Writing My Own Line Drawing Software as an Artist

by

Greg Philbrick

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Greg Philbrick 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	David Mould Professor, Dept. of Computer Science Carleton University
Supervisor:	Craig S. Kaplan Professor, Dept. of Computer Science University of Waterloo
Internal Member:	Dan Vogel Professor, Dept. of Computer Science University of Waterloo
Internal-External Member:	Rob Gorbet Professor, Dept. of Knowledge Integration University of Waterloo
Other Member(s):	Christopher Batty Professor, Dept. of Computer Science University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Chapter 4 comes in part from

Greg Philbrick and Craig S. Kaplan. 2019. Defining Hatching in Art. In Proc. Computational Aesthetics, Sketch-Based Interfaces, Modeling, Non-Photorealistic Animation, and Rendering (Genoa, Italy) (Expressive). Eurographics Association, Goslar, DEU, 111–121. https://doi.org/10.2312/exp.20191082

Chapter 5 comes in part from

Greg Philbrick and Craig S. Kaplan. 2022. A Primitive for Manual Hatching. ACM Trans. Graph. 41, 2, Article 21 (jan 2022). https://doi.org/10.1145/3503460

Abstract

I describe trying to improve my own art—line art, specifically—by developing computer science-based tools. The results of this experience are three technical contributions targeted at an NPR (Non-Photorealistic Rendering) research audience. The first is a formal definition of hatching, a traditional drawing technique. The second is the "hatching shape," a software primitive for rapidly performing hatching. The third is the invented problem of "mashing up" two drawings, along with an algorithm for solving this problem. In addition to these technical contributions, I relate how researching NPR changed me as an artist.

Acknowledgments

Early on, Linda Carson checked the sketchbook I had on me and was disappointed to see the kind of drawings you make just to get practice hours in. She wanted to know what something real from me looked like. Wish I could show you how far I got.

Thanks to Wanying and Becca for drawing with me during the pandemic.

Thanks to Craig for being a friend throughout.

Thanks to my family for getting me through life outside of art.

And thanks to Prof. Sederberg for teaching me B-splines [88].

Table of Contents

Examining Committee Membership ii

Author's Declaration iii

Statement of Contributions iv

Abstract v

Acknowledgments vi

List of Figures xi

List of Tables xvi

1 Introduction 1

1.1 An Artist's Story 1

1.2 Line Art 2

1.3 NPR and Illustration 2

1.4 My Testbed Application 4

1.5 Defining Hatching 6

1.6 Hatching Shapes 7

1.7 Drawing Mashups 8

2 Related Work **12**
2.1 NPR . . . 12
2.2 Interactivity . . . 13
2.3 Line Art . . . 15
2.3.1 Hatching . . . 17
2.3.2 Stippling . . . 19
2.4 NPR Position Papers . . . 20

3 My Testbed Application **22**
3.1 Introduction . . . 23
3.2 Design Decisions . . . 23
3.3 Primary Primitives . . . 24
3.4 Other Features . . . 27
3.4.1 Radial Mode . . . 28
3.4.2 Dot Patterns . . . 28
3.4.3 Perspective/3D Tools . . . 31

4 Defining Hatching **36**
4.1 Introduction . . . 36
4.2 Purposes . . . 37
4.3 Definition . . . 46
4.3.1 Curve . . . 47
4.3.2 Patch . . . 48
4.3.3 Pattern flexibility . . . 51
4.3.4 Pattern simplicity . . . 54
4.4 Future Hatching Research . . . 56
4.4.1 Creating hatching . . . 56
4.4.2 Analyzing hatching . . . 58
4.5 Conclusion . . . 62

5 Hatching Shapes **63**

5.1 Introduction . . . 63

5.2 Parts of a Hatching Shape . . . 66

5.3 Creating and Editing Hatching Shapes . . . 69

5.3.1 Creating a Hatching Shape . . . 69

5.3.2 Mask Editing . . . 70

5.3.3 Field Fill Operations . . . 71

5.3.4 Setting the Width Field From an Image . . . 75

5.3.5 Field Multiplication . . . 75

5.3.6 Crosshatching . . . 76

5.4 Barrier Curves and Operations Involving Them . . . 76

5.4.1 Blend Strokes . . . 79

5.4.2 Multi-Dir . . . 79

5.5 Streamline Advection . . . 81

5.5.1 The Dead Grid and the Collider Grid . . . 83

5.5.2 Occupancy Grid . . . 85

5.5.3 Interpolating Field Data . . . 87

5.5.4 Rasterization Issues . . . 88

5.5.5 When Interpolation Fails . . . 89

5.6 Results . . . 92

5.6.1 Artist Interviews . . . 97

5.7 Future Work . . . 98

5.7.1 Tonal Controls . . . 98

5.7.2 Spacing Artifacts . . . 98

5.7.3 Messy Hatching . . . 99

5.7.4 Depicting 3D Objects . . . 102

6 Drawing Mashups **103**

6.1 Introduction . . . 103

6.2 Related Work . . . 105

6.3 Learning to Mash . . . 108

6.4 Method . . . 112

6.4.1 Crossings . . . 112

6.4.2 Routing . . . 113

6.4.3 Tailable Intervals . . . 118

6.4.4 Tails . . . 120

6.5 Results . . . 125

6.6 Future Work . . . 129

6.7 Conclusion . . . 132

7 Conclusion **135**

7.1 The Fourth Contribution . . . 135

7.1.1 The groundwork for creative development comes from making pictures meant to show off my research. . . . 136

7.1.2 Creativity comes from encountering my research's weaknesses. . . . 139

7.1.3 Adaptations to my research's weaknesses can have unforeseen consequences. . . . 141

7.2 Where I Go from Here . . . 145

References **147**

List of Figures

1.1 Illustration by Beatrix Potter . . . 3
1.2 Digital painting from my undergraduate years . . . 4
1.3 Pen sketches . . . 5
1.4 Hatching by Dürer and myself . . . 6
1.5 Gibson and Doré . . . 7
1.6 Hatching shape . . . 8
1.7 Illustration featuring hatching shapes . . . 9
1.8 Stippling . . . 10
1.9 Drawing mashups . . . 11

2.1 "Computer-Generated Pen-and-Ink Illustration" . . . 13
2.2 Jobard and Lefer . . . 14
2.3 "MoXi" . . . 15
2.4 "Interactive Pen-and-Ink Illustration" . . . 16
2.5 "Digital Facial Engraving" . . . 18
2.6 Hedcut stippling . . . 20

3.1 Testbed application . . . 22
3.2 Intro to testbed application . . . 25
3.3 A stroke . . . 26
3.4 Refining strokes . . . 26

3.5 Strokes' over-simplicity . . . 26
3.6 Radial mode . . . 27
3.7 Regular and irregular dots . . . 29
3.8 Relaxed stippling . . . 30
3.9 Ruler tool . . . 32
3.10 3D rectangle . . . 33
3.11 Box patch and depth patch . . . 34

4.1 Hatching by Gibson . . . 37
4.2 White-on-black hatching . . . 38
4.3 Optical grays . . . 39
4.4 Optical grays from crosshatching . . . 39
4.5 Value performing multiple roles . . . 40
4.6 Ways hatching can show form . . . 40
4.7 Perspective from hatching . . . 40
4.8 Texture from hatching . . . 41
4.9 Texture as microforms . . . 41
4.10 Hatching as shorthand for texture . . . 43
4.11 Radiating hatching . . . 44
4.12 Symbolic hatching . . . 45
4.13 Hatching leading the eye . . . 45
4.14 Hatching's inherent beauty . . . 46
4.15 Closed hatching curves . . . 47
4.16 Ways hatching can trail off . . . 47
4.17 Hatching vs. stippling . . . 48
4.18 Patches . . . 49
4.19 Crosshatching . . . 50
4.20 Hatching with multiple patches . . . 50

4.21 Separability of hatching patches . . . 51
4.22 Context in identifying hatching . . . 52
4.23 Tablecloth example . . . 53
4.24 Silhouette contours vs. hatching . . . 54
4.25 Too complex to be hatching . . . 55
4.26 Trivial vs. sophisticated hatching . . . 57
4.27 Moiré patterns in CG hatching . . . 58
4.28 Flick-of-the-wrist hatching . . . 59
4.29 Hatching by Zorn . . . 60
4.30 Hatching by Kley . . . 61
4.31 Single-mark hatching patch . . . 61

5.1 Hatching shape introduction . . . 64
5.2 Masks and barriers . . . 65
5.3 Barrier-curve detail . . . 66
5.4 Example hatching shape process . . . 67
5.5 Hatching shape's mask . . . 69
5.6 Hatching shape creation . . . 70
5.7 Mask editing . . . 71
5.8 Fill operations on width and spacing fields . . . 72
5.9 Tree foreshortening . . . 73
5.10 Simple/linear fills on direction field . . . 73
5.11 Radial fills on direction field . . . 74
5.12 Width field from image . . . 75
5.13 Doubling density . . . 76
5.14 Crosshatching operation . . . 77
5.15 Dead cells . . . 77
5.16 Blend strokes . . . 78

5.17 Blend strokes and barriers . . . 80
5.18 Multi-dir operation . . . 80
5.19 Dead cells and multi-dir . . . 82
5.20 Streamline tapering . . . 82
5.21 Dead/collider grids . . . 84
5.22 Failing to account for barriers . . . 86
5.23 Excluding cells from interpolation . . . 87
5.24 Leaking across barrier . . . 88
5.25 Rasterization and dead cells . . . 90
5.26 Secondary seeds . . . 91
5.27 Increasing resolution to fix artifacts . . . 91
5.28 *Breeze* and *Twee* . . . 93
5.29 *Houppelande* . . . 94
5.30 *Breeze* has procedural and manual hatching. . . . 95
5.31 Broken egg . . . 96
5.32 Default vs. line-based seeding . . . 100
5.33 Clean and messy hatching . . . 101
5.34 Messy hatching . . . 101
5.35 Difficulty hatching a 3D form . . . 102

6.1 Drawing mashups frontispiece . . . 104
6.2 Making a picture from smaller pictures . . . 106
6.3 Making pictures with multiple readings . . . 107
6.4 Concept art for my drawing mashup technique . . . 109
6.5 The drawing mashup algorithm . . . 111
6.6 Stroke interval . . . 113
6.7 Crossings . . . 114
6.8 Crossing with six stubs . . . 114

6.9 A joint with connecting radius shown . . . 117
6.10 Tailless mashed strokes come after finding routings. . . . 118
6.11 AB field . . . 119
6.12 Tailable intervals . . . 119
6.13 Two ways of delimiting tailable intervals . . . 120
6.14 Tail creation . . . 121
6.15 r_{tail} . . . 123
6.16 d_{along} . . . 124
6.17 Interrupted join-with path . . . 124
6.18 Handle both tails at once. . . . 125
6.19 Computational mashup as just part of process . . . 126
6.20 The mashup algorithm can require experimentation. . . . 127
6.21 The mashup algorithm favors simple patterns. . . . 128
6.22 Computational mashup improved manually . . . 130
6.23 Preserve-drawing mode . . . 131
6.24 Tapering artifacts . . . 132
6.25 Concept art for a future mashup algorithm . . . 133

7.1 *Watch* . . . 137
7.2 *Sculptor* . . . 138
7.3 *Twee* redux . . . 140
7.4 NPR research gave me a fondness for dot patterns. . . . 141
7.5 *Spectacle* . . . 142
7.6 How I use hatching shapes lately . . . 143
7.7 Crude-stroke patterns . . . 144
7.8 James Gurney . . . 145

List of Tables

5.1 Data for *Breeze* and *Twee* 92

Chapter 1

Introduction

1.1 An Artist's Story

This is a dissertation in computer science (CS), meaning the author's job is to present technical contributions to other computer scientists. Traditionally there should not be much autobiography. Yet the *why* driving my research has always been art, and art is less scientific, more personal.

Even before I arrived at Waterloo, I had a plan: I would get my PhD by writing art tools. These tools would outwardly be "novel" and "interesting" enough to write publishable CS articles about, but their real purpose would be to increase my artistic ability. That ulterior motive has become the story tying my technical contributions together. It is a story that ends with a discovery, a discovery that is not just personally interesting: I consider it a fourth contribution. I discovered that CS did not just let me make the kinds of pictures I used to make, only faster. Writing art tools helped me imagine new kinds of pictures.

I do remember guessing at the beginning that CS research would take me to new artistic places, but I was skeptical, and I could not have guessed what those new places would look like.

To introduce a long-running metaphor for this dissertation, CS in the hands of an artist can become an artistic medium. I hope that by the end I will have illustrated that point.

I predict that my situation is generalizable, that other artists who use CS research to assist their artistic practice will be happily surprised by the effect the research has on their creativity. Formally speaking, this prediction is a *position* statement (Section 2.4).

1.2 Line Art

"Line art" is not a well-regulated term. I use it to mean images containing solid-colored shapes, typically black against white. These shapes are usually curves or dots. Shapes' edges are sharp, but not necessarily smooth. My intent with this definition is to capture the kinds of art achievable with old-fashioned print media, where ink is deposited in a binary way (it is there or it is not) from, say, a woodblock to a piece of paper. The sort of art I have in mind could be represented with a binary raster image (true means ink is present; false means it is not), which is not to say that all binarizable art should be considered line art.

1.3 NPR and Illustration

My research is in the subfield of CS known as Non-Photorealistic Rendering, or NPR. The "Non-Photorealistic" excludes CS literature on rendering 3D models photorealistically. (I elaborate on this distinction in Chapter 2.) NPR[1] is a home for CS researchers interested in visual art, whether or not their work involves "rendering" in the computer graphics sense. "Art" in NPR refers almost exclusively to images and video. The NPR community so far has not made many inroads into installation art, conceptual art generally, or even into postmodernism. NPR-tists prefer the kind of art that is meant mainly to look pretty.

Personally I work within the branch of art known as *illustration*. I have considered myself an illustrator since my undergraduate years, when I took many illustration courses. Within illustration, this dissertation focuses on line art.

The illustrator is a tradesperson, spiritually blue-collar. Traditional skills are celebrated in the illustrator's world, arguably more so than personal vision. For the illustrator, the commercial viability of a piece—its mass appeal, its *prettiness*—is paramount. This is why the greatest illustrators, like Norman Rockwell and Beatrix Potter (Figure 1.1), are household names.

An illustrator's understanding of art jibes with NPR. When I started my PhD, I found there were many published papers that matched my artistic sensibility. However, the illustration mentality limited me. I and other NPR researchers tend to focus on the prosaic *how* of art at the expense of the philosophical *what* and *why*. That narrow focus slowed

[1] If "NPR" seems like a poor name for the intersection of CS and art, it should. I use it only because it is the most recognizable term at the moment. A name found in some other literature is "Expressive Rendering."



Figure 1.1: *Illustration* is art intended to have mass appeal, to be pretty. Cover from *The Tale of Peter Rabbit*, Beatrix Potter, 1902, watercolor.

my artistic progress and delayed the discovery that this thesis is about: that writing your own drawing tools can change you as an artist.

Well into my research, publicity around AI image generators like Dall-E [73] and Stable Diffusion [93] forced me to take a wider view of my art, wide enough to glimpse this thesis's discovery. Suddenly I was seeing AI agents capable of making professional-looking illustrations, with no reason not to expect huge improvements in the near future. At first these uncanny pictures dismayed me. I was looking at the imminent death of the craft I had learned. AI art will one day make many of the illustrator's current skills obsolete, perhaps even obsoleting the word "illustrator" the way that CAD[2] software obsoleted "draftsperson."

My dismay turned to reflection. The advance of AI art helped me reimagine the purpose of my own art. It challenged my inner illustrator's belief that a legitimate artist must be traditionally skilled. More importantly, it gave me the perspective to consider CS—not just computer generation, but computer *science* itself—as a part of art, an art *medium*, even.

[2]Computer-Aided Design



Figure 1.2: As an undergraduate I did not connect well with any professional art medium. This digital painting was for a class assignment. Art is by me except when otherwise noted.

1.4 My Testbed Application

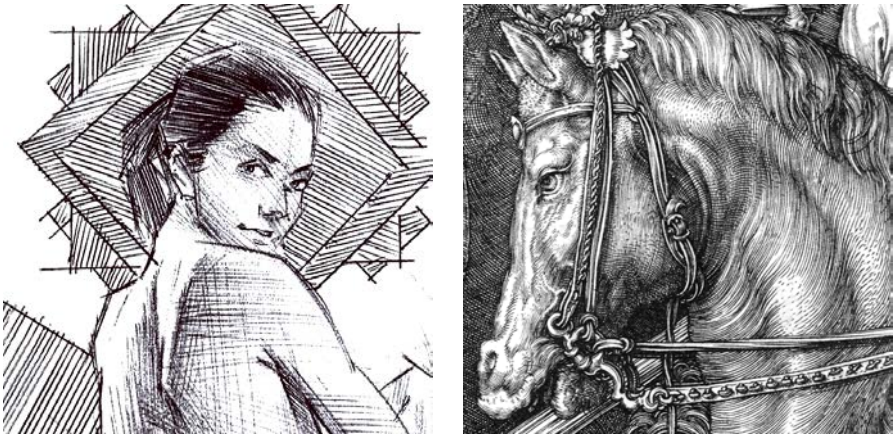
Part of the purpose of an illustration education is to give students competence with professional art media. Unfortunately I did not connect well with any. Digital art (Figure 1.2) did appeal to me for its efficiency: in principle it requires only a single business expense, after which you can make any kind of picture. However, I was prejudiced. My ego would not have called digital art "fake," but my id would have. Also, I did not learn a way to avoid the "slickness" of digital art. I concluded that only a very capable artist, the kind who could excel in physical media anyway, should bother making pictures on a drawing tablet. Heading from college into adult life, I could not justify to myself the cost of buying one.

Between obtaining my undergraduate degree and starting my PhD, art practice for me was just that: *practice*, exercises in sketchbooks (Figure 1.3). I procrastinated on making real pictures, waiting until the day I was skilled enough to justify paying for professional media.

A CS PhD based on art was a gambit to end this stagnation. It was also a tentative choice of digital as my professional medium. Because of this, the first task I gave myself as a doctoral candidate was writing my own art application. Its purpose was to let me test art tool ideas, and to let me draw real pictures. My relationship with this testbed application mirrors my relationship with NPR. The program's quirks have shaped my style over time.



Figure 1.3: Prior to the PhD, my art practice consisted of sketchbook exercises like these.



(a) (b)

Figure 1.4: I have long used *hatching* in my drawings (a). This technique originated with print media (b). Detail from *Knight, Death and the Devil*, Albrecht Dürer, ca. 1513, engraving.

1.5 Defining Hatching

I had one major requirement for any tools I would put in my testbed: they should operate at a low level. In other words, they should not make any high-level compositional decisions on their own, because composition is the part of art I most enjoy. The ideal tool would deal only with the tedium of editing or placing numerous individual marks, subject to *my* compositional requirements.

The clear first choice of tool was one oriented around *hatching*. I have used hatching—patterns of approximately parallel lines—in my sketching since my undergraduate years (Figure 1.4). Hatching is a centuries-old technique invented to accommodate print media. It gained popularity by becoming the engraver and etcher's comfortable answer to many rendering needs. For me, hatching appealed on the same level: it let me portray value (light versus dark), form, and texture conveniently with a ballpoint pen. My hatching habit led to my dissertation's focus on line art.

On advice from Dan Vogel, I let my first research contribution be a formal definition of hatching, a definition meant for NPR scientists. Chapter 4 is based on a published version of my definition.



Charles Dana Gibson



Gustave Doré

Figure 1.5: As I imagined a software hatching tool, I studied other artists' hatching. I prefer the loose hatching of Gibson (*"Home again—as the steamer docks"*, ca. 1907, pen and ink). Yet Doré's more intense hatching seemed better suited for computational generation (detail from *The Inferno, Canto XIX*, 1861, etching).

1.6 Hatching Shapes

After formally defining hatching, I looked for a way to create hatching semi-automatically. I imagined interfaces that would take in a few input marks—artistically intuitive inputs, like guideline curves—and use these inputs to place many hatching marks at once. For inspiration I studied hatching drawn by notable illustrators (Figure 1.5). Gustave Doré in particular caught my eye. His hatching—regular, highly dense, sometimes visually exhausting—is not exactly my favorite. I prefer the relaxed, less space-hungry hatching of more recent artists like Charles Dana Gibson. Yet when looking at Doré's hatching, I could easily imagine software interfaces that would relieve the tedium behind it. I still have difficulty picturing a software interface that would be useful for sparser, more Gibsonesque hatching.

My software answer to Doré-style hatching was the *hatching shape*, an art primitive. By "primitive" I mean a persistent object that a user can add to a virtual workspace—a virtual "canvas," in art language. I cover the hatching shape in Chapter 5, which is based on a published paper. Figure 1.6 shows a single hatching shape. Figure 1.7 has several hatching shapes working together in a full illustration.

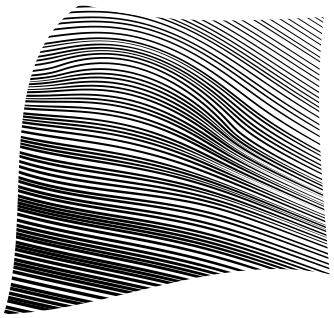


Figure 1.6: The *hatching shape* is my software tool for creating hatching.

1.7 Drawing Mashups

After developing the hatching shape, the intuitive next step was to address its weaknesses. There are two major ones: First, hatching shapes are laborious to create in terms of the number of hand motions required. I naturally feel disinclined to use hatching shapes except in unusual situations where I need a highly dense and regular line pattern, the very sort of hatching I was less attracted to in the first place. The second weakness is more interesting: it is difficult to use hatching shapes to portray 3D forms, especially forms that occlude themselves. This difficulty is apparent in Figure 1.7. Here, hatching shapes render the plants. Geometrically, these plants are spiked cylinders. The 2D nature of hatching shapes makes it tricky for them to handle places where the spikes occlude the cylinders they are attached to. The problem is even more pronounced where a plant's body occludes itself in the center of the picture. I liberally concealed awkward spots like this with shadows.

Such awkward experiences spurred me to move outside of hatching and tackle the more general issue of rendering *any* repeating pattern on a 3D object viewed in perspective. I made strides in this direction—see Chapter 3—but no stride far enough to merit its own publication. The difficulty was in designing a tool that let me perform 3D drawing tasks without essentially being a lightweight CAD tool, and therefore not novel from a CS perspective.

After a few experiments, I abandoned the 3D-rendering angle. I tried to imagine a tool that was like but unlike the hatching shape, something that performed automatic work at the level of individual strokes, but in pursuit of a task other than hatching. I struggled deciding what that task should be. There *are* other drawing tasks that might be considered siblings of hatching. Stippling is an obvious one; it does with points what hatching does



Figure 1.7: Hatching shapes render the plant forms.

with curves (Figure 1.8). (Stippling came naturally to mind since it has received so much attention within NPR [66].) There are other drawing tasks in this family like typography and general pattern generation.

Ultimately I had a breakthrough that led not just to my third technical contribution, but to this dissertation's central narrative. The idea was to *invent* a drawing task. This invented task needed to be complex enough to require a non-trivial, interesting algorithm to solve; plus it needed to appeal to me artistically. The task I came up with was to create

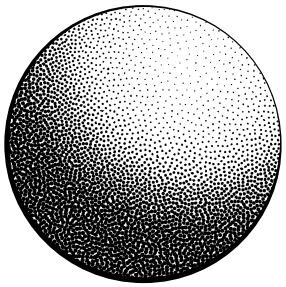


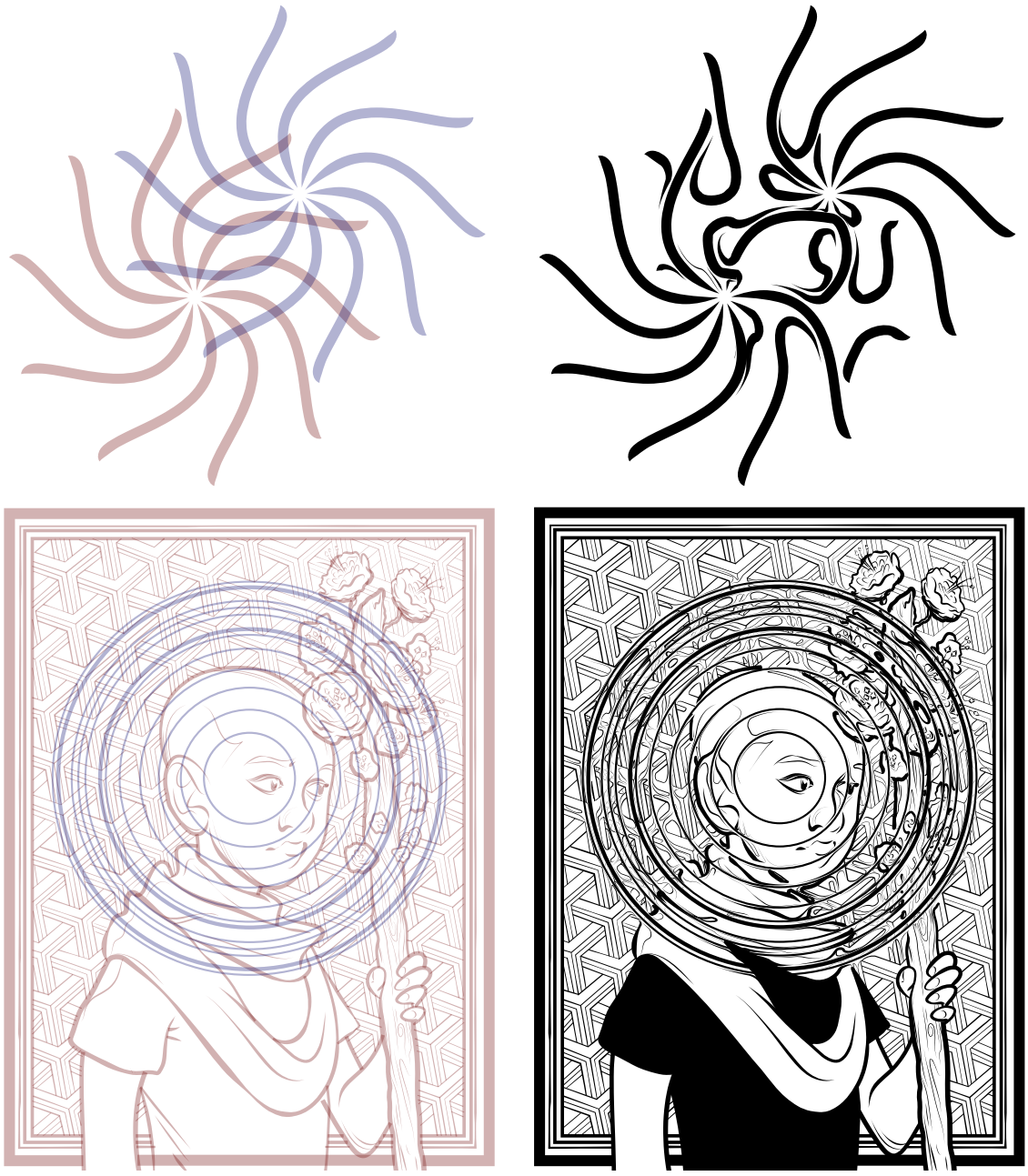
Figure 1.8: After hatching, I considered stippling as my next research subject. The stippling in this image comes from my application—see Chapter 3.

a *drawing mashup* by blending two superimposed drawings (Figure 1.9). Ruminating on this task led me to a by-hand technique for executing it, which led to an algorithm based on reproducing the by-hand technique. I cover this technique and algorithm in Chapter 6.

In summary, my technical contributions are

1. a definition of hatching (Chapter 4),
2. the hatching shape (Chapter 5), and
3. drawing mashups (Chapter 6).

Before presenting these contributions in detail, I review related research and discuss my testbed application in Chapters 2 and 3. At the end of this dissertation, in Chapter 7, I summarize how writing my own art tools influenced me as an artist.



(a) (b)

Figure 1.9: A *drawing mashup* combines two superimposed input drawings (a) in a single blended drawing (b). The colors red and blue distinguish the two input drawings within a superposition.

Chapter 2

Related Work

This chapter gives a sense of the research landscape for those interested in creating line art (Section 1.2) using CS.

2.1 NPR

My technical contributions fall within NPR: Non-Photorealistic Rendering. In Chapter 1 I call NPR the intersection of visual art and CS, but some clarification is necessary. "Rendering" is a fraught term. In the art world it has a more specific meaning than what I intend here. For traditional artists, to render means to depict lighting, whereas I prefer to use NPR to include art tasks that do not necessarily involve lighting. "Render" also has too specific a meaning within the CS world. In CS, "rendering" means generating 2D views of explicit[1] 3D models from the point of view of virtual cameras. More to the point, "rendering" in CS is strongly associated with rasterization and raytracing.

Typically, the goal in CS rendering is photorealism. Early on, "NPR" was about negating this goal: it specifically meant retrofitting rasterization or raytracing techniques for non-photorealistic ends. See, for instance, Gooch et al. explaining how to add object outlines to a 3D mesh renderer [29]. Also see Winkenbach et al.'s modified polygon renderer [104] which, though it operates on explicit 3D input, places individual strokes in screen space using "stroke textures" (Figure 2.1). In other words, "Non-Photorealistic Rendering" used to mean what its name suggests.

[1] By "explicit" I mean that a 3D model resides in computer data, not just in an artist's imagination.

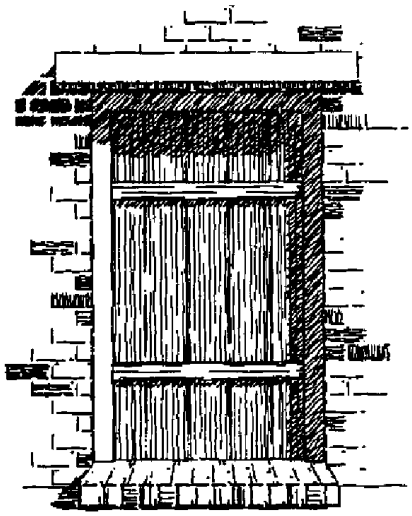


Figure 2.1: This pen and ink illustration is actually the output of a 3D renderer [104].

Over time, NPR has grown to include not just non-photorealistic rendering techniques like the two just mentioned, but visual art-oriented CS in general. Positively, NPR is the intersection of CS and visual art. Negatively, NPR excludes conventional, realism-oriented rasterization and raytracing (even though the products of these might reasonably be called "art").

Taxonomy is an insoluble issue in research. Beware that others in CS may not agree with my definition of NPR. Even were my definition authoritative it would still be tricky to say which research projects are NPR. Institutions force us to arbitrarily split work into exclusive categories. Take, for instance, Jobard and Lefer's treatise on streamline advection [46] (Figure 2.2). Streamline advection motivates my second technical contribution (Chapter 5). Should Jobard and Lefer's algorithm be considered NPR research? In a sense, yes: this algorithm is useful for NPR researchers interested in line art. But Jobard and Lefer consider their work to be part of *scientific visualization*, a different research category. It appears they do not conceive of their work as *art*-related.

2.2 Interactivity

Because my research has revolved around art tools, interactivity has been a consistent concern of mine. Art tools are a major part of NPR, and these tools vary greatly in terms

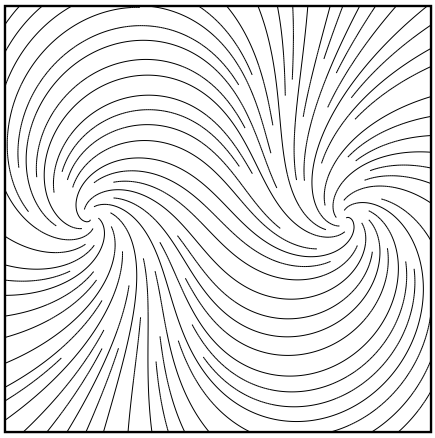


Figure 2.2: Jobard and Lefer's algorithm for visualizing fluid flow fields [46] would generally not be considered part of NPR, but it definitely has NPR applications.

of interactivity. Isenberg lays out an "interaction spectrum" in which tools range from fine-grained and artist-oriented to automatic/semi-automatic and non-artist-oriented [45]. Two of my three contributions are art tools. The first of these, the hatching shape, occupies the interactive end of Isenberg's spectrum. The second tool, the drawing mashups algorithm, falls more in the middle. A tool at the non-interactive end of the spectrum would be, for instance, a script that automatically converts a photograph to a stippled illustration [59].

Although my second tool is on the less interactive side, I have been inspired most in my research by the more interactive tools in NPR literature. The most intuitive examples of interactive tools are those, like "MoXi" [16], that simulate pre-electronic artistic practices (Figure 2.3). Yet some NPR tools are about embracing the digital and rejecting the traditional. These experimental interfaces tend to be more thought-provoking than appealing, but the feeling behind them resonates. Schwarz et al.'s interface [87] has an artist direct numerous brushstroke primitives at the same time, like guiding a school of fish. Loi et al.'s interface [63] has an artist use a functional programming language to define decorative patterns. Both are meaningful experiments in interface design. The latter interface is almost prophetic: nowadays the studio portfolio site ArtStation hosts a lot of art made programmatically within tools like *Substance Designer*.

I prefer an art interface whose primitives, like my hatching shape, encourage traditional intuition, but whose operations are nontraditional. A good example is the interface of Fišer et al. [25]. There, the artist draws one stroke at a time, each stroke automatically adjusted to fit better with previous strokes around it. This requires the software to infer what kind of pattern the artist is trying to draw.

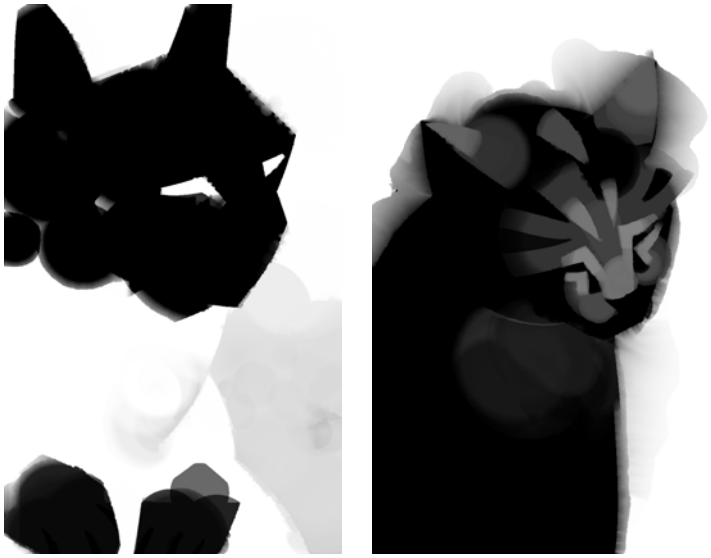


Figure 2.3: The "MoXi" interface [16] simulates watercolor. I recreated "MoXi" for a class project, producing these test images.

My testbed application (Chapter 3) is most directly a descendant of the 1994 interface "Interactive Pen-and-Ink Illustration" by Salisbury et al. [81], which is more forward-oriented than its age and name suggest (Figure 2.4). In that interface, the artist does not individually place all the "ink" strokes, but indicates what patterns of strokes the software should use to automatically fill in different regions. (Salisbury et al.'s interface can be thought of as a companion to Winkenbach and Salesin's 3D renderer [104], taking as its input a 2D grayscale image to match rather than a 3D model.) Automatic stroke placement arrived in my testbed application with the introduction of the hatching shape.

2.3 Line Art

Line art, as defined in Section 1.2, has received wide treatment in NPR. A taxonomy of NPR work within the line art domain would be an academic contribution in its own right. Here, I will confine myself to describing research on two line art topics that relate closely to my work: hatching and stippling.

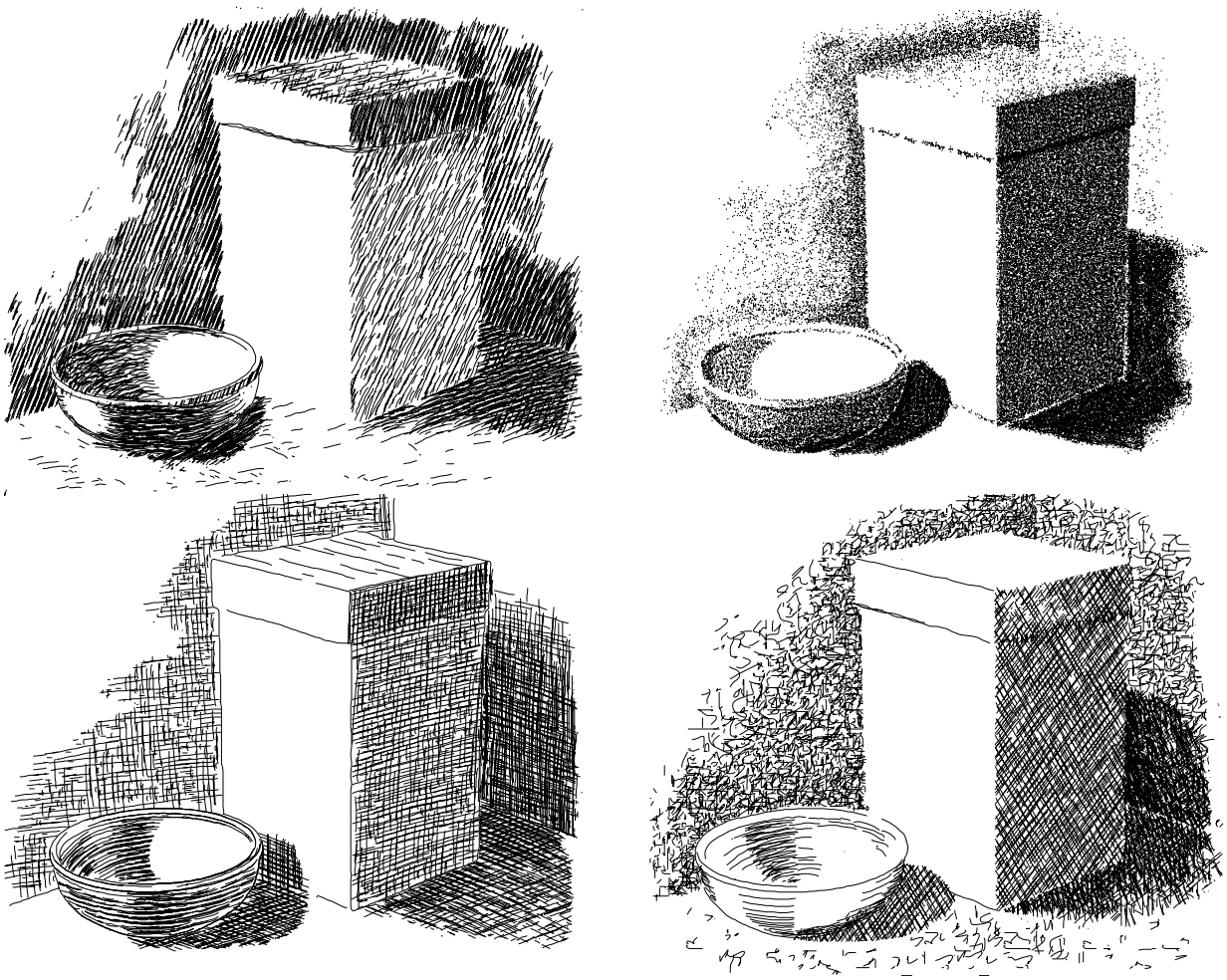


Figure 2.4: The interface of Salisbury et al. [81] simulates pen-and-ink drawing in a non-traditional way. An automatic process handles the placement of individual marks using "stroke textures."

