Random Marks on Paper

Non-Photorealistic Rendering with Small Primitives

by

Adrian Joseph Secord

B.Math., University of Waterloo, 2000

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October, 2002

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Oct. 8, 2002

Abstract

Non-photorealistic rendering is a branch of computer graphics which draws heavily from the traditional artistic disciplines such as painting, drawing, and etching. The emphasis of this research area is to provide rendering styles to artists that communicate, elucidate and express ideas clearly and with style without necessarily reproducing every detail. This dissertation presents two general approaches to reproducing a greyscale input image with small *primitives*: strokes, stipples or hatch marks. The first approach probabilistically places primitives on the output canvas and achieves very fast, "sketchy" renderings. The second approach, complementary to the first and based on weighted centroidal Voronoi diagrams, places each primitive carefully in relation to the others and minimises unnecessary overlap between primitives. The resulting drawings have a very careful appearance and reproduce the input image faithfully.

Contents

List of Figures

Preface

Parts of Chapter 3 were previously published in [23]. The mathematics of Sections 3.1.1, 3.1.2, and 3.2.4, along with the core ideas of Section 3.3 were first laid down by Professor Wolfgang Heidrich. The implementation of the system described in Chapter 3 was achieved by Lisa Streit and the author. Parts of Chapter 4 were previously published in [22]. The entirety of that work is the responsibility of the author.

Acknowledgements

I would like to acknowledge the guidance of my supervisor, Prof. Wolfgang Heidrich, who gave me excellent and honest advice on the realities of academia.

Lisa Streit was a warm and friendly mentor in graphics. She was also largely responsible for holding the Imager lab together during its recent chaos, which I appreciate.

Brian de Alwis, Alex Brodsky, Dima Brodsky, Andrea Bunt, Steph Durocher and Brian Winters were my "wise men", giving me excellent advice when they could and sharing beer when they couldn't.

I have many friends, family and loved who are continuously supportive across the thousands of kilometres in this country and others. They deserve more than a few words in a thesis.

Vicki Weafer supported me unconditionally and with love from afar (5871 km). She listened to me when I needed to talk, whether it was about personal problems, grad school or Voronoi diagrams. Amy Norris helped me along every single day, both with research and staying alive. She made life in Vancouver a very happy thing. Mike Cline was a very good, if new, friend. Finally, in addition to being excellent friends, both Amy and Mike taught me to fear thesis-writing before it was too late. Thanks!

To Mom and Dad.

Part I

Thesis

Chapter 1

Introduction

Photorealism has traditionally dominated much of computer graphics research. Using the physics of light, surfaces and materials, we progress steadily in generating models and algorithms that mimic nature closely. However, photorealism is not always desirable, and artists working in traditional media only occasionally use photorealism. The goals of photorealistic computer graphics are perhaps too encompassing. For example, the ultimate photorealistic renderer is perfectly capable of generating images and animations for cats or extra-terrestrials. While this is a worthy goal in itself, more often the focus of the artist, graphic designer and typographer is to communicate an experience, emotion or idea to the *human* observer.

Artistic techniques in traditional media such as pen-and-ink, water colour, or pastel abstract detail to focus the viewer's attention. For example, a hand-drawn portrait will nearly always include more detail in the subject's face and hands than in the background, because these features are important to human viewers. It is relatively rare to illustrate everything in a scene, simply because some aspects have higher priority than others and the human observer has limited attention. Abstraction, however, is a very difficult goal since to be effective it requires intent and knowledge of human attributes. Computer abstraction of human scenes belongs more properly to artificial intelligence than computer graphics.

However, many of the low-level artistic techniques can be useful to computer graphics since they rely on relatively simple characteristics of the human visual and

related systems. For example, human vision places heavy emphasis on edges and silhouettes, and drawing pictures using only edges gives rise to styles such as cartooning. While computers can be used with some success to generate cartoon-style animations, the intent and focus of such works must still be contributed by a human artist. Having said that, cartoons are a labour-intensive style and a computer-generated cartoon style is useful. In particular, in interactive applications such as immersive environments, there is no other way to generate cartoon styles than to use the computer.

Such low-level artistic techniques have begun to be explored recently in the computer graphics literature. While many have focused on generating images from 3D models or user-driven systems, we will examine automatic image-based algorithms that mimic several low-level techniques similar to etching and stippling. Using simple artistic primitives, we automatically create engaging drawings that reproduce the tone of arbitrary images. Furthermore, through selection of algorithms and parameters, it is possible to trade image quality for speed, resulting in real-time performance for animation or video.

In particular, we have developed a fast probabilistic method that places small arbitrarily-shaped primitives on a blank canvas such that the overall tone reproduces that of an input image. The use of the theory of probability density functions allows us to compensate for the overlap that occurs between primitives. In addition, for roughly point-shaped primitives, we can modify a coarse placement of primitives such that overlap is minimised and all primitives are evenly spaced. The resulting high-quality drawing is accomplished with centroidal Voronoi diagrams, which should be extendible to more complicated primitive types.

Chapter 2

Related Work

2.1 Traditional methods

Traditionally, methods such as pen-and-ink have been used illustrations for books and scientific papers. The amount of detail to depict in these illustrations is easily controlled and the style blends well with diagrams and other forms of visual information. Pen-and-ink and stippling illustrations scale and reproduce particularly well, even with low-quality methods such as photocopiers. For current digital work, if the illustration is encoded using a vector format, then details remain sharp at all levels of magnification, unlike discrete images. An easily-accessible standard reference to pen-and-ink is that of Guptill [5], while the more specialised literature on scientific illustration is well-represented by the standard texts by Wood, Jastrzebski, and Hodges, respectively, [31, 12, 10].

2.2 Geometry-based methods

Saito and Takahashi introduced the concept of *G-buffers*, additional render buffers that contain information such as object IDs, object parametric coordinates, perspective depth, world coordinates and normals [21]. The authors used image processing techniques on these additional buffers to draw contour lines, silhouette and internal edges, and hatching lines from 3D rendered models. The examples presented were

mostly concerned with emphasising information content in renderings. Our hatching method of Chapter 3 uses object ID's and normal buffers to enable additional styles if available.

Winkenbach and Salesin truly introduced pen-and-ink illustration to computer graphics in 1994 [28]. The authors review pen-and-ink, emphasise the challenges, and describe a system for computer-generated pen-and-ink. Tones are rendered using prioritised stroke textures, which are clipped using a 2D BSP tree, and outlines are drawn to show boundaries. Strokes are straight lines with additional waviness and pressure functions to perturb the path and line thickness. Prioritised strokes textures are sets of related strokes that add together to display the texture at increasing tone densities. They also include commonly-used techniques such as indication to improve the quality of the output. The results are some of the highest-quality images in NPR to date. Our hatching method, while technically of the same family of illustration as Winkenbach and Salesin's work, uses short strokes exclusively and is more "loose" and "free".

2.3 Halftoning and dithering

Ulichney introduced the concept of "blue noise" dithering to the halftoning and dithering community [26, 27]. Blue noise has a power spectrum with no power in frequencies below a particular cutoff frequency. That is, blue noise is high-pass filtered white noise[1]. Blue noise is perceptually useful because low-frequency components can appear as structure to the eye. This means that blue noise distributions are relatively good substrates for image approximation algorithms, since they do not impose additional structure of their own on the image. Error diffusion

[1] Blue noise is named as the complement to pink noise, which has no power in the frequencies *above* a particular cutoff frequency.

dithering is an example of a method with moderate blue noise properties. The point distributions of Chapter 4 have been shown to have blue noise properties [2]. In addition, we expect that the stippling methods of Chapter 4 will produce similar results to blue noise dithering if the stipple size is forced to be extremely small, that is, on the order of a single pixel. The particular methods used by Ulichney, however, are not connected to those presented in this dissertation.

2.4 Image-based methods

Haeberli describes one of the first NPR techniques to use an input image [6]. Our methods of Chapters 3 and 4 follow this tradition. His technique uses a colour input image which is approximated with a set of straight coloured strokes. The strokes are roughly placed by the user with the mouse. The orientation can either be fixed by the user or extracted from the current mouse movement direction when a stroke is placed. The size of the stroke can again be either set explicitly, or determined from the current mouse speed. Faster speeds place larger strokes in keeping with the spirit of a fast, loose, sketch. The orientation of strokes is either user-specified, determined from the current mouse direction, or from the gradient of a lowpass-filtered version of the input image. Haeberli anticipates many techniques to be used in NPR, including using Voronoi regions as primitives, attaching strokes onto 3D geometry, and using relaxation techniques[2].

Turk and Banks, working from the scientific visualisation field, detailed a method for placing streamlines over a 2D vector field [25]. Streamlines are lines that are tangential to the underlying vector field at every point. They focused on placing the lines evenly in the image to prevent unwanted visual artifacts from

[2]Haeberli also emphasises the flexibility of using images as input. He calls pre-modifying the input image "adding spice" or "Video-Sodium Glutamate (VSG)".

detracting from the viewer's understanding of the vector field. In particular, they aimed for even overall coverage of lines and used tapering to reduce the jarring effects of a line terminating. They also showed how to express other field attributes in the line appearance. This relates to the current dissertation because it attempts to obtain good spacing of lines, much as we do for stipples in Chapter 4. Also, our fast probabilistic method of Chapter 3 generates a non-uniformly sampled vector field of positions and orientations, which could be treated with Turk and Banks' method. The method is an iterative energy-minimisation method which uses a filtered version of the current output as an energy measure. A set of operators that can improve the energy (lengthening lines, merging lines, etc.) is introduced and applied randomly to the current set of lines, but only energy-decreasing changes are kept. A drawback of this method is that the convergence is quite slow. A more direct approach to this problem is attempted by Jobard and Lefer [13].

Deussen *et al.* make good use of centroidal Voronoi diagrams to generate stipple drawings from a grayscale image and user input [2]. The grayscale reference image is first dithered to produce stipple positions, and the stipple positions are then relaxed using Lloyd's algorithm (see Section 2). However, during the relaxation step, no reference is made to the underlying input image, so Lloyd's algorithm would spread the stipples evenly across the image if not checked. To avoid this, the stipple positions are constrained to stay with user-defined regions of the image. This allows the user to control the diffusion of the stipples and produce sharp edges. The results are quite attractive, but require significant user input. In comparison, our method of Chapter 4 integrates the input image into the relaxation step, which allows us to dispense with the tiresome user entirely.

In a very similar technique, applied to a different domain, Hausner produces simple decorative mosaics from a colour input image [9]. The tiles of the mosaic are square in shape and aligned with a direction field. Again, Lloyd's algorithm is

used to avoid overlap between the tiles, but with the novel use of a Manhattan (or lattice) metric, corresponding nicely to the placement of squares. The user defines edges in the image that the tiles are not allowed to overlap, and the direction field can be generated from the edges by examining the gradient of the distance field of the edges. The final colour of a tile is point-sampled from the input image at the tile's centre.

Chapter 3

Probabilistic Drawings

We need to represent continuous tone images with a set of spatially- and tonally-discrete primitives. We wish to place more primitives in areas where the input image is dark and less where it is light. However, for the first pass at the problem, the exact placement of every primitive does not matter. This intuitive statement is the motivation for treating the input image as a 2D probability density function (PDF). We will generate random positions for the primitives according to the darkness of the input image.

Formally, a PDF describes the differential probability that some event will occur at any point. If $q(\mathbf{x})$ is a 2D PDF, then the probability of some event occurring within an infinitesimal region dA at $\mathbf{x}$ is $q(\mathbf{x})\,dA$. Note that there in no notion of probability at a point — that probability is always zero. Only probability of a region makes sense in this continuous setup. In general, the probability of an event occurring in some region R is $\int_R q(\mathbf{x})\,dA$. The "events" described by our PDF's will be the placement of a primitive at that location.

We generate a PDF from the input image and then generate a set of samples distributed according to that PDF. The samples come from a uniformly-distributed set that are transformed to fit the target PDF. These samples are the positions of the primitives which make up the output drawing.

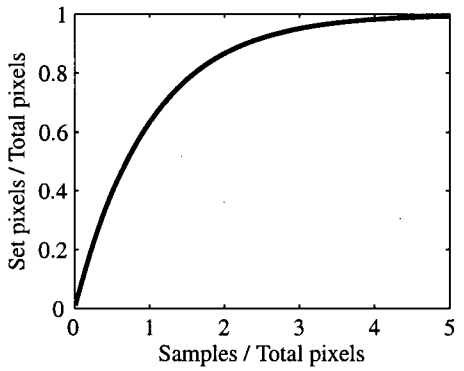


Figure 3.1: Ratio of samples selected uniformly at random versus total number set.

3.1 Deriving a PDF from an image

In stippling, hatching and other applications, the samples that we would like to place have finite area. It is possible, and with increasing density quite likely, that two randomly placed primitives will overlap. Once a pixel has been covered by a primitive, further overlap has no effect. Thus the area covered by all samples does not change linearly with the number of samples. For example, if we were to set N pixels uniformly at random in an image of M total pixels, the ratio of the number of pixels set to the number of pixels chosen is nonlinear and reaches only about 60% when $N = M$. The non-linearity is solely due to overlap — pixels chosen more than once do not change the resulting image. Figure 3.1 demonstrates this simple situation. It is, however, possible to correct for overlap by modifying the density function accordingly. We will first consider the simple case where each primitive is a single pixel.

3.1.1 PDF correction for single-pixel primitives

Let $I(\mathbf{x}_i)$ be the intensity of the base image at pixel $\mathbf{x}_i$ represented as a value between 0 and 1. Note that 2D pixel positions do not play a role in the case of single pixel primitives, so we use the simpler notation with a single subscript to distinguish between pixels. The drawing process should create a binary image where the probability that a pixel $\mathbf{x}_i$ is set is identical to the intensity of the corresponding pixel in the base image, that is, $p(\mathbf{x}_i = 1) = I(\mathbf{x}_i)$. We will use the shorter notation $p(\mathbf{x}_i)$ to mean $p(\mathbf{x}_i = 1)$ where there is no possibility of confusion. Furthermore, we want to be able to independently place random samples for generating this binary image. We need to find a PDF $q(\mathbf{x})$ that can be used with the algorithm from Section 3.2.4 such that once primitives have been placed and overlap taken into consideration, the probability of each pixel being set is as above.

After placing N samples according to the density function $q(\mathbf{x})$, the probability that pixel $\mathbf{x}_i$ has *not* been set is $(1 - q(\mathbf{x}_i))^N$. Therefore, we require that

$$\begin{aligned} p(\mathbf{x}_i) &= 1 - (1 - q(\mathbf{x}_i))^N \\ q(\mathbf{x}_i) &= 1 - \sqrt[N]{1 - p(\mathbf{x}_i)} \\ &= 1 - \sqrt[N]{1 - I(\mathbf{x}_i)} \end{aligned} \tag{3.1}$$

(3.2)

for $i = 1, 2, \ldots, n$. We also require that $q(\mathbf{x}_i)$ integrate to one so that it satisfies the definition of a probability density function. Combining this constraint with

Equation 3.1 gives us

$$\begin{aligned} & \sum_{i=1}^{n} q(\mathbf{x}_i) = 1 \\ \Rightarrow \quad & \sum_{i=1}^{n} 1 - \sqrt[N]{1 - I(\mathbf{x}_i)} = 1 \\ \Rightarrow \quad & \sum_{i=1}^{n} \sqrt[N]{1 - I(\mathbf{x}_i)} = n - 1 \end{aligned} \tag{3.3}$$

This non-linear equation for N can only be solved directly for a constant intensity image $I(\mathbf{x}_i) = I_0$. For general images this equation system cannot be solved directly, however standard numerical methods work well. We bracket the value of N between a minimum and maximum value and perform a binary search. The maximum can be chosen as the number of samples that would be required to cover an image of constant intensity very close to black.

