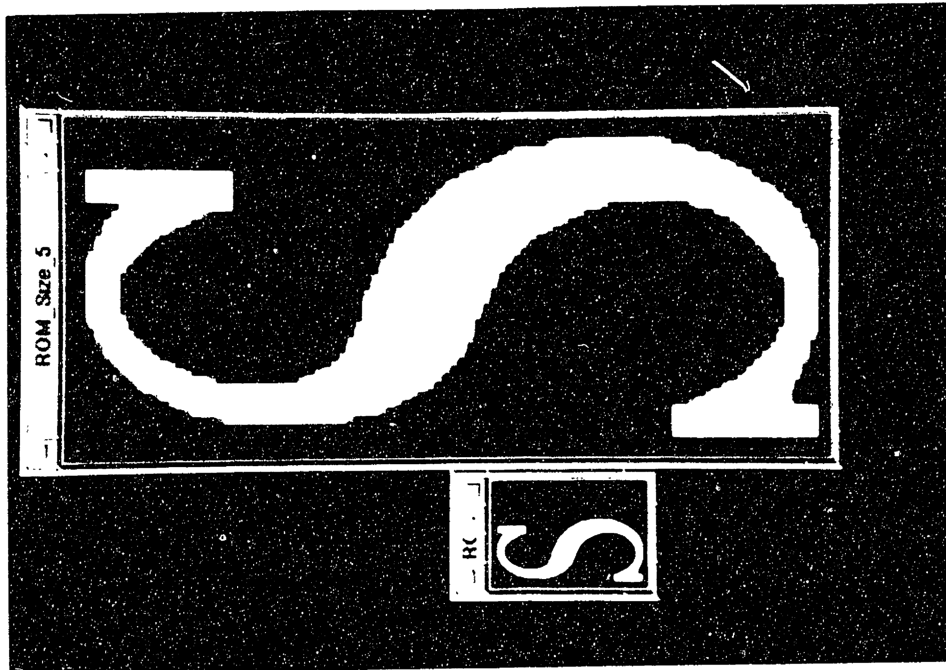


Figure B.[illegible] The letter [illegible]

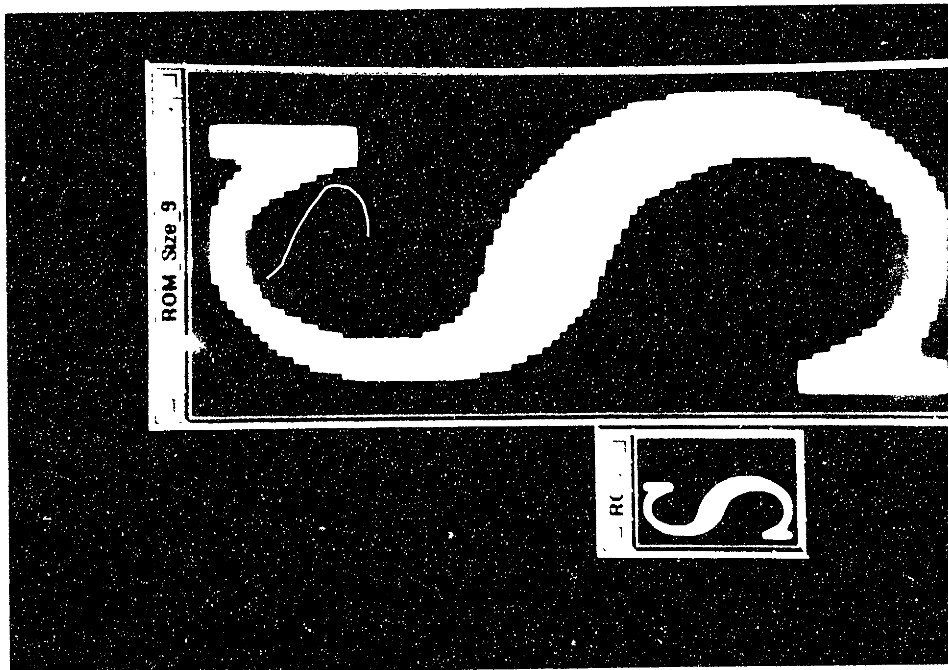


Figure B.40. The letter "S" [illegible]