2.3.1 Hatching

Most NPR hatching work accords with NPR's early definition: it is about putting a 3D renderer to non-photorealistic use. The input is a 3D model and the output is typically a hatching image. Lawonn et al. [57] survey 3D renderers that produce hatching.

An important early innovation in this area is Elber's use of isocurves as a convenient way to hatch parametric surfaces [22]. The idea here is simple: if the surface to render can be parametrized, one can generate a family of isocurves and individually project all of these in order to render the surface with hatching. Hertzmann and Zorin perform hatching this way [42].

I have already mentioned the important "stroke texture"-based hatching renderer of Winkenbach and Salesin [104]. Praun et al. advance the idea of stroke textures with tonal art maps (TAMs) [78]. The TAM rendering pipeline is closer to a traditional CS rendering pipeline. Winkenbach and Salesin's renderer, despite taking 3D input, is very image space-oriented. It involves building a 2D spatial subdivision to facilitate placing strokes in image space, as if drawing with a pen directly onscreen. TAM rendering, on the other hand, is more about modifications to the usual 3D pipeline; as such, it can produce temporally coherent animations, something Winkenbach and Salesin's renderer cannot do.

TAMs have been successful, at least in inspiring further research, but their ability to render animations comes at a cost. When generating hatching with software, there is typically a desire to match the look of hatching made by humans, and the human process of hatching is inherently image-based: it is about strategically placing marks one at a time within image space. To rely on the 3D rendering pipeline to make hatching is to depart from the way hatching is traditionally made, and can result in an overly computer-generated look. Winkenbach and Salesin's renderer, despite being one of the earliest NPR treatments on hatching, manages to avoid the CG aesthetic because of its highly image space-oriented mechanism of mark placement. Other researchers have found ways to tie hatching generation to image space while rendering explicit 3D models. For instance, Orbay et al. [74] calculate screen-space shadow shapes and fill them with marks whose paths are based on the shape they need to fill. They even break complex shadow shapes into separate hatching patches (Section 4.3) the way a human might.

Though most NPR treatment on hatching involves 3D geometry as input, significant work has been done on 2D hatching interfaces [76, 81, 83, 82, 75, 21]. In most of these interfaces, the hatching derives from an input image. I prefer to organize 2D hatching methods according to how they handle hatching marks' directions. One of the first 2D hatching interfaces is "DigiDürer" [76], in which marks' directions come from the input image. The same



Figure 2.5: "Digital Facial Engraving" [75] lets users create hatching by outlining Coons patches. Individual hatching marks are isocurves of these patches.

applies in "Interactive Pen-and-Ink Illustration" [81], though there the user additionally provides an offset angle. "Orientable Textures" [82] gives more direct control over marks' directions. It features a rasterized direction field which the user can edit via brush tools. The most interesting edit operation has the user draw two curves and fills the direction field between them by sampling from interpolating curves. (Sections 5.3 and 5.4 describe my own work with direction field-editing operations.) Interpolation between two curves also seems to feature in *Strokes Maker* [3], a commercial application for creating hatching from a photograph. Adobe *Illustrator*, augmented with plugins like *WidthScribe* by Astute Graphics [31], can achieve quite sophisticated two-curve interpolation. The idea of interpolating curves crystallizes further in "Digital Facial Engraving" [75], where hatching marks' paths are isocurves from Coons patches (Figure 2.5).

In 2D hatching generation, there are two pressing needs, both of which I have tried to address in my own work with hatching. First, as already stated, most 2D hatching generators take a grayscale image as input. As an artist, I am not so interested in applying artistic filters to existing images; I am interested in creating hatching directly from imagination. I prefer a finer level of interactivity than selecting an image to apply a hatching effect to. In Chapter 5, where I describe my hatching shape primitive, note the attention paid to how a user creates and modifies hatching shapes.

The second pressing issue in 2D-based hatching is poor control over hatching marks' directions. My hatching shape primitive represents progress in this area, but there is still work to be done on conveniently and intuitively generating directionally complicated hatching.

2.3.2 Stippling

Martin et al. provide a useful survey of NPR work on stippling [66]. They caution that "stippling" is ambiguous. The word can broadly mean "painting with dots," but I use "stippling" more strictly to mean dots of the same size placed irregularly (not along a simple grid) to create a collective image while maintaining (locally) even inter-dot spacing. I write "dot patterns" when I intend to include other possibilities, such as allowing dots to vary in size or placing them along a simple grid.

As with 2D hatching generation, a grayscale input image typically guides stippling algorithms, though not always. Deussen et al.'s stippling interface [18] is more interactive and does not require an input image. It is hands-on enough to require the user to manually segment the canvas into different stippling areas. It even has a relaxation brush for evening out dots' spacing in a specific place. However, as with hatching, my sense is that there is still much work to be done on stippling directly from imagination.

More automatic NPR stippling typically involves two steps: place an initial collection of dots, then relax the dots' positions to improve spacing. Simple initial dot placement mechanisms include dithering [18] and rejection sampling [27]. Classically, dot relaxation uses Lloyd's algorithm (Voronoi iteration) [18], where dots repeatedly move to the centers of their respective Voronoi cells. Various improved relaxation methods appear in stippling literature. In Chapter 3 I mention using capacity-constrained Voronoi tessellation [65, 5] to relax my dots. In this algorithm, the space to stipple is discretized with a collection of "sites" (dot positions) and a much denser collection of "points." The algorithm iteratively reassigns points to sites and updates sites' positions based on the evolving point-to-site mapping.

In later stippling approaches, initial dot placement is high-quality enough to not require relaxation. For instance, Mould [70] creates finalized stippling dots by placing them along the frontier of a single pass of Dijkstra's algorithm [19] over an input image's pixels.

A reliable strategy in NPR is to modify an existing art-making algorithm in order to hone in on a specific style. In the hedcut style of stippling, dots group along lines that help show the subject's 3D form. This style is associated with author portraits in *The Wall Street Journal*. Kim et al. move toward this style by modifying Lloyd relaxation so that dots align with edges in the input image when they are close to them [51]. Son et al. go further [92], designing a whole new stippling algorithm specifically to achieve the hedcut style (Figure 2.6). First, the algorithm generates an image-wide direction field based on image edges and on isophotes—contours along which image brightness is constant. This direction field helps to create a pair of perpendicularly aligned, periodic distance maps, on which dots can be placed trivially.



Figure 2.6: Son et al.'s stippling algorithm [92] is built around the hedcut style.

2.4 NPR Position Papers

Position ("meta") papers are where NPR researchers clarify what their field is about and where its value lies. Though my dissertation mainly relays technical contributions, it is also effectively an NPR position statement about the value that CS research has when directed toward achieving a specific artist's personal objectives.

Hertzmann is a notable author of NPR positions, some of which resonate with mine. While writing about the creative process [40], he says, "...art comes from a process; I cannot choose the outcome of a new artwork when I begin. I can't even predict it. It's about following the process until I get 'something good'." I myself have discovered the importance of *process*, particularly what happens when my process incorporates CS research. Hertzmann also suggests how we in NPR should think about AI art: in the past, emerging technologies have seemingly threatened to destroy art, but in retrospect have wound up invigorating art instead [38, 39]. His optimism resonates. My research does not involve any AI, but I have experienced the fear Hertzmann discusses, the worry the old-fashioned artist feels as new technology threatens to make traditional skills redundant. Through my research I have come to feel less fearful of and more interested in what digital technology can do for art, including my own art.

An open question in NPR is how to evaluate the success of NPR research. Several position papers treat this. Hertzmann, for example, disparages the community's use of shoddy

user studies meant to appease reviewers more than to actually test scientific claims [41, 12]. Hall and Lehmann suggest that, since NPR is about creating art, we should evaluate NPR results the same way that art critics assess art [35]. Mould suggests how NPR authors might more convincingly validate their work by articulating up front their stylistic goals [71].

I do venture toward formal evaluation of my results when I interview artists about their impressions of my hatching shape primitive (Section 5.6.1). However, evaluation of my research has generally been solipsistic: my underlying concern has always been my tools' capacity to help my art. Success from this point of view is difficult to validate scientifically, especially since my artistic aims shift in response both to my successes and my failures.

Though my solipsistic focus makes my work less scientific, the solipsism is an important part of the larger story I seek to tell, the fourth contribution outside of my three technical contributions. This story distills to an NPR position of my own, specifically a hypothesis. I hypothesize that my personal experience can generalize, that a future artist who uses CS research to help their own art will discover that the research has surprising, positive effects on their creative process.

The place where I most clearly saw my own creativity evolve was within the testbed application which I used for testing my technical contributions. I cover this application in the next chapter.

Chapter 3

My Testbed Application

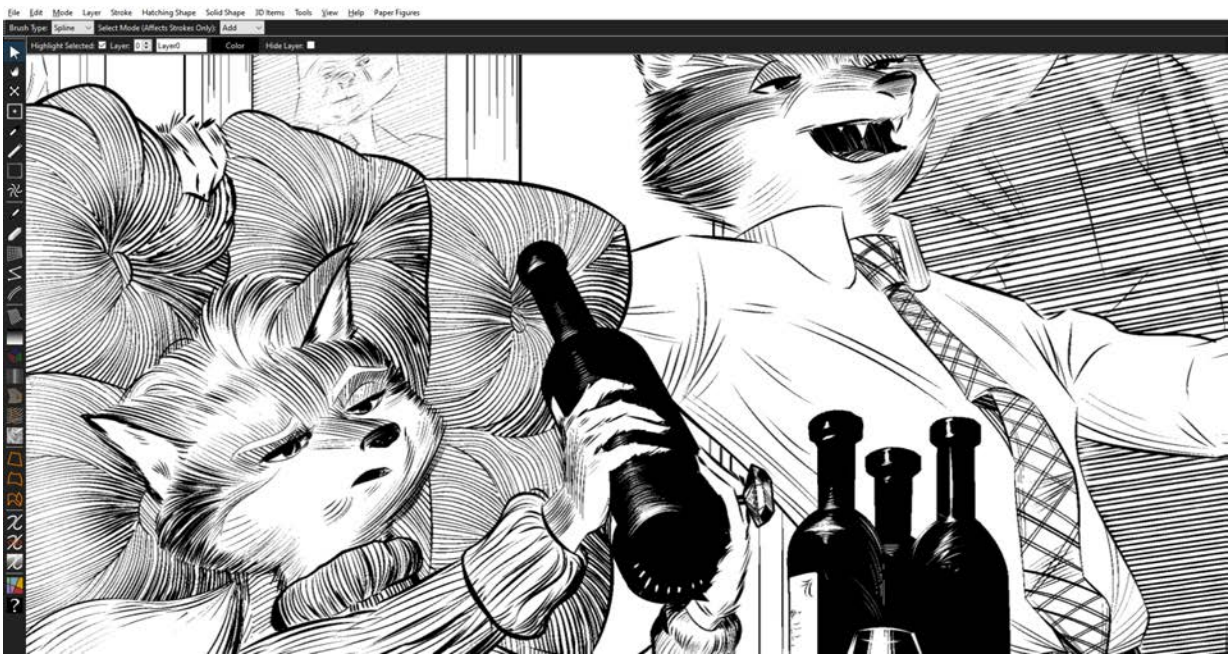


Figure 3.1: I wrote a drawing program as a testbed for new tools. Note the menu at top and toolbars at top and left.

3.1 Introduction

My first research task was to start writing a drawing program (Figure 3.1), a place to test my new art tools. This is a windowed application, visually evocative of programs like *Photoshop* or *Illustrator*. It takes its input mainly from a drawing tablet (I have been using a Cintiq). There are also keyboard commands for switching between tools and changing brush settings. This testbed application (which I have yet to properly name) is a product of engineering more than of scholarship. Still it deserves attention. If nothing else, it has been indispensable in developing my second two technical contributions: hatching shapes and drawing mashups.

In isolation, this chapter explains what my testbed application is to researchers wanting to make their own exploratory art software. Within the larger dissertation, this chapter has another purpose: to highlight my application's quirks. These foibles forced me to adapt creatively when making pictures. They are part of the story of how NPR research changed me as an artist.

3.2 Design Decisions

Initial development choices for my application set the trajectory of my research. First, I chose to use low-level code: C++ with OpenGL. This choice was partly from habit. C++ is my most comfortable language and I had previous experience with OpenGL. There was also a logistical reason: I wanted my application to be roughly as fast as the art programs I already had experience with.

Writing my own application with low-level code did give me a fast application, but it sometimes limited my experimentation. C++ frequently presented me with time-wasting dilemmas. I repeatedly had to make the uncomfortable choice between quickly hacking in a desired feature and making my code brittle versus properly integrating the feature and having to solve a refactoring puzzle. Such delays are not limited to C++; they will occur in any language once the codebase gets large enough. But I suspect this problem is especially bad in C++.

Another crucial choice was for the application to be vector- rather than pixel-based. This choice incurred some overhead, but it was the right one. A vector tool is inherently more powerful than a visually identical pixel-based one. More fundamentally, vector representation makes it easier to explore the kinds of art that can only be produced digitally.

3.3 Primary Primitives

I can place different types of primitives in the virtual canvas. The three most important types, in decreasing order of usefuless, are strokes, filled shapes, and hatching shapes (Figure 3.2).

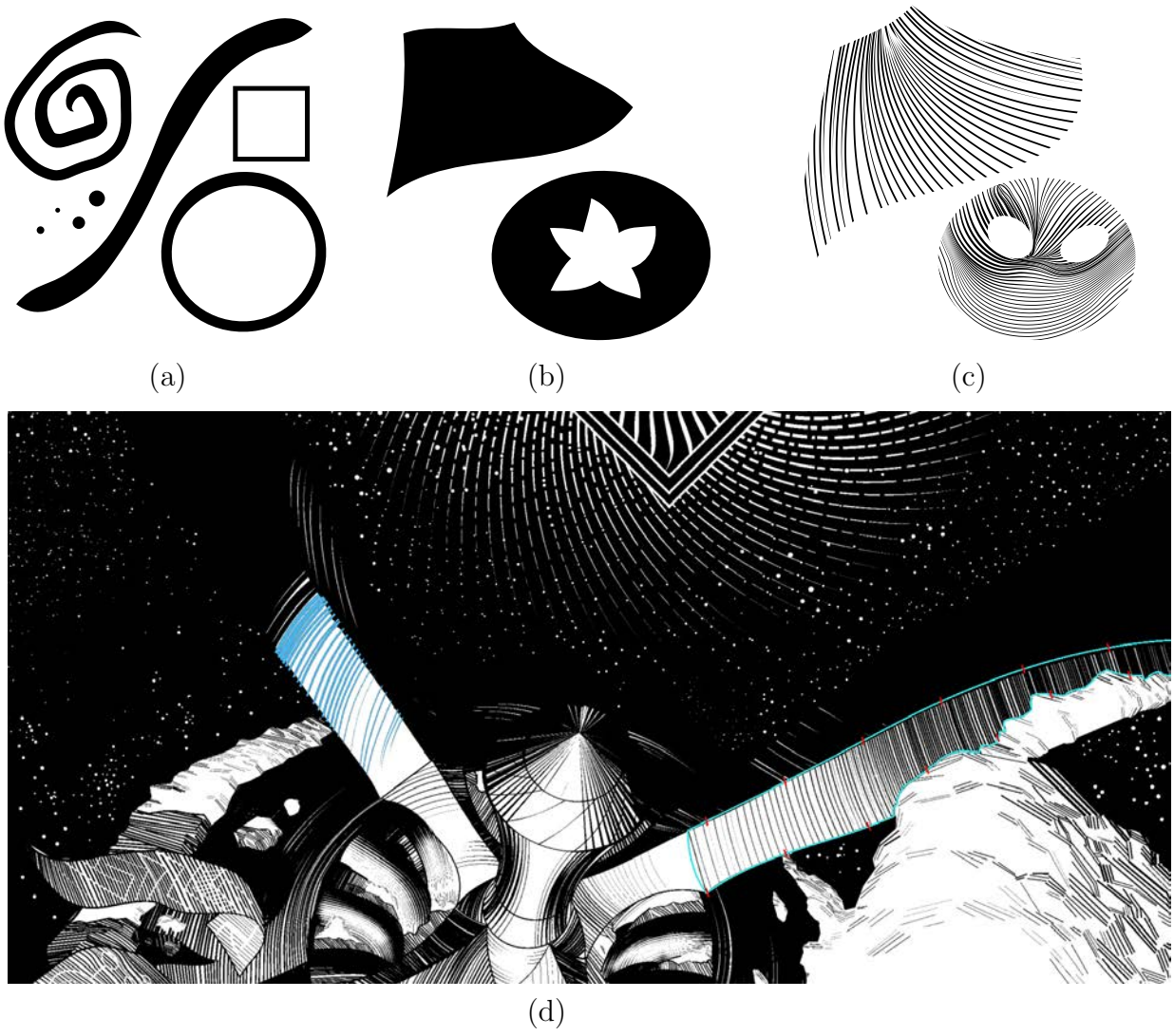
The *stroke* is my most-used primitive by a wide margin. It is based on a position curve and a width curve (Figure 3.3). Position curves, which I refer to as stroke *spines* in Chapter 6, are approximately parametrized by arc length. They may be open or closed, and may have zero length, resulting in dot strokes like those in Figure 3.2a. All curves in my application are B-splines [88].

Since strokes do the most work, their limitations are the most glaring. Two issues have nagged for years. The first, more irritating problem lies with my code for generating a new stroke from position and pressure input. This task is less trivial than I had expected and has received significant academic attention [96, 89, 105, 36, 1]. I patterned my stroke-capture code after Disney's online description of its *Meander* [102] tool. That is, I borrowed the approach of smoothing input points, evenly respacing these points, and then using them as the control points of a quadratic B-spline. My stroke creation code is functional, but imperfect. The *feel* of drawing in my application has always been off. To make it work like *Meander* would require careful tuning. Specifically, I would need to develop a sophisticated "adaptive window" [102] so that when I draw a long, complex stroke, I do not deal with framerate drops caused by redrawing the entire stroke on each window update.

Uncomfortable stroke generation is enough of an issue that I actually sketch all my pictures in other programs—*Krita* [55] currently—and then trace these sketches in my program. You can think of my application as for "inking" rather than for freehand drawing. Beyond outsourcing sketching to other programs, I also mitigate my unsatisfactory stroke generation with a suite of stroke-editing operations. Some of these operations appear in Figure 3.4.

The second major issue with strokes comes from their simplistic definition. A mere position curve and width curve make for a digital-looking mark. I have tried to conceal this in a couple of ways (Figure 3.5). I sometimes force strokes to taper at one or both ends using a tunable taper distance. Though this feature does appear in other drawing programs, it is elevated to the level of a quirk within my application due to how few "brush modes" my application has. A second means for hiding my strokes' digital look is tunable Perlin noise applied to strokes' sides during rendering.

Filled shapes are simpler than strokes. Each is basically a mask made from one or more non-intersecting, closed polycurves, where one polycurve may lie completely inside another.



(a) (b) (c)

(d)

Figure 3.2: My testbed application has three main primitives: strokes (a), filled shapes (b), and hatching shapes (c). The snapshot (d) features all three. Some strokes and a hatching shape (blue and cyan respectively) are selected for editing, and a filled shape creates the night sky. The stars are white strokes.

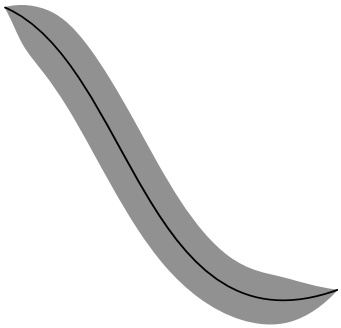
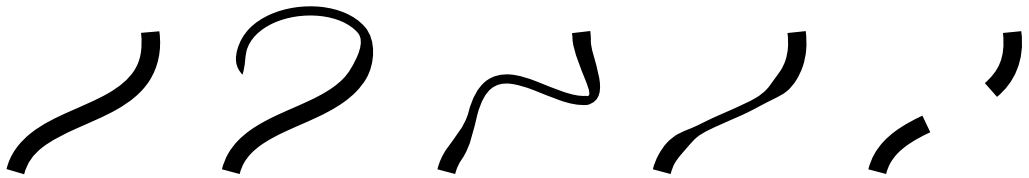
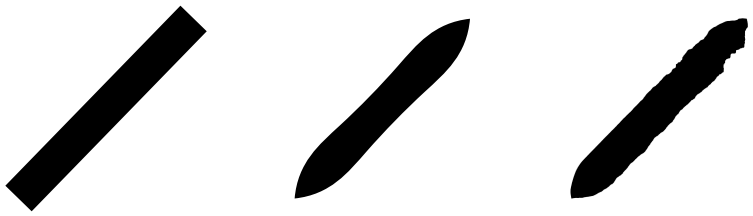


Figure 3.3: A stroke contains a position curve (black) and a width curve. The width curve for this stroke starts at zero, increases to a certain value, and remains at that value until decreasing to zero at the end.



(a) (b) (c) (d) (e)

Figure 3.4: I can refine an initial stroke (a) in various ways: overdrawing to extend (b) or internally edit it (c), overdrawing to change its width curve (d), or erasing parts of it to create multiple strokes (e).



(a) (b) (c)

Figure 3.5: Since a stroke is based just on a position curve and width curve, it will often look too digital (a). I counter this by forcing width curve tapering on new strokes (b), and/or by using Perlin noise in rendering (c).

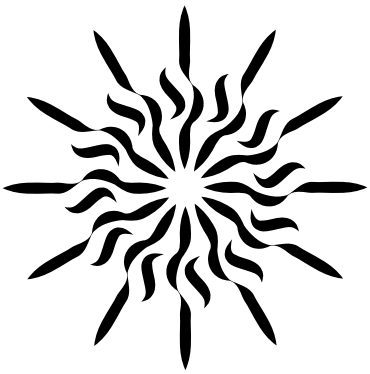


Figure 3.6: Strokes created and edited in radial mode

(A polycurve is a chain of connected curves; a closed polycurve is, colloquially speaking, a shape.) However simple they are, I love filled shapes' utility. As an artist, I care about light versus dark. I appreciated the ability to rapidly create areas of pure black after years of wishing for it in my sketchbooks.

Hatching shapes are also based on multi-polycurve masks, except that the masked regions are filled not with solid "ink" but with hatching marks. In Chapter 5 I explain hatching shape creation and editing. Here, suffice it to say that the resulting hatching patterns are very regular, a little too digital-looking, like my strokes. To avoid confusion, the hatching marks produced by a hatching shape are not the same as strokes. However, my application does enable converting a hatching shape to a collection of strokes. Nowadays I usually do this; I can best counter hatching shapes' digital look by manually editing individual hatching marks.

3.4 Other Features

My application is littered with secondary features. Some are left over from abandoned research projects. The rest are simply tools I had an impulse to make art with. I do not draw an important distinction between these two groups, given that I am telling a story about what it means to be a researcher and an artist at the same time.

3.4.1 Radial Mode

When this feature is enabled, each new primitive gets copied n times about a central point, and any edit to one of the $n + 1$ primitives is propagated to the rest of the family. Radial mode is obviously a trivial feature. But again, as with stroke tapering, since my test application is so feature-sparse, radial mode has become a key part of my artistic workflow, important enough to alter my style. I return to radial mode in Chapter 7.

3.4.2 Dot Patterns

Dot patterns were a natural avenue for me to explore, given their proximity to hatching. In fact, I implemented dot placement as an alternate rendering mode for the hatching shape. Thus my dot-placement mechanisms get to use hatching shapes' underlying data. There are two mechanisms. One places dots on a regular grid, as in halftoning. The other performs stippling, placing dots irregularly to achieve local density constraints. Figure 3.7 shows both mechanisms at work.

Recall from Section 2.3.2 that some NPR stippling algorithms have an initial placement step followed by a relaxation step. This is how my stippling works. I use simple rejection sampling [27] to place initial dots. I then optionally relax the dots using capacity-constrained Voronoi tessellation [65, 5], described in Section 2.3.2. Figure 3.8 shows some relaxed dots. I use relaxation code from Balzer [4], one of the inventors of capacity-constrained relaxation. The process is slow: the just-mentioned illustration takes about five minutes to load and display in my application because of it. This is not to say that stippling is slow in general, or even that capacity-constrained relaxation is always slow. However, I am still unsure how I could have achieved significantly faster stippling in my application without major refactoring. More advanced (and presumably faster) stippling techniques in NPR seem to be tied to an input grayscale image, a setup which does not make sense in my application.

The sluggishness of my first stippling generator killed my interest in stippling as a research topic. This reveals an important quirk of my own: I have consistently avoided projects where I did not see myself making a highly responsive interface in a realistic time frame. This ruled out, among other things, art tools that could only run interactively with the help of GPU-oriented code, which is not my forte. I say this not to perform modesty; what I felt inclined or disinclined to do as a software developer has shaped my interaction with NPR *as an artist*. Were I the sort of developer who enjoyed GPU-izing slow tasks, this dissertation might easily have become a stippling travelogue.

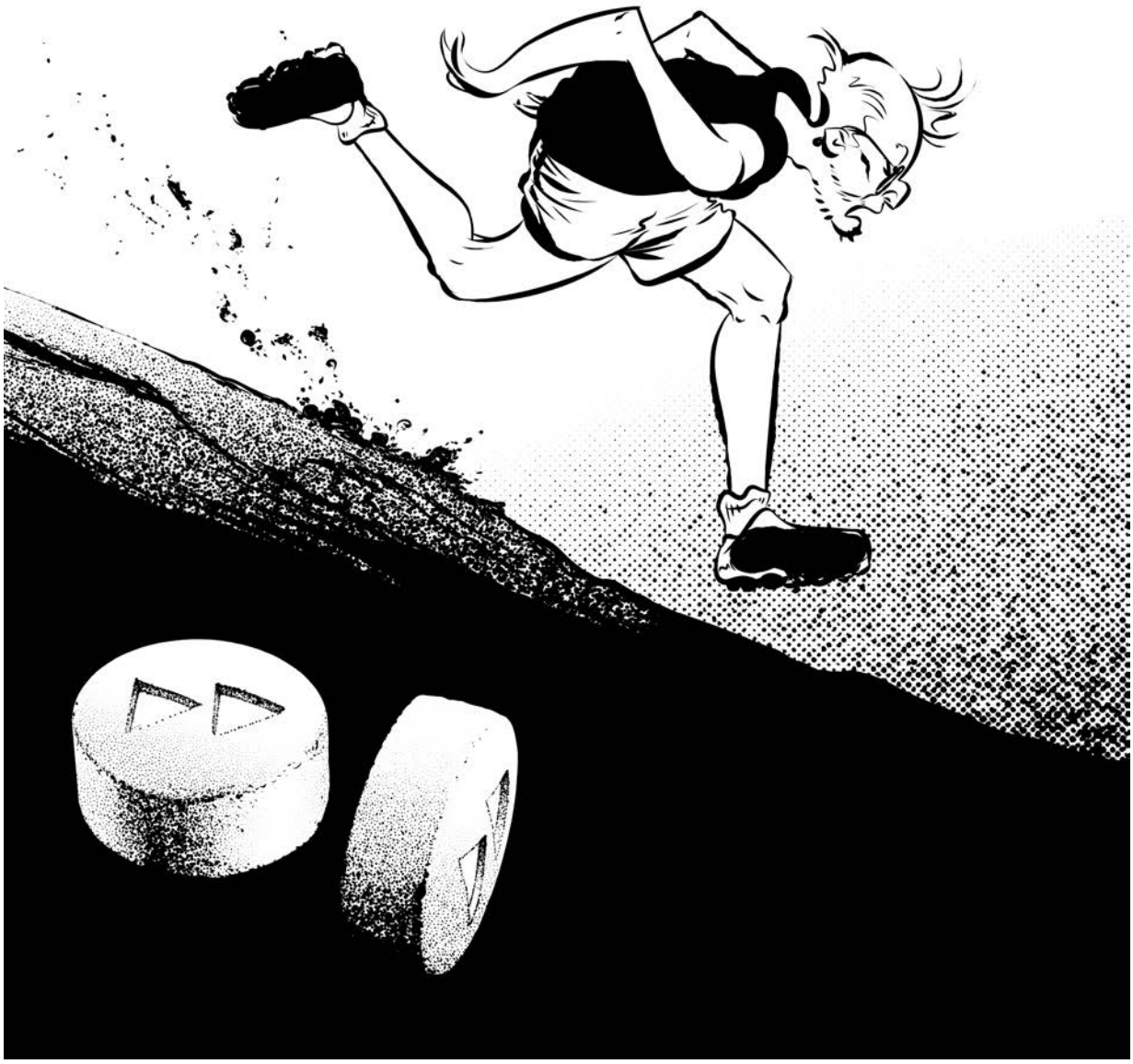


Figure 3.7: Most dots in this image were procedurally generated in my application. Irregularly spaced dots (stippling) appear at left, regularly spaced dots at right.

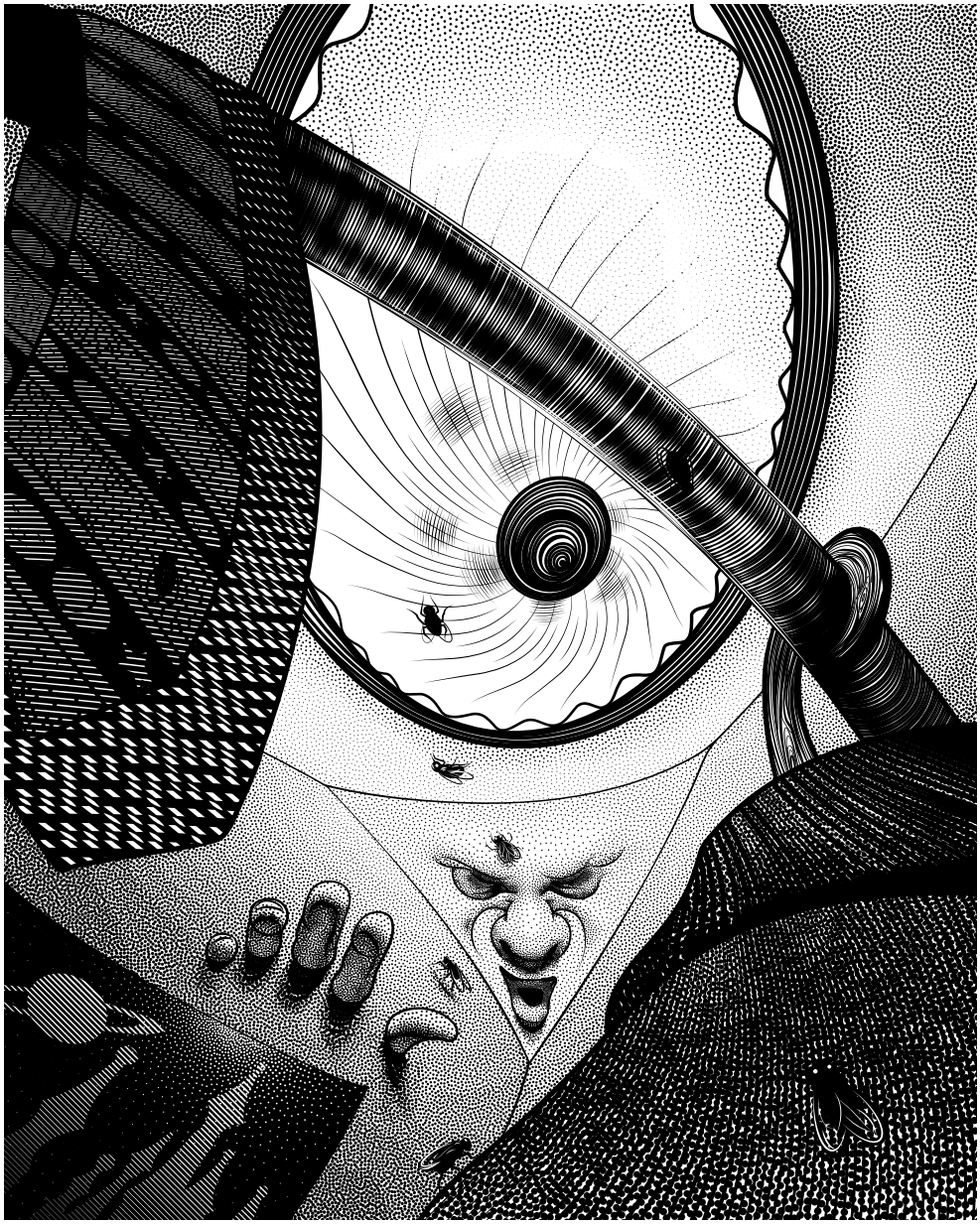


Figure 3.8: I evened out the dots' spacing using capacity-constrained Voronoi tessellation [65, 4, 5].

3.4.3 Perspective/3D Tools

Several secondary features in my application assist with perspective drawing. This is partly because of my frustration with hatching shapes' weakness at portraying 3D forms. More broadly, it is because I have never been good at perspective drawing, nor have I ever enjoyed practicing it. In my encounter with NPR, my strengths and weaknesses in art have been as important as my strengths and weaknesses in programming.

My first step in this direction was a suite of perspective-guide primitives: points, lines, and line segments. Some points are vanishing points; some lines are horizon lines. These guide primitives are visible in the pool scene of Figure 3.9. I illustrated this scene while reading Scott Robertson's book on perspective drawing [80]. The techniques in that book date from the Renaissance, and were commonly used until CAD tools made them redundant. As I pored over these old tricks, I added guide primitives and operations whenever I saw a way to help with my swimming pool geometry. In the end, the guide primitives helped, but perspective construction was still exhausting. Since then I have only used my guide primitives for very simple perspective scenarios.

Perspective aids like my guide primitives are increasingly common in other drawing programs. *Krita* [55], for instance, has powerful perspective-grid tools.

Simple guide primitives were just the beginning of my search for an easier way to draw 3D scenes. Once I saw that the hatching shape was an awkward tool for 3D depiction, I started searching for a more 3D-oriented primitive. I wanted this new primitive to represent a 3D object under perspective, and for its interaction to be canvas-oriented. For the sake of publication, the primitive had to be more than a CAD knockoff. Thus it was very important for this primitive to have a novel interface.

As preparatory work, I programmed a *3D rectangle* primitive. To create one of these in my application, I click four corner points in the canvas. My code uses old-fashioned drawing techniques to plot any pattern on the imaginary projected surface these corners represent. Think of this as a perspective-correct UV mapping. There is no need to build an underlying 3D scene. I used a 3D rectangle for the bricks in Figure 3.10.

I moved on to more complex variations where the imagined 3D object was a parametric surface or *patch* (in the mathematical sense, not a *hatching* patch from Chapter 4). Restricting myself to 3D patches simplified the issue of drawing arbitrary patterns on these surfaces. The idea was for me to draw the 2D outline of an imaginary 3D patch projected to the canvas, and to provide further input to disambiguate the projection.

Figure 3.11 show two attempts at a 3D-patch primitive. The first (top) is the *box patch*. After I draw the 2D projection of an imaginary 3D patch, my code fits a perspective box

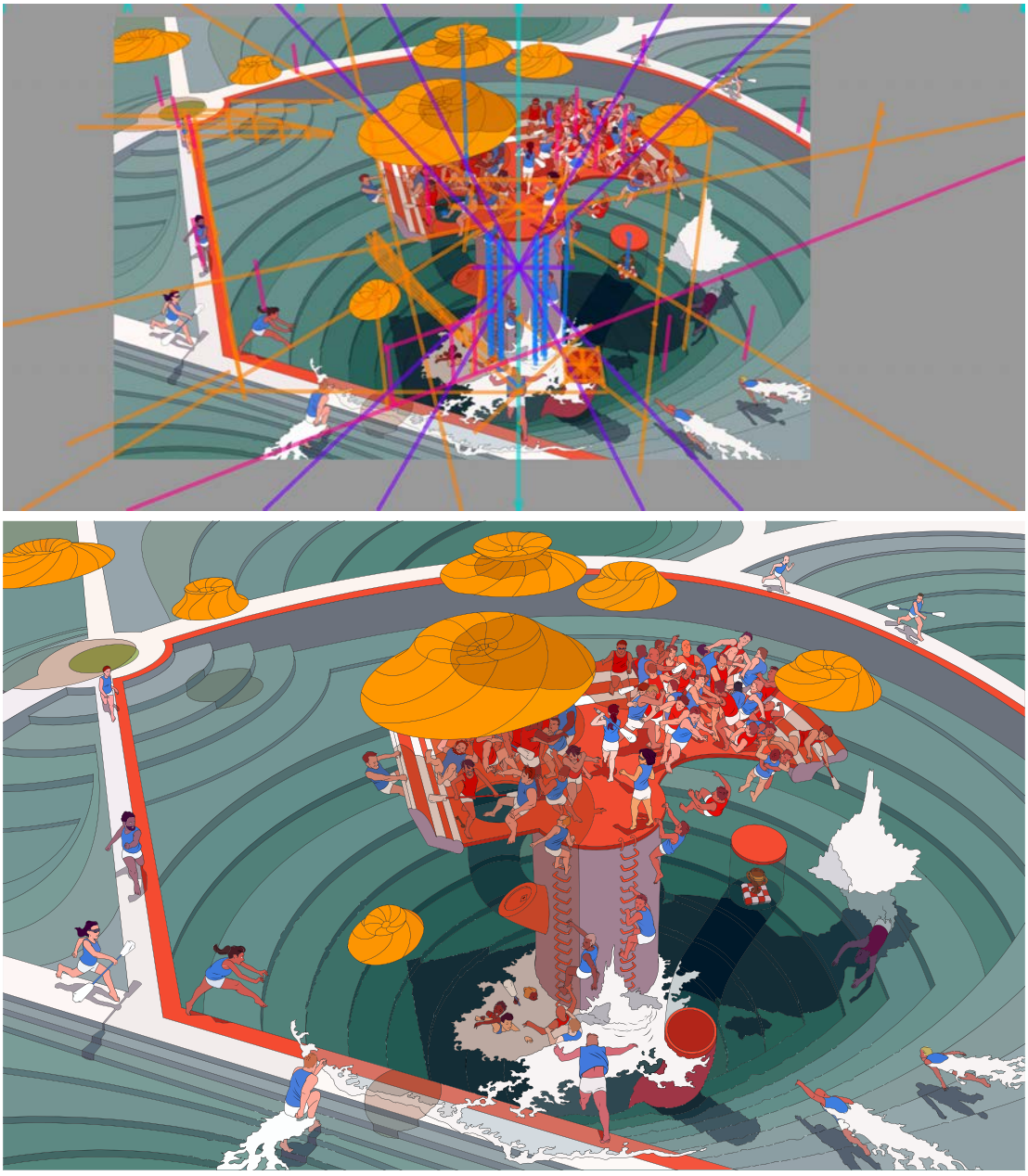


Figure 3.9: The colored lines (top) are guide primitives that help with old-fashioned perspective techniques.