In practice this binary search does not have to extend over the full sum in Equation 3.3. Every term in the sum only depends on the probability $p(\mathbf{x}_i)$ for each pixel, which in turn depends only on the intensity $I(\mathbf{x}_i)$ of the base image, and that is usually quantised, for example to 256 levels. Therefore, if we denote the quantised intensity levels as I_j, $j = 1, 2, \ldots, K$, and the histogram of the intensities in the base image as $H(I_j)$, we can rewrite Equations 3.1 and 3.3 as

$$q(I_j) = 1 - \sqrt[N]{1 - I_j} \tag{3.4}$$

and

$$\sum_{j=1}^{K} H(I_j) \sqrt[N]{1 - I_j} = n - 1 \tag{3.5}$$

which involves sums over only K levels rather than n pixels.

Solving Equations 3.4 and 3.5 gives us both N, the total number of primitives to place, and $q(\mathbf{x}_i)$, the modified PDF value for each intensity level. We can then generate the positions of the primitives using the methods of Section 3.2 and draw the final output.

3.1.2 PDF correction for large stroke sizes

The method of Section 3.1.1 assumed that the primitives to be placed were each single single pixels. The theory guarantees that the probability of a particular pixel being set to black equals the greyscale intensity of the input pixel. This is a probabilistic dithering algorithm. Tone reproduction for larger primitives is possible under two assumptions. First, it is clear that drawing styles that require strokes to cross discontinuities cannot *precisely* preserve tone around those discontinuities. We can therefore only hope for exact tone reproduction if we either align strokes parallel to discontinuities, or clip them accordingly. Second, we can only hope to preserve feature sizes that are no smaller than the width of the primitives used. We are willing to sacrifice exact per-pixel tone reproduction if the style is sufficiently interesting. The resulting output drawing will reproduce tone down to, but not below, the level of a single primitive.

With these limitations in mind, the tone correction procedure from Section 3.1 can be extended to primitives of larger size. Where we could previously ignore spatial relationships between the different pixels, these now have to be taken into account. We adapt our notation to 2D indices for the images and probability tables (e.g. $p(\mathbf{x}_{i,j})$ rather than $p(\mathbf{x}_i)$). In the new notation the image has a width of n_x pixels and a height of n_y pixels.

We can then describe the probability $p(\mathbf{x}_{i,j})$ that a pixel is set after rendering N primitives as

$$p(\mathbf{x}_{i,j}) = 1 - \left(1 - \sum_{k=1}^{n_x}\sum_{l=1}^{n_y} q(\mathbf{x}_{k,l})s(\mathbf{x}_{k-i,l-j})\right)^N \tag{3.6}$$

where $s(\mathbf{x}_{k-i,l-j})$ is the image of a primitive placed at position $\mathbf{x}_{k,l}$. The value of $s(\mathbf{x})$ at any point is either one or zero; we do not consider intermediate values. Comparing Equation 3.1 with Equation 3.6, we see that the simple probability

of a primitive being placed at $\mathbf{x}_{i,j}$ is replaced with the summed probability of a primitive being placed anywhere such that the primitive covers $\mathbf{x}_{i,j}$. The constraint from Equation 3.3 becomes

$$\sum_{k=1}^{n_x}\sum_{l=1}^{n_y} q(\mathbf{x}_{k,l}) = 1 \tag{3.7}$$

To determine $q(\mathbf{x}_{i,j})$ and the number of samples N, Equations 3.6 and 3.7 have to be solved simultaneously. However, this system of equations does not in general have a solution: consider an image with a one-pixel wide dark line on white background, and a circular primitive of radius larger than a single pixel. Since the line is very dark, the primitives have to be densely spaced on it. However, since each individual primitive is wider than the line itself, it will also automatically cover adjacent parts of the image which should be white, i.e. $p(\mathbf{x}_{i,j}) = 0$. Thus it is obvious that the system of Equations 3.6 and 3.7 is not solvable if the image contains features narrower than the primitives used for hatching.

An alternative strategy would be to minimise the difference between the left and the right side of Equation 3.6 subject to Equation 3.7 with the additional constraint that $q(\mathbf{x}_{k,l}) \geq 0$ for all k, l. This is a non-linear constraint optimisation problem with a large number of variables (equal to the number of pixels in the final image). While this system is rather sparse for reasonable stroke sizes, it is beyond hope of solving in real time.

However, we can simplify Equation 3.6 by assuming the target tone is approximately constant over the support of a primitive. This may sound restrictive, but is easily achieved by aligning the strokes orthogonal to the gradient field of the input image or otherwise preventing the stroke from crossing discontinuities as described

above. With this simplification, Equation 3.6 becomes

$$\begin{aligned} p(\mathbf{x}_{ij}) &= 1-\left(1-\sum_{k=1}^{n_x}\sum_{l=1}^{n_y} q(\mathbf{x}_{k,l})s(\mathbf{x}_{k-i,l-j})\right)^N \\ &\approx 1-\left(1-q(\mathbf{x}_{i,j})\sum_{k=1}^{n_x}\sum_{l=1}^{n_y} s(\mathbf{x}_{k-i,l-j})\right)^N \\ &= 1-\left(1-q(\mathbf{x}_{i,j})s_{i,j}\right)^N \end{aligned} \tag{3.8}$$

where $s_{i,j} \equiv \sum_{k=1}^{n_x}\sum_{l=1}^{n_y} s(\mathbf{x}_{k-i,l-j})$. Intuitively, $s_{i,j}$ is the number of positions $\mathbf{x}_{k,l}$ that would cause pixel $\mathbf{x}_{i,j}$ to be covered if a primitive was placed at $\mathbf{x}_{k,l}$. Note that this relationship allows for different stroke sizes and orientations depending on the local intensity or even pixel position. However, this relationship should be simple to compute in order to facilitate a rapid calculation of the $s_{i,j}$.

Once $s_{i,j}$ has been computed, we can solve Equation 3.8 for $q(\mathbf{x}_{i,j})$ just as in Section 3.1. The search for the correct number of primitives to be placed can again be sped up by using a histogram-based approach, where the size of the histogram is the product of the number of quantisation levels for the image intensity and the number of different sizes used for strokes during rendering. For a small set of differently sized strokes this will still be more efficient than a search directly on the full resolution image.

3.2 Generating samples according to a given PDF

Once we have generated an appropriate PDF from an input image using the methods in Section 3.1, we must generate a set of sample points that are distributed according to that PDF. A set of samples $\mathcal{S}$ is distributed according to a PDF q if for *any* region R, the fraction of samples in R equals $\int_R q\,dA$. No particular finite set of samples can represent a continuous PDF, since it will nearly always be possible

to find some region R such that the integral of the PDF is non-zero, but there are no samples in R. However, we can still distribute a finite set of sample if the probability of placing a sample in some region R is equal to $\int_R q dA$. There are several standard methods of generating samples according to a given PDF [19, 20]. We present them in the following sections along with a hybrid approach.

3.2.1 Generating samples analytically

The only common directly-generated distribution is the uniform distribution $U_{0,1}$ and its trivial scalings and translations. However, using the transformation method (see Section 3.2.3), it is possible to generate analytical transformations that map points from $U_{0,1}$ to some easily-solved distributions. An example, given without derivation, is the random variable $y(x) \equiv -\ln(x)$, where $x \sim U_{0,1}$, which has an exponential distribution. That is, its PDF is e^{-y}. Note, however, that this method is limited to distributions with an analytical expression for the PDF, and even then only certain easily-solved forms of expression.

3.2.2 The rejection method

The rejection method is a simple and very powerful method to generate samples distributed according to some PDF $q(\mathbf{x})$. Assume that we have a method of generating points from a different distribution r, where $q(\mathbf{x}) \leq Mr(\mathbf{x})$ everywhere for some constant M. In general r is chosen so that generating r-distributed points is particularly simple or convenient. For example, in one dimension r is often the uniform distribution $U_{0,1}$ and $M = \max(q(\mathbf{x}))$.

To generate the q-distributed set of N points $\mathscr{S}_q$, we use Algorithm 1.

The average efficiency of this procedure, as measured by the ratio of N to the total number of tested points, is the ratio of areas of Mr and q, or M, since r and q

Algorithm 1: The rejection method.

while $|\mathscr{S}_q| < N$ **do**
  Sample $\mathbf{x} \sim r$
  Sample $l \sim U_{0,1}$
  if $q(\mathbf{x})/Mr(\mathbf{x}) \leq l$ **then**
    $\mathscr{S}_q \leftarrow \mathscr{S}_q \cup \mathbf{x}$
  end if
end while

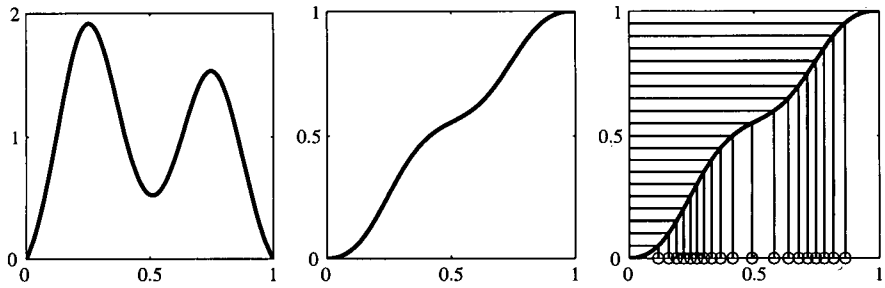
have unit area by definition.

3.2.3 The transformation method in one dimension

Given a one-dimensional probability density function $q(x)$, $x \in [0,1]$, and a set of uniformly distributed random samples $\{x_i\}$ over $[0,1]$, we can redistribute the samples according to $q(x)$. An example of a 1D probability density function is shown in Figure 3.2(a). We compute the *cumulative density function*

$$C(x) = \int_0^x q(t)\,dt$$

as shown in Figure 3.2(b). We then invert $C(x)$ and transform each sample x_i to get $x_i' = C^{-1}(x_i)$. This is shown graphically in Figure 3.2(c) by mapping a set of samples on the y-axis through to the x-axis. The set $\{x_i'\}$, drawn as circles on the x-axis of Figure 3.2(c), is distributed according to the original probability density function $q(x)$. Note that for the purposes of the illustration, the input sample set is regularly distributed, but it should be chosen from a uniform random distribution. Also, an intuitive explanation of the mechanics of the transformation method is possible with Figure 3.2. The peaks in the probability density function are transformed through integration into areas of high slope. These areas of high slope



(a) Probability density function.

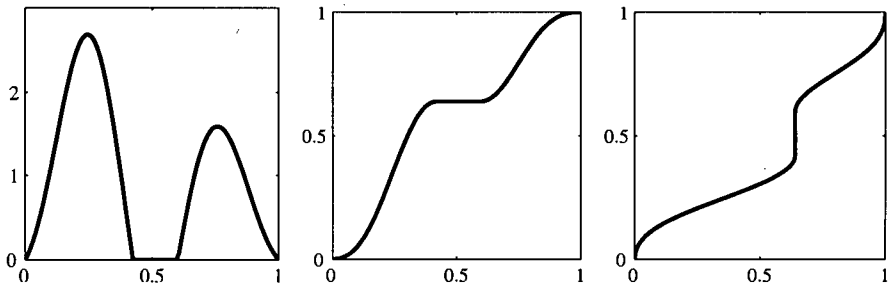
(b) Cumulative density function.

(c) Regular sampling of the inverse of CDF.

Figure 3.2: 1D example of redistributing points according to a probability density function. Compare the samples (circles) at the bottom of 3.2(c) to the density function in 3.2(a).

consequently gather more input samples than other areas. Symmetrically, valleys in the density function correspond to areas of low slope in the cumulative density function, which intersect fewer input samples.

It should be noted that computing the inverse of the cumulative density function is not necessarily straight-forward. Figure 3.3 illustrates the problem. If the PDF has any regions of zero probability, then they integrate to regions of zero slope in the cumulative density function. These regions of zero slope make the inversion impossible since many x-values map to a single y-value. The transformation method is only well-posed if the PDF is strictly positive over its support. Worse still, in general, the vertical section of the inverted graph cannot be represented on a fixed grid, in general. If the cumulative density function is forcibly inverted and stored in a standard array of values, the vertical jump will be distorted from a segment of infinite slope to one of merely high slope. The equivalent effect on the original PDF would be to change the regions of zero probability to have some



(a) Probability density function.

(b) Cumulative density function.

(c) Inverse of 3.3(b).

Figure 3.3: 1D example of the transformation method with zero regions in the PDF. Note that 3.3(c) is no longer a function.

small but positive probability which will generate samples. This introduction of spurious samples is very disturbing since zero-probability regions usually come from visually important areas such as light backgrounds or object highlights.

One correct solution would be to split the original PDF into a set of strictly positive PDF's and generate samples from each PDF independently. However, this is complex, especially for PDF's of greater than one dimension, where the introduction of zero probability "holes" does not conveniently disconnect the PDF. Stepping back from the formalism, zero probability regions in the PDF mean that we should never generate samples in those regions. Generating a sample in a flat region of the CDF is equivalent to generating a sample outside the support of the PDF. If we invert the cumulative density function *on the fly* for each sample, then we can detect when an input sample falls in a flat region and discard that sample. The inversion is accomplished by searching the cumulative density function table for the x-value which maps to a particular input y-value. Since the cumulative density function is increasing, a simple binary search algorithm can do this task

in $O(\log_2 N)$ table accesses, where N is the number of entries in the table. This also solves the problems associated with discretising the inverse of the cumulative density function, since the inverse is never explicitly calculated and stored, only implicitly accessed in the search.

3.2.4 The transformation method in two dimensions

Given a 2D probability density function $q(\mathbf{x})$, $\mathbf{x} \in [0,1]^2$, and a set of uniformly distributed random points $\{\mathbf{p}_i\}$ over $[0,1]^2$, we can redistribute the points according to $q(\mathbf{x})$ using the transformation method.

To find the y-coordinate $\mathbf{p}'_{i,y}$ of a redistributed point $\mathbf{p}'_i$ we compute the *cumulative density function*

$$M(y) = \int_0^y m(t)\,dt, \text{ where } m(y) = \int_0^1 q(\mathbf{x})\,dx,$$

and obtain $\mathbf{p}'_{i,y} = M^{-1}(\mathbf{p}_{i,y})$. The function $m(y)$ is the *marginal density function* of $q(\mathbf{x})$. In a 2D image, $m(y)$ can be considered the total intensity of the scanline y. Note that $M(y)$ is monotonic, but not necessarily strictly monotonic if some scanlines have zero intensity. Hence, the inverse $M^{-1}(y)$ exists almost everywhere except at isolated points.