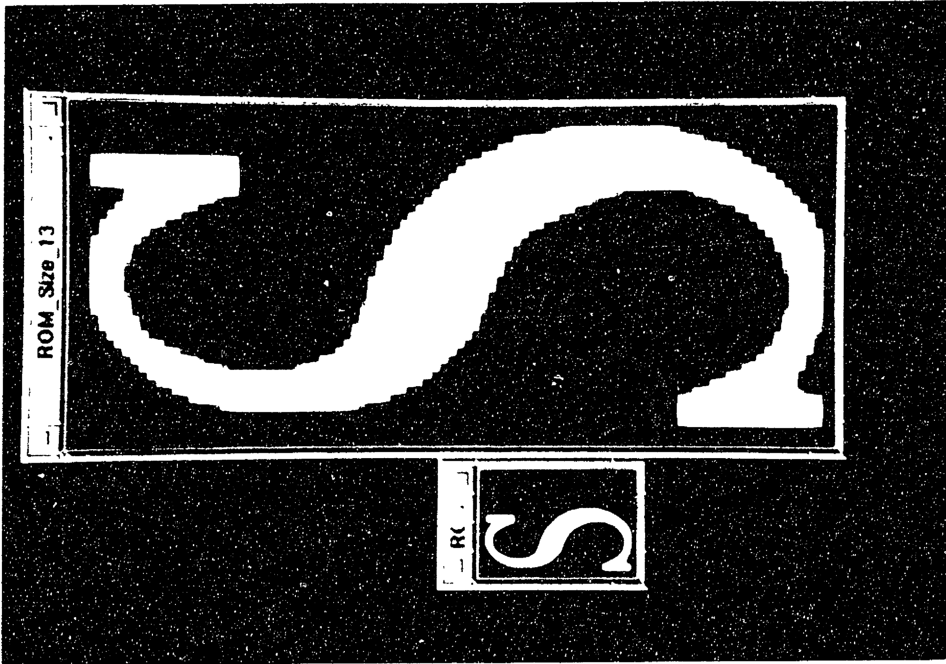


Figure B.41. The letter "S" antialiased with a window size of 13

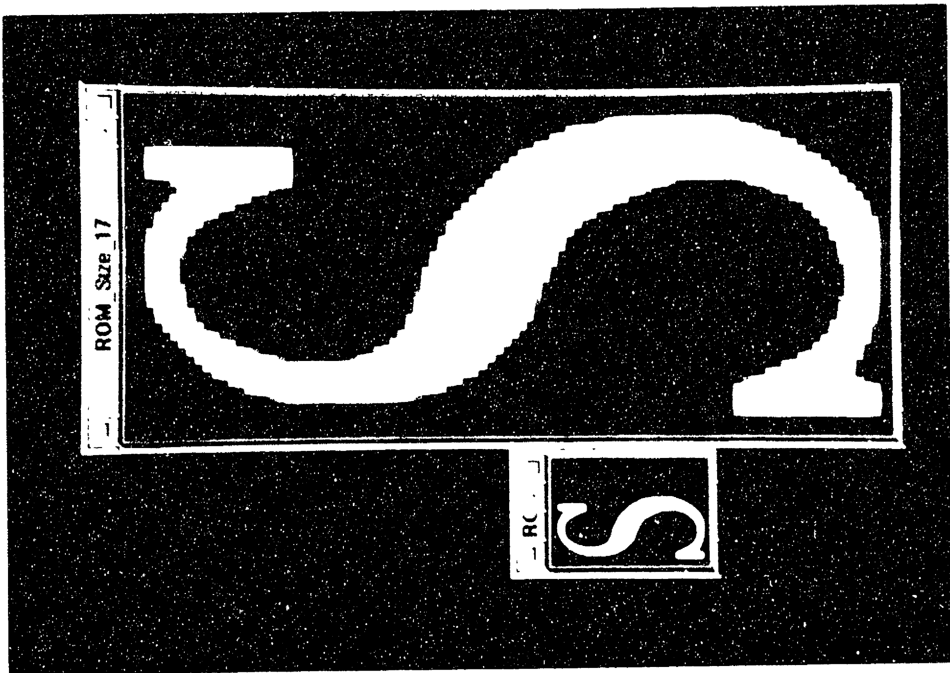


Figure B.42. The letter "S" antialiased with a window size of [illegible]. Note that the new learning pattern performs significantly better than the previous pattern.

Bibliography

[1] G. Abram, L. Westover, and T. Whitted. Efficient alias-free rendering using bit-masks and look-up tables. *Siggraph ACM*, 19(3):53–59, 1985.