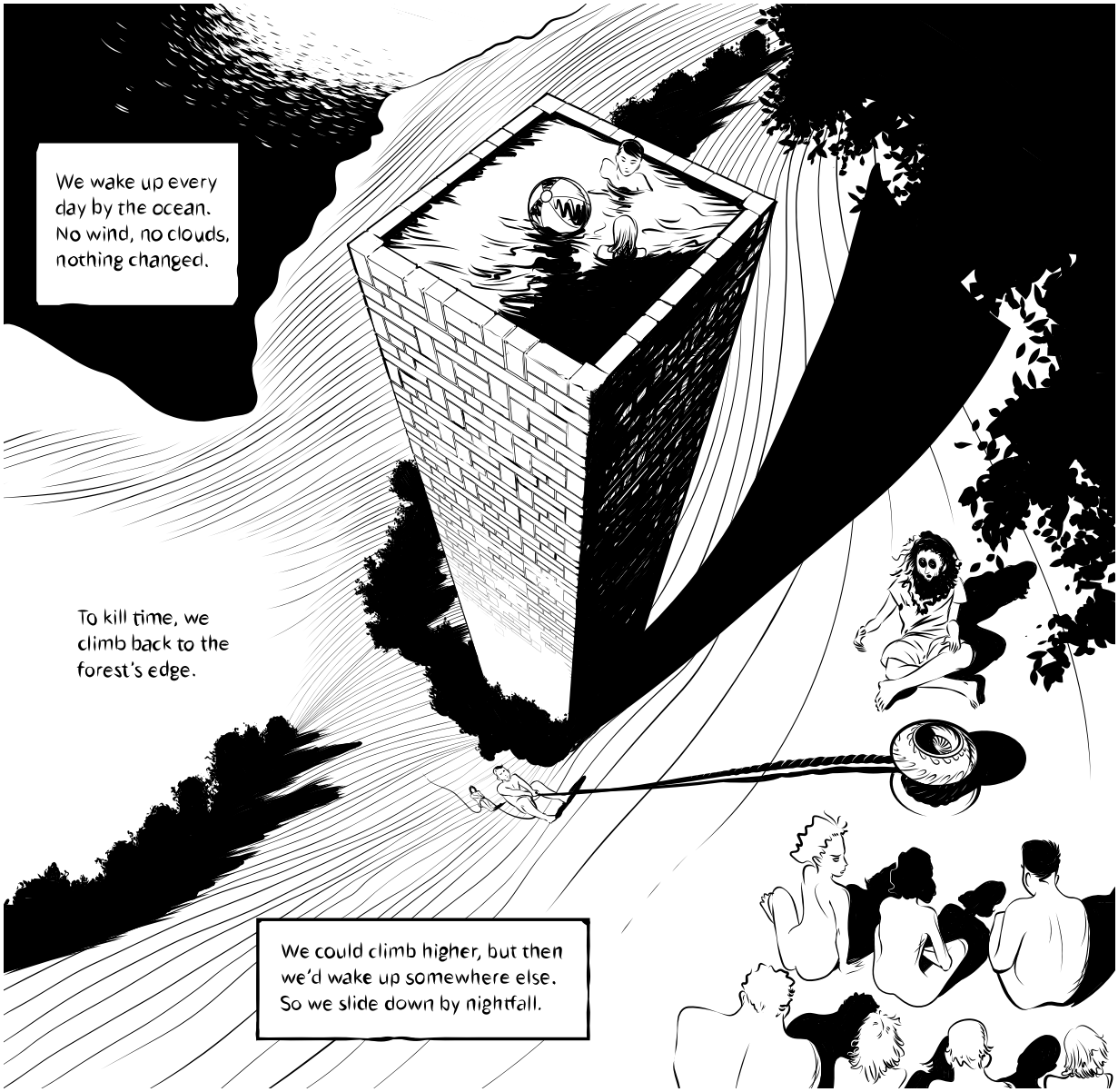


Figure 3.10: I used a *3D rectangle* primitive to place the bricks.

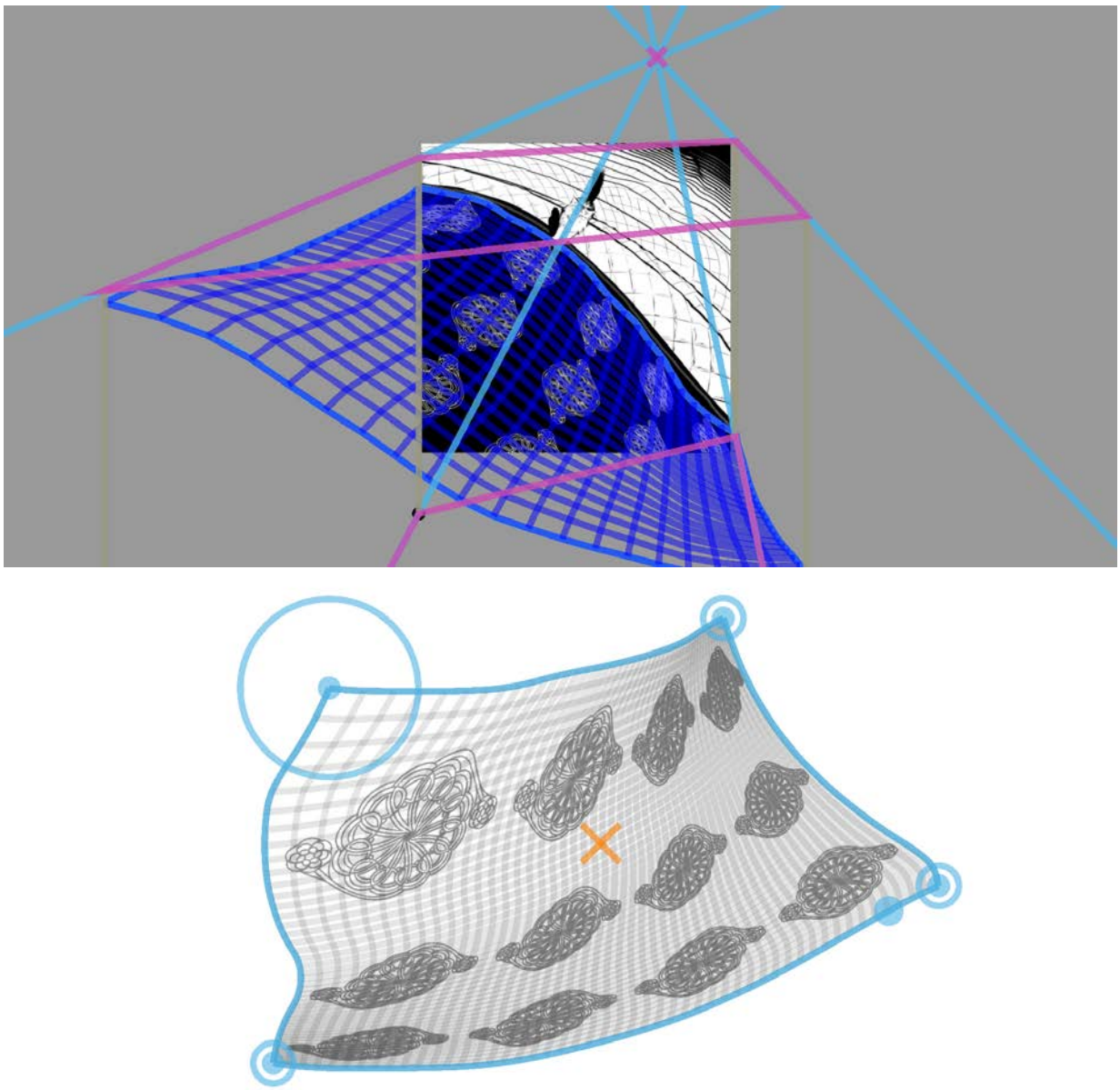


Figure 3.11: With the *box patch* primitive (top), I draw the outline of a projected patch and then modify the vanishing points of a fitted perspective box. With the *depth patch* primitive (bottom), I also draw the outline of a projected patch. I disambiguate the projection by choosing a camera position with canvas controls (orange X) and by drawing circles on the projection to indicate relative depths.

around the projection. I can alter the perspective by interacting with this box, mainly by dragging its vanishing points (one of these is visible at top center). There is no true underlying 3D scene, only draftsmanship tricks taken from Robertson [80].

The second primitive is the *depth patch*. With this one, there *is* an underlying 3D scene, defined intuitively through 2D controls. First, I can adjust the camera's position by moving its projected canvas position (as well as changing its focal length by altering the size of a circle centered at this position). Second, I can draw circles of varying radii on the projected patch to indicate how near or far different parts of the underlying 3D patch are from the camera.

These primitives are a valiant effort, yet bizarre, and more damningly, awkward to use. Again, this is not to self-deprecate; there is an important point here. The oddness of my box/depth patch primitives reflects a specific difficulty, one that is essential to my experience using NPR as an art medium. The limitation here was publishability more than my artistic goals themselves. If all I had really wanted was to illustrate complicated 3D objects under perspective, the obvious choice would have been to use existing CAD software. My position as artist *and* researcher made matters more complicated. I needed a tool that would meet my artistic goals *and* be sufficiently different from existing software techniques.

In the end, I did not manage to conquer perspective drawing with my tools, but my attempt to do so is an important thread to my story. I pick it up again in Chapter 7.

Chapter 4

Defining Hatching

4.1 Introduction

Once my testbed application had basic abilities, like drawing strokes, editing them, and saving files, I was ready to start integrating more experimental art tools. First I wanted to tackle hatching. Before doing so, I took the time to flesh out hatching as a research problem, to write a definition of this drawing technique that would be useful for other NPR researchers. That definition is the subject of this chapter.

Hatching is a drawing technique that uses patterns of roughly parallel curves to achieve multiple ends, the most traditional of which are low-level depictive goals like creating gray levels and showing form and texture. Hatching also fills higher-level symbolic and compositional roles.

Figure 4.1 shows hatching's versatility: marks cooperate to establish lights and darks, show the curvature of forms, and suggest texture. The hatching in this image is utilitarian; it mainly serves to depict geometry so that the viewer can understand the scene at a literal level. Yet the hatching has a beauty of its own, an appeal separate from its depictive efficacy.

Before presenting my definition of hatching in Section 4.3, I elaborate on hatching's roles in Section 4.2. I write about possible future applications of my definition in Section 4.4. Finally I give concluding thoughts in Section 4.5.



Figure 4.1: *Hatpin Girl*, Charles Dana Gibson, pen and ink

4.2 Purposes

Hatching apparently originated out of convenience. Some artistic media require artists to rely exclusively on marks made with a sharp instrument—points and curves in effect. Hatching is a natural response to this limitation: it turns the lowly curve into an all-purpose illustrative mark. (Similarly, stippling [66] turns the point into an all-purpose mark.) Printmaking media such as etching often impose the sharp-instrument constraint, which is why hatching has a long history as a halftoning [56] technique.

Over time, hatching has come to serve three basic depictive purposes: creating value, showing form, and indicating texture. It also has uses as a symbolic device and as a com-



Figure 4.2: White-on-black hatching

positional tool. Finally, hatching is intrinsically appealing, even when not clearly serving one of the former five purposes. This appeal is a sixth purpose in itself.

Value (or "tone" [64]) means brightness or gray level. In line art as defined in Section 1.2, there are typically just two actual values to be found in the canvas: the value where ink is present (typically black) and the value where ink is absent (typically white). Yet hatching can create the impression of other values when groups of marks blend *visually*. This makes hatching a de facto halftoning technique, like stippling. Hatching marks can individually be in any tone or color, of course (Figure 4.2), though we move outside of line art once more than a few ink colors are in play.

Hatching does not directly produce a desired gray; marks act together to create the impression of one—an "optical gray" [23, p. 151]. Artists create different optical grays by altering curves' spacing, their widths, or the value of the ink used to draw them (Figure 4.3). They may also use multiple layers of hatching marks moving in different directions to produce *crosshatching*, as in Figure 4.4. (For some artists, "hatching" and "crosshatching" are mutually exclusive terms, the first applying only to a single layer of hatching marks. In contrast, I treat crosshatching as a specific technique under the hatching umbrella.)

Any drawing marks that establish value can also establish form (and more—see Figure 4.5). This is because value encompasses *shading*: showing which sides of objects are in light or in shadow, along with cast shadows, core shadows [33, p. 46], and highlights. Yet there is a second mechanism for portraying form that is unique to hatching and completely unrelated to value: if the curves wrap around an imagined 3D surface, the eye will

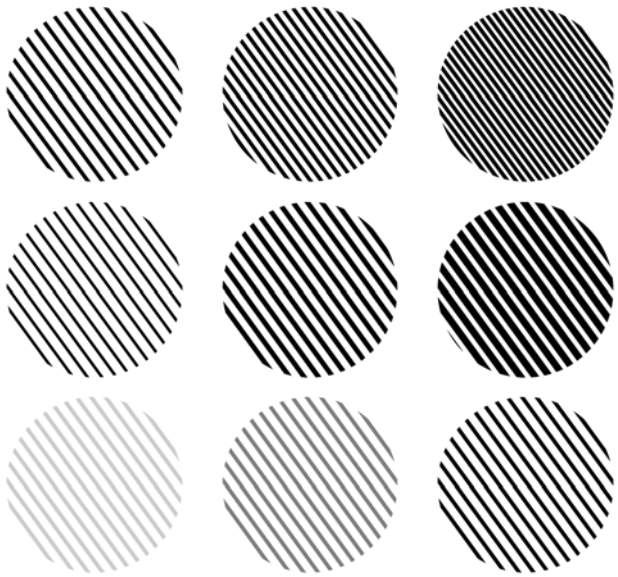


Figure 4.3: To create different "optical grays" with a single layer of hatching, vary marks' density (top row), their widths (middle row), or the value of the ink used to draw them (bottom row).

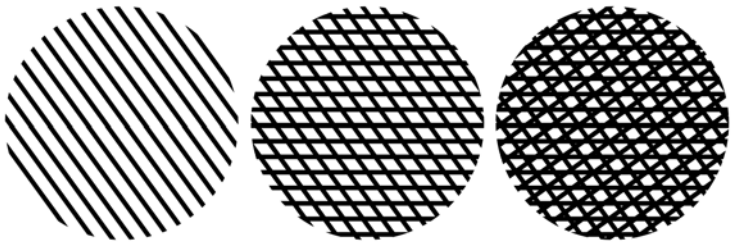
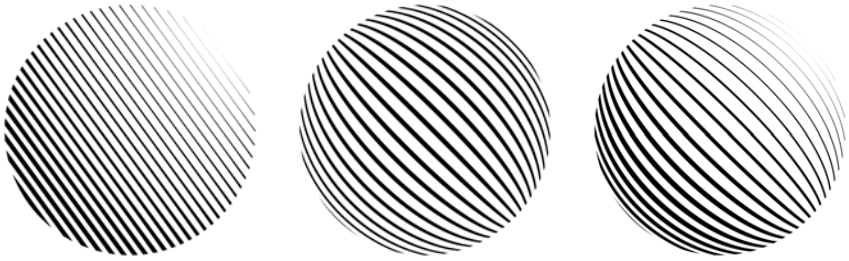


Figure 4.4: Crosshatching is a way to produce different optical grays using multiple layers of hatching.



Figure 4.5: Value does more than just show form. Here, gray levels do establish the sphere's form through light and shadow, but they also establish its (monochromatic) color pattern.



(a) (b) (c)

Figure 4.6: Hatching can show form through value variation (a), suggesting perspective (b), or both at once (c).

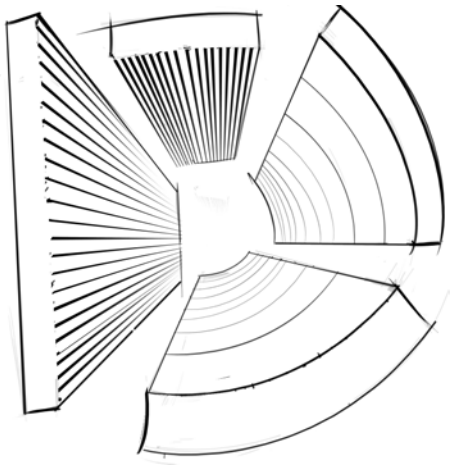


Figure 4.7: Varying hatching marks' spacing or having marks converge can produce the effect of perspective.



Figure 4.8: Hatching can illustrate texture on the surface of macroscopic forms.



Figure 4.9: Hatching can illustrate texture that emerges from numerous separate microforms, like hairs or blades of grass.

perceive depth, even without lighting cues (Figure 4.6b). Imagine starting out with 3D curves based in the scene, specifically isocurves within one of the scene's surfaces. When two of these isocurves are projected to the image plane to produce hatching marks, the varying distance between the marks will show the surface's curvature and the foreshortening caused by perspective (Figure 4.7). Art literature sometimes calls such hatching marks *cross-contours* [23, p. 130]. Principal curvature is often a good choice for guiding the surface curves that get projected into cross-contours [28, 42]. There are more specific tricks for conveying perspective when hatching cylindrical objects [69, 101].

Before explaining how hatching depicts texture, I must disambiguate "texture." Here, that term denotes form at small scale: *microforms* in a three-dimensional scene, or the illusion of these microforms created in the mind of the viewer by hatching. "Texture" does not have the meaning here that it often has in 3D graphics contexts, where it refers to

a color pattern on a perfectly smooth surface, like in Figure 4.5. Nor does it refer to the interplay of 2D marks within the image plane; I use "pattern" for that purpose.

There are two kinds of microforms. In the more intuitive case, a macroscopic object has minor surface details that are forms in their own right, like the ridges on a shell or on the cut surface of a piece of wood (Figure 4.8). Less intuitively, a group of separate microforms can appear as a single form: 100,000 individual hairs simplify into a hairdo; millions of blades of grass coalesce into a giant prism (Figure 4.9). The kinds of texture ideal for hatching are those whose microforms are grooves or strands.

A single hatching mark does not strictly correspond to a single microform. Whether working from life or from imagination, drawing every hair, feather, or blade of grass is prohibitively difficult and often produces an unappealing result anyway. Thus, hatching is a *shorthand* for texture, with varying degrees of abbreviation. Take Woodstock from Schulz's *Peanuts*, whose head of plumage can be represented by as few as three marks (this is hatching as visual abstraction [20]). A related kind of visual abbreviation is to detail the texture only on certain parts of a form, letting the mind fill in the blanks; Winkenbach and Salesin [104] and Grabli et al. [30] call this *indication*. Figure 4.10 displays both types of abbreviation.

There is a principle here: if a group of hatching marks represents a texture to the viewer, they will keep representing more or less the same texture even if some marks are added or taken away. Later I generalize this principle and claim that flexibility is an essential aspect of hatching. Whatever pictorial role a group of hatching marks currently fills, it will keep filling that role subject to minor changes in density, neatness, direction, and so on.

With value, form, and texture in its purview, hatching has a great deal of depictive power. Yet hatching can do more than just render. Figure 4.11 shows two examples of how clusters of radiating lines—a kind of hatching, in my opinion—can read as light sources. In the Dürer detail (top), the effect could be called simply depictive because the eye sees streaks of white against a dark background, matching the experience of looking at a bright light. However, in the stylized night sky (bottom), the groups of radiating lines are black against white, not what a camera or an eye would record. This is more symbolic than literal. In comics and other graphic art, hatching can work on a completely symbolic level, like with action lines (Figure 4.12).

Hatching also takes on a non-rendering role when it starts serving compositional purpose. Composition operates above the level of merely showing what a scene looks like. It is not just about putting information in the image plane, but about guiding the eye through the picture, creating beauty, and inspiring emotion. Figure 4.13 shows hatching applied with a compositional purpose. Finlay uses radiating lines to depict a light source

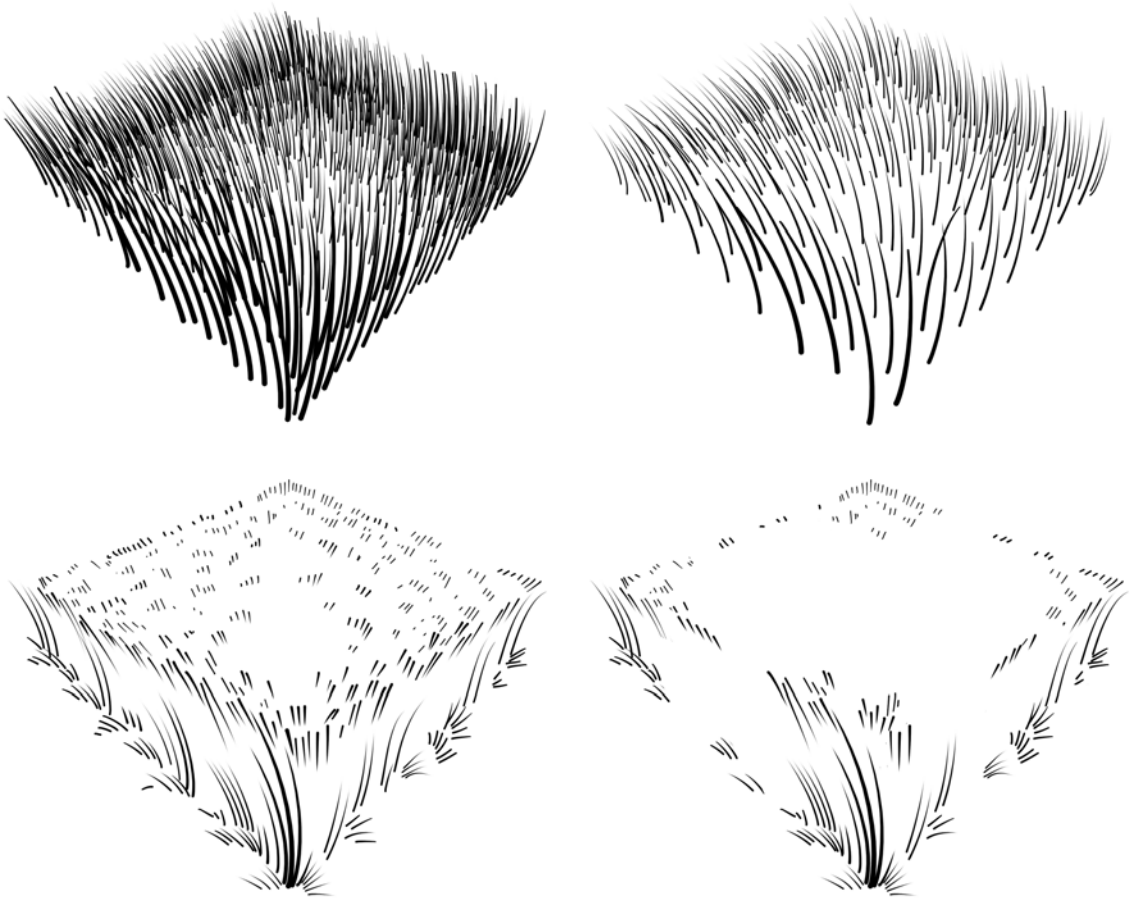


Figure 4.10: Each row has the same texture portrayed with different numbers of hatching marks. That both versions convey the same idea shows how hatching depicts texture abstractly, not literally. In the second row, some areas are free of marks, letting the viewer interpolate the texture mentally; this is called *indication*.

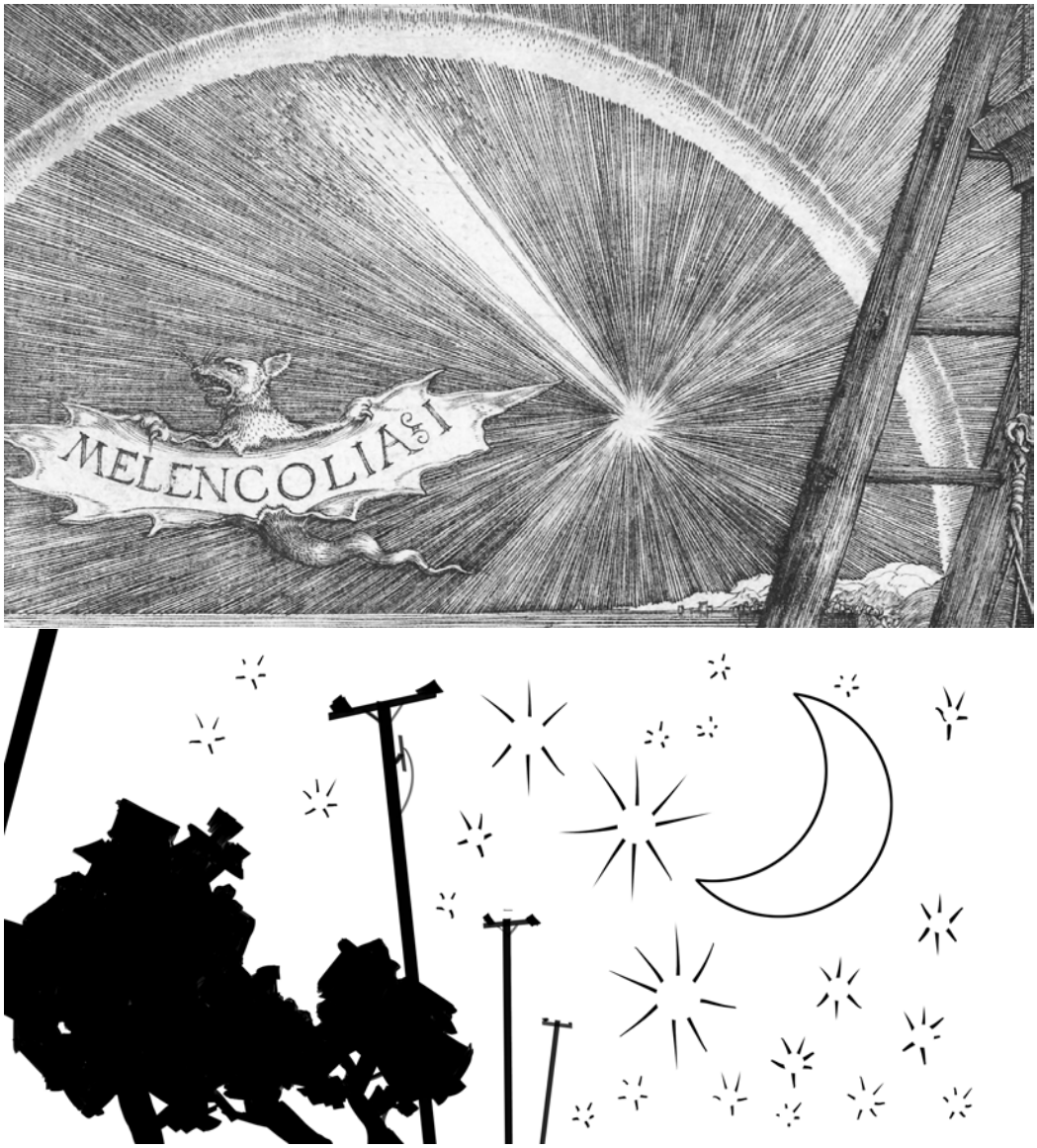


Figure 4.11: Detail from *Melancholia*, Albrecht Dürer, 1514, copperplate (top); night sky with stars as radiating hatching marks (bottom).

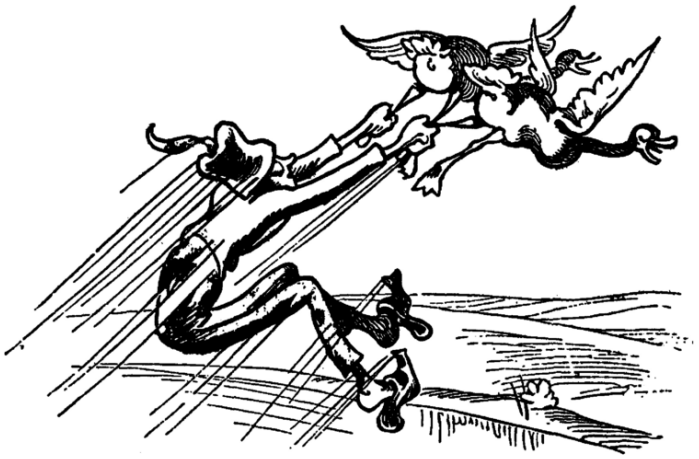


Figure 4.12: Some hatching marks are purely symbolic, like action lines. Illustration from *Max und Moritz*, Wilhelm Busch, 1865, wood engraving.



Figure 4.13: Hatching serves compositional purposes here, energizing the scene and leading the eye. *Trajectory to Taurus*, Virgil Finlay, pen and ink. Copyright Lail M. Finlay, used with permission.

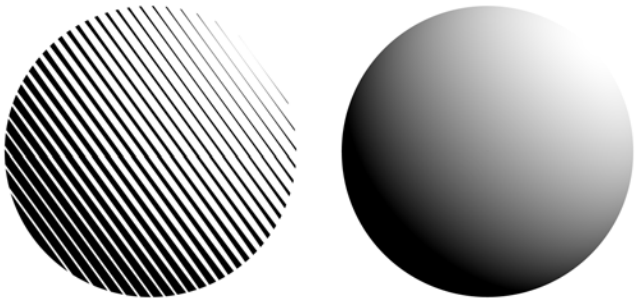


Figure 4.14: The left is functionally a lossy optical-gray approximation of the right, yet because of the hatching the left is more beautiful.

like Dürer does, but in a creative, multipurpose way. The radiation covers the whole scene, not just indicating light but guiding the eye and energizing the composition.

Hatching may even abandon its connection to a scene entirely, becoming pure ornamentation. According to Aldous Huxley, drapery (folded cloth) in art is an end in itself without regard to the scene around it [43]. Hatching has the same context-transcending charm. Its beauty might just derive from the viewer's unconscious appreciation of a correctly rendered scene, or it could be deeper than that. Figure 4.14 illustrates the question. The bending of hatching marks to show surface curvature is deliberately omitted here. The straight hatching marks at left do nothing but produce optical grays. Yet even though the hatching is, in a way, just an imperfect rendering of the "true" scene at right, the hatching is more interesting, suggesting that hatching's appeal is at least partly independent of its rendering function.

4.3 Definition

My definition of hatching implicitly encompasses all the purposes just described and accommodates both human- and computer-generated hatching:

> Hatching is the use of one or more *patches*, where a patch is a group of roughly parallel curves that form a flexible, simple pattern.

For *creating* hatching, this definition needs no further explanation. Yet for *identifying* hatching, I need to elaborate on the terms *curve*, *patch*, *flexible*, and *simple*. Be aware of the subjectivity in this elaboration. There will inevitably be cases where a group of marks is not definitely hatching and not definitely *not* hatching.



Figure 4.15: Hatching curves can be closed. Otherwise, two of the curves on the right would be arbitrarily excluded.



Figure 4.16: Hatching marks can trail off by narrowing, breaking up, or fading (in the last case, we have moved outside of line art).

4.3.1 Curve

Because hatching is so multipurpose, rigorously defining a hatching curve is impossible. Its degree, curvature, and continuity are unstatable. It may intersect itself, be open or closed (Figure 4.15), or have gaps.

Sometimes the artist will have individual hatching marks trail off, such as by varying pen pressure during a single stroke. Along its length, a curve can widen or narrow, break into gaps, or change in transparency (Figure 4.16), though the last technically takes us outside of line art.

"Curve" is a misleading term. With more typical hatching, it is tempting to think of each mark as a one-dimensional path, but even the possibility of this mark varying in width makes a 1D curve insufficient for representation. The mark is not a path but a shape with area, and it may be fuzzy or discontiguous. The sides of this long shape could be ragged, especially in a drawing made with physical media or meant to *look* made with physical media. An artist might even refine an individual curve-shape by drawing other

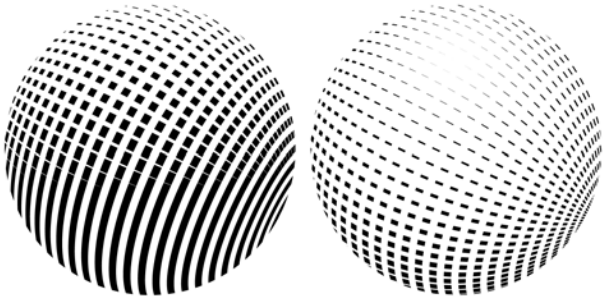


Figure 4.17: Both scenes show a sphere lit by a single light. Changing the position of this light illustrates the fineness of the distinction between hatching and stippling.

curve-shapes on top of it, creating something that reads as a curve to the eye but appears as a spiky mess to algorithms that take either raster- or vector-based input. Work in sketch beautification and simplification [6, 61] is relevant here.

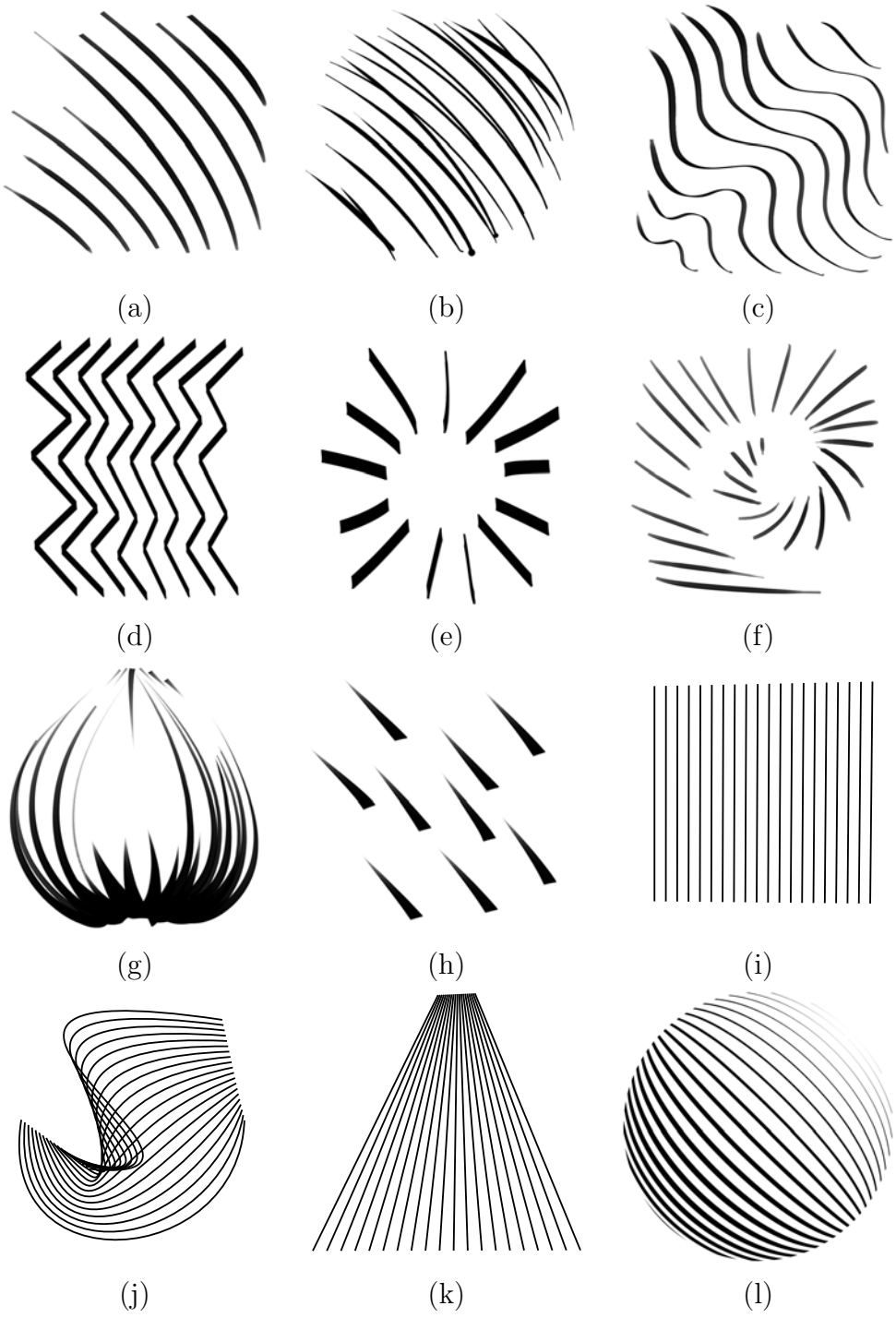
Hatching or stippling?

Figure 4.16 shows curves breaking up as a way to trail off. Past a point, breaking up curves produces a result that most would call stippling, not hatching. Figure 4.17 shows that it is hard to state when one becomes the other (unless one describes the two images as white hatching on top of black hatching). This is one simple way that identifying hatching becomes a necessarily indefinite task.

4.3.2 Patch

A *patch* [26] is a single instance of hatching, a group of curves that run together and can be loosely ordered from first to last such that each curve is about parallel with the next (Figure 4.18). It may be useful to think of each curve in a patch as a parallel offset curve of the one before it, but this need not actually be the case. Curves in a patch can intersect, whether because of sloppiness in their placement (Figure 4.18b) or because the patch overlaps itself (Figure 4.18j).

When separate patches overlap, the result is crosshatching (Figure 4.19). This technique is very old and useful, but I refrain from investigating it in detail since my definition of hatching operates at a simpler level. In my definitional context, crosshatching is just a case of hatching involving more than one patch. Briefly though, inter-patch relationships



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

Figure 4.18: Patches

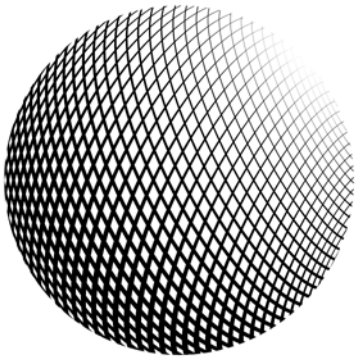
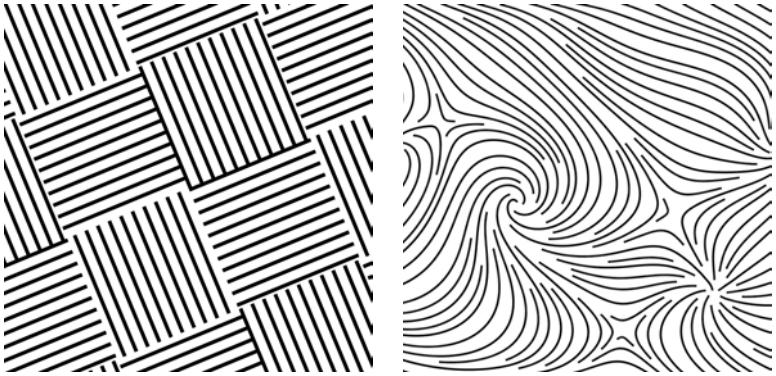


Figure 4.19: Crosshatching



Figure 4.20: The patch on the back of the hand is separate from the patch along the hand's heel, even though one, more highly curved patch could have done the job of both. Using multiple patches is often easier and faster. Detail from Alexander Browne's *Ars Pictoria* [10], 1669, copperplate.