Given $\mathbf{p}'_{i,y}$, we can now determine the x-coordinate $\mathbf{p}'_{i,x}$ by redistributing $\mathbf{p}_{i,x}$ according to the PDF of the respective scanline. Mathematically, this is described by a conditional PDF

$$c(x \,|\, \mathbf{p}'_{i,y}) = \frac{q(x, \mathbf{p}'_{i,y})}{m(\mathbf{p}'_{i,y})}$$

and its cumulative density function

$$C(x \,|\, \mathbf{p}'_{i,y}) = \int_0^x c(t \,|\, \mathbf{p}'_{i,y})\,dt.$$

As before, the x-component of the new point is given by the inverse of that function: $\mathbf{p}'_{i,x} = C^{-1}(\mathbf{p}_{i,x} \,|\, \mathbf{p}'_{i,y})$.

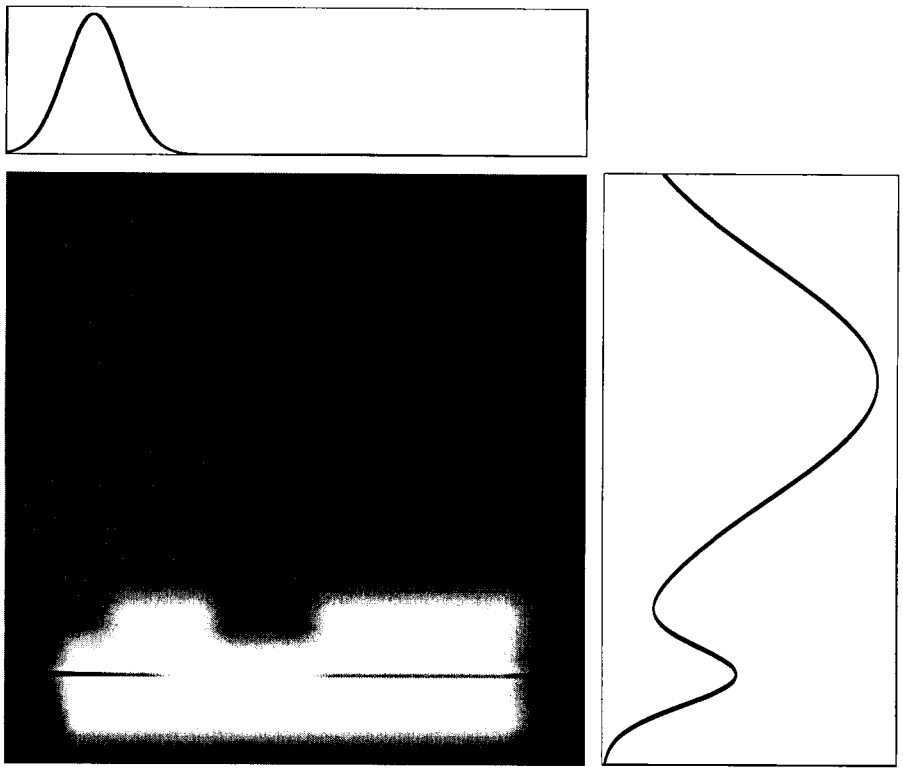


Figure 3.4: A 2D PDF along with its cumulative density function $M(y)$ (right) and a 1D scanline PDF (top).

For a discrete PDF $q(\mathbf{x})$, such as one arising from an image, $M(y)$ and $C(x|y)$ are easily precomputed as a 1D and a 2D table, respectively.

To illustrate the above procedure, Figure 3.4 shows an example 2D PDF, where dark values map to low probabilities and light values map to high probabilities. On the right is the marginal density function $m(y)$ obtained by integrating in the x-direction, and at the top is one of the many scanline PDF's, $c(x|y)$ for some y, highlighted in the 2D PDF.

3.2.5 Computational concerns for sample generation

The most efficient method of generating samples would be the analytical method from Section 3.2.1. However, in our application the PDF is neither known in advance nor expressed in analytical form, making the analytical method impossible. When the rejection method is used with PDF's that are not known in advance, the "enveloping" distribution $r(\mathbf{x})$ is often chosen to be the uniform distribution, and the scaling factor is chosen as the maximum of the target PDF. While this allows the rejection method to work for arbitrary distributions, it can be quite inefficient. The further the target PDF differs from the flat uniform distribution, the larger M must be and the lower the efficiency of the sample tests. On average, M samples must be tested to get one output sample. However, the rejection method has no fixed costs, unlike the transformation method with its integrations. If only a small number of samples are needed, then the relatively expensive testing process of the rejection method will still be faster than the complete integration required by the transformation method. In general, however, the integrations required by the transformation method are not disastrous because they are highly memory-coherent: essentially running sums of linear tracts of memory. This allows the memory caching algorithms found on modern computers to work very efficiently.

3.3 Generating samples for differential PDF's

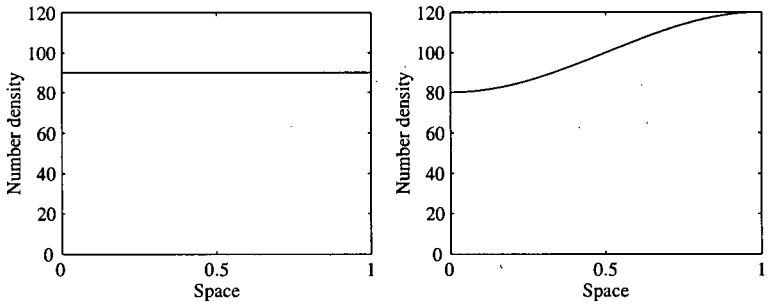
While Section 3.2.3 and Section 3.2.2 gave us methods of generating a set of points that are distributed according to some probability density function (PDF) q, we would also like to consider the situation where we have two PDF's, q_1 and q_2, and a set of points S_1 that distributed according to q_1. From this set we would like to add and subtract points so that the resulting set S_2 is distributed according to q_2. This is

what we call generating a *differential* distribution. A differential distribution will add and delete the minimum number of points required to generate the PDF q_2. If the distributions are used in the rendering of consecutive frames in an animation, the frame-to-frame coherence will be maximised. To do this we will examine the positive and negative regions produced by subtracting the first distribution from the second distribution.

It is important to note that the we cannot use the PDF's directly for this subtraction since, by definition, the PDF's are scaled such that their integrals are unity. Imagine two point distributions, both uniformly distributed, but the first with 1000 points and the second with 100. The difference of their PDF's is identically zero, however, it is clear that to transform the first set of points into the second, many points must be deleted. Instead, we need to look at the PDF's augmented with the total number of points in their distributions. That is, we need to examine number density functions (NDF's) instead of raw probability density functions.

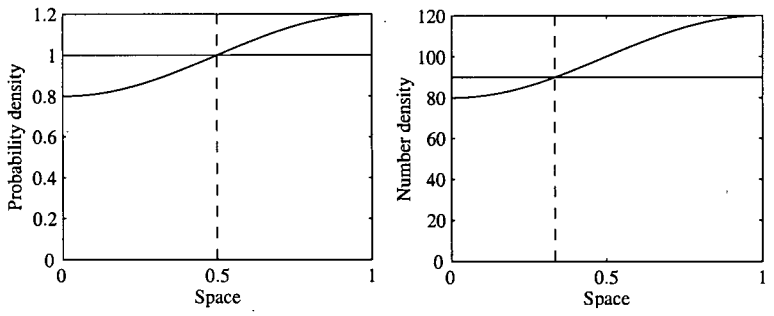
A number density function is defined as a scaling of a PDF by the total number of points in the distribution. All the properties of PDF's carry over to NDF's with the addition of this simple scale. For example, while the integral of a PDF over a particular region gives the probability of a point existing in that region, the integral of a NDF of a region gives the expected number of points found in that region. To illustrate the importance of using NDF's over PDF's in differences, Figure 3.5 shows two NDF's, Figure 3.6(a) shows the incorrect result of using only the corresponding PDF's, and Figure 3.6(b) shows the correct result using the full NDF's.

For a particular set of points $\mathcal{S}$ that is distributed according to a probability density function q, we define a number density function by $n \equiv Nq$, where $N = |S|$. In particular, assume that we have two frames of an animation, processed according to Section 3.1 to give us two PDF's q_1, q_2, and their respective number of primitives



(a) (b)

Figure 3.5: Two number density functions.



(a) Difference of PDF's (incorrect). (b) Difference of NDF's (correct).

Figure 3.6: Comparison between differences of PDF's and NDF's.

N_1 and N_2. Then we examine the difference function

$$d = N_2 q_2 - N_1 q_1$$

and define the *positive difference region* to be $R_\oplus \equiv \{\mathbf{x} \mid d(\mathbf{x}) > 0\}$ and the *negative difference region* to be $R_\ominus \equiv \{\mathbf{x} \mid d(\mathbf{x}) < 0\}$. We define the *positive difference NDF* to be $n_\oplus \equiv \max(d, 0)$ and the *negative difference NDF* to be $n_\ominus \equiv -\min(d, 0)$. Note the minus sign in the definition of $n_\ominus$. The regions where $n_\oplus$ and $n_\ominus$ are non-zero are $R_\oplus$ and $R_\ominus$, respectively.

3.3.1 Processing difference regions

After calculating the difference NDF's $n_\oplus$ and $n_\ominus$, we know that $|n_\oplus|$ points must be added to region $R_\oplus$, distributed according to the PDF $n_\oplus/|n_\oplus|$. Formally, we are doing the following:

$$\begin{aligned} N_1 q_1 + n_\oplus - n_\ominus &= N_1 q_1 + \max(N_2 q_2 - N_1 q_1, 0) + \min(N_2 q_2 - N_1 q_1, 0) \\ &= N_1 q_1 + N_2 q_2 - N_1 q_1 \\ &= N_2 q_2 \end{aligned}$$

Where we use $\max(a,b) + \min(a,b) = a + b$ in the first step. To add points we simply use the transformation method as in Section 3.2 to distribute the new points according to the PDF. The PDF has zero probability outside of $R_\oplus$, so points will not be distributed outside the positive difference region.

However, we must also remove $|n_\ominus|$ points from region $R_\ominus$ according to the negative difference PDF. Note that since we are only removing points from a fixed set, the problem is best represented by a probability distribution over a discrete set of points. The standard rejection method applies naturally here to select points for removal.

3.3.2 Algorithm summary

We summarise the algorithm used to generate the differential probability distributions:

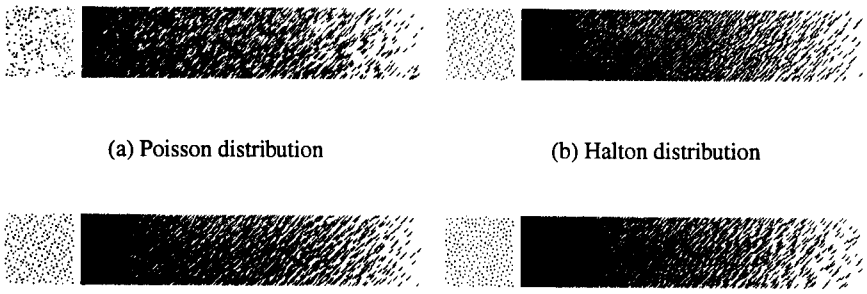
1. Calculate the positive and negative difference NDF's $n_{\oplus}$ and $n_{\ominus}$, identify the regions $R_{\oplus}$ and $R_{\ominus}$
2. Assign points in $R_{\ominus}$ probabilities proportional to $n_{\ominus}$ and scale so that the sum of probabilities is one
3. Use the transformation method to generate $n_{\oplus}$ new points in $R_{\oplus}$ according to the PDF $n_{\oplus}/|n_{\oplus}|$
4. Use rejection sampling to choose points to remove from $R_{\ominus}$

3.4 Input point distributions

To this point we have assumed that the input points are Poisson distributed, i.e. are independently chosen, uniformly distributed random points. This does not necessarily have to be the case; we can choose the input points from a variety of different random distributions and quasi-random sequences.

Poisson distributions tend to form small clusters of points that are relatively close together. It is widely accepted in the stippling and halftoning literature that a more uniform point spacing yields more visually pleasing results, so that Poisson disk distributions ("blue noise") are usually preferred [26, 2].

Our method maps any given input point set through a distribution function to determine the final location of primitives. If the base image used to generate the distribution function is approximately constant locally around a given point, then the mapping obtained by the distribution function is approximately linear locally.



(a) Poisson distribution (b) Halton distribution

(c) Sobol sequence (d) Poisson disk distribution

Figure 3.7: Comparison of different input distributions with a point distribution and a grayscale ramp.

This indicates that desirable properties such as minimal distances in a Poisson disk distribution should be *locally* preserved in regions of approximately constant intensity. This property breaks down in regions of strong changes such as edges, but that is also the case for traditional images.

We use the first N samples of a precomputed point sequence (where N is determined according to Section 3.1) for every frame in order to maintain frame-to-frame coherence. This means that we are restricted to point sequences where any subset consisting of the first N samples is well distributed. This eliminates, for example, the Hammersley point set [8], which is only well-distributed in its entirety, but allows us to use the Halton [7] or Sobol sequences [24].

In addition, we experimented with Poisson disk distributed points generated with the hierarchical approach of McCool and Fiume [16]. Besides being faster than other methods for generating Poisson disk distributed points, this approach also has the advantage of producing a sequence of points that are Poisson disk distributed with a decreasing disk radius. In Figure 3.7 we compare the performance of the different approaches by applying them to a simple grayscale ramp.

It can be seen that the Poisson distribution clusters the individual samples, resulting in a very noisy and irregular image. The other distributions avoid excessive clustering and achieve more uniform sample spacing. The Sobol sequence, however, sometimes tends to form undesirable patterns. The Halton sequence and the hierarchical Poisson disk distribution are free of those artifacts. Based on this experience, we usually choose the Halton sequence for our method since it is computationally more efficient to generate. The other images in this paper are rendered using the Halton sequence unless otherwise noted.

3.5 Drawing styles

The algorithms from Sections 3.1 and 3.2 can be used to determine stroke positions from input images. What remains open are issues of acquiring the input images from 3D models and orienting the strokes. Different choices of methods for these tasks give rise to a large variety of rendering styles.

The basic algorithm for placing the strokes only requires a grayscale input image to generate a PDF. Hence, our algorithm is able to work from data like photographs or paintings. These can be preprocessed with any kind of image processing tool to achieve the desired effect. For example, *indication* can be achieved by increasing or decreasing the brightness in selected parts of the input image [5, 28]. In addition, we include a transfer function in our system, which allows for on-the-fly adjustments of contrast or tone on the whole image.