[2] Huseyin Abut. *Vector Quantization.* IEEE Press, New York, 1990.

[3] Jules Bloomenthal. Edge inference with applications to anti-aliasing. *Computer Graphics*, 17(3):157–162, July 1983.

[4] J.E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, July 1965.

[5] Donald M. Burland and Lawrence B. Schlein. The physics of electrophotography. *Physics Today*, pages 46–53, May 1986.

[6] Bill Burling. Conversations with bill burling, 1990.

[7] Loren Carpenter. The a-buffer, and antialiased hidden surface method. *Computer Graphics*, 18(3):103–108, July 1984.

[8] Inan Chen. Optimization of photoreceptors for digital electrophotography. *Journal of Imaging Science*, 34(1):15–20, January 1990.

[9] Vincent Cordonnier. Antialiasing characters by pattern recognition. *Proceedings of the SID*, 30(1), 1989.

[10] L. D. Dickson. Characteristics of a propagating gaussian beam. *Applied Optics*, 8(9), August 1970.

[11] E. Fiume and A. Fournier. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. *Computer Graphics*, 17(3):141–150, July 1983.

[12] James D. Foley, Andries Van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice*. Addison–Wesley Publishing Company, Reading, MA, 1990.

[13] D. Ghazanfarpour and B. Peroche. Antialiasing by successive steps with a z-buffer. *Eurographics*, pages 235–244, 1989.

[14] Satish Gupta and Robert F. Sproull. Filtering edges for gray-scale displays. *Computer Graphics*, 15(3):1–5, August 1981.

[15] H. M. Haskal. Laser recording with truncated gaussian beams. *Applied Optics*, 18(13):2143–2146, July 1979.

[16] J. Kajiya and M. Ullner. Filtering high quality text for display on raster scan devices. *Computer Graphics*, 15(3):7–15, August 1981.

[17] Don Lancaster. Postscript insider secrets. *Byte Magazine*, pages 293–302, July 1990.

[18] Ho John Lee. Vector quantization of gray scale images. Master's thesis, Massachusetts Institute of Technology, June 1985.

[19] William J. Leller. Human vision, anti-aliasing, and the cheap 4000 line display. *Computer Graphics*, 14(3):308–313, July 1980.

[20] Avi Naiman. Some new ingredients for the cookbook approach to anti-aliased text. *Graphics Interface '88*, pages 99–108, 1984.

[21] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[22] Richard Rubinstein. *Digital Typography*. Addison–Wesley Publishing Company, Reading, MA, 1988.

[23] Christopher Schmandt. Greyscale fonts designed from video signal analysis. *Proceedings 1983 Conference of National Computer Graphics Association*, pages 549–558, 1983.

[24] John M. Sturge, Vivian Walworth, , and Alan Shepp. *Imaging Processes and Materials*. Van Nostrand Reinhold, New York, 1989.

[25] John W. Trueblood. Theory and measurement of anti–aliased line performance. *Sony Corporation, San Diego, CA.*

[26] Robert Ulichney. *Digital Halftoning*. The MIT Press, Cambridge, MA, 1987.

[27] John E. Warnock. The display of characters using gray level sample arrays. *Computer Graphics*, 14(3):302–307, July 1980.

[28] Carl F. R. Weiman. Continuous anti–aliased rotation and zoom of raster images. *Computer Graphics*, 14(3):286–293, July 1980.

[29] Turner Whitted. Anti–aliased line drawing using brush extension. *Computer Graphics*, 17(3):151–156, July 1983.

[30] G. Wyvill and P. Sharp. Fast antialiasing of ray traced images. *New Advances in Computer Sciences (Proc. CG Intl. '89)*, pages 579–587, 1989.

THESIS PROCESSING SLIP

FIXED FIELD: ill. ______________ name ______________

index ______________ biblio ______________

► COPIES: Archives Aero Dewey Barker Hum

Lindgren Music Rotch Science Sche-Plough

TITLE VARIES: ► ☐ ______________

____

____

NAME VARIES: ► ☐ ______________

____

IMPRINT: (COPYRIGHT) ______________

____

► COLLATION: ______________

____

► ADD: DEGREE: ________ ► DEPT.: ______________

► ADD: DEGREE: ________ ► DEPT.: ______________

SUPERVISORS: ______________

____

NOTES:

cat'r date

page

► DEPT: ______________ ►

► YEAR: ________ ► DEGREE: ______________

► NAME: ______________

____