(a) (b)

Figure 4.21: Some patches are easily separable (a), others less so (b). Second image from Jobard and Lefer [46].

are important and worth exploring further elsewhere, for one because artists often use overlapping patches to deal with complicated forms (Figure 4.20).

Isolating patches

To decide which curves in a drawing constitute a patch is nontrivial. My approach is to be generous. Figure 4.21 shows two images, each with multiple patches. The second is an illustration of a vector field using streamlines—not conceived of as hatching but in the spirit of the term as I see it. Not all its patches are easily separable. It might be worth clarifying that two patches should be allowed to share curves, thus making the streamlines image more comfortable to classify as hatching. Too much generosity produces absurdity, however. Simply grouping all the lines in a drawing that move in the same direction would seem to satisfy the definition of a patch just given, but such a disparate collection should not usually be considered a patch. This is where the last part of my definition of hatching becomes relevant: the marks in a patch should form a "flexible, simple pattern."

4.3.3 Pattern flexibility

A patch should form a pattern that is not so rigid that any alteration of marks' directions, curvatures, or density would change the semantics of the image.

Take the reticulated cube in Figure 4.22. Ostensibly, each of its sides contains two patches overlaid to form crosshatching. Yet the pattern is highly regular, and is easily read

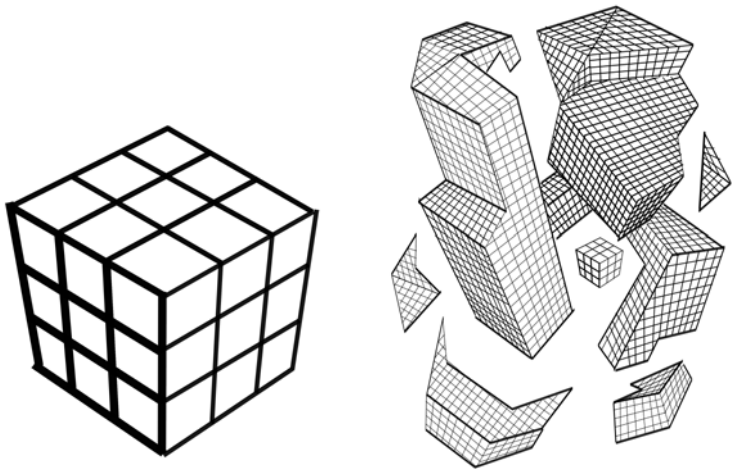


Figure 4.22: In isolation, the cube's marks look too regular to be hatching, but this changes when the cube is in a context full of similar marks that are obviously hatching.

as the grooves in a Rubik's Cube. Changing the density of the hatching—the number of sub-cubes, in effect—would make the cube no longer a Rubik's Cube, thus changing the semantics of the scene. In other words, the pattern is inflexible. Context matters, though. If the same cube appears as a small part of a scene full of other reticulated forms, its own marks seem much less semantically important and more like a hatching pattern.

Figure 4.23 explores the flexibility test with a question applied to four test drawings: in each drawing, can all approximately vertical marks be grouped in a single patch? In the first test drawing (a), there is a similar problem as with the cube example: the scale of the pattern is too coarse. Each contour on the inside of the tablecloth's silhouette is the side of a fold, and the folds are so large relative to the composition that they are almost forms in their own right rather than microforms on the surface of a macroform. There is another snag, one not addressed in the cube example. Two of the vertical marks up for consideration as members of a patch happen to be silhouette contours of the whole tablecloth, a form that is undeniably macroscopic. It is traditional to think of silhouette contours as separate from hatching, though I do not feel strictly beholden to this distinction—see Figure 4.24.

The second test drawing (Figure 4.23b) introduces more marks next to each of the marks in the first test drawing (a), creating several different patches. For the same reason as in the first drawing, i.e., the relatively largeness of individual folds in the cloth, it is still incorrect to group *all* the vertical marks as one patch. But the presence of hatching in the scene is now indisputable.

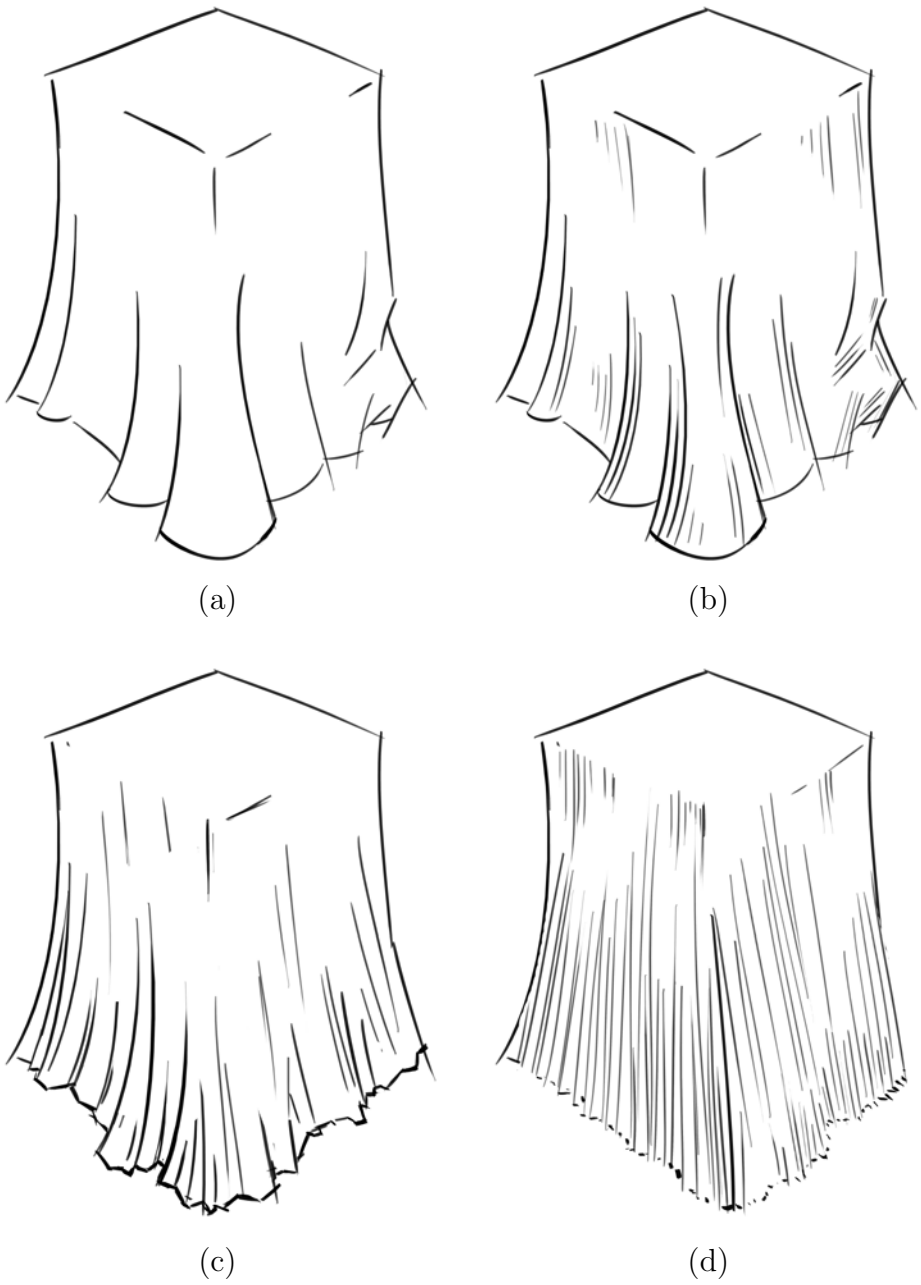


Figure 4.23: If the folds are too large (a), it is hard to call their outlines hatching. However, additional marks (b) meant to reinforce the folds without defining new ones are hatching. When the folds are small enough (c,d), their outlines become hatching.

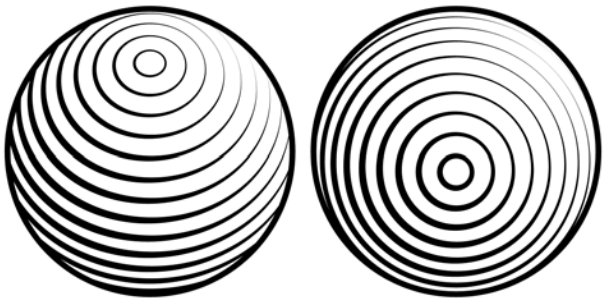


Figure 4.24: Silhouette contours are customarily distinct from hatching (left), though a patch may be aligned such that it would be unreasonable to exclude a silhouette contour from it (right).

There is a more drastic change in the third test drawing (c). Now the scene geometry is different from that of the first and second. Perhaps the cloth is of a different material. The folds in the third drawing are small enough to enter the microform domain, making it easier to group the image's vertical marks into one patch, maybe two.

In the fourth test drawing (d), the folds are even smaller. Yet the change is less significant. Moving from the second drawing to the third means going from macro- to microforms. The transition from the third drawing to the fourth is less significant.

The flexibility test distills to two points: First, a patch's curves must share a purpose (which means that to identify a patch is to apply the Common Fate principle from Gestalt psychology). Second, the fewer the curves, the less confidently they can be regarded as a patch, especially when the specific number of curves appears to be important. Conversely, if it is possible to vary the number of curves in a group without affecting the purpose they serve or altering the image's semantics, then that group is more likely to be a patch.

4.3.4 Pattern simplicity

The previous test requires that curves in a patch work together in a flexible pattern. However, curves can cooperate too much, creating inter-curve patterns that make it harder to classify the group as a patch. To clarify, there is no problem with patterns contained inside each curve individually. There is no issue with a patch made of sine curves offset from each other, for instance, or with the third and fourth patches (c,d) of Figure 4.18. The problem is when one curve in a patch has a special relationship with some other curve in the same patch, a relationship it does not have with others in the group.



(a) (b) (c)

Figure 4.25: When the curves of a single patch (a) start to relate to each other in particular ways (b,c), the patch is no longer definitely a patch.

Figure 4.25 shows three grass tussocks. It is easy to classify the first as a single patch. The second is trickier. The longer curves represent blades of grass while the smaller ones represent seeds. All the curves move together, but each seed curve has a relationship with a particular blade curve. It would be cleaner to call the grass blades a patch and each blade's group of seeds another two patches.

In the third tussock, inter-curve relationships are more numerous and particular. The curves outline individual blades of grass and honor occlusion relationships between the blades. There are special relationships wherever two curves outline a single blade or where one curve stops at its intersection with another in order to create occlusion. These relationships make it hard to call the whole tussock a patch, yet I cannot use the previous trick of partitioning the group into multiple patches either. Two curves outlining a single blade of grass are not a flexible pattern—what would it mean to increase this pattern's density?

4.4 Future Hatching Research

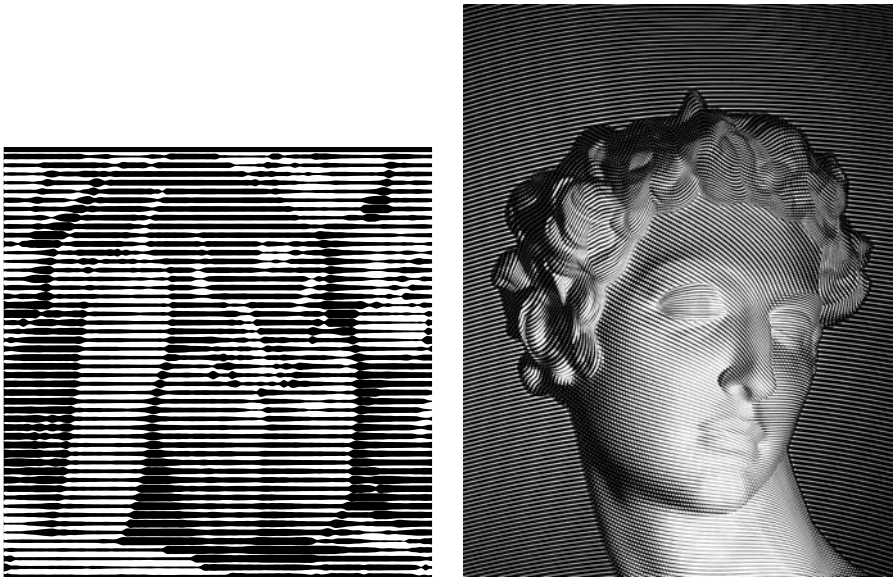
4.4.1 Creating hatching

Past NPR research has investigated creating hatching in three different contexts. One is rendering 3D models [104]. Then there are image-to-image applications where an input picture becomes a hatched illustration [81, 82, 75]. The third and least developed area contains applications like "Vignette" [50] and *Strokes Maker* [3], where the user directs hatching through curves drawn on-screen. These categories are fluid: Kalnins et al.'s application [48] is technically a 3D renderer, but the user can draw hatching marks in screen space. See Lawonn et al.'s survey of techniques for generating hatching [57].

The issues faced within these three areas are similar, with the exception of some concerns specific to 3D rendering.

Intuitively, software should outperform a human at hatching. For humans, creating patterns of curves requires exercising fine motor control over sustained periods. At the very least, I can expect software to draw a given patch far more quickly than any human. Yet with software there is a danger of creating hatching that looks too correct, sterile, or cheap. It is possible for a computer to generate a hatched version of any image in a facile way akin to screening [94]: a single patch covers the image, its marks varying in width according to the image's local tone (Figure 4.26a). The technique has no longevity; only the first few images generated with it will look impressive. A human would have difficulty making the same marks by hand, but this does not make the computer's output much more appealing. When converting images to hatching, it is better to imitate a human artist, breaking the input image into areas of light, shadow, and form and choosing where to place patches according to this breakdown. "Digital Facial Engraving" [75], in which an artist turns a photograph into a hatched illustration, lets that artist decide where patches should go and how their curves should bend (Figure 4.26b). To apply the same spirit in a different application, a 3D hatching renderer can imitate the old artistic practice of simplifying a scene's values into a small number of groups like highlights, midtones, and shadows, and then hatch consistently within these groups [42].

Another danger in procedural hatching is moiré patterns. Figure 4.27 shows how crosshatching can cause these when the marks from different patches are too close to parallel. One solution is to imitate the irregularity of the human hand in placing and drawing individual marks, preventing unintended patterns from arising [101]. See AlMeraj et al. [2], who present a way to draw a single curve with anthropomorphic irregularity. One can also use human-made hatching examples to train (in the general or machine learning-specific sense) software that generates hatching [47, 49, 26].



(a) (b)

Figure 4.26: The simplest way to turn any image into hatching is via a single patch with straight curves (a). It is better to break up the image into different patches whose curves have purposeful directions (b). The original for the first image comes from www.imageprocessingplace.com. The second is from "Digital Facial Engraving" [75].

Imitating human output is a worthy goal in hatching applications generally. Understanding the human *process* of hatching is perhaps even more crucial for interactive applications where mouse or stylus input drives hatching. Recall the vagueness of the hatching curve definition in Section 4.3.1. This vagueness is necessary to encompass all marks considered hatching. A specific interactive application can use a much stricter definition. In the right context, one might assume that hatching marks come only from simple flicks of the wrist (a comfortable way to draw, especially for those trained in life drawing). Under this assumption, I may define a hatching curve as an open, degree/2 Bézier curve (Figure 4.28), this representation presumably being capable of encompassing most flick-of-the-wrist marks. This point applies not just to the definition of curves, but to the whole definition of hatching. All its parts can become more technical and exclusive according to a project's specific goals.

In 3D rendering, hatching marks often start out as object-space entities to be projected to image space. In "Real-Time Hatching" [78], a hatching mark originates as a shape in a texture that is mapped to an object's surface. In the sphere renderings I have used

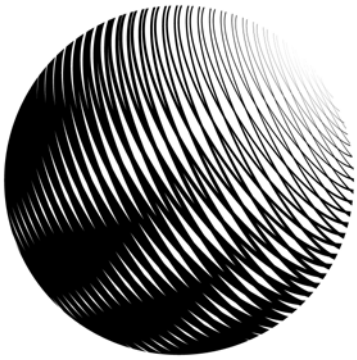


Figure 4.27: Computer-generated hatching marks are susceptible to moiré patterns.

throughout this chapter, hatching marks are just procedurally generated stripes in UV space, which is a simplified version of how Knöppel et al. [53] produce hatching.

Originating hatching marks in object space like this is convenient because the object-world-camera chain easily handles the complexity of bending and spacing marks to convey form as in Figure 4.6b. However, hatching is ultimately a 2D design tool, and only when treated thus can it show all its capability, including its compositional applications. The Zorn etching of Figure 4.29 has multiple "lost edges." That is, some silhouette edges are deliberately omitted for simplification and emphasis. The most (or rather, least) prominent of these is the edge between Zorn's jacket and his wife. This technique is fundamentally two-dimensional, and hatching contributes to it. A hatching renderer that deals with hatching marks as object-space curves cannot approach such techniques easily.

3D rendering has access to more of hatching's potential when it treats hatching as an image-space concern, informed by the 3D pipeline but operating outside it. Work is already moving in this direction. In the renderer of Lengyel et al. [58], the placement of hatching marks occurs completely in screen space. The renderer of Orbay et al. [74] goes much further. It calculates screen-space shadow shapes and fills them with marks whose paths are based on the shape they need to fill. It even breaks up complex shadow shapes into separate patches the way a human might.

4.4.2 Analyzing hatching

The obvious motive for analyzing existing hatching is to improve procedural hatching, but there are other reasons. Most involve deriving a scene or object from a hatched illustration. The ability to generate a 3D form from hatching could be useful in product prototyping, for



Figure 4.28: Hatching marks made with flicks of the wrist can be treated as quadratic Bézier curves.

example. BlendFields [44] and CrossShade [90] almost get to the point of producing form from hatching in the product design context. They process drawn silhouette contours and cross-contours to produce geometry. Whether cross-contours can always be called hatching is debatable, but certainly the process by which cross-contours guide geometry generation could apply to more conventional hatching.

Product prototypes are not the only forms to be inferred from hatching. Most hatched drawings contain inferable 3D information, and this information can be much more interesting than mere smooth surface geometry. A scene generator could read texture from hatching, perhaps in a scale-agnostic or scale-flexible manner, according to the loose way in which hatching communicates texture. Even if an algorithm that produces a 3D scene from a 2D illustration has no particular use for hatching, it will benefit from being able to detect and filter out hatching marks. Deep learning seems suitable for this sort of drawing analysis.

More unusual applications may lie in the future. Consider hatching's demonstrated ability to describe forms in medical illustrations [69, 95, 79]. One might use procedural hatching to create a medical illustration from a ground-truth 3D model, use a form-from-hatching algorithm to derive a new 3D model from the hatching, and measure the difference between the original and new models to study the reliability of either or both of the algorithms involved.

I can imagine algorithms at a more rarefied level that digest a hatched illustration to produce a "2D scene": a high-level breakdown of the image's pieces and an analysis of how the pieces work together, perhaps even an indication of why beauty emerges or does not emerge from the design. A preparatory step would be to infer a smooth grayscale image from a hatched one, that is, to move from left to right in Figure 4.14. Such esoteric investigations would at least be useful for artists who hatch.



Figure 4.29: *Zorn and His Wife*, Anders Zorn, 1890, etching

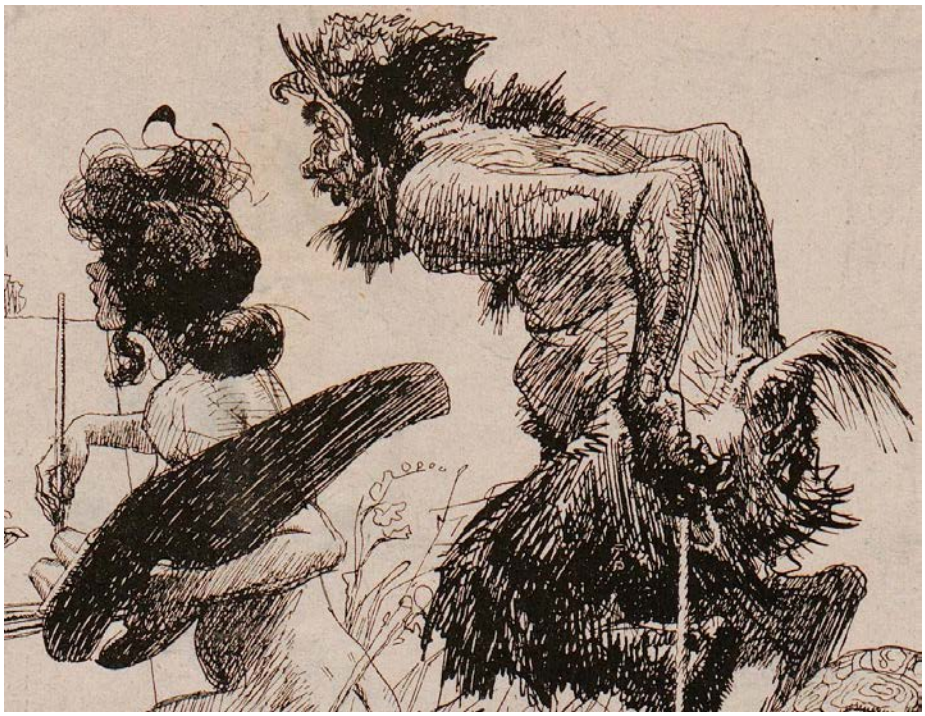


Figure 4.30: Detail from *Der Faun*, Heinrich Kley, 1912, pen and ink



Figure 4.31: I might interpret the curve at left as an entire patch, nearly equivalent to the patch at the right except made without lifting the pen between individual strokes.

Whatever the application, software that detects or interprets hatching must be robust enough to deal with how messy hatching can be. Figure 4.30 contains hatching that could be challenging for an algorithm to parse. Look closely and notice that some of its patches are actually single strokes, the result of hatching quickly without lifting the pen. Thus, a complicated back-and-forth curve might be best interpreted as a patch of simple curves glued together (Figure 4.31), or it might be a member of a patch along with other zigzag curves. Context must inform the decision.

When inferring 3D geometry from hatching patches, there is a risk of perceiving depictive intent from decisions that are stylistic and non-depictive, decisions that are only meant to accommodate the artist's hand, or decisions that are just mistakes. We have already seen Figure 4.20, in which the artist uses multiple patches where a single, more complicated one would be more geometrically correct. An algorithm meant to infer form from hatching might erroneously perceive extra surface detail from patch multiplicity when it should instead account for incidental reasons why two patches might appear instead of one.

The challenge of analyzing hatching does not arise solely from the artist's haste or ineptitude. Some quite deliberate hatching decisions can pose difficulty, like in Figure 4.29, where Zorn produces lost edges using hatching marks that obscure shapes by moving across their boundaries. A sophisticated algorithm is necessary for properly reading the hatching of such an image without being confounded by its numerous individually misleading marks.

4.5 Conclusion

Hatching is art, and art resists taxonomy. My definition of hatching cannot be authoritative. It is permissive enough to include marks that some artists would consider too complicated, like the streamlines in Figure 4.21; or too symbolic, like the black-on-white stars of Figure 4.11. Others might call my definition too restrictive. Rigidly labeling a subset of drawing marks is antithetical to art, which to some extent is about breaking rules. Yet NPR benefits from tentative guidelines and simplifications. The ones given here should be useful to that community, if only as something to argue against.

Chapter 5

Hatching Shapes

5.1 Introduction

After formally defining hatching, I began work on my second technical contribution, a tool for creating hatching. The value of such a tool was obvious. When working by hand, hatching is hard to create and even harder to fix. Editing means adding marks to an existing hatching pattern, which is prone to messiness; carefully erasing specific marks; or erasing the entire pattern and starting over. The difficulty is highest when hatching is dense, neat, and complex.

I reasoned that in a digital context, an artist might use a specialized primitive to direct a group of hatching marks collectively rather than adding or removing them individually. I wanted this primitive to operate just above the level of individual marks, giving me as much control as possible while sparing me from manual mark placement. The primitive I came up with was the *hatching shape* (Figure 5.1). A hatching shape renders a group of marks that run together, which is to say that a hatching shape is a *patch* as described in the previous chapter (Section 4.3).

A hatching shape defines hatching within a bounding box. Three fields give hatching marks' widths, directions, and the spacing between them. I manually edit these fields, after which streamline advection [46] uses them to produce hatching marks. A mask specifies where marks are visible within the bounding box.

A hatching shape also includes *barrier curves*, which interrupt the hatching. Figure 5.2 shows two hatching shapes, one with and one without a barrier curve. Figure 5.3 shows the importance of barrier curves in an actual illustration. The torso has discontinuities

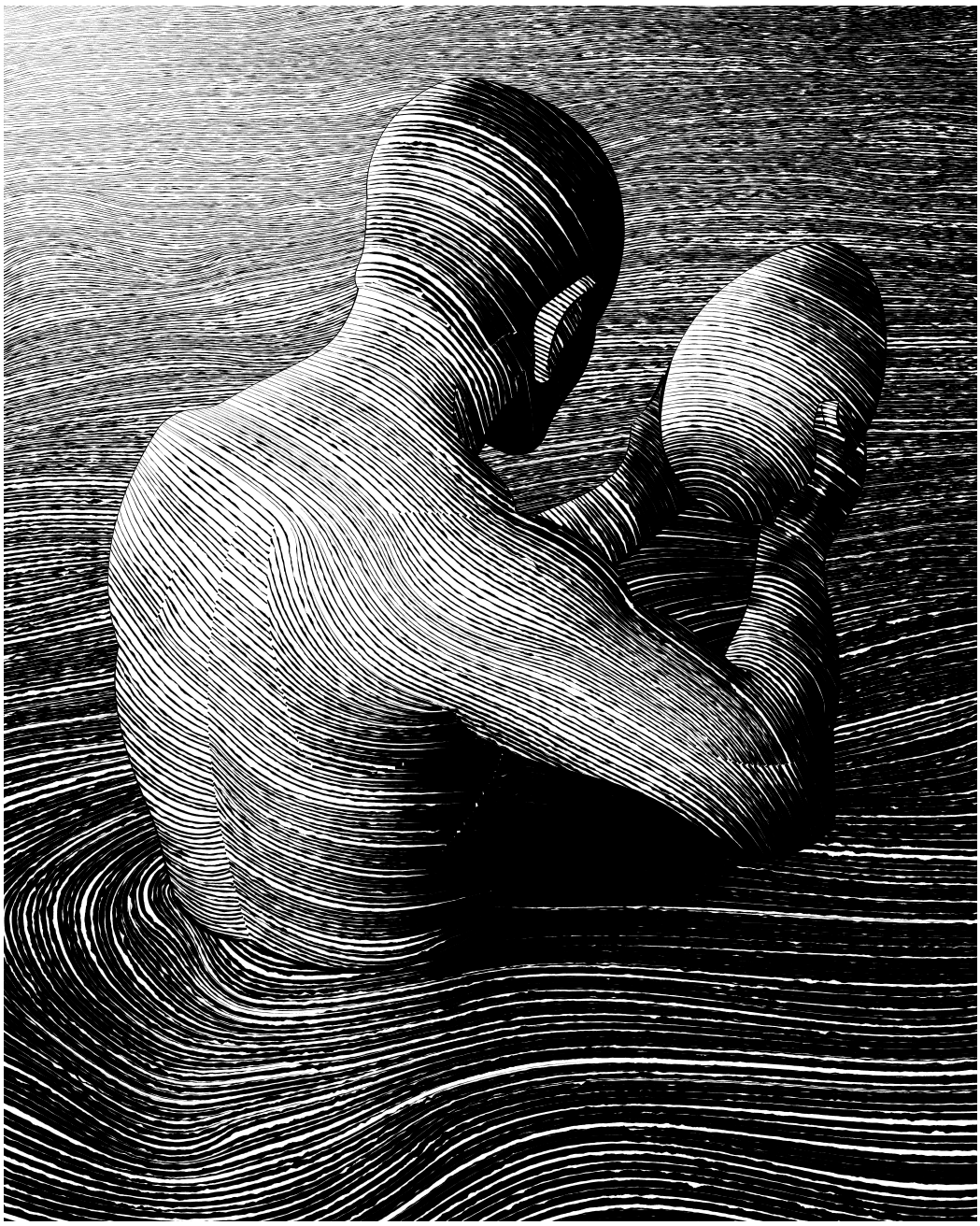


Figure 5.1: The *hatching shape* primitive is for making and editing procedural hatching. Four hatching shapes feature in this picture: one for the human figure, one for the egg, and two for the water.

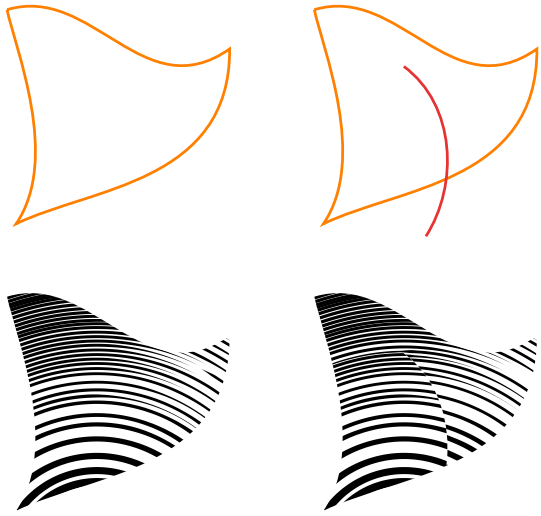


Figure 5.2: At top left is a hatching shape mask. At top right is the mask plus a barrier curve. Below is the hatching produced without the barrier curve (left) and with it (right).

when projected to the picture plane, such as the edge between the right arm and the right armpit. Barrier curves keep hatching marks from crossing such edges.

It is significant that a single hatching patch (produced by a single hatching shape) does the job of rendering the entire torso. Using a single patch to render a complex form like this is especially difficult by hand; it is more common to break such a form into multiple patches, as in Figure 4.20. It is not necessarily better from a pictorial standpoint to use a single patch, but since it is usually harder, it becomes a show of virtuosity when done by a human. To mimic such virtuosity in Figure 5.1 would have been impossible without barrier curves.

As a technical contribution, the hatching shape has two parts:

1. hatching shape operations with implementation details (Figure 5.4) and
2. modifications to streamline advection for producing hatching marks from a hatching shape.

The rest of this chapter is organized as follows: Section 5.2 explains the parts of a hatching shape. Sections 5.3 and 5.4 treat hatching shape operations. (Section 5.3 covers

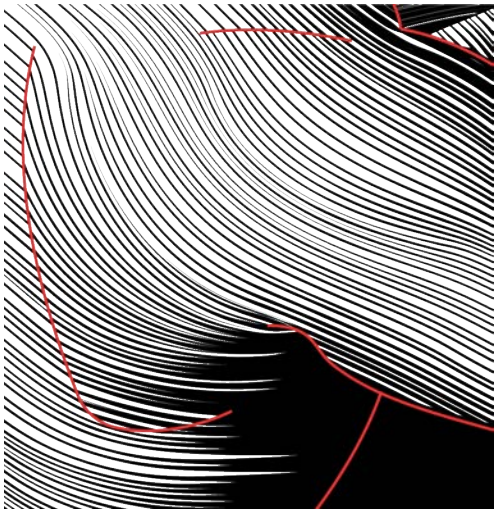


Figure 5.3: This detail from Figure 5.1 shows the importance of barrier curves for portraying a 3D form that is discontinuous in the picture plane. (Hatching marks are smoother here than in Figure 5.1 because per-mark Perlin noise-based roughness is disabled.)

basic operations and Section 5.4 explains more advanced ones that involve barrier curves.) Section 5.5 gives streamline advection details. Section 5.6 lays out the merits and limitations of hatching shapes, and in Section 5.7 I consider future work on the hatching shape.

5.2 Parts of a Hatching Shape

A hatching shape stores information for producing hatching within a bounding box. There are three fields, a mask, and a set of barrier curves.

The three fields—width, spacing, and direction—are 2D arrays with a common resolution. At each grid location, the hatching shape defines

1. a scalar *width*, the width of a hatching mark passing through that location;
2. a scalar *spacing*, the distance between the spines of two hatching marks; and
3. a *direction*, represented as the three unique components of a structure tensor.

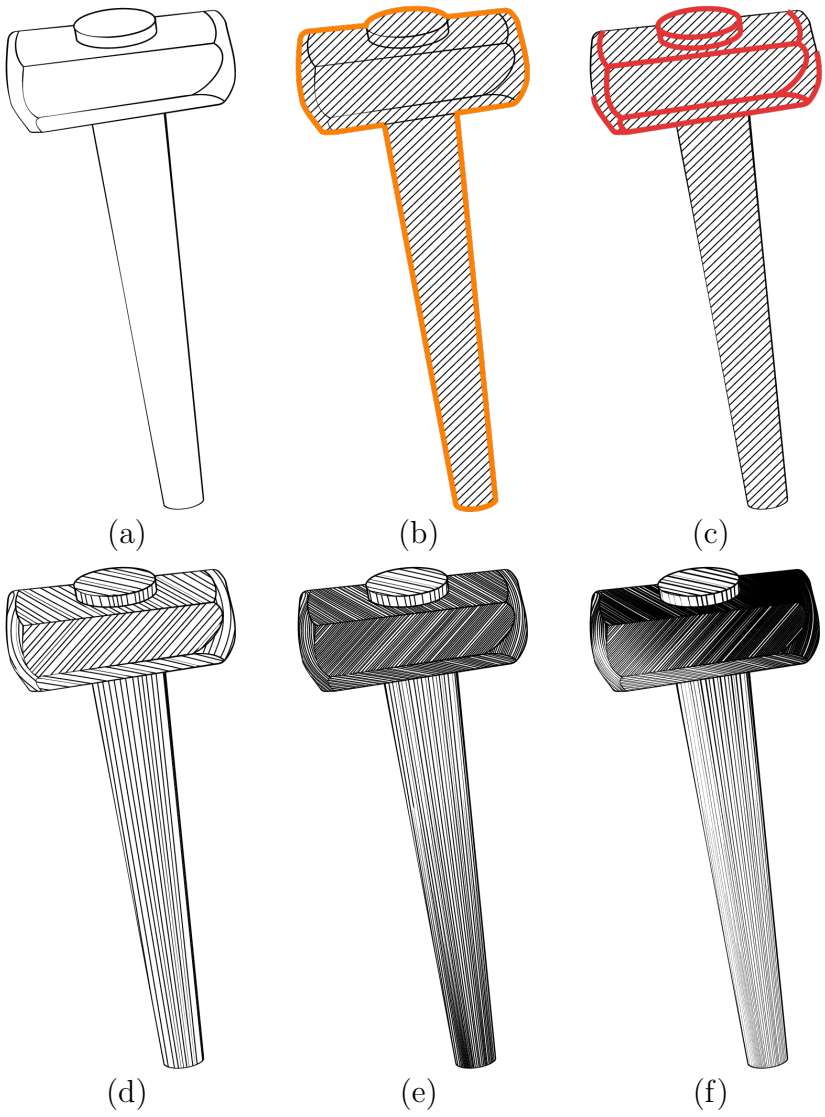


Figure 5.4: Example process for an illustration that includes a hatching shape: Draw strokes in the canvas (a). Create a hatching shape (b), possibly using some of those strokes to construct its mask (Section 5.3.1). Specify barrier curves (c), again possibly snapping to existing strokes. Set the shape's direction field (d) using the *multi-dir* operation (Section 5.4). Tweak the spacing field (e) and the width field (f) using fill and *blend stroke* operations (Sections 5.3.3 and 5.4).

I use structure tensors rather than 2D vectors or polar values since directions need to be 180°-symmetric. My tensors have the form provided by Knutsson [54]: given a vector $\mathbf{v} = (x, y)$, the structure tensor is

$$\begin{bmatrix} x^2 & xy \\ xy & y^2 \end{bmatrix} \tag{5.1}$$

Structure tensors are a useful representation because the vectors $\mathbf{v}$ and $-\mathbf{v}$ come out identical. This matters not so much when storing a hatching shape in memory, but is crucial when editing its direction field. I do ultimately convert tensors back to 2D vectors when sampling the direction field during streamline advection.

The resolution of a just-created hatching shape's fields depends on the shape's relative size in the canvas. By default, a shape that completely fills a square canvas will have a 50×50 grid. I can change the resolution if desired. The torso hatching shape in Figure 5.1 has a 200×243 grid.

To sample the width, spacing, or direction at any target position inside the bounding box, not just at exact grid locations, I must interpolate from nearby grid values. When no barrier curves are nearby, this means standard bilinear interpolation; otherwise, it is more complicated (Section 5.5).

In Sections 5.3 and 5.4, I discuss operations that modify a hatching shape's three fields. Often, the goal of these operations is to interpolate a smooth field from a sparse set of constraints. In such cases I solve *grid-Laplacians* [13]. A grid-Laplacian is a linear system that interpolates a scalar field f on a grid. The field has known values—Dirichlet constraints—at some grid locations and $\Delta f = 0$ (alternately written as $\nabla^2 f = 0$) elsewhere, meaning that f is as smooth as possible. For each unknown $f_{i,j}$, the system contains this equation:

$$-4f_{i,j} + f_{i-1,j} + f_{i+1,j} + f_{i,j-1} + f_{i,j+1} = 0 \tag{5.2}$$

When interpolating a smooth direction field, I solve separate grid-Laplacians for the three unique components of a structure tensor. My grid-Laplacians all take the four-neighbor form shown in Equation 5.2, though an eight-neighbor form does exist [17]. The importance of this choice becomes clear in Section 5.4, where I discuss barrier curves. I solve grid-Laplacians with the sparse simplicial LDLT solver from the Eigen library [32].

Since an artist might want to confine hatching marks to a shape other than a rectangle, a hatching shape also has a mask: a group of closed polycurves that do not intersect each other but may contain each other (Figure 5.5). Recall that a polycurve is a chain of connected curves.

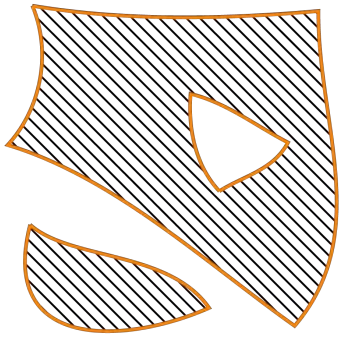


Figure 5.5: A hatching shape's mask is a collection of closed polycurves, three in this case.

I perform the masking itself in two ways. In OpenGL rendering, I fill a stencil buffer by rendering each polyline-approximated mask curve using a triangle fan, incrementing the stencil value at drawn locations [106]. This produces stencil values that are odd (with the least significant bit set to one) inside the mask, where hatching marks should be visible. Second, when I export drawings in PostScript form (as a precursor to creating PDF files), I use the mask curves as clipping paths.

The last part of a hatching shape is zero or more barrier curves (Figure 5.2). These must extend inside the bounding box to have effect.