Starting from a 3D model rather than an input image, we first use an OpenGL rendering pass with traditional shading and specular and diffuse lighting to generate an image. However, by starting from a 3D model we can easily generate additional information, encode it into the image, and later use it to drive different drawing styles. For example, we can draw silhouette outlines either using environ-

ment maps, as in the work of Gooch *et al.* [4] or by explicitly traversing the input mesh and finding silhouette edges. Similarly, we can encode additional information for determining the stroke *direction* into other colour channels. In an RGBA framebuffer, this gives us three channels to be used for such additional information. We have experimented with the following options and more are certainly possible:

- Computing a per-vertex tangent or parametric direction, projecting it into image space, and interpolating it for every pixel (all steps are possible in a single pass in hardware) to be used as a stroke direction. This yields results similar to the ones described by Winkenbach and Salesin [29].
- Computing and encoding a normal direction in much the same way. This is an interesting way of drawing short fur or grass.
- Encoding 2D stroke directions into a camera-aligned environment map. This can be used to give areas facing different directions different stroke properties.
- Encoding object or material identifications for individual parts of the scene, which can then be used to render different objects in different styles.

Based on this information, we can generate variations on three basic drawing styles:

3.5.1 Point stippling

The simplest rendering style is point stippling, which does not require any directional information. We simply place OpenGL points or scanned stipple textures at the positions obtained with the method described in Sections 3.1 and 3.2.

Figure 3.8: A stippled image of foot bones.

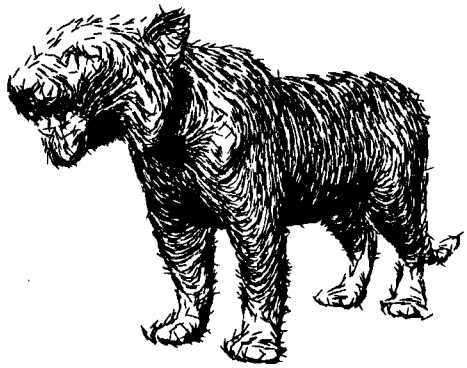


Figure 3.9: A tiger model hatched with large strokes to achieve a furry look. The stroke orientations are perpendicular to the value at the object's normal map.

3.5.2 Hatching

For hatching we either use OpenGL lines of different widths and lengths, or scanned texture maps of real pen-and-ink strokes. For the stroke orientation we can choose between a constant direction or any kind of direction encoded into colour channels as described above. Finally, we can also compute the gradient field of the image intensity, and orient strokes orthogonal to it. This works both for images and 3D models, and orients strokes parallel to discontinuity edges, which helps preserving the tone of the input image as described above. In areas in which the intensity gradient is not strong enough, we can either default to a constant stroke direction, or search for a stronger gradient in an image pyramid.

3.5.3 Cross hatching

For cross hatching we can vary the hatch direction probabilistically. Say we want to start cross hatching for some intensity threshold $I(\mathbf{x}) > \alpha$. Whenever we place a

stroke at pixel $\mathbf{x}_{i,j}$, we choose the orientation as follows: if the intensity of the input image $I(\mathbf{x}_{i,j})$ at that location is less then the threshold, we compute the stroke direction with any of the methods described above. If $I(\mathbf{x}_{i,j})$ is larger than the threshold, however, then we choose the original orientation with a probability of $\alpha/I(\mathbf{x}_{i,j})$, and a secondary direction with probability $(1-\alpha)/I(\mathbf{x}_{i,j})$. This secondary orientation can either be a fixed direction, the original direction rotated by some fixed angle, or any rotation inferred from an encoding of an object-space property as discussed above.

3.6 Results

We have implemented a system demonstrating our approach to distributing primitives that preserves continuous tone in a scale-independent manner. We can either load images or 3D scenes as a starting point for our method. The different drawing methods described in the previous section, as well as the parameters for the initial rendering of the 3D model, can produce a multitude of different styles of which we can only show a small subset.

Figure 3.11 shows a bust model rendered with hatching, where the hatches are aligned against the image gradient. Only the rendered scene is used to align the strokes, and not the 3D model information. Figure 3.11 shows a good preservation of sharp features. Figure 3.8 and 3.9 show results for stippling and hatching, respectively. Figure 3.10 shows a bee rendered with cross hatching, which gives the legs and the antennae a hairy appearance. Figure 3.13 shows how our system accurately preserves continuous tone. Figure 3.14 (Top) shows every fifth frame of an twenty-five frame animation. All of the primitives in the frame on the far left appear in the frame on the far right at adjusted locations. Extra primitives have been added over the twenty-five frames since the image darkens. The primitives

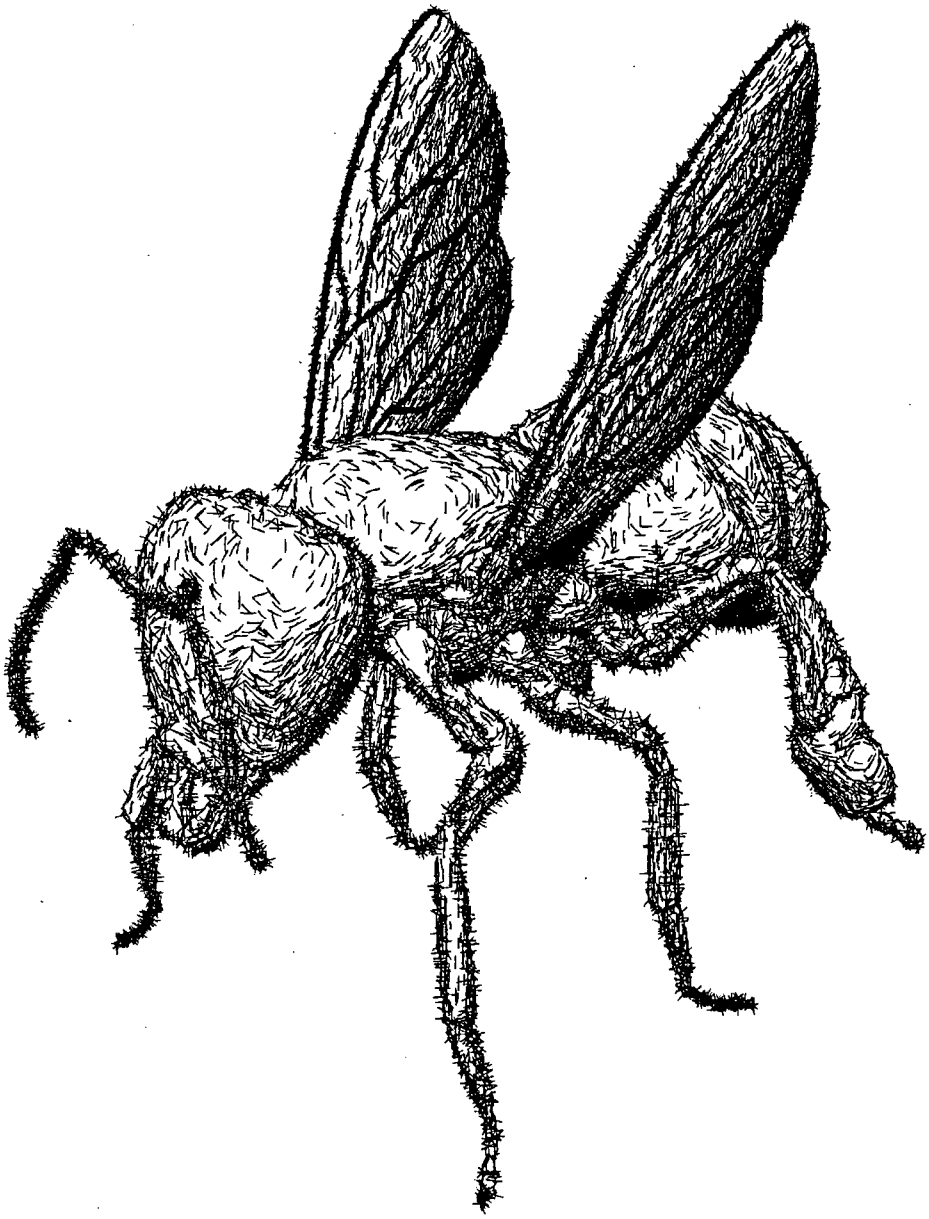


Figure 3.10: Cross hatching in very dark areas together with silhouette extraction cause a hairy appearance of the legs and antennae.

Figure 3.11: A hatched pen-and-ink illustration using the image gradient for primitive orientation.

have been "shape-encoded" to help trace their movement. More examples can be found at the end of this paper.

The performance of our method depends on several factors. The first one is the size of the table used for the PDF and the distribution functions. This size is algorithmically independent of the resolution of the final image, so small table sizes can be used to reduce rendering cost. However, if the table gets too small, sharp edges will start to alias. The resolution at which these artifacts start to be visible depends on the size of strokes, but we generally found that using a PDF table of half the size of the target image produces good results except for stippling with very small point sizes. The second factor for performance is the rendering style. Computing image-space gradients for orienting the strokes is relatively expensive, while generating the orientation from encoding a 3D direction is practically free. The cost of rendering the initial 3D model is negligible since the meshes do not need to be very detailed to serve as a basis for pen-and-ink illustration. The number of strokes rendered is another cost factor. Very dark images take longer than light

ones, and images with large strokes render faster than images with very small ones. The major cost here is the rendering of the individual points, lines, or texture-mapped strokes using OpenGL (which is a geometry-bound operation on current graphics hardware).

Depending on the above parameters, we were able to achieve between 2 frames per second and 15 frames per second on a 1.5GHz Pentium with a GeForce 3 card, using PDF sizes between 256×256 and 512×512 for the scenes shown in this paper. No specialised hardware was used with our software implementation to obtain these frame rates.

A common goal of many real-time NPR techniques is to achieve frame-to-frame coherence of primitive placement. In our approach individual primitives are placed in image-space, are therefore not attached to 3D geometry and can move around freely. Thus, we cannot expect them to move exactly with the 3D object points. However, since we re-use the same set of uniformly distributed points, small changes in the input image (and therefore in the PDF) will only result in small adjustments of the point positions. For example, if a highlight moves across a 3D object due to a moving light source, then the individual strokes will rearrange themselves continuously around the highlight as shown in the top of Figure 3.14. Similarly, if the transfer function is adjusted continuously, the strokes will slowly move across the image to adapt for the changes. These effects are demonstrated in the video. Since our approach distributes primitives in real-time based on intensity and primitive size, continuous tone is precisely replicated. Even with the use of slightly larger primitives the global tone is accurately preserved and is scale-independent. Our method is based entirely on the location of primitives, allowing us to easily simulate many drawing styles by altering the primitive type or direction.

Figure 3.12: Rendering of the Lena image using hatching and cross hatching. Primary strokes are aligned perpendicular to the gradient in regions of strong gradients and at a 45° angle in areas where the gradient is small.

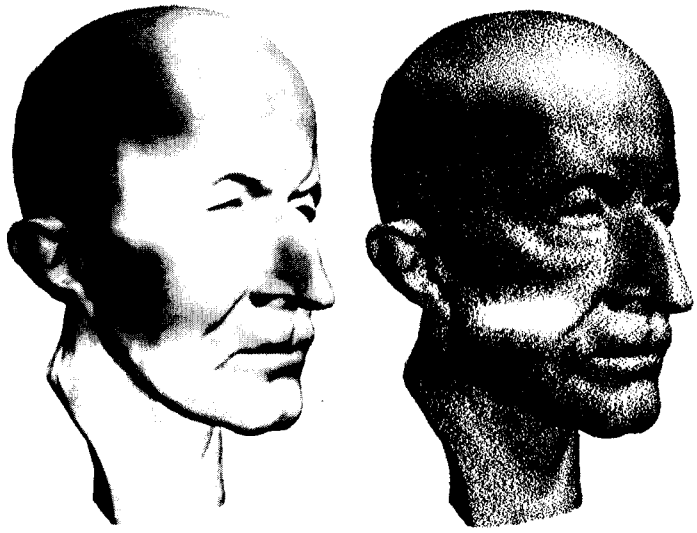


Figure 3.13: A bust stippled with pixel-sized stipples. The left image is the original greyscale input and the right is the stippled image. This result is similar to the ones that would be obtained by halftoning methods. Any apparent differences in tone between these two images, when viewed from a distance of several feet, is due to the halftoning screen of the individual printer.

Figure 3.14: Frames of an animation of a moving highlight showing stroke movement.

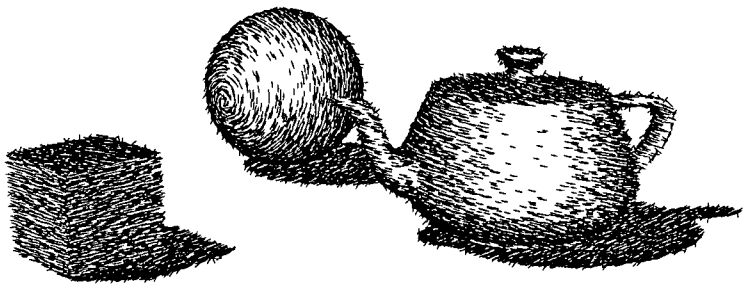


Figure 3.15: A simple 3D scene rendered with different stroke directions for the different objects using material ID's and object-coded directions.

Figure 3.16: Stanford bunny hatched with curved strokes.

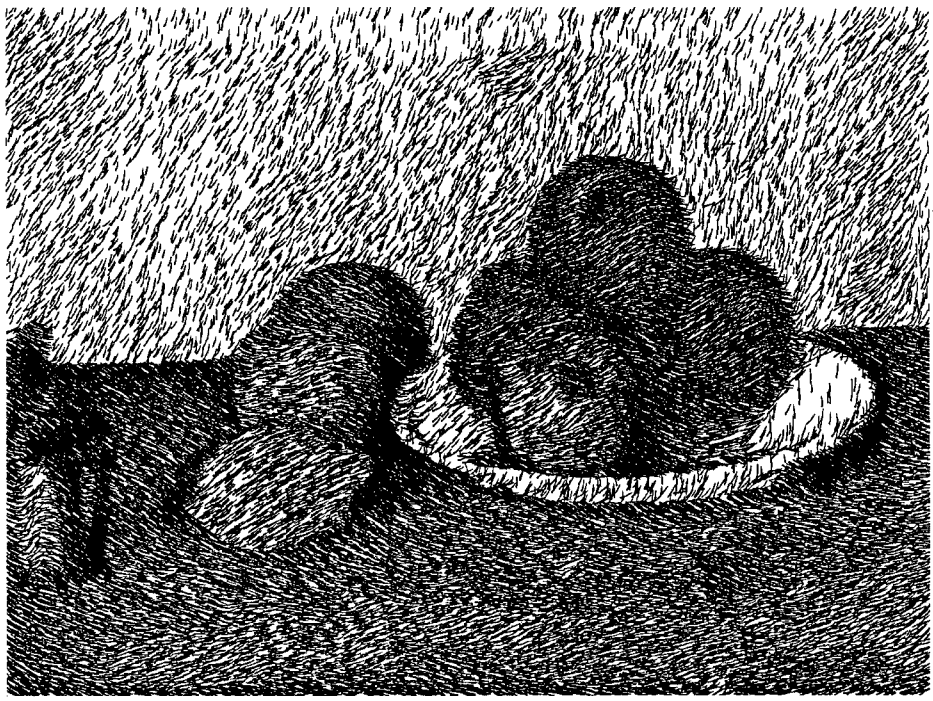


Figure 3.17: A hatched rendering using Cezanne's "Still Life With Apples" as input. The stroke directions are a function of the colour in the original image.