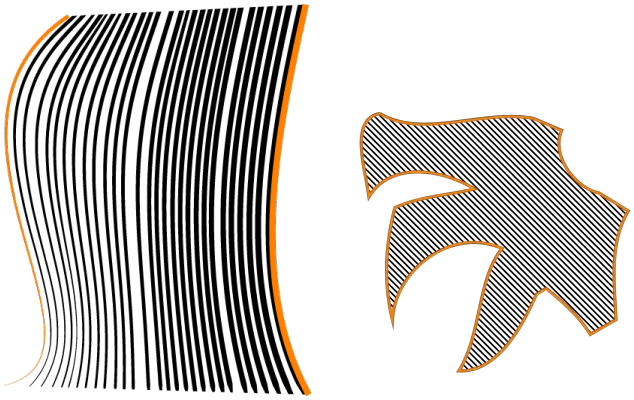
5.3 Creating and Editing Hatching Shapes

With the parts of a hatching shape defined, I can now relate operations on this primitive. In this section, I cover simpler operations that do not involve barrier curves but do not rule out their presence. These operations are hatching-shape creation (Section 5.3.1), mask editing (Section 5.3.2), fill operations (Section 5.3.3), sourcing a hatching shape's width field from a reference image (Section 5.3.4), field multiplication (Section 5.3.5), and crosshatching (Section 5.3.6).

Section 5.4 presents more complex operations that respect barrier curves.

5.3.1 Creating a Hatching Shape

One way to create a hatching shape is to draw two source strokes, as in Figure 5.6a. The strokes' combined bounding box becomes the hatching shape's bounding box. The spacing



(a) (b)

Figure 5.6: One way to create a hatching shape (a) is to draw two strokes defining opposite sides of a Coons patch. Another way (b) is to draw a closed polycurve, which becomes the new hatching shape's mask.

field starts out with a tunable uniform value (the irregular spacings in the figure are an artifact of the streamline placement algorithm—see Section 5.7.2). The width and direction fields come from solving grid-Laplacians, where the Dirichlet constraints are widths and directions sampled from the strokes.

Another way to create a hatching shape is to draw a closed polycurve, possibly snapping to strokes in the canvas as in Figure 5.4. The bounding box of the polycurve becomes the bounding box of the hatching shape. The three fields start out with uniform values: a diagonal direction for the direction field and tunable values for the width and spacing fields. The initiating curve becomes the shape's single mask curve (Figure 5.6b).

5.3.2 Mask Editing

I can draw a closed curve and remove its enclosed area from a hatching shape's visible area (Figure 5.7b). This affects only the hatching shape's mask. I use planar arrangements (provided by CGAL [97]) to derive a new collection of mask curves. *Adding* area to a hatching shape (Figure 5.7c) is more involved, since it might require expanding the hatching shape's bounding box. In that event, I recompute all three fields. They take on bilinear-interpolated original values inside the original box. Outside the original box, their values come from solving grid-Laplacians with Dirichlet constraints along the border of the original box.



(a) (b) (c)

Figure 5.7: Removing the orange shape from the hatching shape's mask (b) requires only coming up with a new mask curve using planar arrangements. *Adding* the orange shape to the mask (c) is more complicated, since it expands the hatching shape's bounding box, requiring new versions of the width, spacing, and direction fields.

5.3.3 Field Fill Operations

I offer several means of editing a hatching shape's fields, including three kinds of fill operations. A simple fill takes a fill value and an alpha in $[0, 1]$. A linear gradient or radial gradient fill operation takes start and end positions, two fill values, and two alphas. All fill operations reduce to updating each of a field's cells to an alpha-weighted mixture of its old value and a fill value. In the linear gradient operation, a cell's fill alpha and fill value are based on the nearest point on the gradient line segment. In the radial gradient operation, a cell's fill alpha and fill value are based on distance to the gradient's start position. No grid-Laplacian is involved in gradient fill operations. Figure 5.8 shows linear and radial gradient fill operations applied to the width and spacing fields of a hatching shape. Figure 5.9 shows a linear gradient fill on a hatching shape's spacing field in the context of an illustration.

I numerically specify alpha values and fill values, except for direction-field fill values, which come from the input stroke. The simple fill uses the direction from one end of the stroke to the other. The linear gradient fill interpolates between the stroke's endpoint tangents. In the radial gradient fill, fill values point either toward the stroke's start position (source/sink mode) or perpendicular to that direction (vortex mode). Figures 5.10 and 5.11 show fill operations on the direction field.

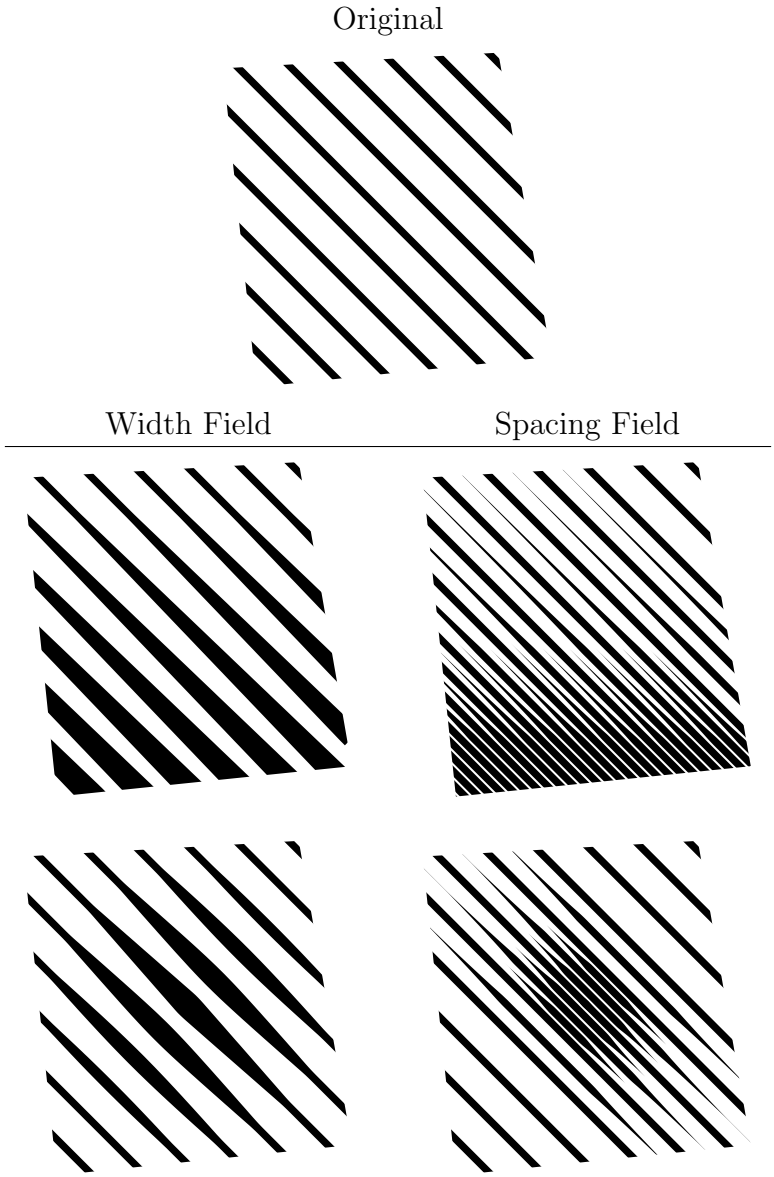
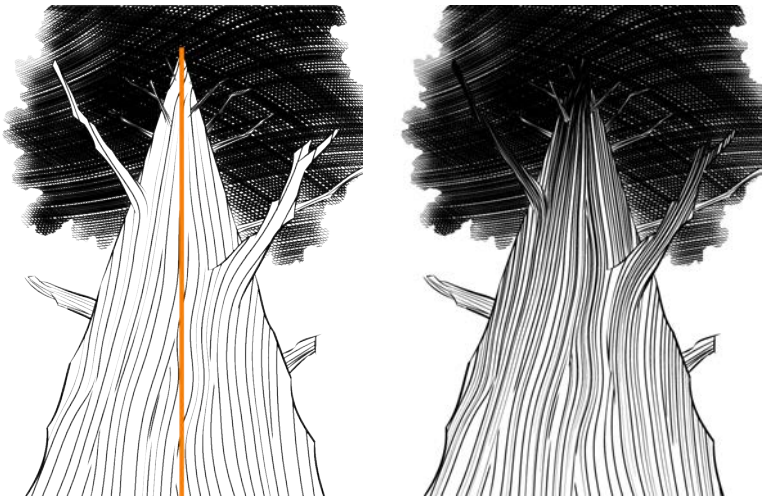
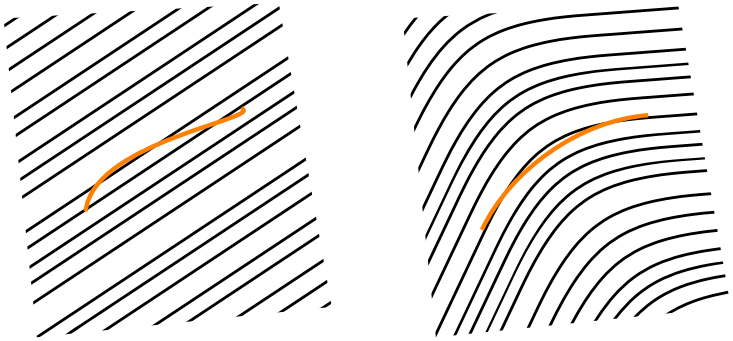


Figure 5.8: An initial hatching shape modified with linear (top row) and radial (bottom row) gradient fill operations on its width and spacing fields.



(a) (b)

Figure 5.9: Initially, the hatching shape representing the trunk has a uniform spacing field (a). A linear gradient fill operation guided by the orange stroke applies a smaller spacing value with an alpha that decreases from top to bottom. This produces the appearance of foreshortening (b).



(a) (b)

Figure 5.10: Simple fill and linear gradient fill operations on the direction field. The simple fill operation uses a stroke's endpoint-to-endpoint direction (a). The linear gradient fill operation uses a stroke's endpoint tangents (b).

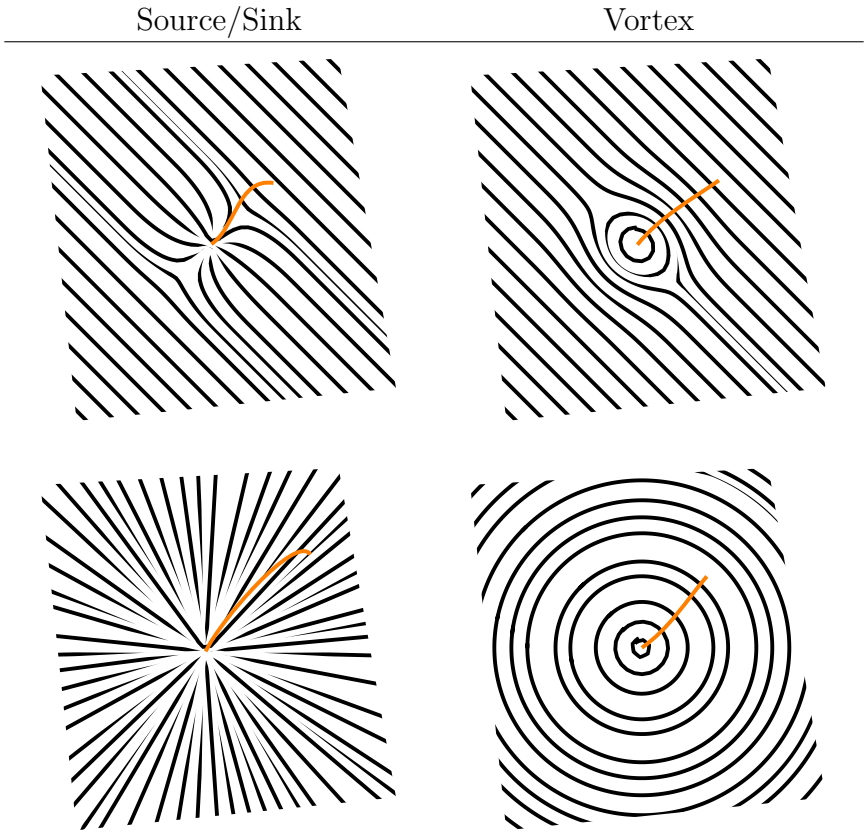
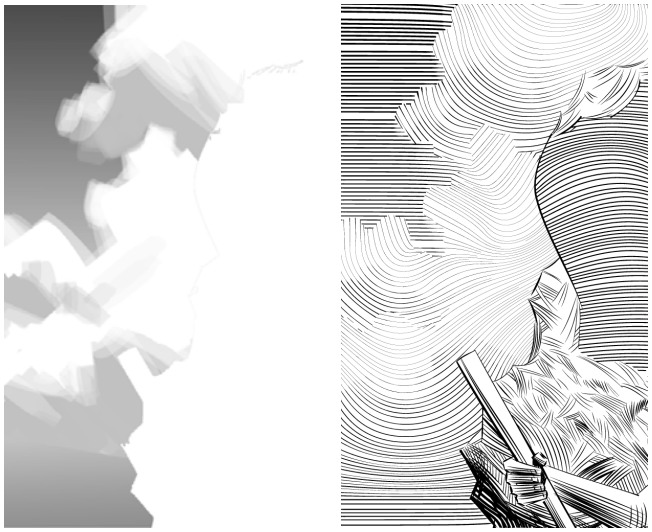


Figure 5.11: Radial gradient fills on the direction field, shown with the precipitating strokes. The first column shows source/sink mode; the second shows vortex mode. In all four cases, the first alpha value, associated with the beginning of the stroke (the center of the field), is 1. The second alpha value is 0 in the first row and 1 in the second row.



(a) (b)

Figure 5.12: A *Photoshop* painting (a) provides the width field used for a background hatching shape (b).

5.3.4 Setting the Width Field From an Image

I can set a hatching shape's width field using a grayscale reference image—normalized to be $\in [0, 1]$—along with a maximum width value. This option is a fallback for when modifying the width field manually is too tedious. Figure 5.12 presents such a case. In a widths-from-image operation, the hatching shape's resolution changes: its cell width becomes twice the size of a pixel from the reference image, assuming the reference image is mapped to the bounding box of the canvas.

5.3.5 Field Multiplication

I can multiply either the width field or the spacing field by some factor f. I can also perform a density-change operation by specifying a density factor f and multiplying both the width and spacing fields by $1/f$. Though trivial, these operations make clear the value of procedural hatching. Consider how long it would take to perform the edit shown in Figure 5.13 by hand.

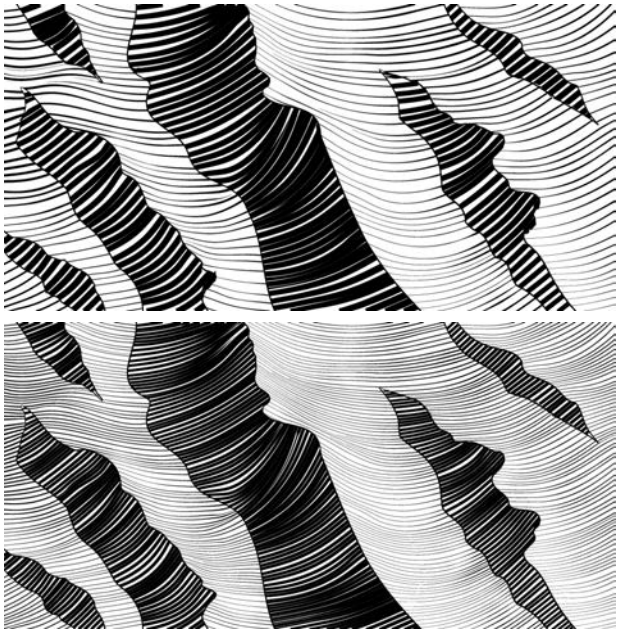


Figure 5.13: I can modify the density of a hatching shape, doubling it here.

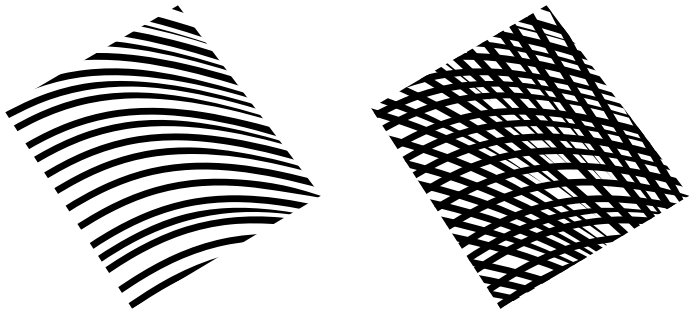
5.3.6 Crosshatching

The crosshatching operation, demonstrated in Figure 5.14 and also discernible in the foliage of Figure 5.9, takes an angular range α between 0 and 90°, plus a number n of crosshatching layers to create. Each crosshatching layer is a copy of the original hatching shape with its direction field (not its marks) rotated clockwise by $i\alpha/n$, where $i = 1 \ldots n$. The copies, once created, have no special relationship with their source hatching shape and can be edited independently.

5.4 Barrier Curves and Operations Involving Them

Having explained the simpler hatching shape operations, I now discuss two more complex ones: *blend stroke* and *multi-dir* operations. These respect a hatching shape's barrier curves, if there are any. Remember that barrier curves interrupt hatching marks.

These two operations require tracking *dead cells* (Figure 5.15) in a hatching shape's grid. I find dead cells simply by rasterizing barrier curves to the grid. Dead cells let barrier curves



(a) (b)

Figure 5.14: The crosshatching operation copies a hatching shape and rotates the direction field in the copies. Here, the crosshatching angle α is 45° and the number of copies n is 1.

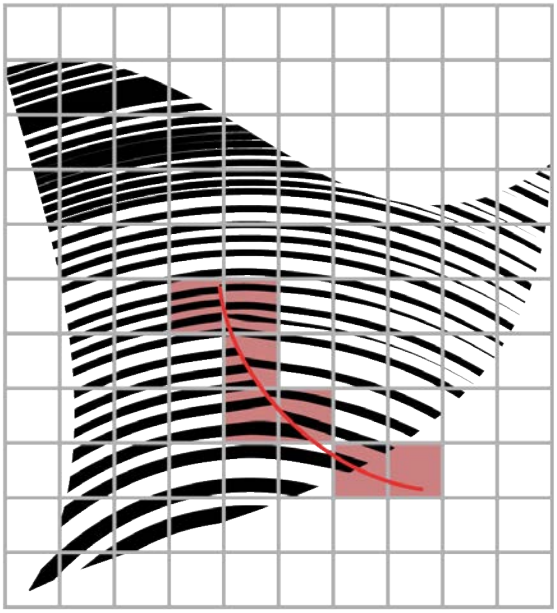


Figure 5.15: Barrier curves create *dead cells* (red) in a hatching shape's grid. Widths, spacings, and directions from dead cells are considered unreliable for use in streamline advection.

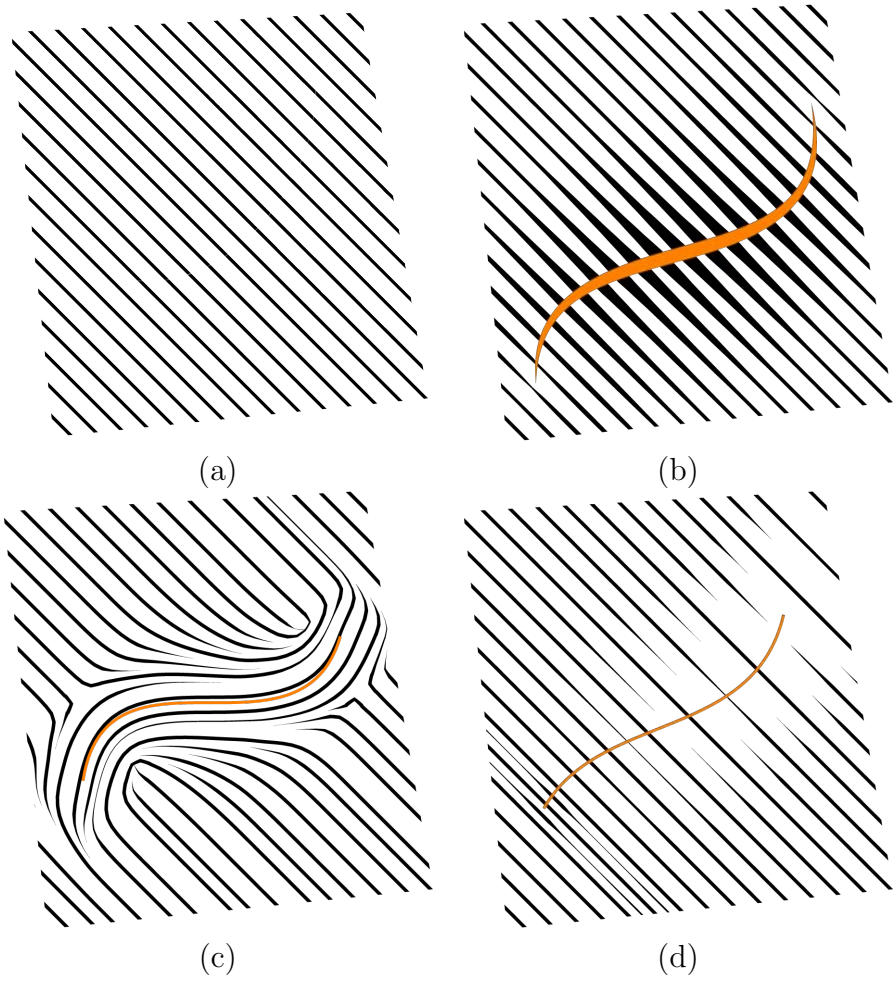


Figure 5.16: *Blend strokes* applied to the width (b), direction (c), and spacing (d) fields of an initial hatching shape (a).

exert an effect in this section's two operations. They also influence streamline advection (Section 5.5).

5.4.1 Blend Strokes

A blend stroke operation lets me locally and smoothly modify one of a hatching shape's fields with a stroke. Effects propagate from the stroke up to a tunable radius (Figure 5.16). Achieving this means updating each of the field's values using a fill value and an alpha value, like in the fill operations described earlier. Here, however, fill values and alpha values are more complex to calculate.

There are three steps to a blend stroke operation. First, I compute a distance transform of the stroke at the same resolution as the hatching shape's three fields. When there are no barrier curves, I use a simple two-pass distance transform [98]. Otherwise I use Dijkstra's algorithm [19] modified to ignore dead cells. This means that for every cell position, I find the length of the shortest path to the stroke, where paths are a mixture of horizontal and vertical steps that do not visit dead cells. (Diagonal steps are not allowed; these are four-neighbor, not eight-neighbor paths.) The result is that barrier curves conveniently block a blend stroke's effects, as shown in Figure 5.17.

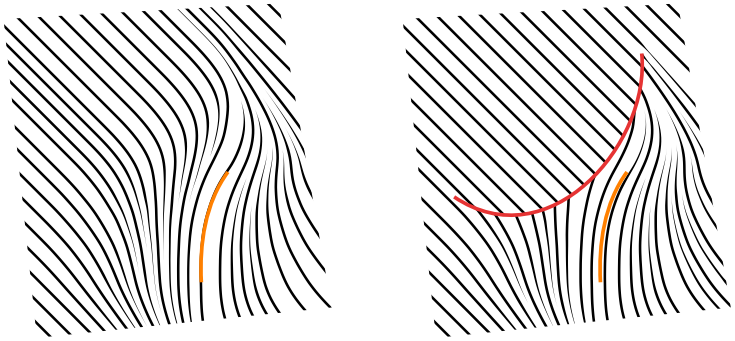
Second, I compute an alternate version of the field using a grid-Laplacian with Dirichlet constraints along the stroke. When editing the direction field or width field, I densely sample the stroke's tangents or widths for Dirichlet values. In a spacing field edit, I interpolate between two tunable spacing values along the stroke.

Third, I blend this alternate version of the field with the current field. Each cell's alpha value comes from the distance map, the effect radius, and a tunable master alpha value.

Dead cells are unchanged by blend stroke operations, which means their values are liable to stop agreeing with their neighbors. To be safe, then, I do not use dead cells' values when interpolating data from the three fields (Section 5.5.3).

5.4.2 Multi-Dir

The multi-dir operation is an expansion of the blend stroke operation, meant just for the direction field. Bending hatching marks to show the curvature of an imagined 3D surface is a crucial part of hatching, and I find it too difficult to accomplish using individual blend strokes. In multi-dir, I do away with the effect radius and do not alpha blend with the old direction field. Instead I produce a completely new direction field that interpolates



(a) Without barrier (b) With barrier

Figure 5.17: Blend strokes' effects stop at barriers.

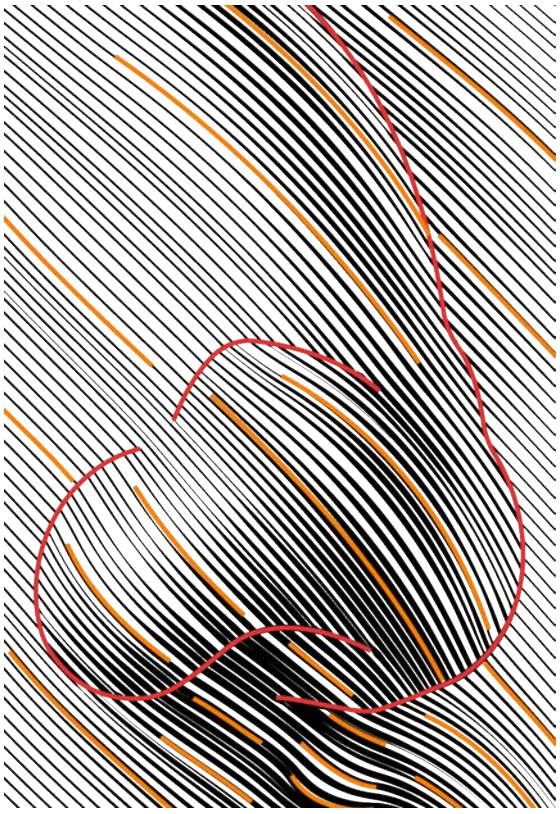


Figure 5.18: The multi-dir operation fills the direction field using the barrier curves (red) and multiple guide curves (orange).

the densely sampled directions of one or more guide curves (Figure 5.18). The input in multi-dir evokes the "directional guides" of "FLOWPAK" [85], the "curvature lines" of "BendFields" [44], the "cross sections" of "CrossShade" [90], and the "hatching strokes" of Bui et al. [11].

Multi-dir requires another grid-Laplacian, with Dirichlet values set to the guide curves' tangents. There is a critical difference between this grid-Laplacian and those discussed earlier. I exclude dead cells from this linear system, which lets barrier curves take effect: cells on opposite sides of a barrier are no longer as closely connected, if at all. Excluding dead cells from the system is simple: if a cell has a dead cell for a neighbor, I remove the dead neighbor's contribution on both sides of Equation 5.2. (A similar technique creates discontinuities in the color gradients generated by *diffusion curves* [24, 9].)

(I do *not* exclude dead cells from the grid-Laplacian used for a blend stroke operation. Blend strokes do respect barrier curves, but by way of the distance map used for blending. In a blend stroke operation, the grid-Laplacian only needs to produce a smooth field that interpolates sample values from the input stroke. I could use multi-dir's more advanced, barrier-respecting grid-Laplacian for this interpolation, but have not seen the need.)

In the dead cell-excluding grid-Laplacian, the four-neighbor versus eight-neighbor distinction is just as critical as it is in the Dijkstra distance transform used in the blend stroke operation. If I used eight-neighbor paths in my distance transforms or an eight-neighbor grid-Laplacian for multi-dir, the rasterized barriers would not be effective. Rather, they would not be effective unless I used a "thick" rasterization algorithm that marked *every* cell touched by a barrier curve as dead. I use "thin" rasterization, keeping as many cells as possible "live" and thereby throwing away no more data than necessary. Section 5.5.4 discusses rasterization further.

Removing dead cells from the grid-Laplacian sometimes means that a small pocket of cells is unreachable from any of the Dirichlet constraints (Figure 5.19). These pockets can make the linear solver fail. Thus, the system includes only live cells reachable from Dirichlet constraints via four-neighbor paths that do not include dead cells.

5.5 Streamline Advection

In this section I explain how to produce hatching marks from a hatching shape using an adapted version of Jobard and Lefer's streamline placement algorithm [46].

The original algorithm fills a rectangle with streamlines one at a time, popping streamline seed locations from a queue. If a candidate location is too close to already placed

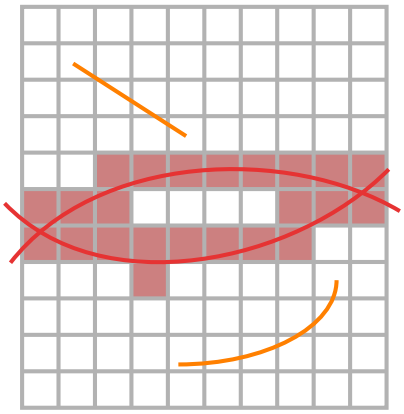


Figure 5.19: In the multi-dir operation, the grid-Laplacian leaves out dead cells. There might be pockets of live cells that are unreachable from Dirichlet constraints (orange). The system leaves these out as well to prevent the solver from failing.

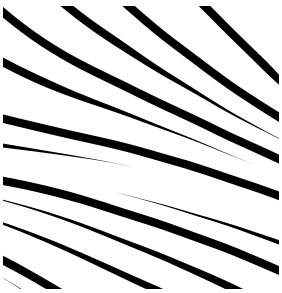


Figure 5.20: Streamlines taper as they approach each other.

streamlines, discard it; otherwise, integrate through a vector field (a hatching shape's direction field, in my case) and terminate the streamline upon going out of bounds or coming within stopping distance of an older streamline.

There are many low-level implementation choices. Here are some of the simpler details:

- Streamline density is variable [86, 14, 60, 91, 67]. The separation distance between streamlines, called d_{sep} by Jobard and Lefer, comes from a hatching shape's spacing field.
- Like Jobard and Lefer, I use midpoint integration.
- For a given hatching shape, the integration step size is a constant: the smallest d_{sep} found in the spacing field.

- Streamlines taper when they come close together, again as per Jobard and Lefer. The local value d_{test} is the minimum spacing allowed between two streamlines: the "stopping distance" mentioned above. A streamline in progress will taper to nothing and terminate if its distance to a previous streamline falls to this value. I set d_{test} to half of the local spacing value ($d_{\text{sep}}/2$). See Figure 5.20.

- A streamline seed must be at least d_{sep} (not d_{test}) away from previous streamlines in order not to be discarded.

- Before placing any hatching marks, I fill the seed queue using the approach of "Suggestive Hatching" [91]. I take seeds from the vertices of a coarse-to-fine series of grids, the finest grid having a cell size equal to the step size. As "Suggestive Hatching" explains, this is a cheap way to get the benefits of farthest-point seeding [68]. A second seed queue handles gaps caused by barriers (Section 5.5.5).

- I use a minimum length test: if, after advection in both directions, a streamline contains fewer than five points, I discard it.

There are several advanced streamline placement features that my implementation lacks, such as the adaptive step size and loop prevention of "ADVESS" [62] and the critical-point handling of Verma et al. [100].

With these lighter details covered, I move to the more involved parts of the streamlines algorithm. Most of the complexity comes from the need to respect barrier curves.

5.5.1 The Dead Grid and the Collider Grid

I use auxiliary 2D arrays to control streamline placement. One is the *dead grid*, a boolean image to which I rasterize barrier curves. I have already alluded to checking the dead-or-alive status of cells; that is the dead grid's role. The *collider grid* cheaply determines whether a given line segment intersects any barrier curve. Each cell in the collider grid stores line segments to use in intersection tests. Both grids have the same resolution as a hatching shape's three fields.

To fill these two grids, I break each barrier curve into line segments. For each segment $\overline{AB}$, I find the integral coordinates it rasterizes to (Section 5.5.4) and mark these in the dead grid. I also add $\overline{AB}$ to the bin at each of the same positions in the collider grid. Finally, I dilate the information in the collider grid with a 3×3 kernel (Figure 5.21).

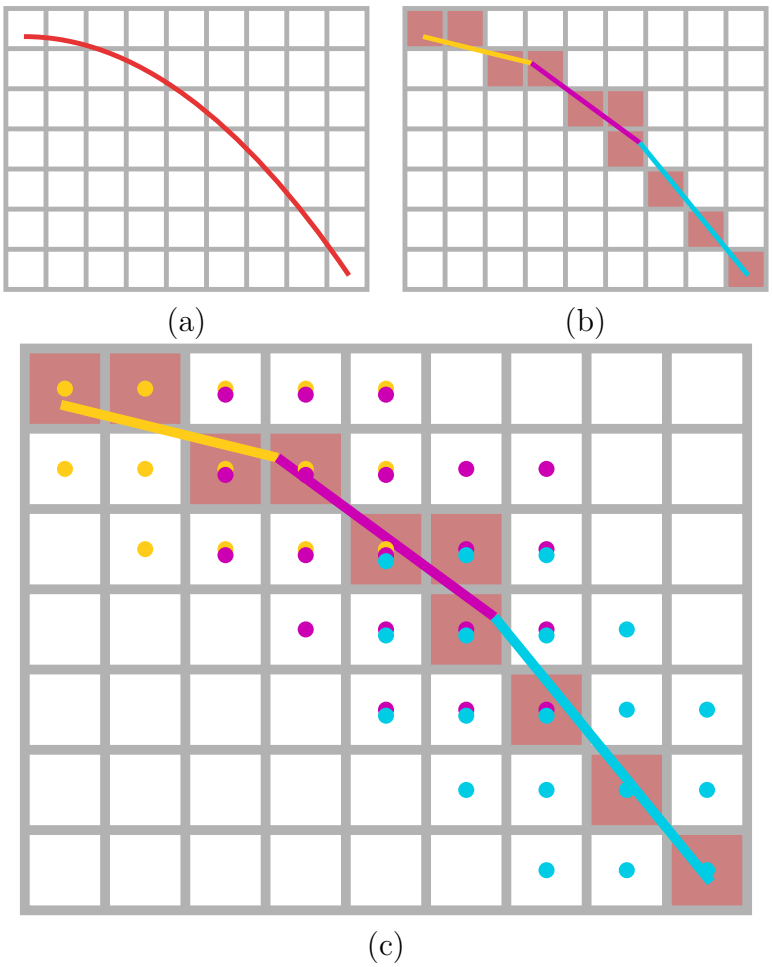


Figure 5.21: Given a barrier curve (a), I split it into line segments. The boolean *dead grid* rasterizes these line segments (b). The *collider grid* (c) stores copies of barrier line segments. For example, a cell with a yellow dot and a purple dot contains a copy of the yellow line segment and a copy of the purple line segment.

This dilation is necessary because of the thin rasterization I use. Without it, I would miss diagonal intersections.

I can find the first intersection of any directed line segment $\overline{CD}$ with any barrier curve by finding the rasterized coordinates of $\overline{CD}$. At each I check for intersection between $\overline{CD}$ and the segments stored in the collider grid there. I return the intersection closest to C, if any.

5.5.2 Occupancy Grid

The *occupancy grid* is a third auxiliary array, meant for finding the distance to the nearest streamline at a query position. Mine resembles the occupancy grid of "ADVESS" [62]: each cell stores points visited by completed streamlines (not the points from the streamline in progress). The cell size for this grid is independent of the cell sizes of other grids. This permits a trade-off between more memory usage and more distance calculations. I set the occupancy grid's cell size to the minimum value found in a hatching shape's spacing field, clamped so the occupancy grid's minimum dimension is no smaller than 15 and its maximum dimension is no larger than 500.

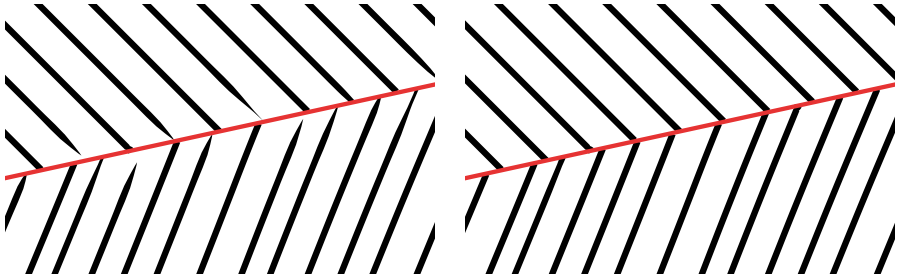
There are two use cases for the occupancy grid: checking a streamline in progress and checking a streamline seed.

Streamline in Progress

Every time a streamline takes a step, I need to check how close it is to previous streamlines. If the nearest streamline is d_{test} away—recall that d_{test} is half of d_{sep}—then the current streamline tapers to nothing and takes no more steps in the current direction.

The occupancy grid does not need to return the true nearest distance to an already placed streamline at the current location. It just needs to return the true distance clamped to the range $[d_{\text{test}}, d_{\text{sep}}]$.

This yields the routine DistToNearestStreamline, which takes a query position p along with lower and upper bounds on the distance to return: $d_{\min}$ and $d_{\max}$. First, I look at the occupancy grid cell containing p and check if it has any stored points, then the 3×3 ring of cells around that cell, then the 5×5 ring of cells around that cell. I proceed until completing the $k \times k$ ring, where $k = \min(2l_{\max} + 1, 2\lceil d_{\max}/w_{\text{occ}} \rceil + 1)$, $l_{\max}$ is the maximum dimension of the occupancy grid, and w_{occ} is the width of one of the occupancy grid's cells. Any points stored outside of the $k \times k$ ring will be at least $d_{\max}$ away from



(a) (b)

Figure 5.22: Failing to account for barriers in occupancy-grid tests causes unnecessary tapering (a). When checking the distance to previously placed hatching marks from a query position, I ignore marks on the other side of barriers (b).

p, so there is no need to check any more cells afterward. If I find a stored point less than or equal to $d_{\min}$ away from p, I immediately return $d_{\min}$. Otherwise I return the minimum distance found, or $d_{\max}$ if I encounter no stored positions.

Barrier curves present a complication. The occupancy information from one side of a barrier needs to be imperceptible from the other side, otherwise a streamline will taper unnecessarily as it approaches a barrier and detects previously placed streamlines on the other side (Figure 5.22). I use the collider grid to test whether the line segment from the query position to each stored position intersects any barrier curve, and ignore stored positions blocked by barriers.

If the next integration step for a streamline has it run into a barrier, as indicated by the collider grid, then the streamline terminates 5% short of reaching the intersection. The padding ensures that the last stored position for the streamline is on the correct side of the barrier. Without it, subsequent streamlines on the other side might spuriously detect the just completed streamline's influence in the occupancy grid.