Figure 3.18: A hatched rendering using Van Gogh's "Self Portrait With Felt Hat, 1888" as input. The stroke directions are a function of the colour in the original image.

Chapter 4

Voronoi-based Drawings

Stippling as a technique came into existence to give artists control over the half-toning processes used when printing images in books was a new and difficult task [12]. The technique consists of carefully placing many small dots of ink on paper to approximate different tones. Stipples are placed closer together to form dark regions and further apart to form lighter regions. The stipples must be placed evenly yet randomly so that the human eye does not see spurious patterns that are not a part of the intended impression. The stipples may vary in size and occasionally shape to convey subtle details.

The original advantage of stippling was its ease of reproduction. The half-toning used to print images in books was of highly variable quality and often drawings were drastically resized to meet space requirements. While normal drawings suffered from such treatment stipple drawings retained their attributes more faithfully. In addition, printing a stippled drawing requires only the ability to produce dots of a single colour, making it an inexpensive technique [31].

However, stippling has significant artistic merit independent of its utility. The stipples can represent fine detail and texture with little cost in complexity. Stippling is particularly good at clearly representing smooth, rounded objects without sharp edges and so is often used in medical and archaeological texts.

We wish to generate stipple drawings from images with as little user input as possible. The goal is to develop a tool which can generate high-quality stipple

drawings from any source whatsoever, which implies that we use images as input and not 3D models. While this limits the amount of information we have to work with, it allows us a greater variety of input sources. For example, a user could start from a scanned pencil sketch, a photograph, the output of a 3D interactive application, frames of an animation, etc.

One of the features of a good stipple drawing is that the stipples are *well-spaced*, that is, the stipples do not clump together, leave uneven voids, or form unwanted patterns. The artist achieves this by carefully placing each stipple onto the page, explaining why stipple drawings often take weeks to create by hand.

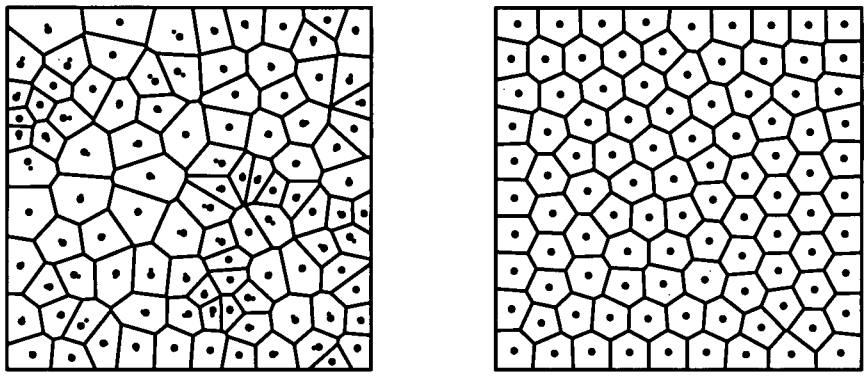
Central to our approach is the use of centroidal Voronoi diagrams to produce good distributions of points, as explained in Section 4.1.1. The input image can be used directly as a weighting function to create a distribution of points that approximate its tones. This method produces images of high quality but takes more processing time than the algorithm of Chapter 3, as explained in Section 4.2.

4.1 Voronoi diagrams

An ordinary Voronoi diagram is formed by a set of points in the plane called the *generators* or *generating points*. Every point in the plane is identified with the generator which is closest to it by some metric. The common choice is to use the Euclidean L_2 distance metric

$$|\mathbf{x}_1 - \mathbf{x}_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

where $\mathbf{x}_1 = (x_1, y_1)$ and $\mathbf{x}_2 = (x_2, y_2)$ are any two points in the plane. The set of points in the plane identified with a particular generator form that generator's Voronoi region, and the set of Voronoi regions covers the entire plane. Figure 4.1(a) illustrates a set of generating points and their associated Voronoi regions.



(a) Voronoi diagram generated by the set of generators (large dots). Centroids of each Voronoi region are marked by the small dots.

(b) Centroidal Voronoi diagram

Figure 4.1: General and centroidal Voronoi diagrams.

There exist $O(n \log n)$ algorithms for the geometric computation of 2D Voronoi diagrams.

We implemented the fast 3D graphics hardware-based algorithm in Hoff [11] and originally suggested in [30] to compute our Voronoi diagrams. The algorithm draws a set of right cones with their apexes at each generator. The cones all have the same height and are viewed from above the apexes with an orthogonal projection. In addition, each cone is given a unique colour which acts as the generator's identity. Since the cones must intersect if there is more than one generator, the z-buffer determines for each pixel which cone is closer to the viewer and assigns that pixel the appropriate colour value. We can then scan the resulting image and determine which generator is closest to each pixel by using the unique colours. This technique allows us to compute discrete Voronoi diagrams extremely quickly

and perform computations on the resulting regions.

4.1.1 Centroidal Voronoi diagrams

A *centroidal* Voronoi diagram has the interesting property that each generating point lies exactly on the centroid of its Voronoi region. The centroid of a region is defined as

$$\mathbf{C}_i = \frac{\int_A \mathbf{x}\rho(\mathbf{x})\,dA}{\int_A \rho(\mathbf{x})\,dA} \tag{4.1}$$

where A is the region, $\mathbf{x}$ is the position and $\rho(\mathbf{x})$ is the density function. For a region of constant density ρ, the centroid can be considered as the "centre of mass." Figure 4.1(a) has the centroids of each region marked with small circles.

A centroidal Voronoi diagram is a minimum-energy configuration in the sense that it minimises $\int_A \rho(\mathbf{x})|\mathbf{C}_i - \mathbf{x}|^2$ [3]. Practically speaking, a centroidal distribution of points is useful because the points are *well-spaced* in a definite sense. Figure 4.1(b) shows a centroidal Voronoi diagram.

4.1.2 Generating centroidal Voronoi diagrams

Lloyd's method [18] is an iterative algorithm to generate a centroidal Voronoi diagram from any set of generating points. The algorithm can simply be stated:

while Generating points $\mathbf{x}_i$ not converged to centroids **do**
  Compute the Voronoi diagram of $\mathbf{x}_i$
  Compute the centroids $\mathbf{C}_i$ using equation (4.1)
  Move each generating point $\mathbf{x}_i$ to its centroid $\mathbf{C}_i$
end while

Algorithm 2: Lloyd's method.

Figure 4.1(a) relaxes under Lloyd's algorithm to become Figure 4.1(b). The convergence of Lloyd's algorithm to a centroidal Voronoi diagram has been proven for the one-dimensional case. The higher dimensional cases seem to act similarly in practice, though no proof is known [3]. There are several different convergence criteria which should be equivalent in the limiting case as the algorithm runs forever. The obvious criterion to use would be that the computed centroids are numerically equal to the generating points. However, for most applications this criterion is far too stringent and it would be perhaps better to look at the average distance moved by all generating points. Since we are interested in generating well-spaced sets of points, we look at the average change in inter-point distance, or equivalently, the average change in Voronoi region area.

Efficient computation of centroids

Calculating the centroids requires efficiently evaluating the integrals in equation (4.1). Since the integrals are over arbitrary Voronoi regions, we convert to iterated integrals and integrate the region row by row. In this manner we can precompute much of the integral.

The denominator of the centroid is transformed as follows:

$$\begin{aligned}\int_A \rho(\mathbf{x})\,dA &= \int_{y_1}^{y_2}\int_{x_1(y)}^{x_2(y)} \rho(\mathbf{x})\,dx\,dy \\ &= \int_{y_1}^{y_2} [P]_{x_1}^{x_2}\,dy\end{aligned}$$

where $P \equiv P(\mathbf{x}) \equiv \int_0^x \rho(s,y)\,ds$ can be precomputed from the density function[1]. Note that we cannot precompute the entire integral because we do not know the boundaries of the Voronoi regions beforehand.

[1]Recall that $[\int_0^x f(s)\,ds]_a^b = \int_0^b f(s)\,ds - \int_0^a f(s)\,ds = \int_a^b f(s)\,ds$

The numerator of the y-coordinate of the centroid is transformed similarly:

$$\begin{aligned}\int_A y\rho(\mathbf{x})\,dA &= \int_{y_1}^{y_2}\int_{x_1(y)}^{x_2(y)} y\rho(\mathbf{x})\,dx\,dy \\ &= \int_{y_1}^{y_2} y\,[P]_{x_1}^{x_2}\,dy\end{aligned}$$

The numerator of the x-coordinate of the centroid involves integration by parts:

$$\begin{aligned}\int_A x\rho(\mathbf{x})\,dA &= \int_{y_1}^{y_2}\int_{x_1(y)}^{x_2(y)} x\rho(\mathbf{x})\,dx\,dy \\ &= \int_{y_1}^{y_2}\left\{[xP]_{x_1}^{x_2} - \int_{x_1}^{x_2} P\,dx\right\}dy \\ &= \int_{y_1}^{y_2}[xP-Q]_{x_1}^{x_2}\,dy\end{aligned}$$

where $Q \equiv Q(\mathbf{x}) \equiv \int_0^x P(s,y)\,ds$ can also be precomputed from the density function.

Note that the final expressions require numerical integration only in the y-direction and otherwise involve expressions only at the region boundaries x_1 and x_2. P and Q are precomputed once from the density function and then evaluated at the horizontal end points x_1 and x_2 as needed. This allows us to compute the integrands only at region boundaries and not at every pixel. Otherwise we would have to compute the integrands $x\rho$ and $y\rho$ for every span of pixels across a region and numerically integrated. The above integrand computation is particularly simple — at worst two look-ups for P and Q, a multiplication and a subtraction. In addition, if the Voronoi region is non-convex for numerical reasons, the scan conversion effectively decomposes the region into convex sub-regions, that is, single spans of pixels.

4.1.3 Resolution of Voronoi calculation

One disadvantage of using a discrete calculation of the Voronoi regions is the calculation of the centroids is affected by the resolution of the diagram. The relative error of the calculated centroid location will increase as the number of pixels per Voronoi region decreases. A related problem is that if the resolution is low enough, two generating points can effectively overlap and one of the regions will disappear. The solution is to split the diagram into tiles and compute each tile at the full resolution available and then stitch the full diagram back together at a higher virtual resolution [11]. The virtual resolution can be increased arbitrarily to meet a lower bound on Voronoi region pixel area.

4.2 Stippling with weighted CVDs

The centroidal Voronoi diagrams in Section 4.1.1 incorporate the idea of a density function $\rho(\mathbf{x})$ which weights the centroid calculation. Regions with higher values of ρ will pack generating points closer than regions with lower values. During the iteration of Algorithm 2, the darker regions of the image appear to "attract" more points. We can use Algorithm 2 directly to generate high-quality stippling images by treating a grayscale image as a discrete two-dimensional function $f(\mathbf{x})$ where $\mathbf{x} \in [0,1]^2$ and $0 \leq f(\mathbf{x}) \leq 1$ is the range from a black pixel to a white pixel. Define a density function $\rho(\mathbf{x}) = 1 - f(\mathbf{x})$. We can then stipple a given image by first distributing n points in the image and using Algorithm 2. Although any distribution of initial points will eventually converge, it is useful to start with a distribution that approximates the final form. Deussen *et al.* use a dithering algorithm and we use simple rejection sampling to generate an initial distribution [2].

4.3 Non-linearity of the density/area relationship

One difficulty that the reader will notice is that we have no well-defined relationship between *input density* and the resulting *area* of a Voronoi region. The previous discussion makes clear that areas with higher density will attract more generators, which in turn will cause the Voronoi regions in that area to shrink. Smaller Voronoi regions mean that the generators are closer together, but in no well-defined way with respect to the density. Ideally, we would like a linear inverse relationship between region area and region density: that is, the greater the integrated density in a particular Voronoi region, the smaller the region's area. If a fixed-sized primitive is placed in each Voronoi region, then the coverage of ink on the paper would scale linearly with input density, which is our ultimate goal. Figure 4.2 shows a scatter plot of the ink coverage versus the input image density of 5000 Voronoi regions. The scatter plot is obtained by using an input image consisting of a linear ramp which goes from zero density to full density. The system is then seeded with 5000 points and allowed to relax. The integrated density, area, and other attributes of each Voronoi region are then written to file for examination. The ink coverage for a particular primitive is estimated by dividing the primitive's area by the uncovered area surrounding the primitive, that is, the area of the Voronoi region. In Figure 4.2 the primitive is fixed as a disc whose area equals the area of the smallest Voronoi region. This value is chosen so that in black regions, where the Voronoi regions are smallest, the primitive completely covers its Voronoi region. No fixed primitive size is ideal, however, and different sizes could be chosen.

There are several notable features of Figure 4.2. First, the relationship between input density and output ink coverage is far from linear. In particular, the ink coverage is too low for most of the input density range, which would result in washed-out midtones and shadows. Second, the output coverages become more scattered with

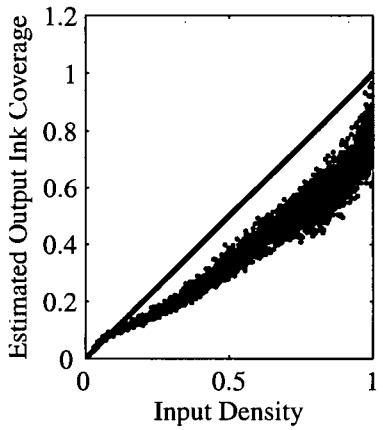


Figure 4.2: Input image density and corresponding ink coverage using reasonable fixed-radius stipples.

increased input density. The reason for this is that the Voronoi regions for higher-density areas are quite small and close to the resolution of the discrete Voronoi diagram. Fitting a single curve to the coverage relationship would be difficult. Increasing the resolution of the diagram and allowing it to relax for more iterations reduces the variation in the higher density areas because the Voronoi regions become better formed and closer to true hexagons. However, increasing the regularity of the smallest Voronoi regions is undesirable for several reasons: large memory usage, long computation times, and overly-regular patterns in the output. Reducing the variability present in Figure 4.2 is not a viable option.

The practical consequence of the non-linearity between input image density and output ink coverage is that any particular fixed primitive size cannot faithfully reproduce the input image tones. Changing the primitive size is equivalent to scaling the points in Figure 4.2 by a constant factor, but it is tedious to find a reasonable value even if some non-linearity is accepted. Figure 4.3 shows the resulting ink coverage if the user happens to select a primitive size half that of Figure 4.2.

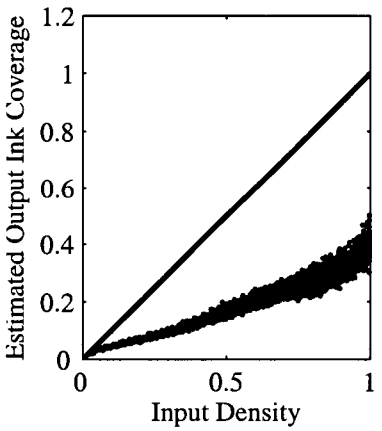


Figure 4.3: Input image density and corresponding ink coverage using fixed-radius stipples half of reasonable value.

Clearly input tones are not being reproduced linearly.

A solution is to set the primitive size individually for each Voronoi region such that the ratio of the primitive's area to the Voronoi region's area equals the input density. The details for doing so for disc-shaped stipple primitives can be found in Section 4.4. Figure 4.4 shows the ink coverage if this method is applied. Note that the coverage is almost ideal for most values of input density and only slightly deviates from linear. Also, the over-coverage for input densities greater than 0.9 compensates for the overlap between primitives. If overlap is not compensated for, then very dark shadow regions will wash out. The actual coverage of ink is very nearly linear for all values of input density. The size calculations, overlap compensation and reasons for small non-linearities are explained in Section 4.4.

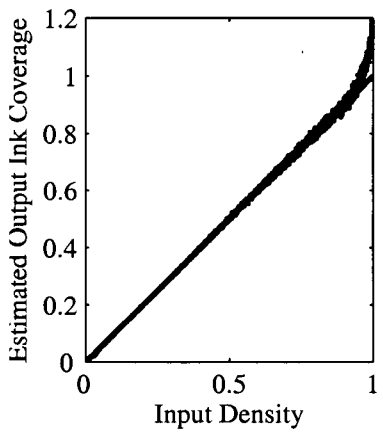


Figure 4.4: Input image density and corresponding ink coverage using variable-radius stipples.

4.4 Correlating primitive size and ink coverage

We have the desired ink coverage for each primitive over its Voronoi region by integrating the input image. We will adjust the scale of each primitive to achieve this coverage, but we need to know the relationship between primitive scale and coverage. Note that we will be only using simple scales to increase or decrease the ink coverage of a particular "base" primitive. We could adjust the ink opacity or even the colour of the background Voronoi region, but these effects are not easily achievable on many common devices such as printers. Furthermore, keeping at least one eye on tradition, adjusting the opacity of the ink for each of several thousand primitives violates the spirit of our endeavour.

In general, this problem is hard, since many scales will cause primitives to overlap with each other, and this overlap affects the ink coverage. This was discussed previously with respect to modifying the probability density functions in Section 3.1. In the worst case, the overlap between any particular primitive de-

pends on all the primitives which surround it, including *their* individual scales. Clearly the general problem must be solved for all primitives simultaneously, and not for each primitive individually. In this most general case, it is difficult to decouple the ink coverage of any particular primitive from its neighbours.

However, in the case of stippling with relaxed point distributions, there is a nice simplification which allows the problem to be treated analytically. The optimal packing of discs in the plane is a hexagonal grid, like that of a honeycomb. The Voronoi regions produced by Lloyd's algorithm are approximately congruent to hexagons, depending on how many iterations are used [17]. Indeed, in the constant-density case on the infinite plane, Lloyd's algorithm will produce a perfectly regular hexagonal grid.[2] Even when a non-constant density function is used, such as is the case for our stippling algorithm, the Voronoi regions form near-hexagonal shapes of various scales.

We will assume that when the generators have become sufficiently relaxed, each Voronoi region approximates a regular hexagon. Furthermore, we will assume that the scales of the Voronoi region do not change drastically over the scale of a single hexagon. These assumptions let us assume that any Voronoi region is surrounded by a regular hexagonal grid of identical size, denoted by s, where s is measured from the centre of the hexagon to the centre of a side. Clearly this assumption becomes more valid with larger numbers of generators, since the density variation a particular Voronoi region covers will shrink. To illustrate the validity of this assumption for a typical input image, Figure 4.5(a) shows a test input image and the centroidal Voronoi diagram of 400 generators. This is a very small number of stipples to use for the test image, as is illustrated its stipple drawing in Figure 4.5(b). Note that in the central region the Voronoi regions are indeed

[2]Finite regions, such as the ones used in this dissertation, show slight edge effects near the boundaries. This does not change our argument.

roughly hexagonal, but the regions centred on the horizontal lines are far from hexagonal. The regions on these lines violate our assumption that each Voronoi region is approximately in a grid of hexagons, but any deviations from correct tone reproduction near edges is very difficult to detect in practice.

To calculate the relationship between coverage and stipple scale, we centre a disc of radius r in each hexagon. Figure 4.5 shows the configuration. Note that the disc will completely cover the hexagon when $r/s = 2/\sqrt{3}$. As we increase the radius of the disc from zero to $2/\sqrt{3}s$, there are clearly three distinct regimes: the disc is completely contained in the hexagon, the disc touches or overlaps the hexagon's edges, and the disc completely covers the hexagon. These are conveniently parameterised by $\sigma^2 \equiv r^2/s^2$ in the following ranges:

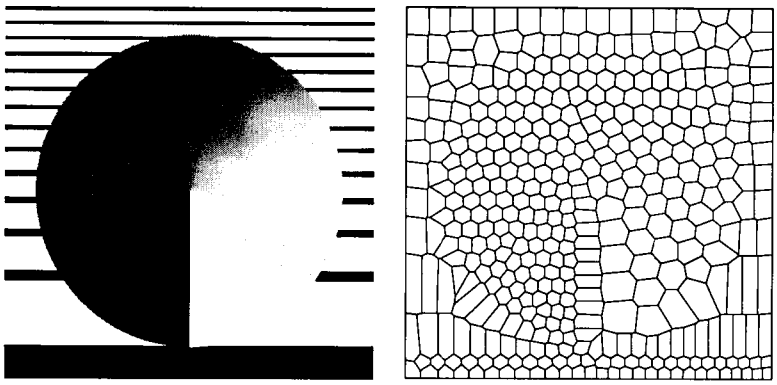
$$\begin{aligned} 0 \leq \sigma^2 < 1 &\quad \text{disc contained,} \\ 1 \leq \sigma^2 < 4/3 &\quad \text{disc overlapping,} \\ \sigma^2 > 4/3 &\quad \text{disc covers.} \end{aligned}$$

4.4.1 Disc completely contained in hexagon

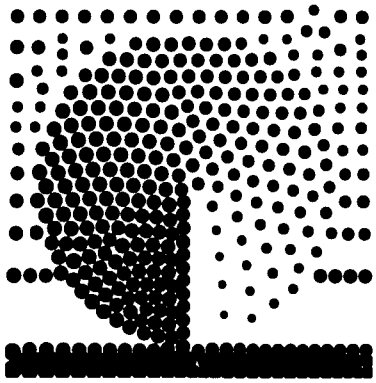
If $\sigma^2 \leq 1$, then we have the simplest case, shown in Figure 4.5, where the disc is completely contained in the hexagon. In this case each disc in the assumed hexagonal grid is independent and does not overlap any other. The disc coverage is then just the ratio of areas:

$$\frac{\text{area of disc}}{\text{area of hex}} = \frac{\pi r^2}{2\sqrt{3}s^2} = \frac{\pi}{2\sqrt{3}}\sigma^2 \tag{4.2}$$

where we have used the area of a hexagon as $2\sqrt{3}s^2$.



(a) Test input image and resulting centroidal Voronoi diagram of 400 generators.



(b) Stipple drawing of the test input image.

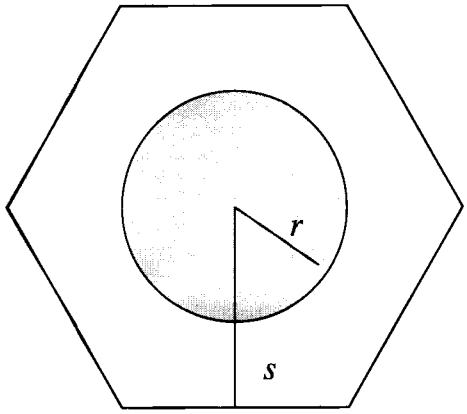


Figure 4.5: The general case of a single tile of the disc packing.

4.4.2 Disc completely covers hexagon

If $\sigma^2 \geq 4/3$ then the disc completely covers the hexagon and we have

$$\frac{\text{area of disc}}{\text{area of hex}} = 1. \tag{4.3}$$

4.4.3 Disc overlaps hexagon edges

If $1 < \sigma^2 < 4/3$ then the disc will overlap the edges of the hexagon and the neighbouring discs. This case is shown in Figure 4.6. The area of the wedge $A_1 + A_2$ is simply $\pi r^2 \frac{\alpha}{2\pi} = \alpha r^2/2$ where α is the angle contained by r and s: $\alpha = \arccos s/r$. The area A_1 of the triangle is $\frac{s}{2}\sqrt{r^2 - s^2}$. Hence, A_2, the partial overlap region is $(\alpha r^2 - s\sqrt{r^2 - s^2})/2$. There are twelve such regions about the hexagon, so the effective area of the disc is $\pi r^2 - 12A_2$ or

$$A_{disc} = \pi r^2 - 6(\alpha r^2 - s\sqrt{r^2 - s^2}). \tag{4.4}$$

The ratio of areas covered by the disc is (after some shuffling)

$$\frac{\text{area of disc}}{\text{area of hex}} = \frac{(\pi - 6\alpha)\,\sigma^2 + 6\sqrt{\sigma^2 - 1}}{2\sqrt{3}} \tag{4.5}$$

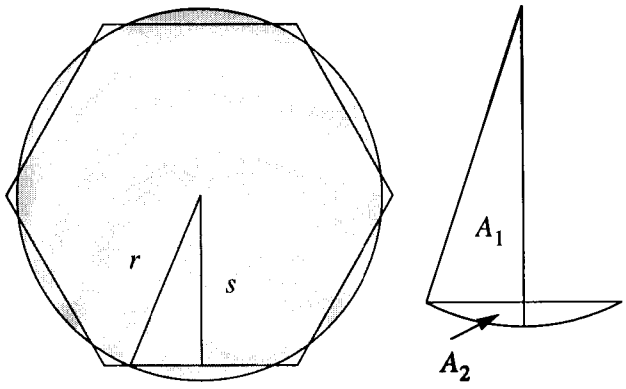


Figure 4.6: A single tile of the disc packing where the disc overlaps the hexagon edges (left). Detail of the overlapping region (right).

The two base cases are when the disc just touches the hexagon ($\sigma^2 = 1$) and when the disc fully covers the hexagon ($\sigma^2 = 4/3$). If $\sigma^2 = 1$ then $\alpha = 0$ and the square root cancels, leaving just

$$\frac{\text{area of disc}}{\text{area of hex}} = \frac{\pi\sigma^2}{2\sqrt{3}} \tag{4.6}$$

which matches Equation 4.2. If $\sigma^2 = 4/3$ then $\alpha = \pi/6$ and we have

$$\begin{aligned} \frac{\text{area of disc}}{\text{area of hex}} &= \frac{(\pi - 6(\pi/6))4/3 + 6\sqrt{4/3 - 1}}{2\sqrt{3}} \\ &= \frac{2\sqrt{3}}{2\sqrt{3}} \\ &= 1 \end{aligned}$$

which matches Equation 4.3.

Figure 4.7 shows the complete disc coverage function for all pertinent values of σ^2. Note that σ^2 has units of dimensionless area, the same as coverage. If we plotted r/s, which has units of dimensionless length, then the plot would show a typical x^2-like shape.

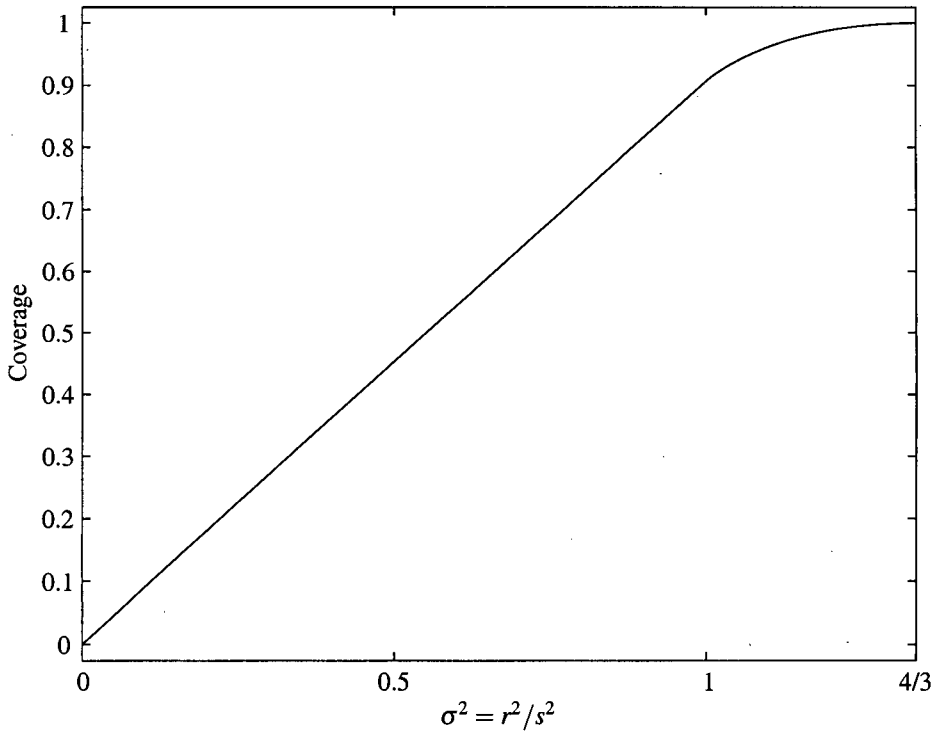


Figure 4.7: Ratio of disc area to hexagon area for values of $\sigma^2 = r^2/s^2$.

4.4.4 Correcting non-linear tone reproduction

In Section 4.4 we showed how to calculate the coverage of a disc in a hexagon for any radius r relative to a hexagon radius s. This relationship can be analytically inverted to give a radius that a disc must be to achieve a given coverage for a given hexagon radius. We use this relationship to set the size of each stipple based on the integrated input image darkness and the Voronoi region's area. Using the assumption that every Voronoi region is approximately hexagonal, we compute the radius of a hexagon with the same area as the Voronoi region. We want the ratio of area covered by ink to area not covered by ink to match the integrated darkness of the input image for every Voronoi region. Once we have calculated this integrated darkness, we invert the relationship shown in Figure 4.7 to get σ^2, and then use our estimated value of the hexagon radius s to get r. This radius is used to set the size of each stipple in the output. Note the correspondence between the overlap regions in Figures 4.7 and 4.4 from the previous section. Figures 4.11 and 4.12 in the next section compares a stipple drawing with fixed-sized stipples to one with variable-sized stipples.

4.5 Results

We expect that at the limit of large numbers of very small stipples, the stipple drawing will approximate the grayscale image. Centroidal Voronoi diagrams produce distributions of points that approximate a blue noise distribution, that is, a random distribution with a constraint on the minimum distance between points. Blue noise distributions are useful because they do not introduce spurious patterns such as lines or grids. They can also approximate a constant tone because of the minimum distance constraint. Blue noise distributions have been used to create very high-

quality dither patterns for colour reduction [27]. Figure 4.8 shows a grey-scale close-up image of some Peperomia leaves with a drawing of 20000 stipples. The fine stippling approximates the tones of the image very well, including the textures inside the leaves.