Streamline Seed

The second use case for the occupancy grid is when creating a new streamline, given a seed position. If the seed is less than d_{sep} from an existing streamline, I discard it. Here I use the occupancy grid routine AtLeastThisFarAway, which takes a query position p and a minimum distance $d_{\min}$, returning a boolean. This routine is simply the boolean expression DistToNearestStreamline$(p, (1-\epsilon)d_{\min}, d_{\min}) \geq d_{\min}$. The constant $\epsilon = 0.001$ is necessary to keep the expression from always evaluating to true. This constant actually

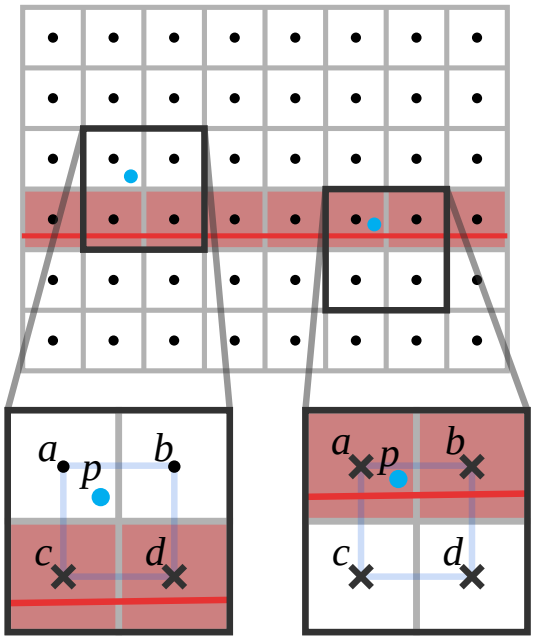


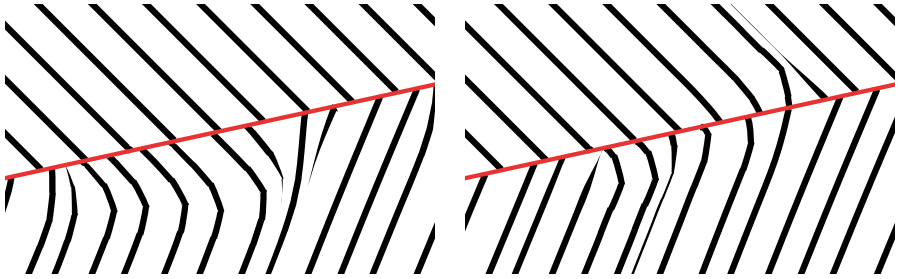
Figure 5.23: In the left case, I exclude cells c and d from interpolation because they are dead. In the right case, I exclude a and b for the same reason. There I also exclude c and d since their centers are separated from p by a segment of a barrier curve.

features twice: when I call AtLeastThisFarAway, I set d_{min} to $(1-\epsilon)d_{\text{sep}}$ instead of just d_{sep}, as floating point error can otherwise lead perfectly placed seeds to be discarded for being detected as just barely too close to existing streamlines.

5.5.3 Interpolating Field Data

The width, direction, and spacing values for a query position inside a hatching shape come from interpolating the values surrounding that position. Barrier curves pose a couple of problems: I must avoid sampling dead cells' values and I must not sample values from the wrong side of a barrier.

By default, I use bilinear interpolation between four cells' values. To prevent interpolation problems caused by barriers, I may disqualify some of these four cells. Two tests can disqualify a cell. First, I ignore dead cells, since their values are unreliable, as explained in Section 5.4. Second, I ignore cells whose centers are blocked from the query point by barriers. Figure 5.23 shows both of these tests. When using the collider grid to answer the



(a) (b)

Figure 5.24: At left, I allow dead cells to participate in interpolation. Some of the streamlines at bottom veer undesirably as they enter the dead cells' zone of influence and start incorporating out-of-date direction values. At right, I allow cells on the wrong side of barrier curves to participate. Some of the streamlines below the barrier receive influence from direction values above the barrier, and one streamline from above the barrier receives influence from below it.

necessary intersection queries, I err on the false-positive side. Remember that the collider grid returns the first collision between a directed line segment $\overline{CD}$ and barriers. In this case I start with C equal to the query point and D equal to the center of the cell of interest. To avoid missing any intersections, I extend the length of $\overline{CD}$ by a factor of 1.001 in both directions before checking it against the collider grid.

Figure 5.24a shows the consequence of leaving out the first test and letting dead cells participate in the interpolation of streamline directions: old data pollute the pattern. Figure 5.24b shows the result of leaving out the second test: "leaks" result when streamlines approach a barrier and start sampling data from the other side.

If only three of the surrounding cells pass the tests, I use distance-weighted averaging to interpolate their values. In two-cell cases where the two passing cells have the same X or Y coordinate, I linearly interpolate using the other coordinate. In diagonal two-cell cases, I use distance-weighted averaging.

5.5.4 Rasterization Issues

I took care in choosing how to rasterize barriers to the dead grid. A subtle problem frequently arises with integer-based algorithms. Situations occur like that shown in Figure 5.25. Here, a barrier segment is so far from its rasterized cells that there are areas of live space below the true barrier segment yet above the rasterized version. In this situation,

the collider test for excluding interpolation values is unreliable, resulting in leaks. I use a floating-point DDA algorithm [103] that steps a distance of one cell width until capping off with a step less than or equal to one cell width. This algorithm keeps barrier segments closer to their rasterized versions, though to my knowledge it does not guarantee that leaks will never occur.

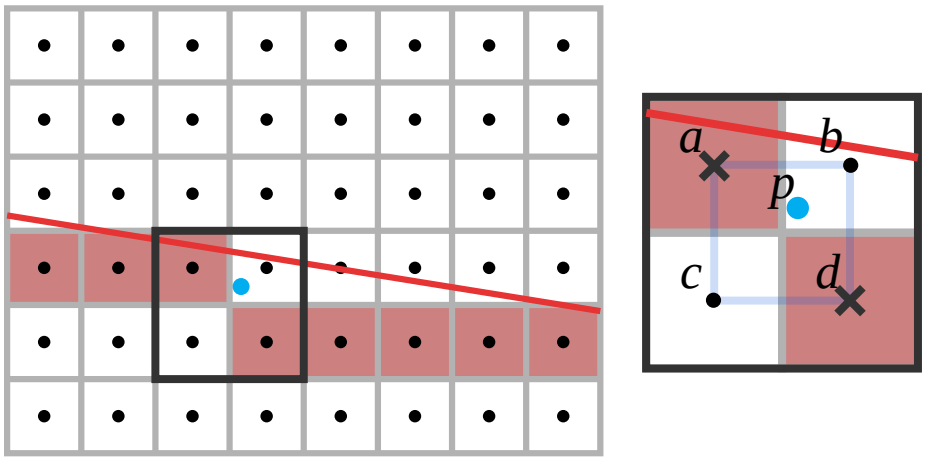
5.5.5 When Interpolation Fails

Sometimes a query point is surrounded by four disqualified cells, meaning that width, spacing, and direction are indeterminate there. If a streamline enters indeterminate space, it keeps advecting using its last known data until it terminates or reenters determinate space.

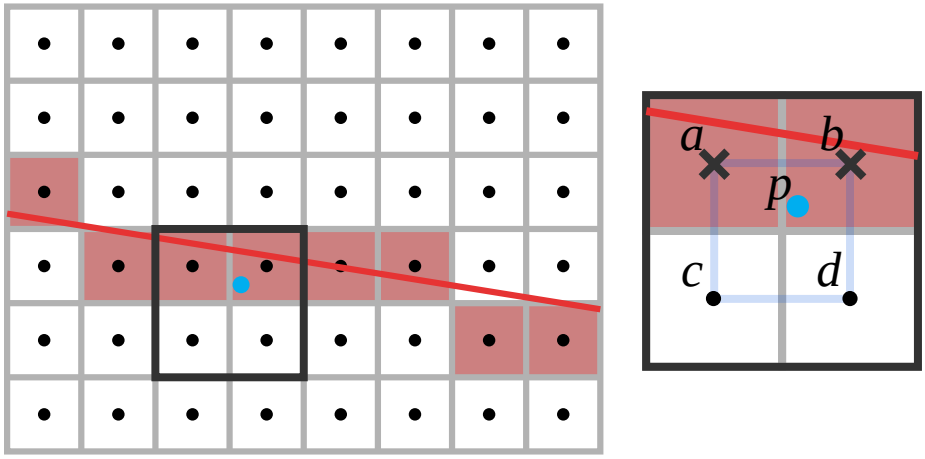
When I pop a seed from the primary queue and find that it falls in indeterminate space, I discard it. If enough seeds fail to sprout, gaps appear in the hatching. To fill these, I introduce a secondary seed queue, which I do not use until the primary queue is empty. While advecting a streamline, at each step I look d_{sep} to its left and right, creating two candidate positions p_l and p_r. If a p falls within a dead cell and there is no barrier curve between p and the streamline's current position, I tentatively create a secondary seed at p. (I do not enqueue the secondary seeds produced by a streamline until that streamline finishes and passes my minimum length test. Otherwise an infinite loop can result from zero-length streamlines reseeding each other.) Secondary seeds, unlike primary seeds, store initial spacing, width, and direction data, sourced from parent streamlines. When I pop a seed from the secondary queue, the resulting streamline starts out using these stored data. Figure 5.26 shows how secondary seeds fill some gaps, but not areas of indeterminate space completely walled off by barriers.

Hatching inevitably degrades near barrier curves. This degradation owes not just to indeterminate space, but more generally to the loss of interpolation data as query points approach barriers. Reducing artifacts caused by barrier curves is one of the main reasons to increase the resolution of a hatching shape (Figure 5.27).

Readers interested in implementing hatching shapes should know that there are more sophisticated data structures and mechanisms for solving grid-Laplacians in 2D than simple rectangular grids. Some of these better methods are likely better equipped for dealing with barrier curves. For instance, it is possible to solve grid-Laplacians on a triangular mesh [99], perhaps in this case an adaptive triangulation over a hatching shape's bounding box that updates with the addition of barrier curves.

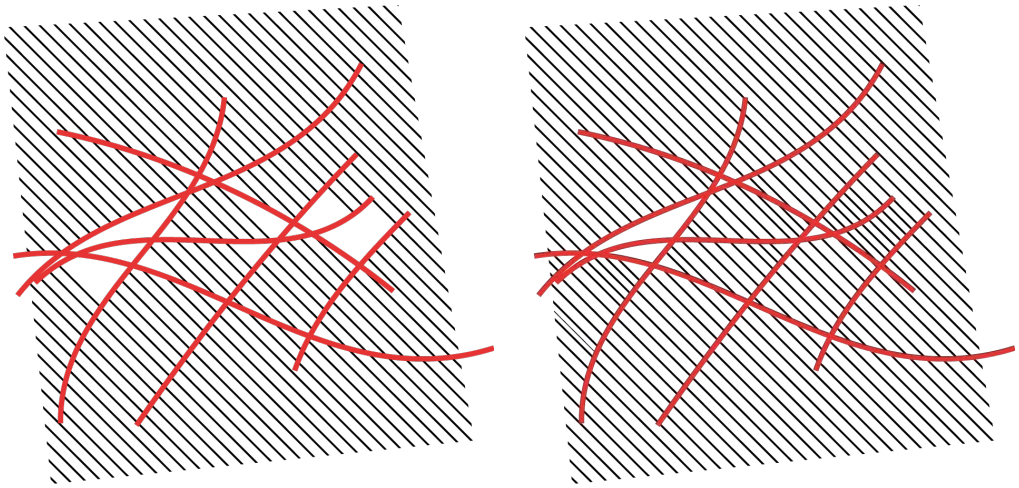


(a) Integer-based midpoint [77]



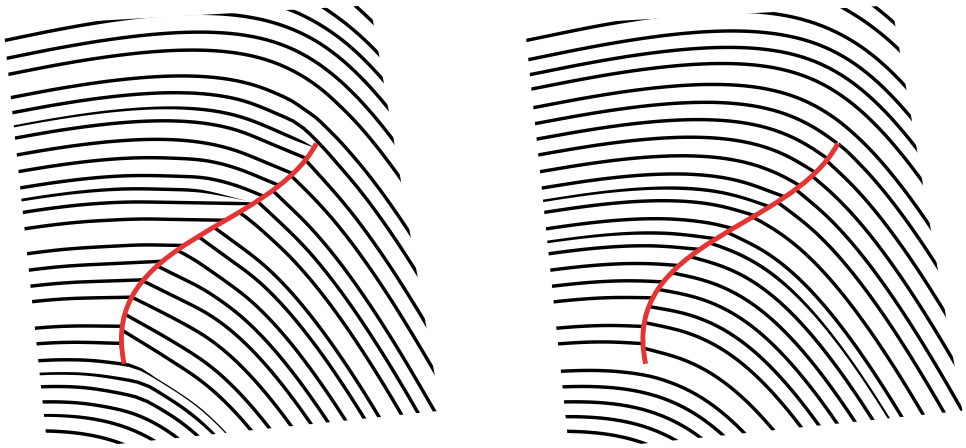
(b) Floating point DDA [103]

Figure 5.25: If I mark dead cells using integer-based rasterization, I often cannot rely on my tests to prevent the wrong cells from participating in interpolation. The first scenario is an example (a). Cells b and c qualify to participate. Neither is dead nor separated from the query point by the barrier. But these cells are on opposite sides of the *rasterized* barrier, so using them both at once means leaking data across. Floating-point DDA [103] rasterization (b) keeps the rasterized barrier curve closer to the actual barrier curve.



(a) (b)

Figure 5.26: Normal streamlines cannot sprout in indeterminate space, which causes gaps where multiple barrier curves come close together (a). Secondary seeds, which can sprout in indeterminate space, fill some of these gaps (b).



(a) (b)

Figure 5.27: The fields of the first hatching shape (a) are 8×9. There are artifacts caused by streamlines losing interpolation data as they approach the barrier curve. Increasing the resolution to 20×21 reduces these artifacts.

	Breeze	*Twee*	*Houppelande*
Hatching Shapes	8	4	2
Hatching Marks	3425	3669	1294
Hand-Drawn Marks	5462	3450	3883
Total Draw Time (s)	3.85893	1.77037	0.35684
Hatching-Only Draw Time (s)	3.8353	1.76357	0.328343

Table 5.1: Data for *Breeze* and *Twee* (Figure 5.28), and for *Houppelande* (Figure 5.29). Each time figure is an average of ten runs.

5.6 Results

I have tested hatching shapes' viability in full illustrations like *Breeze* and *Twee* (Figure 5.28). Not all marks in these pictures are hatching, and not all hatching comes from hatching shapes (Figure 5.30).

In my exploration I have found a couple of criteria for deciding if a hatching task is suited for hatching shapes. First, the best subjects for hatching shapes are forms that read visually as smooth surfaces, like the cliffs in *Breeze* and the concrete walls in *Twee*. Good subjects may have many discontinuities, as long as there are smooth areas visible between. Figure 5.31 shows an extreme example: the jagged interior of the egg.

Second, individual hatching marks should have low pictorial importance individually. In *Houppelande* (Figure 5.29), hatching shapes only feature in the dark drapery and the egg; other hatching is completely manual. The woman's face is not ideal for a hatching shape, despite being a smooth surface, because individual marks have too much visual weight. In the face, marks' widths and spacings are large relative to the size of crucial facial features, which makes it possible for the placement of individual marks to ruin the drawing. A hatching shape *could* produce the hatching here, but because individual marks are so visually powerful, this would require a lot of careful editing, enough to justify hatching manually instead. The same goes for the hatching on the turban (which needs to jibe with the hatching on the face), plus the hatching on the arms and hands.

Hatching produced by hatching shapes has a distinct style. It is often so regular that it draws the eye more than hatching normally does. This can be pleasing or painful to the eye, and may harm a composition. In *Breeze*, the marks on the cliffs are primarily from hatching shapes, but there are additional hand-drawn marks to break up the uniformity. Small manual crosswise strokes accomplish the same task in Figure 1.7. Future work with hatching shapes could impart such naturalism automatically by mimicking traditional hatching,

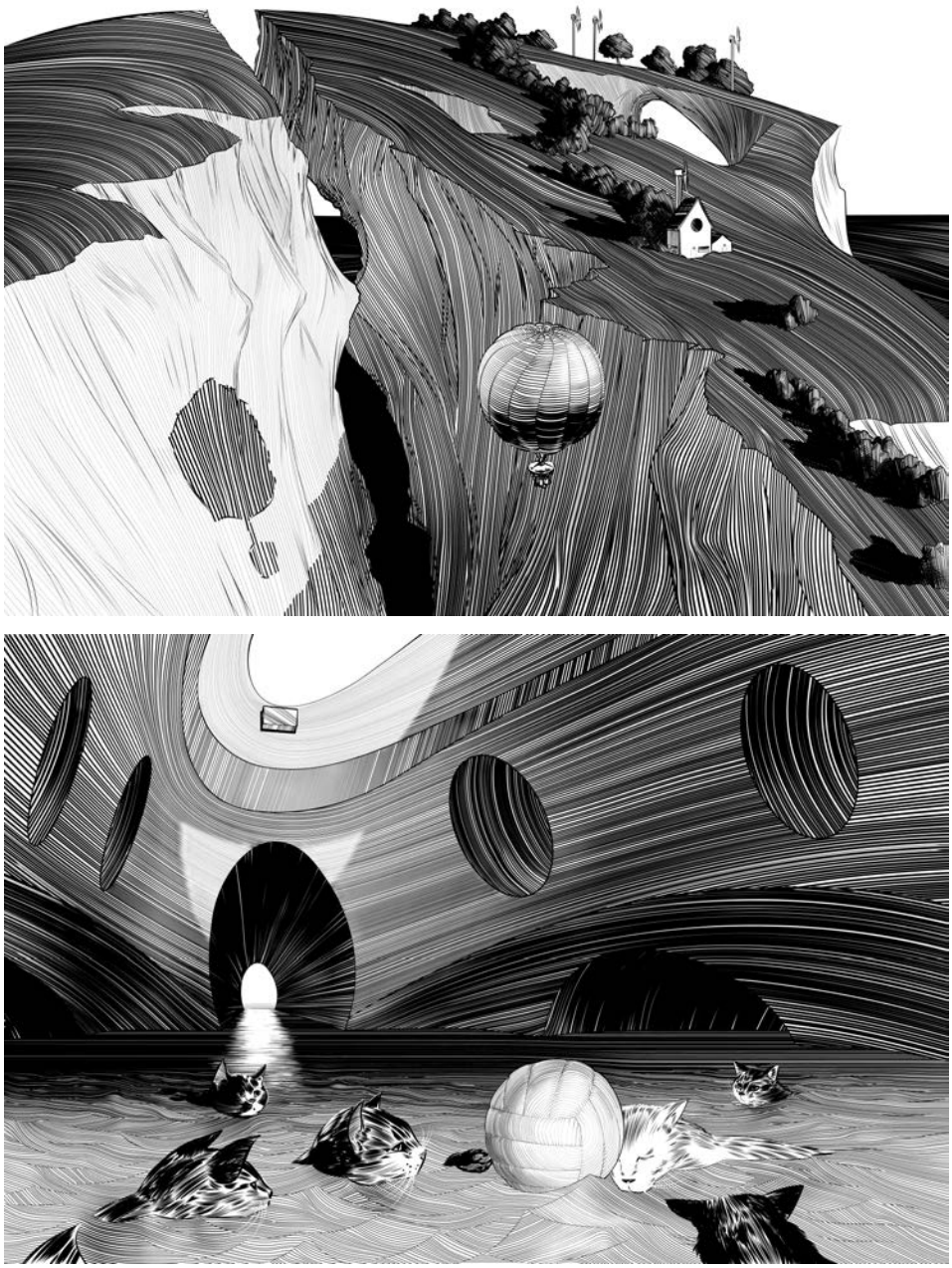


Figure 5.28: *Breeze* (top) and *Twee* (bottom) rely heavily on hatching shapes.



Figure 5.29: *Houppelande* contains two hatching shapes, which produce the hatching of the dark fabric and of the egg's outer surface. Everything else was more practical to accomplish with individual hand-drawn strokes.



Figure 5.30: *Breeze* contains procedurally generated marks from hatching shapes (top) and individually hand-drawn marks (bottom).

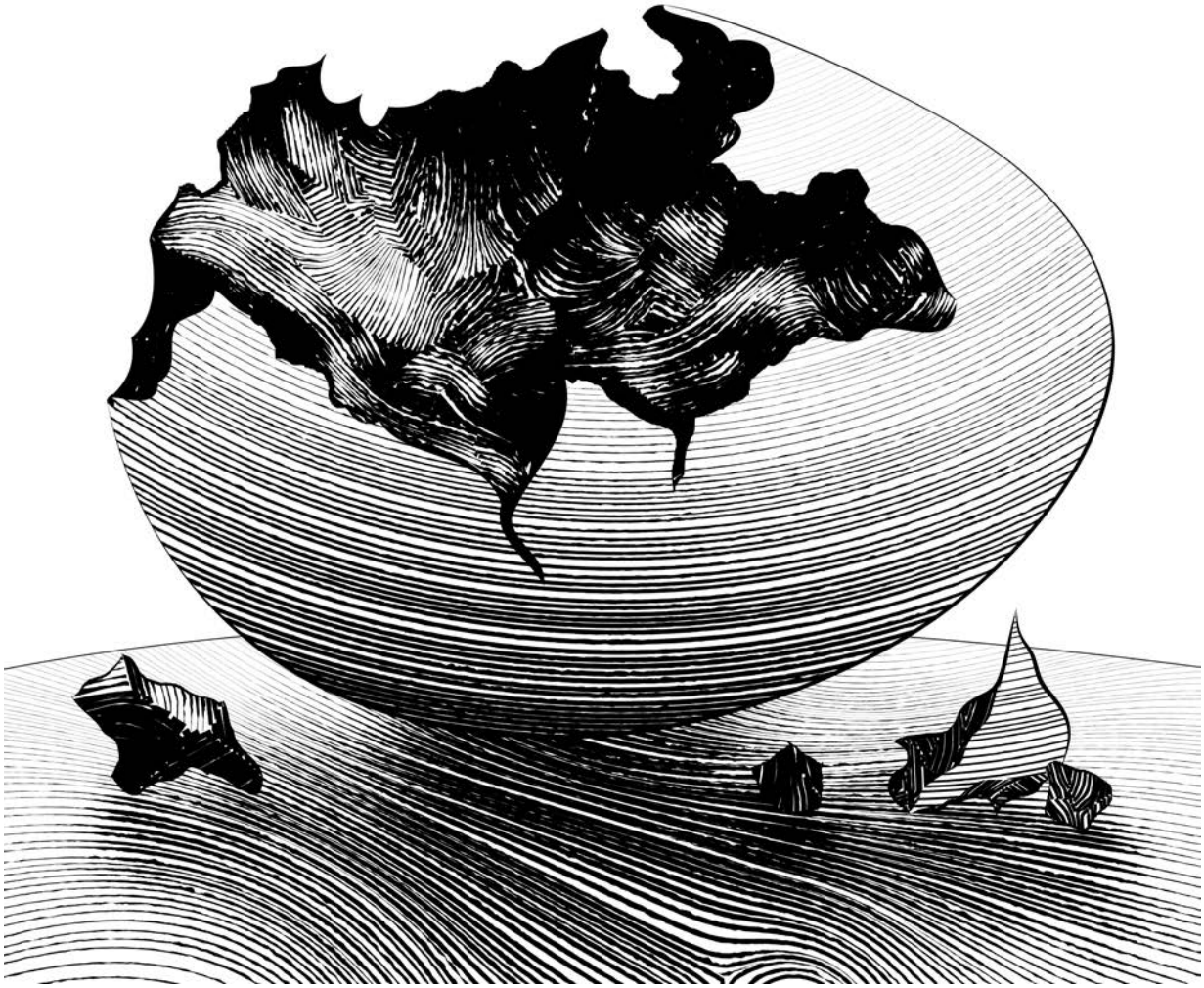


Figure 5.31: Hatching shapes are ideal for the egg because it is easy to visually separate into smooth surfaces, plus individual hatching marks have little pictorial importance.

whether at the level of individual marks [2] or at the level of a group of hatching marks [47, 26], i.e., at the level of a hatching shape.

Table 5.1 gives render times for *Breeze*, *Twee*, and *Houppelande*. It lists the time required for my interface to completely redraw the canvas, plus the time required just to redraw all hatching shapes. Drawing times for hatching shapes include the streamline advection process.

5.6.1 Artist Interviews

I conducted semi-structured interviews with three professional artists to help assess hatching shapes' value. In each interview, we explained what hatching shapes are, showed some operations, and looked at example illustrations. We solicited reactions and asked about prior experience with hatching. Responses to hatching shapes were encouraging.

Artist 1 had over forty years of hatching-oriented professional experience. His preferred medium is scratchboard. He could be considered a master of hatching, experienced enough not to benefit much from hatching shapes, though he did muse that his lines might be more "consistent" if he had hatching shapes at his disposal. When we showed him my illustrations, most of his criticisms were about my draftsmanship, not the hatching shape primitive. His reaction to the primitive itself was encouraging: "You are on your way to creating extraordinary effects with your hatching."

Artist 2 had about 13 years of professional experience, primarily digital. His work has a clean, op art style. He often creates hatching with Adobe *Illustrator* "blends"—interpolations between two curves, like the Coons patch-based hatching of "Digital Facial Engraving" [75]. The affordance these blends give him is limited. He said that if he could use hatching shapes, he would feel more inclined toward intricate, organic hatching.

Artist 3 had a similar reaction. He also carried 13 years of professional art experience, also largely digital, though little if any involving hatching. He admires hatching, but feels intimidated by its difficulty, saying hatching seems like a "magic trick" when done by experts. Early in the interview, as we showed operations on hatching shapes, he called what he saw "phenomenal," and later said he would do much more hatching if hatching shapes were on the table. However, he disliked the untapered ends of hatching marks that terminate against barrier curves. Such ugly mark-ends are discernible in Figures 5.1 and 5.2. Normally I mask this effect by placing a stroke on top of each barrier curve.

These interviews, plus the pictures I have made, suggest that the hatching shape and its operations have value. The pictures almost communicate this on their own, but remember

that much of the hatching shape's utility is the ease of repeated editing. That, if nothing else, is the hatching shape's reason for being.

5.7 Future Work

Hatching shapes could grow to cover a broader range of hatching tasks. Some potential enhancements are simple. For example, currently a barrier curve applies to all three hatching fields at once, but there are situations where it would be better for a barrier to apply only to one or two of the fields, such as in the dunes of Figure 5.13. Here, streamlines stop at the barriers representing shadows, which makes the forms harder to read. The hatching would be more legible if those barriers applied only to the width field, so that streamlines did not terminate at them but abruptly changed their widths instead.

Other advancements could be much more involved, entailing the addition of completely new data fields, a more sophisticated mask definition, or new mechanisms for seeding and advecting streamlines.

5.7.1 Tonal Controls

The previous 2D hatching solutions that inspired hatching shapes tend to make tone an explicit part of their interfaces and algorithms. In practice this means that hatching patterns must approximate the tones of reference images. With hatching shapes, however, I operate below the level of tone, instead directly accessing hatching marks' widths and spacing, which are the two components of tone. I take this approach partly because I am not specifically concerned with matching reference images, but more importantly because I wanted to keep control as low-level as possible. However, it would be useful to wrap hatching shapes with higher-level tone-based controls. These controls ought to be tunable, since it is possible to achieve a desired tonal pattern by adjusting a hatching shape's width field, its spacing field, or both at once.

5.7.2 Spacing Artifacts

Hatching shapes sometimes produce spacing artifacts, as shown earlier in the left half of Figure 5.6. Figure 5.32 shows two more cases (first two rows). These artifacts stem from my streamline seeding strategy. As a reminder, I enqueue seed positions on a coarse-to-fine series of grids. This process is mostly oblivious to the ideal spacing between marks.

Consider the first case (topleft) in Figure 5.32, where the direction and spacing fields are constant. Streamlines initially grow from far-apart seed positions taken from a very coarse grid, then from a finer but still coarse grid, and so on. At some point, two streamlines are placed such that the vertical distance between them is not an integer multiple of d_{sep}, which leads to the artifacts. (Sometimes a constant-spacing, constant-direction hatching shape does not manifest this problem—see the right half of Figure 5.6.) In the second case (middle row, left), the spacing field varies linearly from top to bottom but the resulting pattern features banding in inter-mark spacings. The bands correspond to the different grids used for enqueuing seed positions.

As a stopgap, I have implemented an optional, line-based seeding mode. I fill the primary seed queue by starting at the center of the shape and placing seeds while walking perpendicular to the direction field, using the spacing field for each step distance. In case of gaps, I append all the seeds produced by the default method. Line-based seeding works well if the direction field is constant and the spacing field only varies perpendicular to the direction field. However, it is too crude to handle more complex hatching shapes in a visually uniform way. The bottom-right square of Figure 5.32 shows a failure case. Seeds from the line-based method produce the hatching in the bottom left of the square, while seeds from the default method fill in the top right, which would otherwise be blank. An unwanted false contour separates the two regions.

Moving beyond stopgaps, there may be multiple valid ways to resolve spacing artifacts. One might use a more intelligent seeding strategy that accounts for the structure of both the spacing and the direction fields. For instance, seed placement could be based not on a rectangular grid, but on a conforming triangulation built from barrier curves (taking inspiration from "Painting With Triangles" [8]). Or one might fix spacing irregularities in a post-processing step. I write about an attempt at post-facto spacing improvement in Chapter 7.

5.7.3 Messy Hatching

Hatching shapes favor *clean* hatching (Figure 5.33a), where marks are as long as possible and a single mark has clear left and right neighbors. In *messy* hatching (Figure 5.33b), marks are more chaotically placed and choppy. Subjects like fur or dense vegetation are not separable into smooth surfaces divided by barrier curves. For now, hatching shapes are insufficient for such hatching. They could not easily reproduce the output from "Orientable Textures" [82], an example of which appears in Figure 5.34. Enabling that would require augmentations, like a choppiness field. It would also help to give hatching shapes soft rather than hard masks.

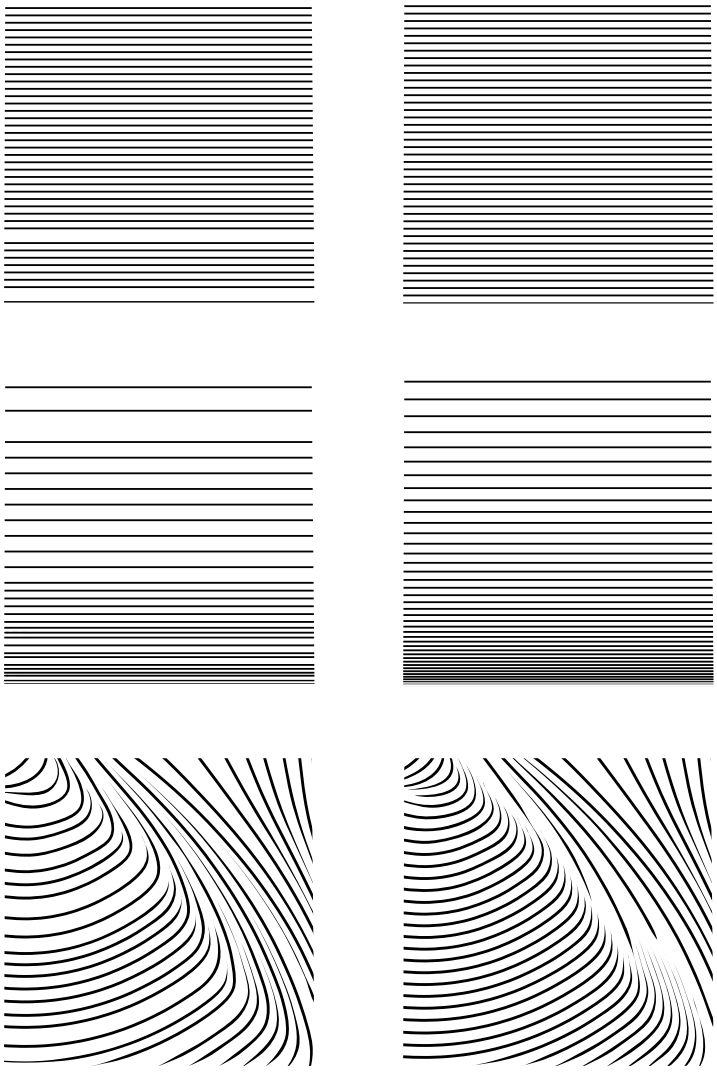
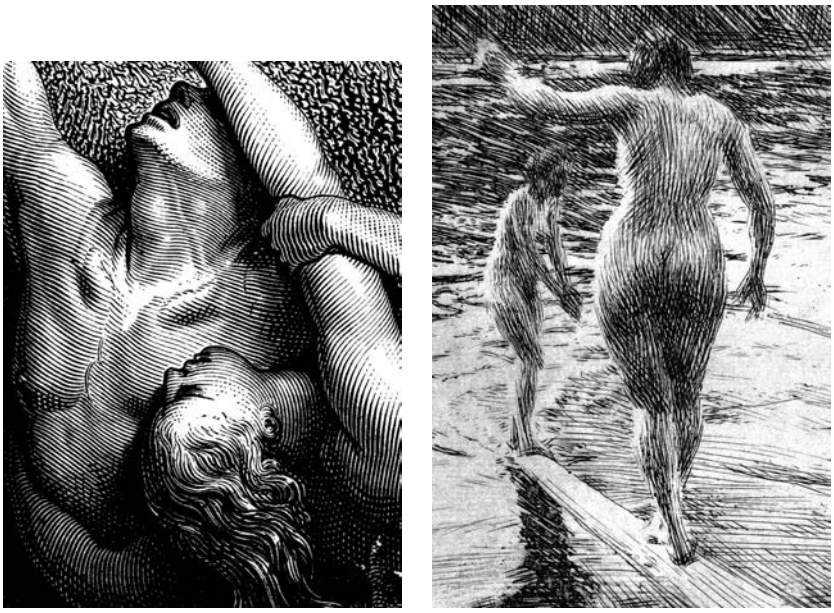


Figure 5.32: The left column shows the default seeding method; the right shows line-based seeding. The latter method fixes spacing artifacts for simpler hatching shapes (first two rows), but is too simplistic for more complex ones (bottom row), where it fails to produce a uniform look.



(a) (b)

Figure 5.33: *Clean* and *messy* hatching, demonstrated respectively by details from *The Deluge*, Gustave Doré, circa 1866, engraving (a); and from *Balans*, Anders Zorn, 1919, etching (b).



Figure 5.34: Messy hatching of a furry subject, from Orientable Textures [82].

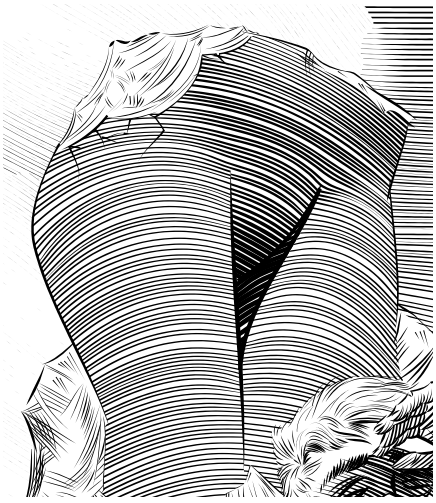


Figure 5.35: It can be a struggle to depict a 3D object with a hatching shape.

5.7.4 Depicting 3D Objects

As is apparent in examples like Figure 5.32, hatching shapes are quite capable of generating abstract line patterns. However, I always intended to use them for depicting 3D objects. Despite barrier curves and blend stroke/multi-dir operations, hatching shapes are still awkward at performing this task. In Figure 5.35, the legs do not look truly round, and do not blend convincingly with the torso. I could have gotten a better 3D look had I been willing to tinker longer with the spacing and direction fields; the results shown are the best my patience would allow.

There may be a future version of the hatching shape that is as inherently 2D as the present version but that can handle 3D-looking hatching more comfortably. I predict that this better version would rely more heavily on directional guide curves like the ones used in the multi-dir operation, promoting them to persistent entities that are part of a hatching shape's definition, like barrier curves. However, it may be that the hatching shape is too 2D-oriented to ever be a good tool for 3D-looking hatching. This suspicion led me to abandon hatching shapes and explore the 3D-oriented primitives described in Section 3.4.3. After that exploration, I left behind 3D depiction to develop the problem of combining two superimposed drawings, which problem is the subject of the next chapter.

Chapter 6

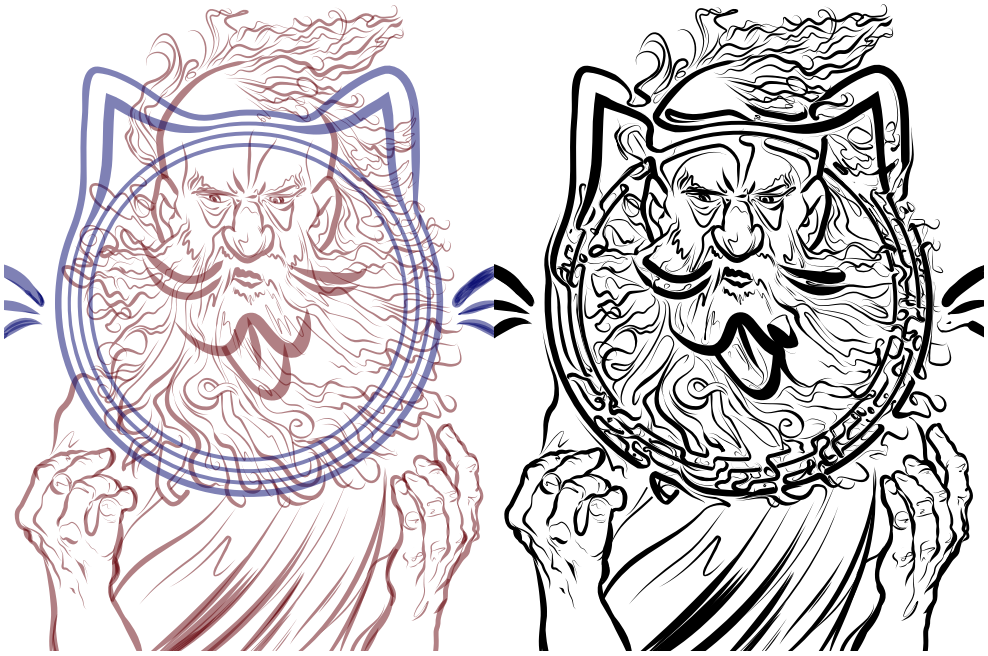
Drawing Mashups

6.1 Introduction

After my first two technical contributions—a formal defining of hatching, then the hatching shape—I moved beyond hatching. My search for a third contribution took an interesting turn. Instead of continuing to use CS to solve traditional art problems, I would invent a new problem. This new problem needed to appeal to me artistically, it needed to be solvable with my knowledge of CS techniques, and it needed to be scientifically novel enough to merit publication.

The problem I came up with is to place two input drawings on top of each other and make a *drawing mashup* from them. My third technical contribution is two things at once: First, it is this new problem for NPR researchers to solve. Second, it is my initial algorithm for solving the problem. A drawing mashup produced by my algorithm appears in Figure 6.1.

On an artistic level, the mashup problem appealed to me because my art often deals with *two realities juxtaposed*. I have been looking for ways to illustrate dreaming turning to waking, the banal meeting the psychedelic, the profane encountering the sacred. Once I had the idea of mashing up drawings, I right away appreciated how useful this could be for such existential juxtapositioning. I visualized illustrations that I could use as test cases for my mashup technique/algorithm. The first image that came to mind was that of a prophet blending with a halo: a once ordinary flesh and blood person now subsumed by bestowed authority. I also imagined landscapes blending into regular patterns as a way to depict dissociation (Section 6.5 features a stab at this idea). For me, the primary value of drawing mashups is how much they can help with this sort of visual storytelling.



(a) (b)

Figure 6.1: Two input drawings (red and blue) combine to create a mashup (b). The mashup shown here is computationally generated with no manual editing afterward.

I must emphasize that simply *conceiving* of drawing mashups was valuable for me as an artist, and would still have been valuable had I never devised a mashup algorithm. My technique for mashing up two drawings—the technique I based my algorithm on—is something any artist could do manually. This technique is itself a discovery. It is not a contribution in the scientific sense. Neither is it necessarily a new frontier in drawing. I do not know whether another artist has ever used a similar technique. To ask that question is to treat art too much like science. My point is that drawing mashups are a *personal* creative development. They are this dissertation's prime example of CS research changing me *as an artist*. At the same time, I am confident that the *technical problem* of mashing up drawings, along with my initial algorithmic solution, will interest NPR audiences enough to be called a scientific contribution.

Inventing drawing mashups led me to view NPR as an art medium. Art media have limitations, and adapting to these limitations inspires creativity. Traditional print media inspired the creation of hatching via their limitations. NPR pushed me in an analogous way. I wanted to do something artistically interesting, but that something needed to be

publishable on NPR's terms. The contrived problem of nontrivially blending two drawings was a creative leap over both hurdles. First, the problem linked with my own visual storytelling needs. Second, I saw that if I found an algorithmic solution to this problem, I would have entered scientifically novel territory.

My testbed application plays an interesting role here. For the purposes of the mashup problem, I formally define a *drawing*—whether input or output—as an unordered collection of strokes. Each stroke has a position curve and a width curve. This definition obviously comes from the way I designed my application. The application then, in an organic way, informed how I responded to NPR's requirements.

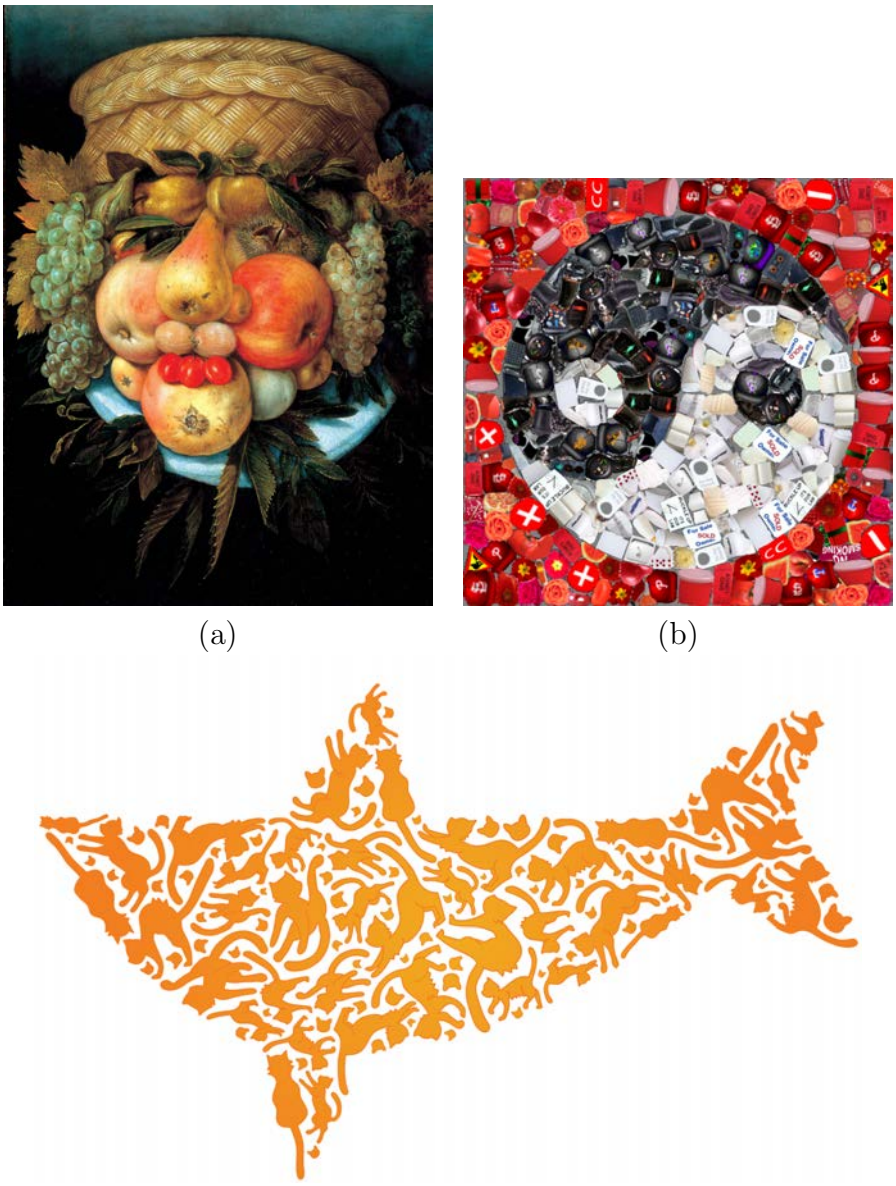
Of course, *drawing* in this chapter does not encompass all the kinds of drawings I can make with my testbed application. I do not just mean the exclusion of primitives other than strokes. For mashup purposes, drawings are binary: the strokes have one color, the background another. This chapter's figures use red and blue for Drawings A and B, but this is only for clarity. The lack of actual inter-stroke color variation means I can treat the strokes within a drawing as unordered. It also means that the input drawings have no ordering. Drawing A is not above Drawing B, or vice versa; like the red and blue, these names are only for explanatory convenience.

Given the definition of a drawing as a collection of strokes, I define the mashup problem as *resolving intersections between different-drawing strokes in a superposition of two drawings, in order to create a single cohesive drawing.*

Section 6.2 mentions related work. Section 6.3 explains how I came up with the drawing mashup problem and a manual technique for solving it. The algorithm that reproduces my manual technique appears in Section 6.4. Section 6.5 describes this algorithm's value and its drawbacks. In Section 6.6 I consider future work on the mashup problem, and in Section 6.7 I say more about this problem's scholarly significance.

6.2 Related Work

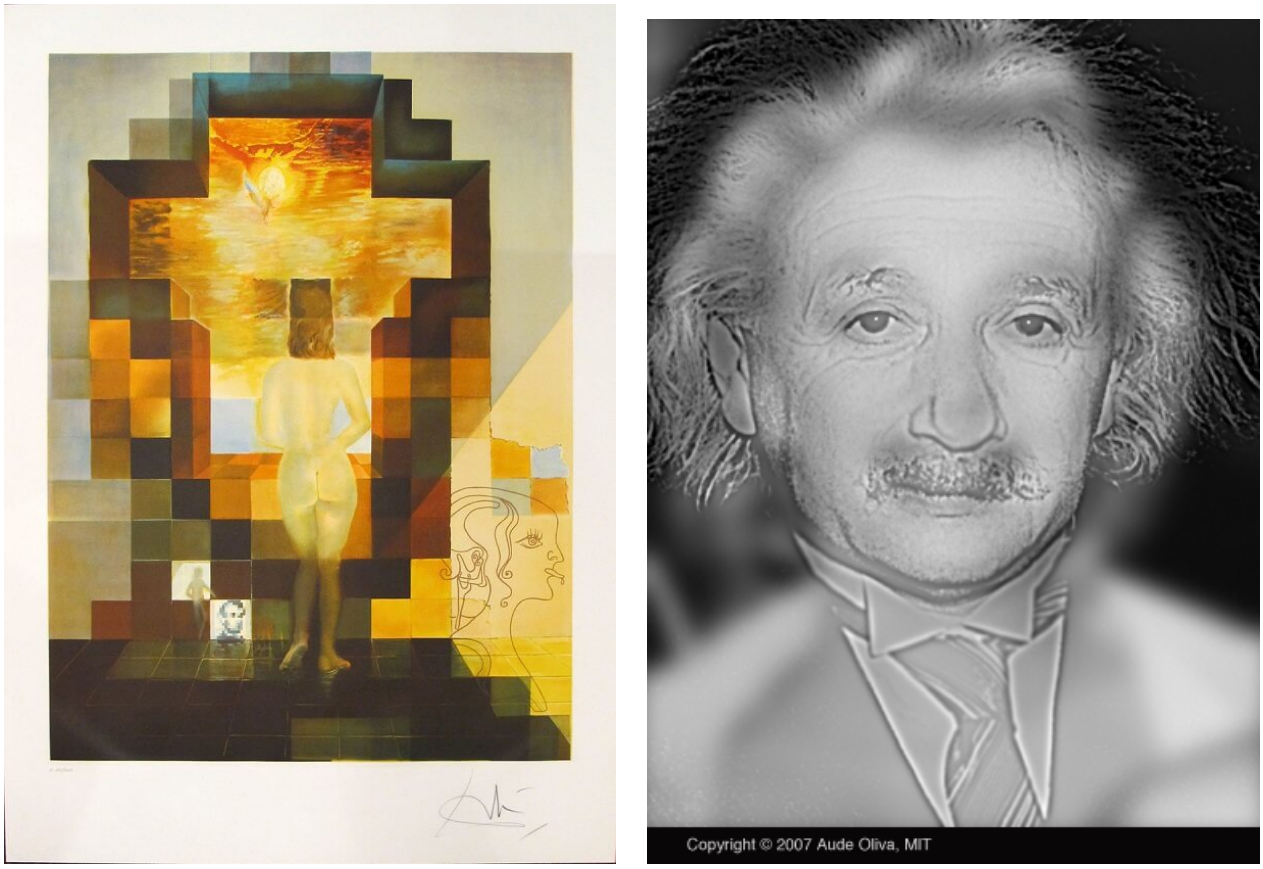
I am unaware of prior work on creating a single drawing from a two-drawing superposition. However, there is precedent, both within art in general and in art-oriented CS, for creating a single picture from multiple input pictures. One artistic task that is conceptually close to the mashup problem is to organize multiple small images into a single cohesive picture. Figure 6.2 shows a few approaches. The Renaissance artist Giuseppe Arcimboldo is famous for his whimsical portraits made out of images of household items, particularly fruits and vegetables (a). Battiato et al., in their survey on *photomosaic* creation [7], name Arcimboldo



(a) (b)

(c)

Figure 6.2: A relevant art problem is to create a single large picture from multiple small pictures. Consider the Renaissance precedent (a) set by Giuseppe Arcimboldo (*Fruit Basket*, ca. 1590, oil on panel). Arcimboldo's work motivates the photomosaic technique (b) of Kim and Pellacini [52], which evokes "element packings" (c) by Saputra et al. [85, 84].



(a) (b)

Figure 6.3: Another relevant art problem is to make a single image that reads differently at different distances—in effect, multiple images in one. Salvador Dalí (a) helped introduce this problem (*Lincoln in Dalivision*, 1977, lithograph). Oliva et al. [72] address this problem with "hybrid images" (b).

as one of the inspirations for this task. He is the explicit inspiration for the irregular photomosaic technique (b) of Kim and Pellacini [52]. Not all NPR work in the Arcimboldo vein is about stitching photos together. Consider the "element packing" designs (c) of Saputra et al. [85, 84], in which small graphics shuffle and warp to collectively fill out intended silhouettes.

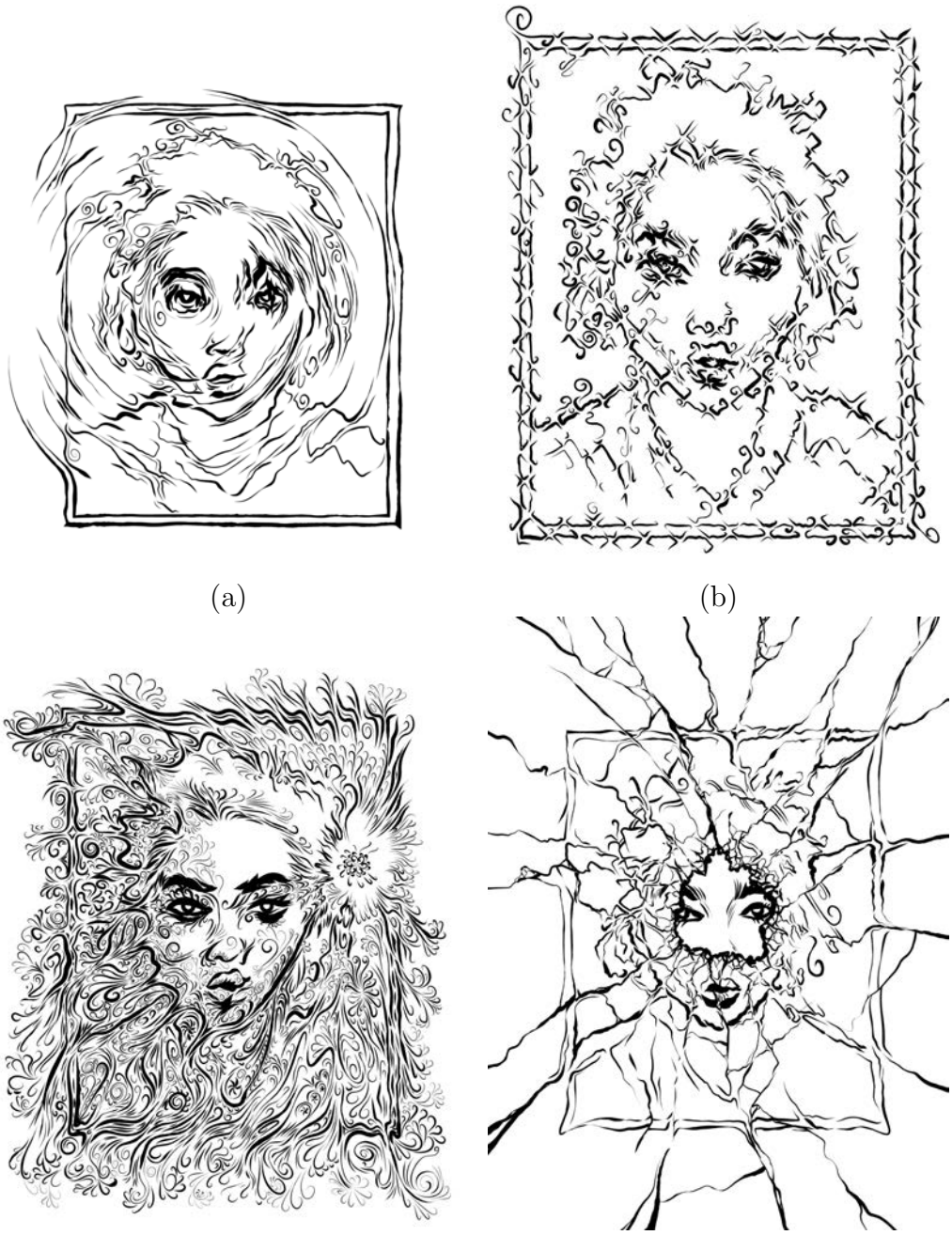
Another art task in drawing mashups' territory is to create a single image that reads differently at different distances. Salvador Dalí's *Lincoln in Dalivision* (Figure 6.3a) is a pioneering image in this direction (it is also one of the inspirations for photomosaics as per Battiato et al. [7]). Up close, this picture reads as a figure painting of Dalí's wife, and at a distance it reads as a portrait of Abraham Lincoln. The "hybrid images" of Oliva et al. [72] have the same multiplicity; see Oliva's *Marilyn Einstein* (Figure 6.3b). Oliva et al.'s research inspired work by Chu et al. on "camouflage images," where a foreground element blends strategically into its background so the viewer must work to discover it [15].

6.3 Learning to Mash

The term *mashup* comes from music mixing. A mashup of two songs is not simply those two songs played dissonantly together; it is a single song that uses pieces of the two inputs in an appealing way. Similarly, a mashup of two drawings should function as a single drawing, using elements (strokes or stroke intervals) of the two input drawings arranged strategically across canvas space.

At the start, I had little more in mind than the abstract concept of mashing up drawings. What combining two drawings should look like was unclear. To refine the problem and develop a technique for solving it, I made a series of concept sketches, some of which appear in Figure 6.4. The algorithm I present in Section 6.4 does not encompass all the effects shown in these pictures. For instance, it is not capable of generating random aesthetic strokes to fill gaps in a mashup (c). However, these drawings establish a guiding principle for the mashup problem: *when strokes from different drawings interrupt (intersect) each other, there must be some resolution.* The concept sketches also demonstrate two methods of resolution for when an A-stroke interrupts a B-stroke:

1. Smoothly connect the two strokes.
2. Have one drawing *dominate* the other, meaning that this drawing's stroke remains in place. The other drawing yields gracefully: its stroke deflects and tapers off into a *tail*.



(a) (b)

(c) (d)

Figure 6.4: Handmade concept art for the mashup problem. My algorithm captures the general approach shown in these pictures, but not all their effects.

The first method is most apparent in the first concept drawing (a), where one of the input drawings was a set of concentric circles and the other was a portrait. When portrait strokes meet circle strokes, they turn and transition into circle strokes. The second method is evident in all four drawings, wherever one stroke turns and runs alongside a second stroke rather than intersecting it. For more visual explanation of the first and second methods of resolution, see Figure 6.5b and Figure 6.5c, respectively.

I emerged from the sketching stage with a solid manual technique for mashing up drawings. What I needed was an algorithm that captured this technique. A crucial insight quickly arose: the importance of the input drawings' relative positions. In the case of mashing up a halo with a figure, relative positioning is about how large the halo is relative to the figure and which parts of the figure it covers. These are important compositional decisions. One could design a mashup algorithm that automatically made choices in this area, but as previously stated, I like to handle composition myself. All I wanted the mashup algorithm to do was handle stroke-level busywork. Hence, I define the drawing mashup problem relative to a specific superposition of two input drawings rather than relative to just a pair of input drawings.

Once I decided on the importance of input images' superpositioning as a priori information, I had an important insight: mashing up two drawings comes down to dealing with *crossings*. A crossing is where two or more strokes intersect, where not all the strokes are from the same drawing. If the two input drawings spatially overlap but no A-stroke intersects any B-stroke, then there is no work to be done and the result is simply the two inputs copied in place. The mashup problem is ill-posed in this case. It is like asking for a musical mashup where Song A ends at five minutes and Song B starts at six minutes.

Figure 6.5 shows my mashup algorithm applied to a simple scenario. Each input drawing contains one closed stroke forming a rectangle. There are two crossings, each with four inputs, two from the A-stroke and two from the B-stroke. A *stub* is one of the inputs to a crossing ("stroke" does not work here, since a crossing can have multiple inputs from the same stroke). For each crossing, I determine a *routing*. Finding a routing means deciding which pairs of stubs should connect. For each stub-stub connection, I generate a smooth *joint*. In Figure 6.5, Crossing 1 produces two joints, each joining part of the A-stroke to part of the B-stroke. Crossing 2 produces one joint which reconnects the two A-stubs. The figure's dotted lines (b,c) are not the actual joints; they merely indicate stub pairs to connect. The joints themselves are discernible in the mashup (d,e).

Some routings will leave stubs unconnected. After determining all crossings' routings, I extend these severed stroke-ends into tails (Figure 6.5e), which taper to zero width while turning toward and then running alongside other strokes in the mashup.

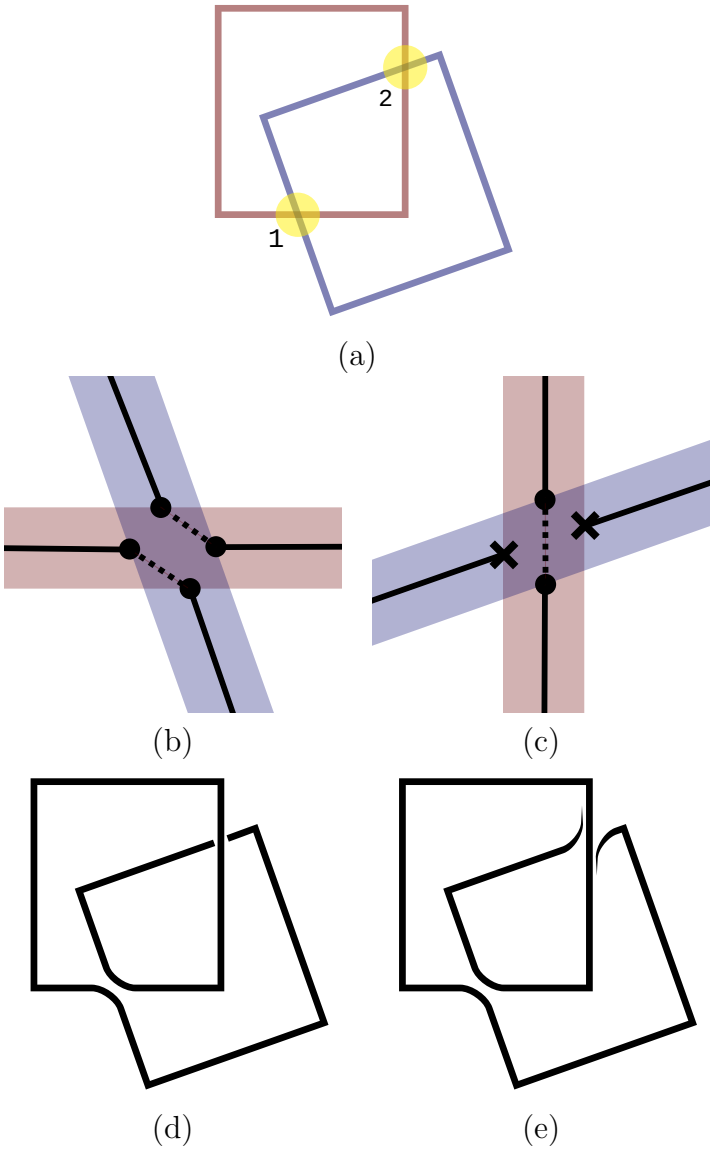


Figure 6.5: The drawing mashup algorithm. Given overlapping input drawings (red and blue), first determine a *routing* at each *crossing*, in this case a routing for Crossing 1 (b) and a routing for Crossing 2 (c). Using the routings, stitch together *mashed strokes* (d). All that remains is to have severed stroke-ends taper off into *tails* (e). Note the two tails which result from Crossing 2's routing.

Broadly speaking, these are the algorithm's two pieces: routing and tail generation. Together, these pieces fulfill the overarching task of resolving inter-drawing interruptions. Because resolving such interruptions is the primary goal, care must be taken within each part of the algorithm not to create new stroke-stroke intersections, though there is no issue with preserving stroke-stroke intersections found within one of the original drawings.

6.4 Method

The algorithm's inputs are two drawings—A and B—plus tunable parameters which I will introduce as they become relevant. To contextualize parameters representing distances, imagine the operation taking place within a region of canvas around 1000×1000 units. The output is a collection of new strokes, which I call *mashed strokes* to distinguish them from input strokes. Note that *mashed stroke* can refer to an output stroke in progress, such as an output stroke that still needs a tail at one or both ends.

Here is the algorithm in outline:

1. Find the crossings (Section 6.4.1).
2. At each crossing, find a routing. After finding all routings, generate mashed strokes without calculating any tails (Section 6.4.2).
3. Wherever a mashed stroke needs to end with a tail, find a corresponding *tailable interval* on that stroke (Section 6.4.3).
4. Convert tailable intervals into tails (Section 6.4.4).

6.4.1 Crossings

A crossing is the intersection of two or more strokes not all from the same drawing. It is a case of one drawing interrupting the other, requiring resolution. If two or more strokes *all from the same drawing* intersect each other, this is not an issue for the algorithm to solve. It is just a feature of that drawing, which might as well persist in the mashup.

Recall that each stroke contains a position curve and a width curve, the two of which combine to define the stroke's shape (Section 3.3). A stroke's position curve can be thought of as its *spine*. For the purposes of this algorithm, a stroke's spine is parametrized by arc

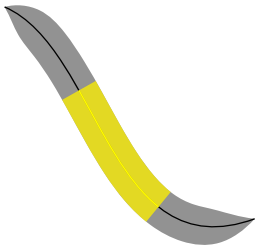


Figure 6.6: A *stroke interval* highlighted in yellow

length. A *stroke interval* is part of a stroke based on an interval along its spine, as shown in Figure 6.6.

To find crossings, I first determine each stroke's *unoccluded intervals*: intervals along which the stroke does not intersect any stroke from the other drawing. *Intersection* between strokes means any overlap between the full width-based stroke shapes, not just a crossing of the strokes' spines. With the unoccluded intervals found, I build a collection of crossings by grouping intersecting *occluded* stroke intervals. Figure 6.7 shows some crossings found this way.

Each of a crossing's inputs is a *stub*. A stub is the point where some unoccluded stroke interval terminates at/feeds into the crossing. Typically, a stroke involved with a crossing will contribute two stubs to it, as in Figure 6.5, though it might contribute only one or more than two.

6.4.2 Routing

Mashing up two drawings is about resolving inter-drawing interruptions, either by smoothly connecting intersecting strokes or by having strokes from one drawing dominate strokes from the other. These choices are made within each crossing's routing. For a given crossing, a routing is a set of connections between the crossing's stubs, each connection represented by a joint. Each stub gets to connect to one other stub, but might get no connection, resulting in a severed stroke-end which must receive a tail later.

Figure 6.8 shows a single crossing which I will use to explain the routing algorithm. Note the following about the input drawings:

- Drawing A has three strokes and Drawing B has two.

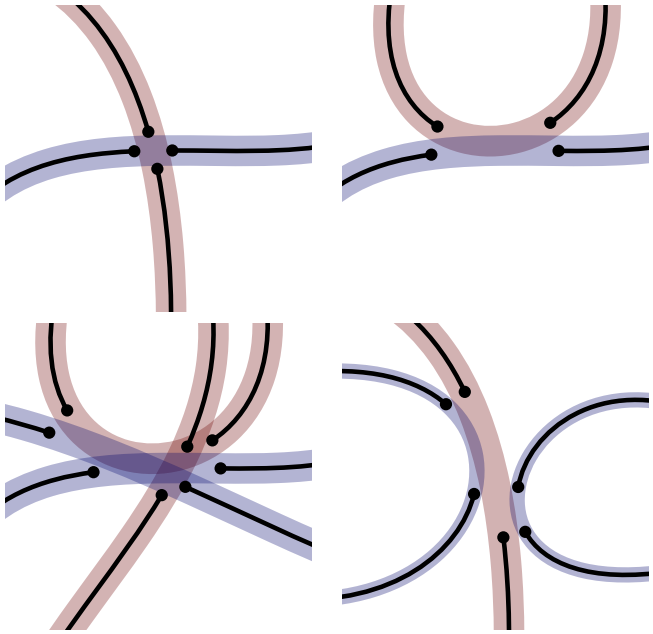


Figure 6.7: Four crossings. A black dot marks a stroke's entry into a crossing, that is, a *stub*.

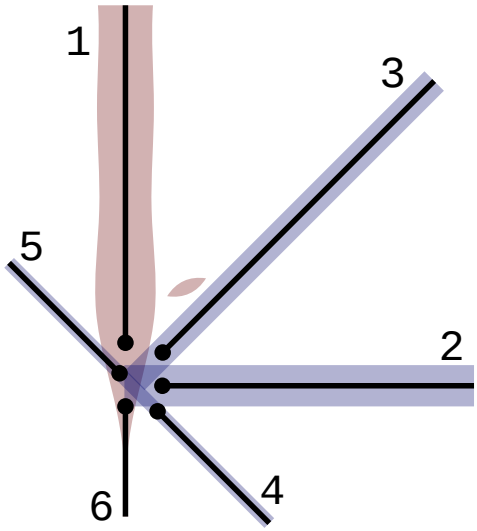


Figure 6.8: Drawing A (red) and Drawing B (blue) produce one crossing. This crossing has six stubs, numbered by decreasing weight (stroke width).

- There is a single crossing. It has four A-stubs and two B-stubs.
- Two of the A-stubs (1 and 6) come from the same stroke, as with two of the B-stubs (4 and 5).
- The small A-stroke between Stubs 1 and 3 is not involved in the crossing. I will refer to it as an "outsider" stroke.

First I assign weights to the stubs. The strokes associated with higher-weighted stubs are more likely to dominate the strokes associated with lower-weighted stubs. To reiterate, domination is the second of the two ways to resolve an inter-drawing interruption; see for instance how the red stroke dominates the blue stroke in Figure 6.5c. I normally set a stub's weight to its stroke width, that is, to the width of the stroke where it enters the crossing. This results in heavier strokes dominating lighter strokes. I have numbered the stubs in Figure 6.8 by decreasing weight (Stubs 4 and 5 have the same weight).

With stubs' weights determined, I assign connections greedily, moving through the stubs by decreasing weight. Stub 1 has the highest weight so it is the first to receive a connection.

To determine a connection for a stub, I pick a first and second choice, then select one of these based on a random function. If a stub has an "original" connection in the crossing, meaning it and some other stub represent the same stroke's entry into and subsequent exit from the crossing, then the first choice is that original connection. Stub 1 has an original connection represented by Stub 6, so Stub 6 is the first connect-to choice. The second choice is the highest-weighted unconnected stub other than Stub 6 with which Stub 1 can legally join. Two stubs can legally join if there is a smooth joint stroke connecting them which

- does not intersect any A- or B-stroke outside of the crossing,
- does not intersect any already established stub-stub connection in the crossing, and
- does not cut off more than half of any unoccluded stroke interval feeding into the crossing.

The first two criteria guard against creating new inter-drawing interruptions in the mashup. Remember that creating a mashup is about *resolving* these interruptions. The third criterion lets crossings' routings be determined independently of each other, which simplifies the mashup algorithm.

Stub 1 can, at this point, legally join with any of the other stubs, so its first connect-to choice is Stub 6 (its original connection) and its second is Stub 2, the highest-weighted of the other stubs. If Stub 6 did not exist, the choices would be 2 and 3.

Given two choices with weights w_1 and $w_2 \geq 0$, a simple probability $p \in [0, 1]$ of a "flip"—of choosing the second instead of the first choice—is defined thus[1]:

$$p = w_2/(w_1 + w_2) \tag{6.1}$$

I use a "sharpened" probability p_s:

$$p_s = 0.5 + \frac{(2|p - 0.5|)^\gamma}{2}\text{sgn}(p - 0.5) \tag{6.2}$$

The function $\text{sgn}(x)$ returns 1 or -1 based on the sign of x. Equation 6.2 pushes p_s toward 0 if $p < 0.5$ and toward 1 if $p > 0.5$. The tunable $\gamma \in [0, 1]$ modulates the strength of this push. If γ is 1 (the default value), then Equation 6.2 reduces to Equation 6.1. If it is 0, then p_s is either 0, 0.5, or 1, and randomization disappears except when $w_1 = w_2$.

In the present situation, a flip happens to occur, so I choose the second option—Stub 2—as the connection for Stub 1.

Figure 6.9 shows the joint between Stubs 1 and 2. To make a joint between two not-originally-connected stubs, I pick a *cut position* (a parametric value) on each of the stubs' stroke intervals. I use the two cut positions to truncate the stroke intervals—see the yellow dots—and then construct an interpolating substroke that smoothly joins the truncated stroke intervals. The range of cut positions on a stroke interval lies within a circle of tunable radius r_{joint} placed at the interval's original endpoint. I also limit cut positions so that no more than half of a stroke interval's original length is removed. (Again, this lets crossings' routings be determined independently from each other.) The two circles in Figure 6.9—centered around the stroke intervals' endpoints pre-truncation—show the chosen r_{joint}, 300 units of canvas space. Based on this setting alone, the joint could have cut off more of the stroke intervals, but a more conservative joint avoids intersecting the outsider A-stroke.

Choosing the cut positions for two stroke intervals (corresponding to two stubs) is a brute-force search over a space with two degrees of freedom. Each dimension of this space is a bounded interval of parametric positions on a stroke interval, this bounded interval limited based on r_{joint} and on the rule against cutting off more than half of the interval. I

[1] If w_1 and w_2 are both 0, treat them both as some nonzero k.

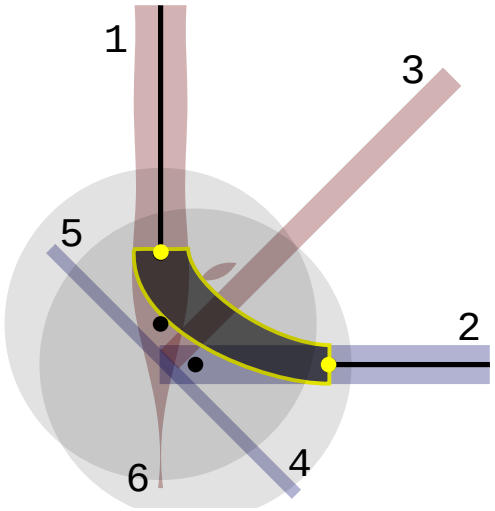


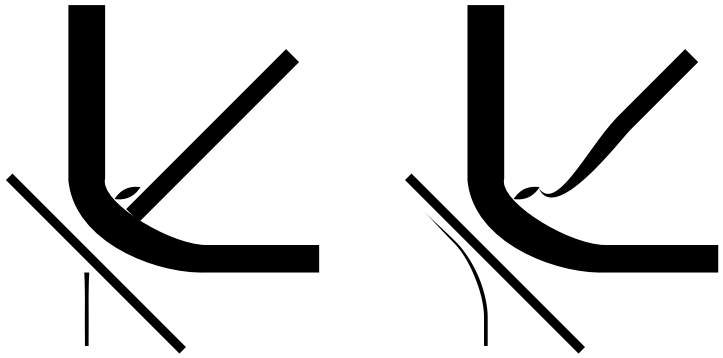
Figure 6.9: To make a joint between Stubs 1 and 2, find *cut positions* (yellow dots) on their corresponding stroke intervals and then smoothly join the truncated stroke intervals. The range of possible cut positions for a stub lies within a circle of radius r_{joint} centered at the stub's original location (black dots).

highly discretize this search space and pick more aggressive cut position pairs before more conservative ones, concluding upon finding the first joint that does not intersect any joints already found for the crossing or intersect any strokes not associated with the crossing.

With Stubs 1 and 2 taken care of, the next to consider is 3. The just-established joint has limited the usable range of Stub 3. This joint also cuts off Stub 3 from all connection options; any of my algorithm's tentative joints from Stub 3 to Stubs 4-6 would intersect this joint. Therefore I leave Stub 3 unassigned, meaning I must generate a tail there later.

For Stub 4, the two connection choices are first Stub 5 and second Stub 6. The first option prevails and Stub 5 connects with its original connection. There is no search for cut positions; the joint is simply the occluded part of the B-stroke that generated Stubs 4 and 5. After that, the only stub left to give a connection to is Stub 6. The only other unassigned stub is the unreachable 3, so Stub 6, like Stub 3, becomes the site of a future tail.

After determining routings for all crossings, it is straightforward to chain unoccluded substrokes and joints together to form mashed strokes. Figure 6.10a shows the five mashed strokes I can derive after finding the routing for this section's crossing. One of these mashed strokes is the outsider stroke. Sections 6.4.3 and 6.4.4 explain how to create tails for mashed



(a) (b)

Figure 6.10: With all crossings' routings determined, I can derive tailless mashed strokes (a). Later I explain how to give tails (b) to mashed strokes that terminate at unconnected stubs.

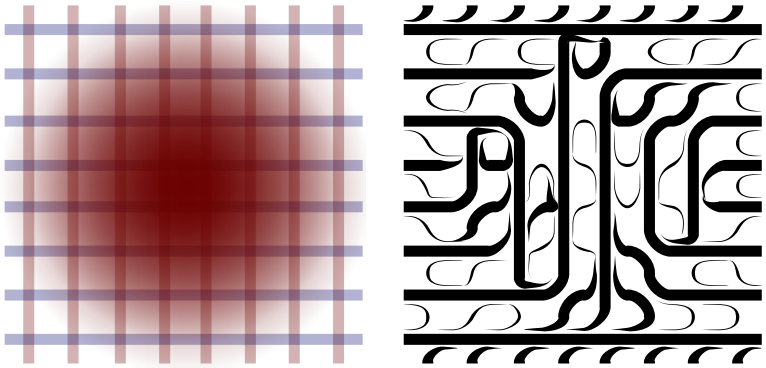
strokes that terminate at unconnected stubs (Figure 6.10b).

AB Field. An alternate way to determine stubs' weights is via an *AB field*, a unit scalar field f_{AB} defined across the canvas. At position $\boldsymbol{p} = (x, y)$, A-stubs have weight $f_{\text{AB}}(\boldsymbol{p})$ and B-stubs have weight $1 - f_{\text{AB}}(\boldsymbol{p})$. In Figure 6.11, f_{AB} is 1 at the center of the canvas and fades radially outward to 0. As a consequence, Drawing A tends to dominate Drawing B nearer the center.

6.4.3 Tailable Intervals

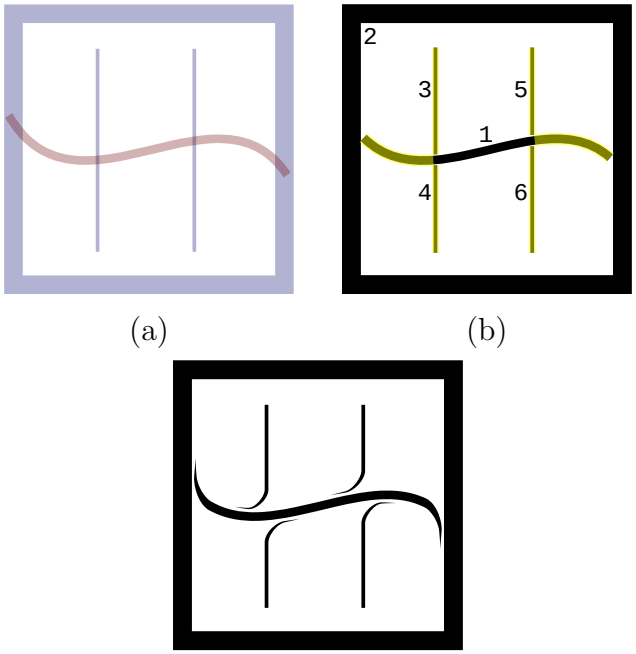
Once I have found crossings, determined a routing at each, and derived tailless mashed strokes, the next step is to determine these mashed strokes' tailable intervals—intervals that may be turned into tails. Imagine that wherever a mashed stroke has a severed end, i.e., it ends with an unconnected stub, I will prune off some of that end and allow it to regrow in an aesthetic way. This section is about determining how much *may* be trimmed off (but not necessarily how much *will* be).

A mashed stroke can have up to two tails according to which of its ends (if any) terminate at unconnected stubs. Platonically, a mashed stroke needing two tails has two tailable intervals, one at each end, but in practice these often overlap, leaving the whole mashed stroke tailable. A mashed stroke needing only one tail may also be completely tailable.