Figure 4.9 shows a different small Peperomia plant, lit from the side, with 20000 stipples. Although the number of stipples per square inch is less than in Figure 4.8, the large number of stipples still renders a faithful image. In particular, note the hard edges maintained by the stipple drawing. Figure 4.10 shows the full Peperomia plant from Figure 4.8 with 20000 stipples. Observe the colouration of the centre of the leaf facing the viewer. While the method of Deussen *et al.* can easily produce sharp edges through user interaction, producing the gradual change in tone visible on the leaf would be difficult. The even spacing of points along the edges of the leaves is the result of the interaction of the centroidal Voronoi diagram, which attempts to space all points evenly, and the density function ρ, which restricts points to the essentially one-dimensional edge.

However, a more interesting test is to apply the method with low stipple counts. Smaller numbers of stipples mean that we cannot rely upon the eye to fuse the tiny size and spacing of the dots into a continuous tone. Figure 4.11 shows an image of an artist's mannequin and the stippled version with 1000 stipples. Figure 4.12 shows a climbing shoe in the same format. Note that both the stipple drawings are quite recognisable, especially in comparison to Figure 4.13, where the source images have been reduced in resolution until they contain approximately 1000 pixels each[3].

Finally, we note that the most striking drawings come from neither very-high

[3]This comparison is not quite fair, as the 1000 pixels are forced to be equally spread across the image whereas the stipples are free to move. The point is that the stippling maintains edges and silhouettes even at very low resolutions.

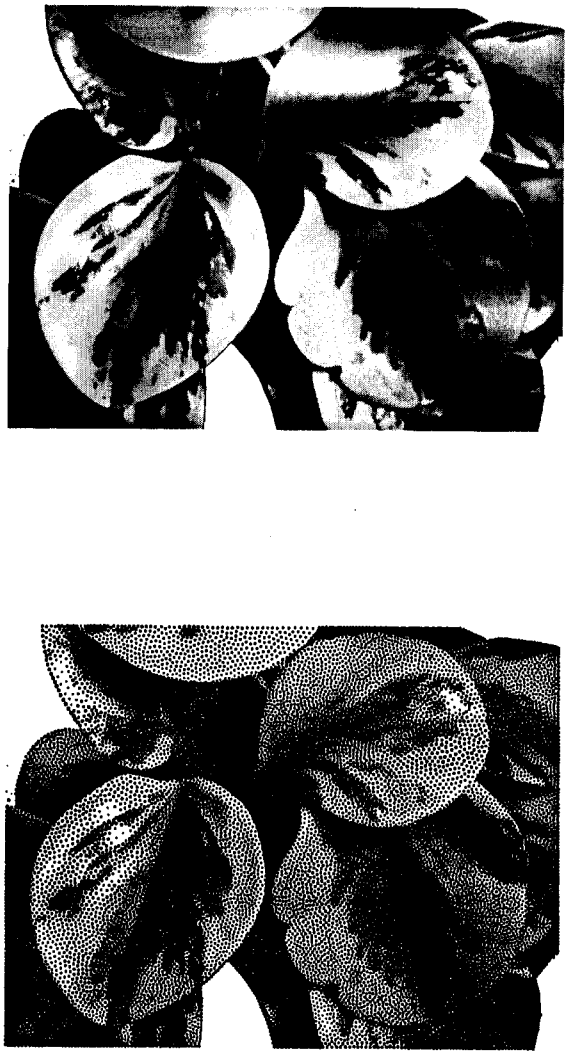


Figure 4.8: Close-up of large Peperomia leaves with 20000 stipples.

Figure 4.9: Small Peperomia plant, lit brightly from the right, with 20000 stipples of fixed radius 1.0×10^{-3}.

Figure 4.10: Large Peperomia plant with 20000 stipples.

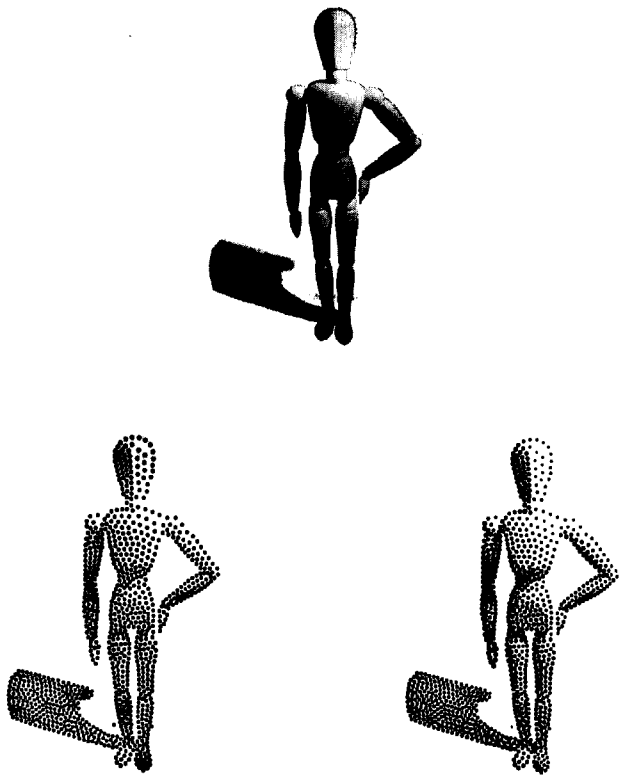


Figure 4.11: Figure with 1000 stipples. Stipples in the bottom left have fixed radius 5×10^{-3} and those in the bottom right have variable radius.

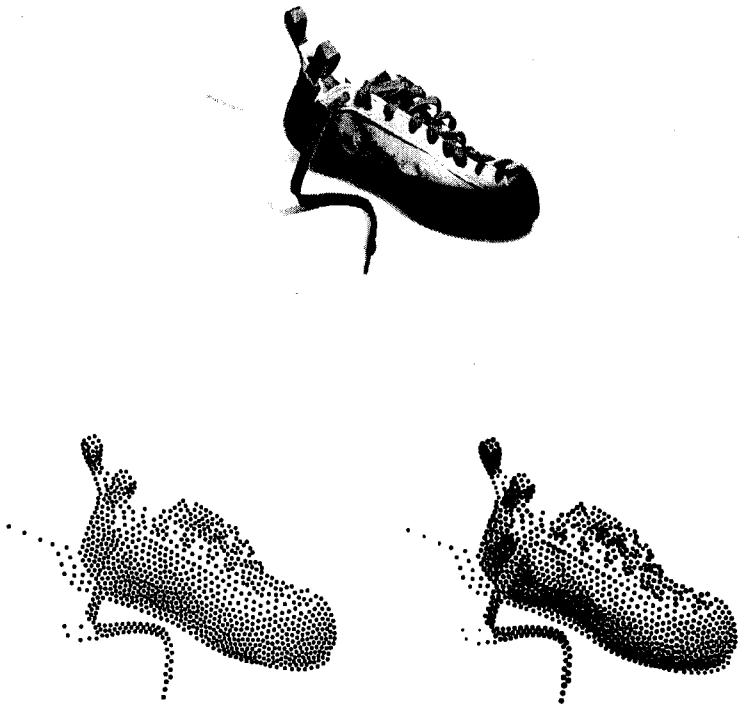


Figure 4.12: Climbing shoe with 1000 stipples. Stipples in the bottom left have fixed radius 5×10^{-3} and those in the bottom right have variable radius.

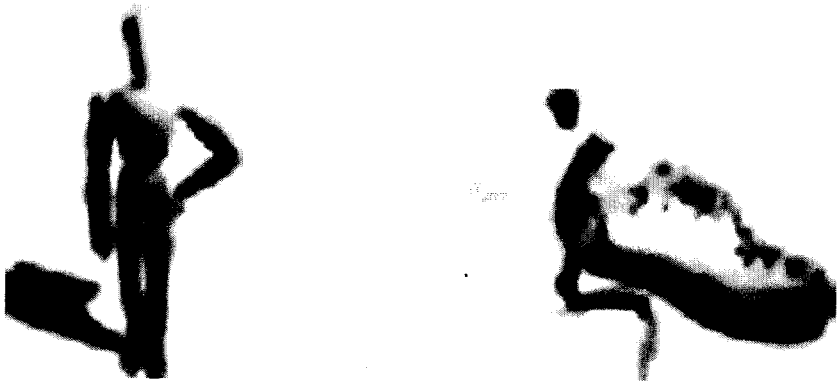


Figure 4.13: Source images of Figures 4.11 and 4.12 rendered with approximately 1000 pixels instead of stipples

nor very-low numbers of stipples, but medium ranges. Figure 4.14 shows the climbing shoe of Figure 4.12 rendered with 5000 stipples. This drawing seems to both reproduce the range of tones from the original and have the "feel" of a real stipple drawing. Figure 4.15 shows a corn plant rendered with 20000 stipples and displaying both colouration on the leaves and sharp boundaries on the edges. We feel that this image begins to live up to the quote by Hodges in [10], page 111, in which he attests to the vibrancy of stippled images: "Like a pointallist painting, the drawing will appear to vibrate slightly."

4.6 Parameters and timings

We computed all the stipple drawings of Section 4.2 on an Intel Pentium III 1000 MHz machine with 256 Mb of RAM and a NVIDIA GeForce2 MX graphics accelerator. As discussed in Section 4.1.3, we require the Voronoi regions to have an average area of at least 500 pixels, which forces a virtual resolution of up to 3600 by 3600 pixels for the 20000 stipple drawings. Since we precompute the in-

Figure 4.14: Climbing shoe with 5000 stipples.

tegrals P and Q from Section 4.1.2 at full virtual resolution, this requires upwards of 100 Mb of memory. The memory requirement could be reduced by an order of magnitude by computing the integrals in tiles in the same way that the Voronoi diagrams are computed, but this did not seem necessary.

The iterations were stopped and the stipple drawing output when the difference in the standard deviation of the area of the Voronoi regions in successive iterations was less than 1×10^{-4}. Because the background of the input images was not always pure white, stipples were only output if the input image value at that location was greater than 99% of pure white.

On the system used, the stipple drawings with up to 5000 stipples completed in under a minute and the drawings with 40000 stipples complete in about 20 minutes on an otherwise unloaded machine. The 1×10^{-4} stopping limit was arbitrarily chosen and different values will lead to different runtimes.

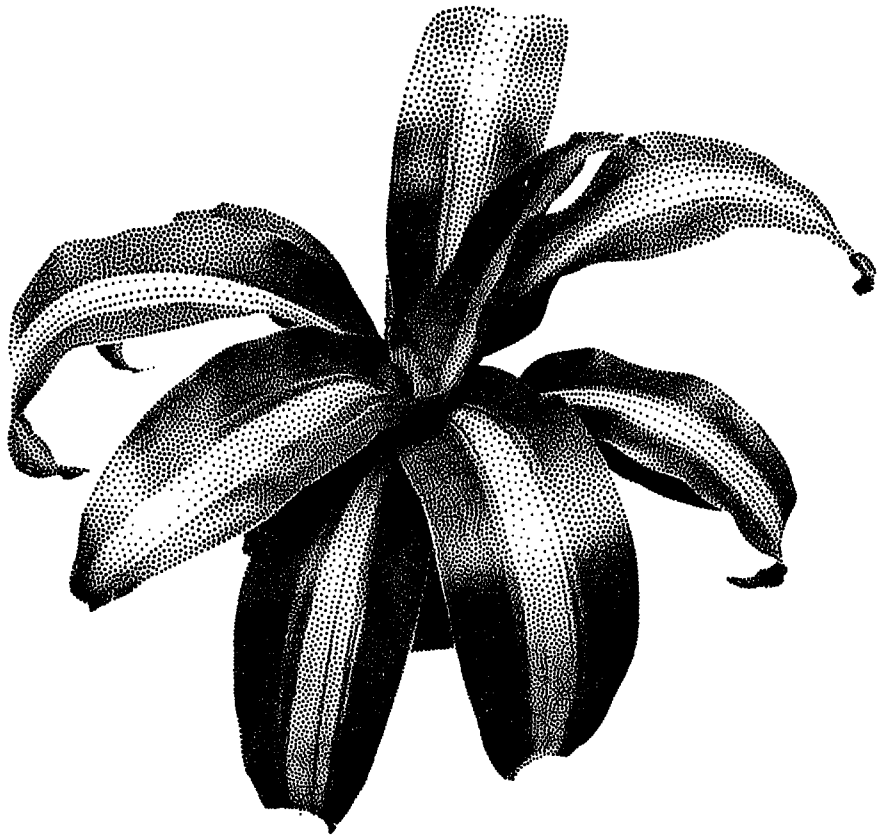


Figure 4.15: Corn plant with 20000 stipples.

4.7 Extensions and future work

Lloyd's algorithm involves an averaging filter. The movement of a generator is based solely on its computed centroid, which is essentially the weighted average position of the Voronoi region. This should not be surprising, since many non-linear solvers use some form of filtering of their data, particularly if the data is noisy. Lloyd's algorithm is interesting, however, because as the iteration continues, areas with higher weights will accumulate more generators than areas with lower weights. In turn, the Voronoi regions must shrink in area, and so the averaging filter will operate on smaller regions. The result is that areas with higher density will be filtered less than areas with lower density. In the language normally used with filters, the width of the filter kernel adapts to the underlying region's weight.

We can illustrate this easily in our image context by generating a "mosaic". Mosaics are an art form where images are made from small ceramic, glass or stone tiles, often decorating walls or table surfaces. Hoff *et al.* made extremely simple Voronoi-based image mosaics by generating a Voronoi diagram with an arbitrary number of points [11]. Each Voronoi region is then replaced with the average colour of an input image over that region. The result is an image with a number of roughly hexagonal constant-colour regions that represent the image.

We generate an image in which the pixels belonging to a particular Voronoi region are replaced with the average pixel value in the region. Figure 4.16 shows an example mosaic and Voronoi diagram. Note that the density function is the same as the input image, as we have previously presented, so the size of the Voronoi regions are smaller where the input image is darker. The original motivation for this was to pack *constant-sized* stipples closer together in darker regions to increase the ink coverage in the output. However, since we are dynamically adjusting the stipple sizes to reproduce tone linearly, we are no longer required to use the input image as

the density function. Figure 4.17 shows a mosaic and Voronoi diagram for the same input image, but in this example the density function is the arbitrary image shown at top. Clearly the Voronoi regions at the top of the mosaic and diagram are much smaller than the bottom. That is, the density function can control the "resolution" of the output drawing. Here we intend the term resolution to roughly mean the scale of smallest features detectable. Finally, Figure 4.18 shows a possibly more fruitful use of the density image. The density image in this example was created by filtering the input image with a Sobel filter and processing the resulting edge pixels with a Gaussian filter of width 25 pixels. The result is related to a distance function from the edges. From the diagram, it is clear that the density image is forming smaller Voronoi regions, and hence higher "resolution", near the edges.

In general, however, we would like to associate higher resolution with "interesting" features of the image, as is generally done in many other areas. For example, using a portrait as an input image, the density image could be darker around the face of the subject, bringing higher resolution to that area.

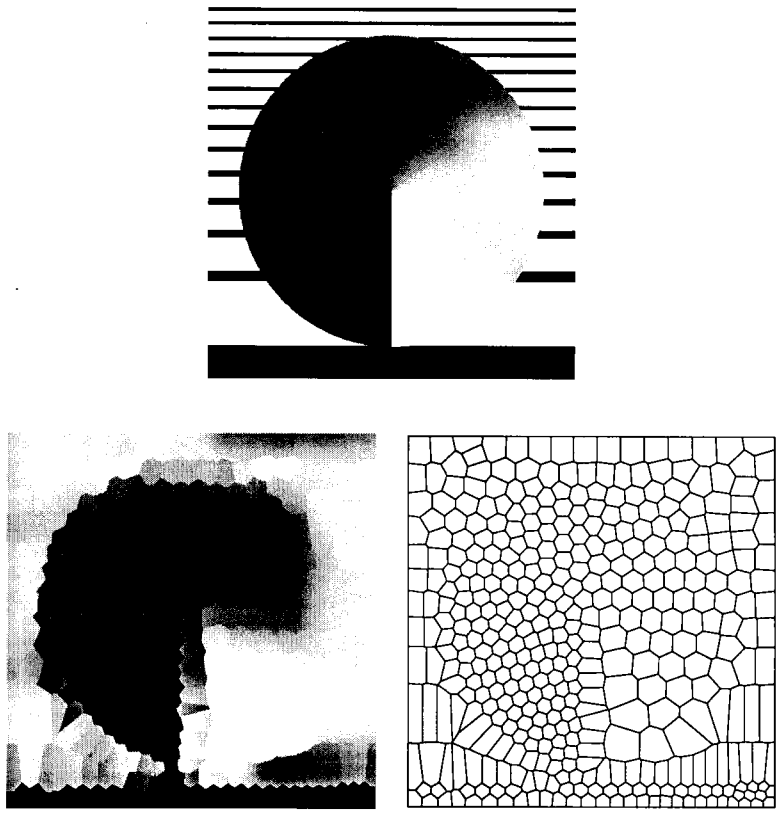


Figure 4.16: Sample image with mosaic and Voronoi diagram. The density image is the same as the input image.

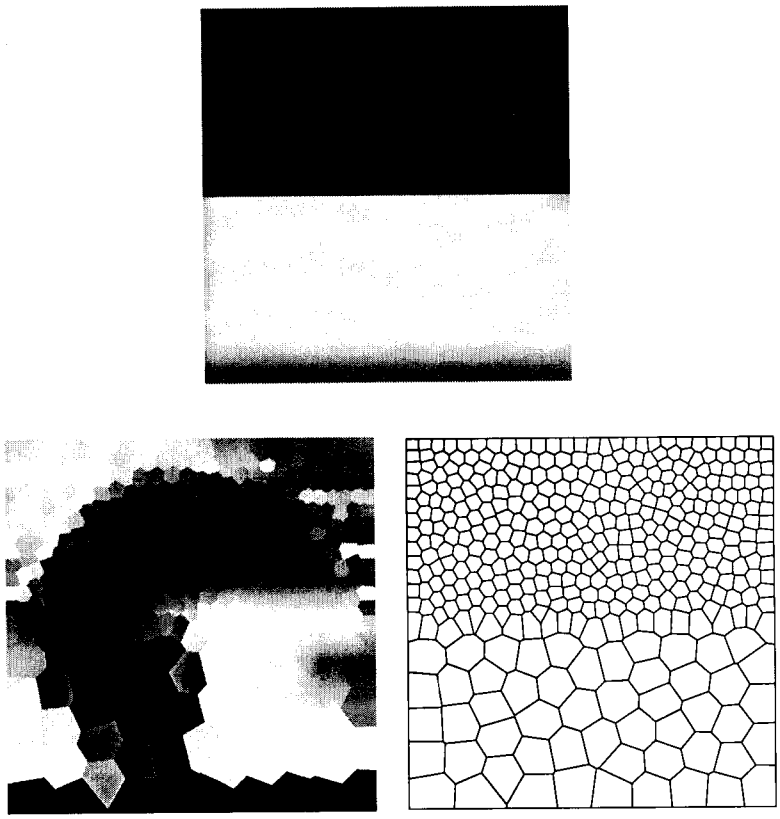


Figure 4.17: Sample image mosaic and Voronoi diagram. The density image is at the top.

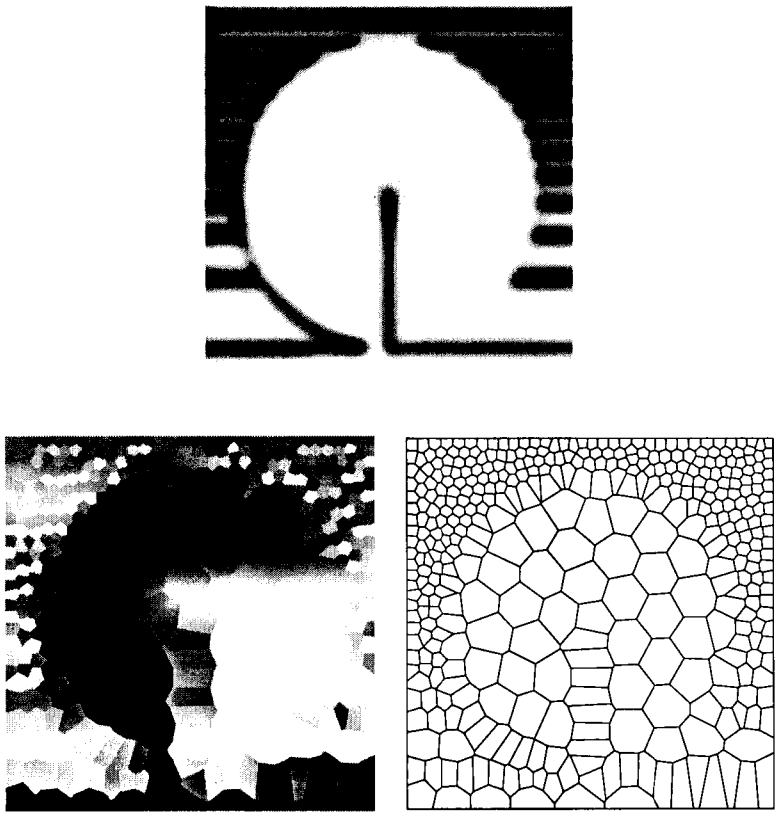


Figure 4.18: Sample image mosaic and Voronoi diagram. The density image at the top was obtained by applying the Sobel filter to the original image and blurring with a Gaussian kernel of width 25 pixels.

Chapter 5

Discussion

We have presented two flexible image-space algorithms for non-photorealistic rendering with small primitives. Our probabilistic algorithm generates continuous tone illustrations at interactive frame rates. Our approach for distributing primitives naturally provides scale independent results with some measure of frame-to-frame coherency. That is, individual strokes move continuously as the input image changes. We demonstrate that a number of different drawing styles are possible using only an image as an input. More styles are possible if additional information such as silhouette edges or material IDs can be derived from a 3D model.

Preservation of continuous tone with our approach is dependent on the size of the primitive, the feature to be preserved (i.e. edges) and the viewing distance. This is a standard limitation, since, as with any halftoning technique, the formation of continuous tone from a discrete palette relies on the spatial integration of the eye. Thus, our approach preserves tone if the choice of drawing primitive, image or model and viewing distance is reasonable. Our frame rate depends on several factors (brightness, size of PDF table, drawing style etc.) and is thus not constant. This could limit the applicability of our method in some situations. A final limitation is the distance a primitive moves from frame to frame. While primitives move continuously, the apparent movement from frame to frame can be distracting, especially at the lower frame rates.

In the future we would like to extend the tone correction method to work for

grayscale and colour primitives or graftals [14]. Grayscale or semi-transparent strokes or brushes could be used to simulate other non-photorealistic rendering effects such as watercolour [1]. Additional pen-and-ink styles could also be incorporated into our method. For example, it would be easy to clip strokes against object masks rendered using an ID buffer algorithm. This would preserve discontinuities in the image without having to align strokes orthogonal to the gradient field. Finally, it should be possible in the future to offload the placement of the stipples onto the graphics chip, which would drastically reduce the bandwidth requirements of the method. This would require a programmable vertex engine similar to the one described by Lindholm *et al.* [15], but with the additional possibility to perform per-vertex table lookups.

Our weighted centroidal Voronoi-based algorithm has very few user-specified parameters and requires no user interaction. The input data are grayscale images which can be produced by a wide variety of sources. Apart from simply requiring less work to generate a given stipple drawing, this independence allows cheap stippling to be used in a wider variety of situations than before.

A fundamental problem with weighted centroidal Voronoi diagrams is the unknown relationship between the underlying density and the size of the Voronoi regions. This problem is general and not specific to our application. It is this problem that is discussed in Section 4.3 and required the per-primitive size adjustments to maintain a linear tone mapping from input image to output drawing. Although adjusting the size of the primitives does fix the problem, in a sense it is redundant. That is, the stipple positions could be arbitrarily placed in the output image, and linear tone could still be achieved by the brute-force changing the size of each primitive. However, using Lloyd's algorithm to position the stipples ensures that a reasonable drawing is produced even if constant-sized stipples are used. Furthermore, a good placement of stipples reduces the variation in stipple size required to

reproduce the input tone.

Several interesting extensions to the current algorithm could be investigated including the use of colour stipples, and stippled animations should be investigated, as well.

Chapter 6

Conclusions

We have presented two novel and complementary non-photorealistic rendering methods which use arbitrary greyscale images as input. Our fast, probabilistic method generates a probability density function (PDF) from the input image which is constructed to compensate for the overlap of primitives. The PDF is then used to distribute the primitives on a canvas, reproducing the input greyscale tone. The style of the final rendering is controlled by the user through the type, scale and orientation of the primitives. The probabilistic algorithm is capable of real-time performance on modern hardware and many different styles. Using centroidal Voronoi diagrams weighted with the input image, our iterative algorithm positions stipples carefully to avoid overlap. We use Lloyd's algorithm to relax an initial distribution of positions into a well-spaced arrangement. Finally, the sizes of the resulting stipples are adjusted to produce linear tone mapping from the input image to the output drawing. The result is that of a carefully-stippled drawing of the input image.

Bibliography

[1] C. J. Curtis, S. E. Anderson, J. E. Seims, K. W. Fleischer, and David H. Salesin. Computer-generated watercolor. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 421–430. ACM Press/Addison-Wesley Publishing Co., 1997.

[2] O. Deussen, S. Hiller, C. van Overveld, and T. Strothotte. Floating Points: A method for computing stipple drawings. *Computer Graphics Forum*, 19(3), August 2000.

[3] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi tessellations: Applications and algorithms. *SIAM Review*, 41(4):637–676, December 1999.

[4] B. Gooch, P. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999.

[5] A. Guptill. *Rendering in Pen and Ink*. Watson and Guptill Publications, 1997. 60th anniversary edition, edited by S. Meyer.

[6] P. Haeberli. Paint by numbers: Abstract image representations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 207–214. ACM Press, 1990.

[7] J. Halton and G. Weller. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7(12):701–702, July 1964.

[8] J. Hammersley and D. Handscomb. *Monte Carlo Methods*. Wiley, 1964.

[9] A. Hausner. Simulating decorative mosaics. In Eugene Fiume, editor, *Proceedings of SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 573–578, New York, 2001. ACM, ACM Press / ACM SIGGRAPH.

[10] E. Hodges, editor. *The Guild Handbook of Scientific Illustration*. Van Nostrand Reinhold, 1989.

[11] K. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 277–286, New York, 1999. ACM, ACM Press / ACM SIGGRAPH.

[12] Z. Jastrzebski. *Scientific Illustration*. Prentice-Hall, 1985.

[13] B. Jobard and W. Lefer. Creating evenly-spaced streamlines of arbitrary density. *Proceedings of Eighth Eurographics Workshop on Visualization in Scientific Computing*, April 1997.

[14] M. A. Kowalski, L. Markosian, J. D. Northrup, L. Bourdev, R. Barzel, L. S. Holden, and J. F. Hughes. Art-based rendering of fur, grass, and trees. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 433–438. ACM Press/Addison-Wesley Publishing Co., 1999.

[15] E. Lindholm, M. Kilgard, and H. Moreton. A user programmable vertex engine. In *Proceedings of SIGGRAPH 2001*, pages 149–158, August 2001.

[16] M. McCool and E. Fiume. Hierarchical poisson disk sampling distributions. In *Proc. of the Graphics Interface '92*, pages 94–105, Vancouver, Canada, 1992.

[17] D. J. Newman. The hexagon theorem. *IEEE Transactions on information theory*, 28(2):137–139, 1982.

[18] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Wiley & Sons, 1992.

[19] J. Pitman. *Probability*. Springer, 1992.

[20] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1993.

[21] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 197–206. ACM Press, 1990.

[22] A. Secord. Weighted Voronoi stippling. In *Proceedings of the second international symposium on Non-photorealistic animation and rendering*, pages 37–43. ACM Press, 2002.

[23] A. Secord, W. Heidrich, and L. Streit. Fast primitive distribution for illustration. In Paul Debevec and Simon Gibson, editors, *Rendering Techniques 2002*, Eurographics. Springer-Verlag Wien New York, 2002. Proc. 13th Eurographics Rendering Workshop, Pisa, Italy, June 26–28, 2002.

[24] I. Sobol. On the distribution of points in a cube and the approximate evaluation of integrals. *U.S.S.R. Computational Math. And Math. Phys*, (4):86–112, 1967.

[25] G. Turk and D. Banks. Image-guided streamline placement. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 453–460. ACM Press, 1996.

[26] R. A. Ulichney. *Digital Halftoning*. MIT Press, 1987.

[27] R. A. Ulichney. Dithering with blue noise. *Proceedings of the IEEE*, 76(1):56–79, 1988.

[28] G. Winkenbach and D. Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of SIGGRAPH 94*, pages 91–100, July 1994.

[29] G. Winkenbach and D. Salesin. Rendering parametric surfaces in pen and ink. In *Proceedings of SIGGRAPH 96*, pages 469–476, August 1996.

[30] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide*. Addison-Wesley, second edition, 1997.

[31] P. Wood. *Scientific Illustration*. Van Nostrand Reinhold, second edition, 1994.

Appendix A

Implementation Notes

A.1 Colour and gamma issues

We attempt to ensure that the entire system is linear in converting encoded values from the input images into ratios of coverage of black regions to white regions in the output. Whether or not this results in a linear mapping from encoded values to perceived lightness is entirely dependent on the input encodings themselves and the methods used to render the black and white regions. Since we wanted our system to handle a wide variety of input images, many of which have no context at all with respect to the interpretation of their values, we believe that this is the best we can do at present. If the characteristics of the output medium are known (for example, dot-gain curves for a particular printer or gamma values for a monitor), then the *inverse* of that transfer function can be applied to the input image encoded values. Since the system is linear, the inverse transfer function will be carried through the system unmodified and the rendered output will be appropriately compensated.

The stippling system can read 16-bit image channels for image types that support them, and internally represents images with 32-bit IEEE floating points numbers. Thus images can have transfer functions applied to them, stored in a 16-bit format such as TIFF or PNG, and then loaded normally. If this 16-bit precision is still not enough for a particular transfer function, then the system could be trivially modified to load a separate transfer function from an arbitrary format that uses full

32-bit IEEE floating point for storage.

The hatching system includes a transfer function that could be used for the same purpose.