(a) (b)

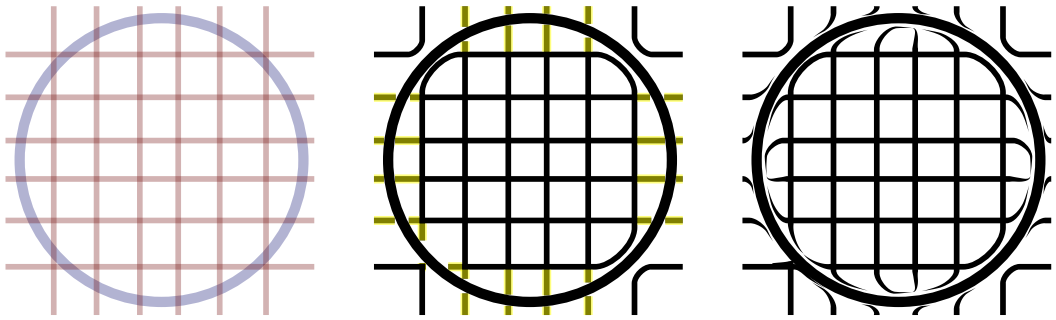
Figure 6.11: An AB field locally defines the relative dominance of Drawing A (vertical strokes) versus Drawing B (horizontal strokes). In the center, Drawing A dominates. At the edges, Drawing B dominates.



(a) (b)

(c)

Figure 6.12: After finding routings for the inputs (a), I derive six mashed strokes (b), numbered arbitrarily. Some of these mashed strokes have tailable intervals (yellow). I convert (parts of) these intervals into tails (c).



(a) (b) (c)

Figure 6.13: I restrict a mashed stroke's tailable intervals not just by different-drawing crossings, but by same-drawing intersections. The strokes of Drawing A (red) cut each other off, limiting the tailable intervals (yellow) available to their derived mashed strokes. The effect is to preserve Drawing A's self-intersections in the mashup (c).

A mashed stroke that starts or ends with an unconnected stub has a tailable interval that begins at the unconnected stub and extends either all the way to the stroke's other end or until the stroke encounters

1. a crossing, specifically the start of the stroke's joint heading into said crossing, or
2. a same-drawing intersection.

Figure 6.12 shows how crossings limit tailable intervals. Mashed Stroke 1 terminates at each end with an unconnected stub, meaning it needs two tails. It also passes all the way through two crossings. Both of its tailable intervals extend only up to entry into these two crossings. Mashed Strokes 3 through 6, each of which needs one tail, are fully tailable for lack of any such interruptions. Mashed Stroke 2 does not need tails.

In Figure 6.13, same-drawing intersections rather than crossings are the delimiters. I include same-drawing intersections as tailable interval delimiters so that same-drawing intersections will be preserved in the mashup where possible.

6.4.4 Tails

The first method of resolving an inter-drawing interruption is to smoothly connect strokes from different drawings. The smooth joints in Section 6.4.2 accomplish this. The second

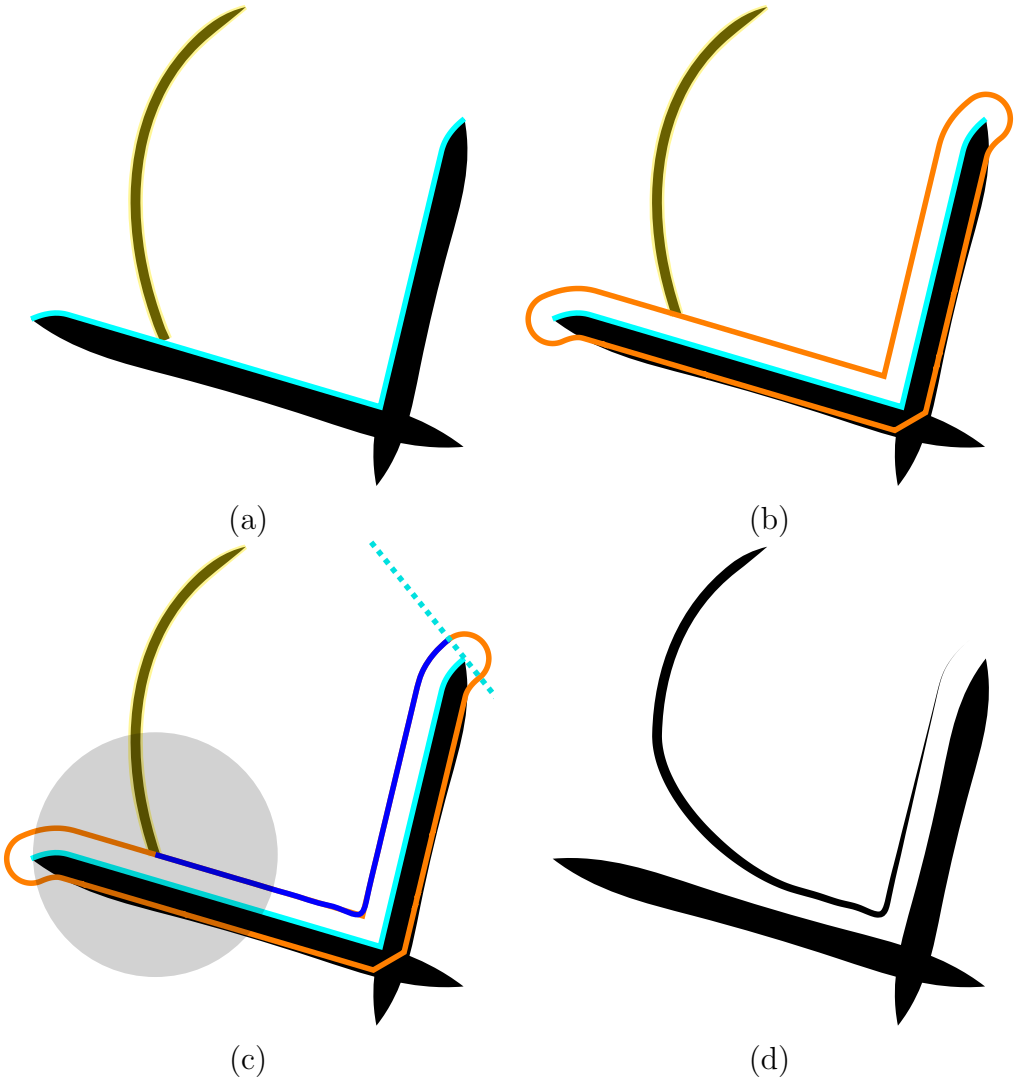


Figure 6.14: To create a tail for the mashed stroke s (yellow), find an on-barrier path (cyan) at s's endpoint. Inflate this path and cut off s where it meets the inflated shape (b). Next (c) establish a join-with path (blue) lying on the inflated shape and ending at the "wall" defined at the end of the on-barrier path (dotted line). Join s with the join-with path using cut positions found within the circle defined by radius r (c). The tail itself (d) smoothly joins between these cut positions, then runs along the join-with path.

method of resolution when Drawing A and Drawing B overlap is for one drawing to dominate and the other to yield. The yielding strokes should taper off appealingly without creating further interruptions. A mashed stroke s that ends in an unconnected stub represents a situation where one drawing has dominated the other. This section is about trimming off part of that end (or ends) of s and creating a tail (or tails). This brings about the yielding of one drawing to the other.

Creating a tail for a mashed stroke s means turning toward and then running alongside *barriers*: other mashed strokes, some of which have their tails, some of which are still waiting. Figure 6.14 illustrates the process for an s that needs one tail and whose tailable interval covers its whole length.

As preparation, I find a "tail radius" (c). Each tail has its own radius r based on the tunable r_{tail} (Figure 6.15). I set r for s as the maximum of

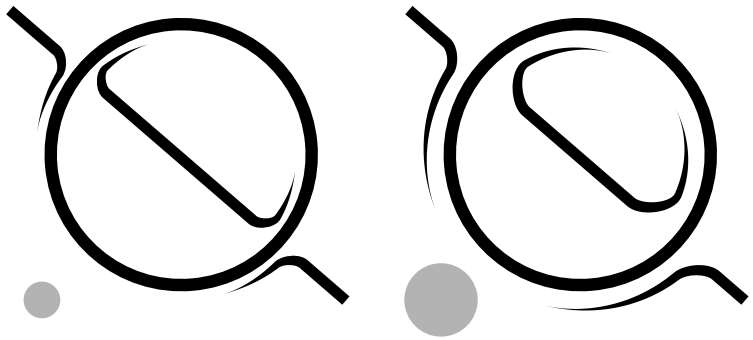
1. the average stroke width of s times 1.5, and
2. the minimum of r_{tail} and 0.6 times the length of s.

The intent is for r to be r_{tail} except when it needs to increase to prevent a very wide s from having to taper too quickly, or when it needs to decrease so that a very short s does not have a tail far longer than itself. This is to maintain a consistent look when strokes vary drastically in scale.

With r determined, I cast a ray from the end of s; find its first intersection with a barrier; and record, based on the relation between ray direction and the barrier's normal, whether the tail ought to turn clockwise or counterclockwise. Walking from this intersection point in both directions produces an on-barrier path, colored cyan in Figure 6.14. The on-barrier path does not go past the tip of any mashed stroke it walks along, which prevents the tail of s from cutting off other mashed strokes' tails.

Inflating the on-barrier path using an offset distance d_{off} produces a shape with which I trim s (b). Let d_{off} be the minimum of

1. r times the tunable $d_{\text{tail}} \in (0, 1]$, and
2. 0.3 times the maximum distance of the on-barrier path from the part of s inside a circle of radius r placed at the ray-barrier intersection (this is a different circle from that in Figure 6.14c).



(a) $r_{\text{tail}} = 50$ (b) $r_{\text{tail}} = 100$

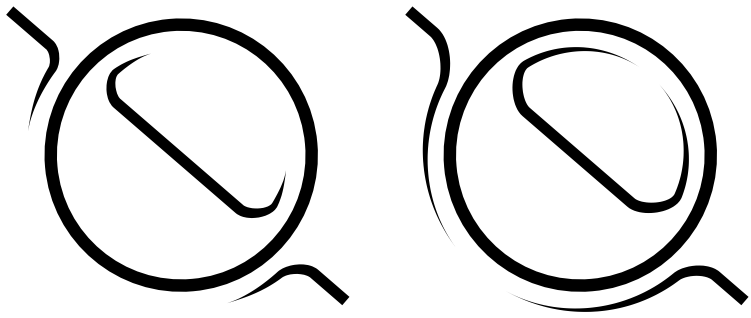
Figure 6.15: A larger r_{tail} means tails define wider turns before running alongside barriers. The scene is 1000×1000 canvas units.

The formulation of d_{off} attempts to keep the inflated shape from completely consuming s.

Next I find the *join-with* path from which I will make the tail. I determine this path by starting where s intersects the inflated shape, then running along it after turning clockwise or counterclockwise as per the earlier determination. The join-with path runs until it exits the circle of radius r placed at the (updated) endpoint of s (Figure 6.14c), after which it travels an additional distance of rd_{along}. The tunable d_{along} (Figure 6.16), expressed relative to local r, gives intuitive control over how long tails run parallel to barriers after turning toward them. The join-with path terminates early if it encounters a barrier (Figure 6.17) or if it reaches the "wall" perpendicular to the on-barrier path, placed at the on-barrier path's end (this enforces the aforementioned desire for tails not to cut each other off). The dotted line in Figure 6.14c represents this wall. In the example scenario, $d_{\text{along}} = 3$, giving the join-with path enough length to terminate at the wall.

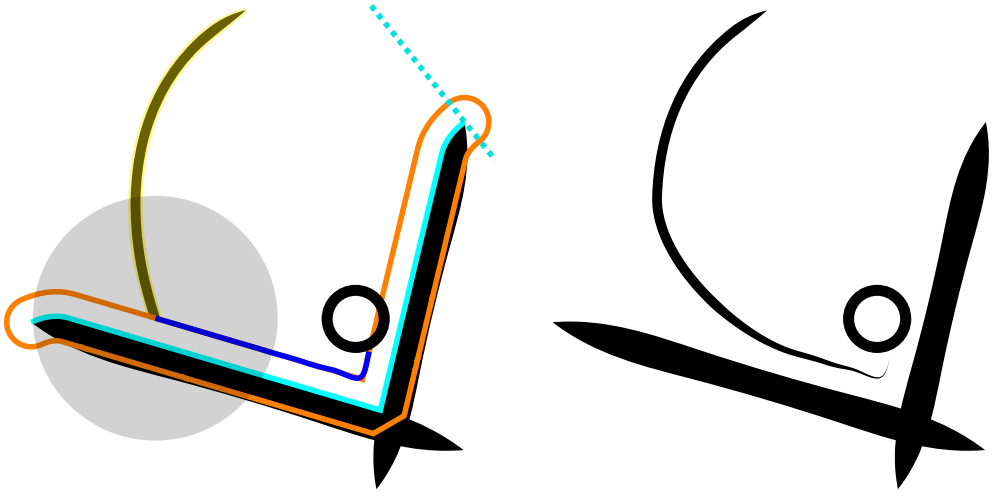
I smoothly join the join-with path to s within the radius-r circle shown in Figure 6.14c. This means picking two cut positions, similar to the joining described in Section 6.4.2. I prefer cut positions as aggressive (far from the circle's center) as possible. Joints that intersect barriers are illegal. I additionally disqualify a joint if the resulting tail's bounding box does not contain at least 80% of the join-with path's bounding box. This makes it more clear to the viewer which strokes intersect which (and where) in the input.

With the tail's path defined, I define its width curve so that its width equals that of s at the start and tapers to 0 at its end, possibly with extra local tapering to avoid intersecting nearby barriers.



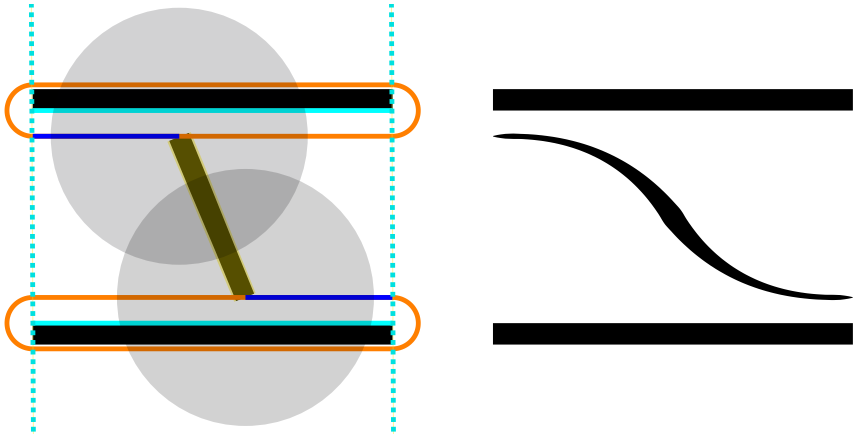
(a) $d_{\text{along}} = 1$ (b) $d_{\text{along}} = 5$

Figure 6.16: The parameter d_{along} says how long tails run alongside barriers after turning toward them.



(a) (b)

Figure 6.17: In a modified version of the scenario in Figure 6.14, a circular obstacle cuts off the blue join-with path (a), abruptly shortening the tail (b).



(a) (b)

Figure 6.18: When a mashed stroke needs two tails, solve for both simultaneously.

When a mashed stroke requires two tails, I solve for both of them at once (Figure 6.18). If the two r values are large enough for the tails' joint circles to overlap, some cut position pairs will completely eliminate the mashed stroke itself and leave the two join-with paths to connect directly to each other. I force such joints to pass through the mashed stroke's midpoint.

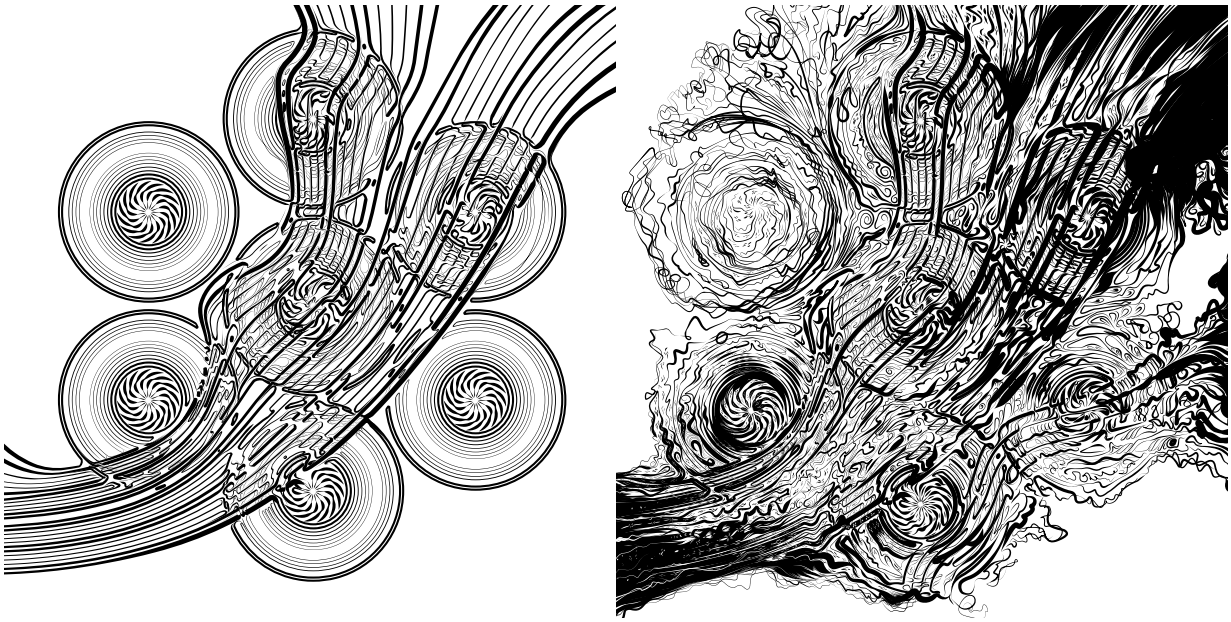
After giving tails to all mashed strokes that require them, the mashup is finished.

6.5 Results

Figures 6.19, 6.20, 6.21, 6.22, and 6.23 give a sense of how powerful the mashup algorithm is when making finished pictures. I hoped at the start of coding that my algorithm would create professional-grade art. This hope is neither completely met nor dashed.

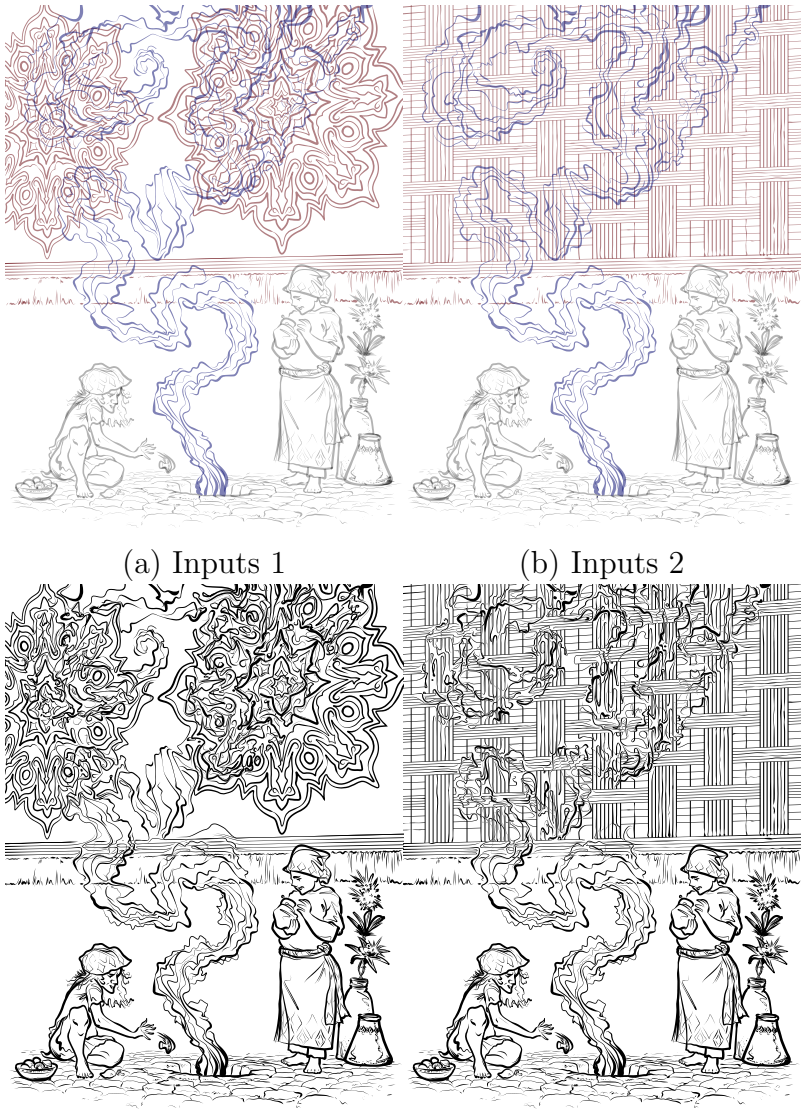
The ability to rapidly experiment with different superpositions and parameter settings has been useful. However, in general the algorithm is too primitive to produce results that can be called finished without manual refinement (Figure 6.19). At present, creating an illustration that showcases computational mashing requires experimentation and an understanding of the algorithm's weaknesses.

I use discretion around which parts of the input drawings overlap and which do not. In Figure 6.20, two mashups combine the same foreground smoke with different background patterns. In both mashups, some of the smoke has nothing behind it in order to help the



(a) (b)

Figure 6.19: A computationally generated mashup (a) does not have to be a finished picture on its own, but might instead be a stage on the way to one. The final picture (b) involved significant manual editing of the computational output.



(a) Inputs 1 (b) Inputs 2

(c) Mashup 1 (d) Mashup 2

Figure 6.20: I mash up the same smoke drawing (blue) with two different background drawings (red). (The gray strokes do not participate in the mashup algorithm.) I wanted the smoke to blend interestingly with a background pattern while remaining easily discernible as smoke. However, in the first mashup (c), the smoke vanishes into the background. The second mashup (d) is more successful: strokes intertwine with appealing complexity, yet smoke and background both remain perceptible. Both mashups are unedited computational results.



Figure 6.21: Computational output tends to look better when at least one of the input drawings is abstract and simple. Blending a landscape with a regular pattern was one of the tasks I had in mind when formulating the drawing mashup problem.

viewer understand the scene, and to understand the mashed-up areas. Even so, results are not entirely successful. In the first smoke-on-background mashup (c), the smoke seemingly vanishes into the background rather than mixing with it. You have to look closely to perceive that there is a mashup at all. The two drawings being mashed are not different enough for the effect to work. A more distinct background (b) gives a more comprehensible and therefore appealing result (d).

Input drawings should not feature overdrawing, at least not near crossings. Overdrawing means creating the impression of a single stroke by drawing multiple strokes on top of each other. The mashup algorithm lacks the sophistication to identify and interact with such an aggregate stroke. (There is some overdrawing in the whiskers of Figure 6.1, but it is far enough from the crossings not to matter.)

Input drawings should also be sparse. Specifically, inter-stroke spacings should be on the same order as r_{joint} and r_{tail}. Otherwise the algorithm does not have room to do its work. In practice this is the same as requiring that input drawings have low numbers of strokes, because mashing up two very dense drawings would require such small r_{joint} and r_{tail} settings that tails and joints would be too small to see.

The requirement for sparseness in the inputs is especially galling since it defeats one

of the purposes of an NPR tool, to save time in highly laborious situations. Still, there are cases where the implementation performs well enough that its time-saving capability is appreciable. The ideal cases seem to be ones where at least one of the input drawings is a simple, repeating abstract pattern, like in Figure 6.21 and Figure 6.20b.

There is no rule saying that the algorithm (or any NPR tool) needs to produce a quality piece of art on its own. See Figure 6.22. After the algorithm generated a mashup (a) from sparse inputs, I performed significant manual work—editing the algorithm's produced strokes, plus drawing new strokes—to produce a more advanced mashup. This semi-automatic mashup is denser, richer, and more semantically informed than what the algorithm could produce on its own.

When computational mashups are unsatisfying, I sometimes fall back on *preserve-drawing* mode, where one of the input drawings is completely unchanged in the result and the other input drawing strictly yields to it (Figure 6.23). If, say, I preserve Drawing A, then A-strokes do not make routing connections. They only act as barriers to tentative joints between B-stubs, and as barriers to mashed strokes' tails. Preserve-drawing mode is comparable to using an AB field (Section 6.4.2) and setting it completely to 0 or completely to 1.

Is my drawing mashup algorithm successful on a personal level? Will it help me in future attempts to portray the confluence of different realities? As for the algorithm itself, my feeling is uncertain. This algorithm is, for me, mostly a proof of concept. The concept itself—an idea produced by combined academic and artistic pressure—is, for me, where the real value lies. I am certain, at least, that in my future art, I will keep mashing up drawings, perhaps manually, maybe using techniques more advanced than the one this chapter's algorithm is based on. Whether I ever use the algorithm again, the thought and work put into making it will influence my pictures from now on.

6.6 Future Work

The piece of the algorithm most suitable for improvement is naturally the most complex: tail generation. The intent here—that a mashed stroke should taper off in an appealing way by turning toward and then running alongside barriers—is open-ended, and the approach outlined in Section 6.4.4 is crude compared to drawing tails by hand.

One area for tails improvement may already be apparent to the reader. Section 6.4.4 describes finding paths for tails to turn toward and join with. These paths come from inflating a *subset* of surrounding barrier geometry using an inflation distance d_{off} dependent

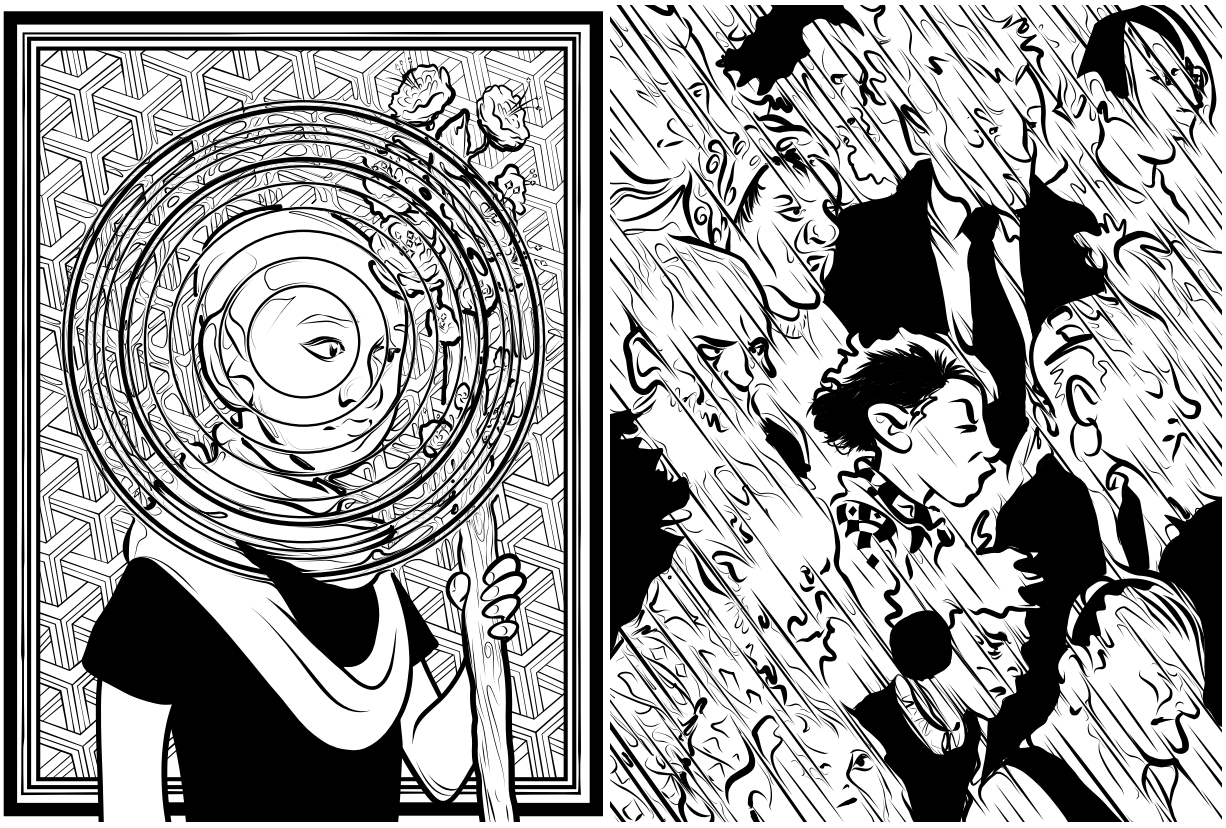


(a)



(b)

Figure 6.22: Output from the algorithm (a) served as the starting point for creating a more advanced by-hand mashup (b).



(a) (b)

Figure 6.23: *Preserve-drawing* mode simplifies the algorithm. One of the input drawings—a halo (a) or some lines representing rain (b)—is unchanged in the result. Both mashups are unedited computational output.

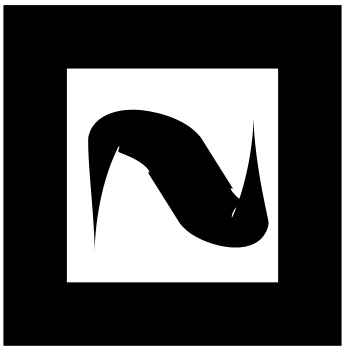


Figure 6.24: Artifacts result when wide strokes must taper too quickly.

on local context. Sometimes a tail will have to terminate abruptly when it runs into barrier geometry not included in the inflation step (Figure 6.17). It would be better to inflate *all* surrounding barrier geometry when finding each new tail, but this would too drastically increase execution time, given changing per-tail inflation distances. Solving this problem would require a sophisticated spatial data structure that evolves as tails are added.

Another tails problem, seemingly more trivial but actually more significant, appears in Figure 6.24. When a wide mashed stroke needs a tail and has limited room to place it, the result is a severe taper, which does not render well. Various tweaks could ameliorate this issue, but the proper way to solve it would be to move beyond the definition of a stroke as a position curve with a width curve, or even to move beyond the definition of a drawing as a collection of strokes.

Consider the illustrations of Figure 6.23, which feature a few filled shapes (Section 3.3). It was frustrating that these filled shapes, not being strokes, could not participate in the mashup algorithm. To enable their participation would require a whole new method for mashing up drawings. This method would be a generalized version of mine capable of dealing with all shapes, not just shapes represented by *strokes* as here defined. Concept sketches for this future, generalized mashup algorithm appear in Figure 6.25.

6.7 Conclusion

In NPR literature, artists and software developers are traditionally separate. Artists produce artifacts assumed to be worthy of the label "art," after which NPR's job is to understand how these artifacts function, i.e., to divine the "science" of art [37], or at least

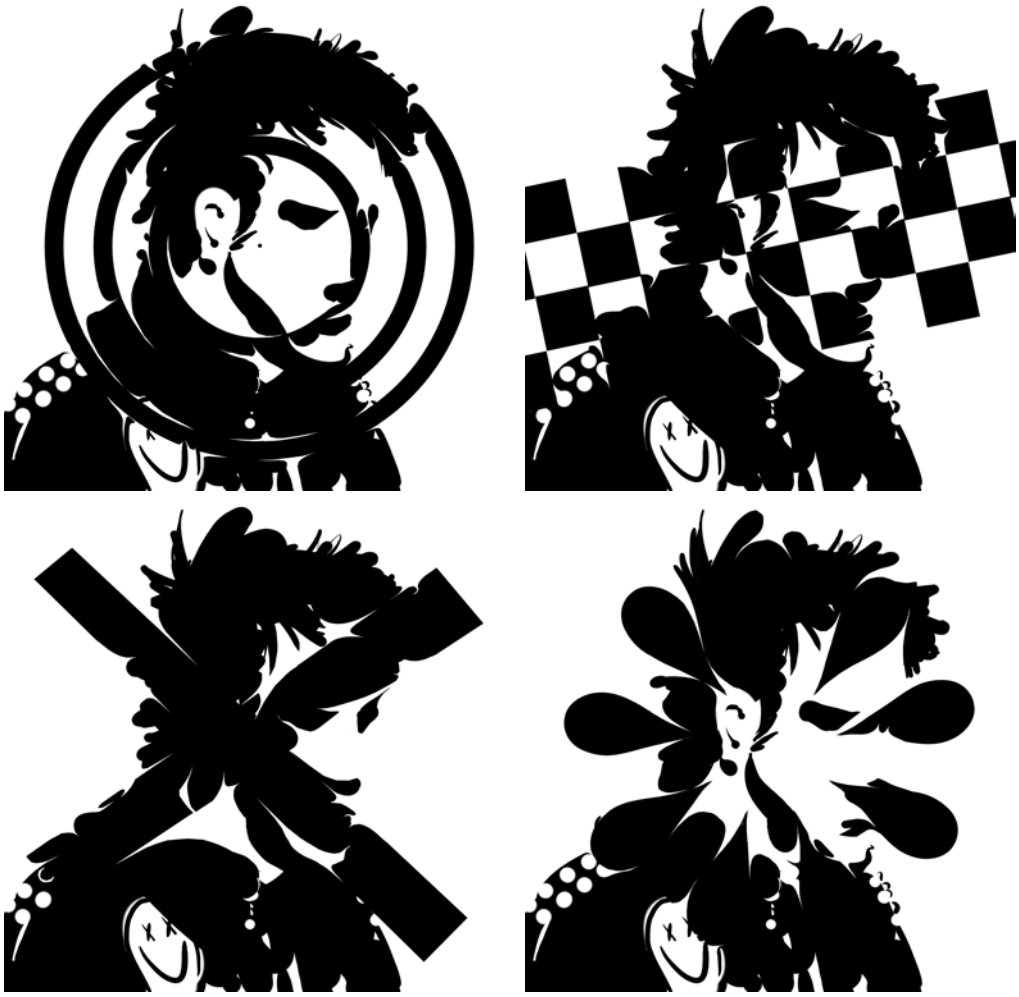


Figure 6.25: Handmade concept art for a future mashup algorithm that can handle shapes in general, not just stroke-based shapes made from combining a spine curve with a width curve.

to algorithmically produce similar artifacts more quickly than artists themselves can. In the NPR literature, the value of art itself is usually implicit. To speak about *why* art is worthwhile, or *what* is so interesting about it, is nonscientific, and therefore outside of NPR's scope.

Recently, however, the bigger art questions are coming to the fore. Certain "intelligent" NPR tools have become so commercially viable that the whole world—at least any scientific community concerned with art—has to confront art's *what* and *why*. Though my mashup problem has little to do with "AI art," its story also involves the big art questions.

When Hertzmann considers the future of art under the shadow of GANs [38], his outlook is positive: "Art maintains its vitality through continual innovation, and technology is one of the main engines of that innovation." I see in my mashups a modest validation of this statement, and a new insight for the NPR community: *when an artist develops NPR tools, NPR itself becomes an art medium.*

An art medium is a tool, and tools have limitations. Figuring out how to make effective pictures in spite of these limitations is a perennial, perhaps fundamental aspect of art-making. Take hatching. This is not simply a drawing technique; it is a creative product. It is a response to the limitations of art media—say woodcut or etching—that require an artist to laboriously scratch marks into a hard surface using a sharp point. Hatching, once invented, took on its own artistic value, and an entire aesthetic developed around it. People still use it today despite not being under the technological limitations that originally motivated it. What was once a necessity became the source of further invention.

An artist who wishes to use hatching must discover personally how to turn the pictures in their head into pictures on the page using groups of roughly parallel lines. To succeed at this is to change as an artist. From then on, hatching is part of the artist's visual vocabulary. New kinds of pictures will appear in the artist's head. The artist will naturally test the limits of what they know how to do with hatching, or what they understand hatching to be capable of in general. This is one of the ways in which creativity works, and if I might betray a postmodern sensibility, part of what makes art worthwhile.

The constraints of NPR pushed me to invent a line art problem I would never otherwise have thought of. Coming up with this problem forced me to imagine pictures I never otherwise would have imagined. Once I implemented an algorithm for solving my invented problem, trying to make good-looking pictures with it showed me its weaknesses and inspired me even more, resulting in the sketches of Figure 6.25. Whether or not I ever write the code to bring these sketches to fruition, the act of sketching them alone added a little more to my visual vocabulary.

Chapter 7

Conclusion

I have presented my technical contributions' merits and weaknesses, along with future work possibilities, in Chapters 4 through 6. In this chapter, I move beyond the technical contributions to focus on this dissertation's larger narrative, the story of an artist using science to make his own tools.

7.1 The Fourth Contribution

My doctoral experience was privately about rescuing my future as an artist. At the beginning, this personal narrative felt incongruous, even shameful. As I defined hatching and invented the hatching shape primitive, I felt a dissonance between the roles of scholar and artist. But the feeling changed with drawing mashups. This third contribution represented a shift in my relationship with NPR. I saw the extent to which NPR had become a part of my artistic practice. Its modalities, its limitations—particularly the requirement for scientific novelty—had begun to apply to my *artist* self, not just to my scholar self. NPR had become an art medium for me.

Had I done nothing but research hatching, this dissertation would not have benefited so much from my personal narrative. The turn at the end to drawing mashups elevated the narrative's importance. What was just dissonance—my ulterior artistic motives clashing with the expectations of a PhD candidate—turned into a pleasant surprise worth sharing.

I have said repeatedly that NPR in the hands of an artist can become an art medium. Let me express this idea as a hypothesis: *if an artist uses NPR research to make their own art, they should expect the research itself to inspire creativity, but in unexpected ways.*

I cannot claim definitely that an artist who conducts CS research to help their art will get more total inspiration than if they did not. Yet I do suspect that using science for art is especially creativity provoking. I can name a few specific ways that science has provoked creativity in me. One I have covered: the pressure for scholarly novelty. There are other, subtler ways which I describe in the next three sections.

7.1.1 The groundwork for creative development comes from making pictures meant to show off my research.

Consider *Watch* (Figure 7.1). This illustration comes from experimentation with hatching relaxation. At the end of Chapter 5 I suggest improving hatching marks' spacing in a post-processing step: relaxation is just that. The idea is to run a simulation that, at each step, slightly adjusts individual points along each hatching mark. Adjustments are based on the conflicting requirements of avoiding kinks and achieving better spacing.

In spite of curvature-preserving forces in the simulation, relaxed hatching marks develop kinks. These kinks evoke the ridges on sand dunes (as well as the kinks in Knöppel et al.'s stripe patterns [53]). In *Watch*, relaxed hatching creates the bulk of the dunes texture. My hope when starting this picture was that relaxed hatching would make a very dense dune pattern beyond what I would have the patience to draw manually. In the end, I was unsatisfied, and if I draw sand dunes again I will take a more manual approach. Still, *Watch* persists, maybe not as evidence of my artistic growth, but certainly as a creative artifact in its own right.

Watch is an unusual instance of a picture inspired after the fact by an unexpected research development. It has been more typical for me to conceive of a picture in advance of writing code, as motivation. *Sculptor* (Figure 7.2) is an example. Leading up to the hatching shape, I knew I wanted any computational hatching tool of mine to be fit for rendering a torso. Much of the work in Chapter 5 comes from trying to meet this specific requirement.

Watch and *Sculptor* are not on their own strong evidence of me growing as an artist due to researching NPR. However, making pictures to demonstrate research sets up the next stage of creative influence:

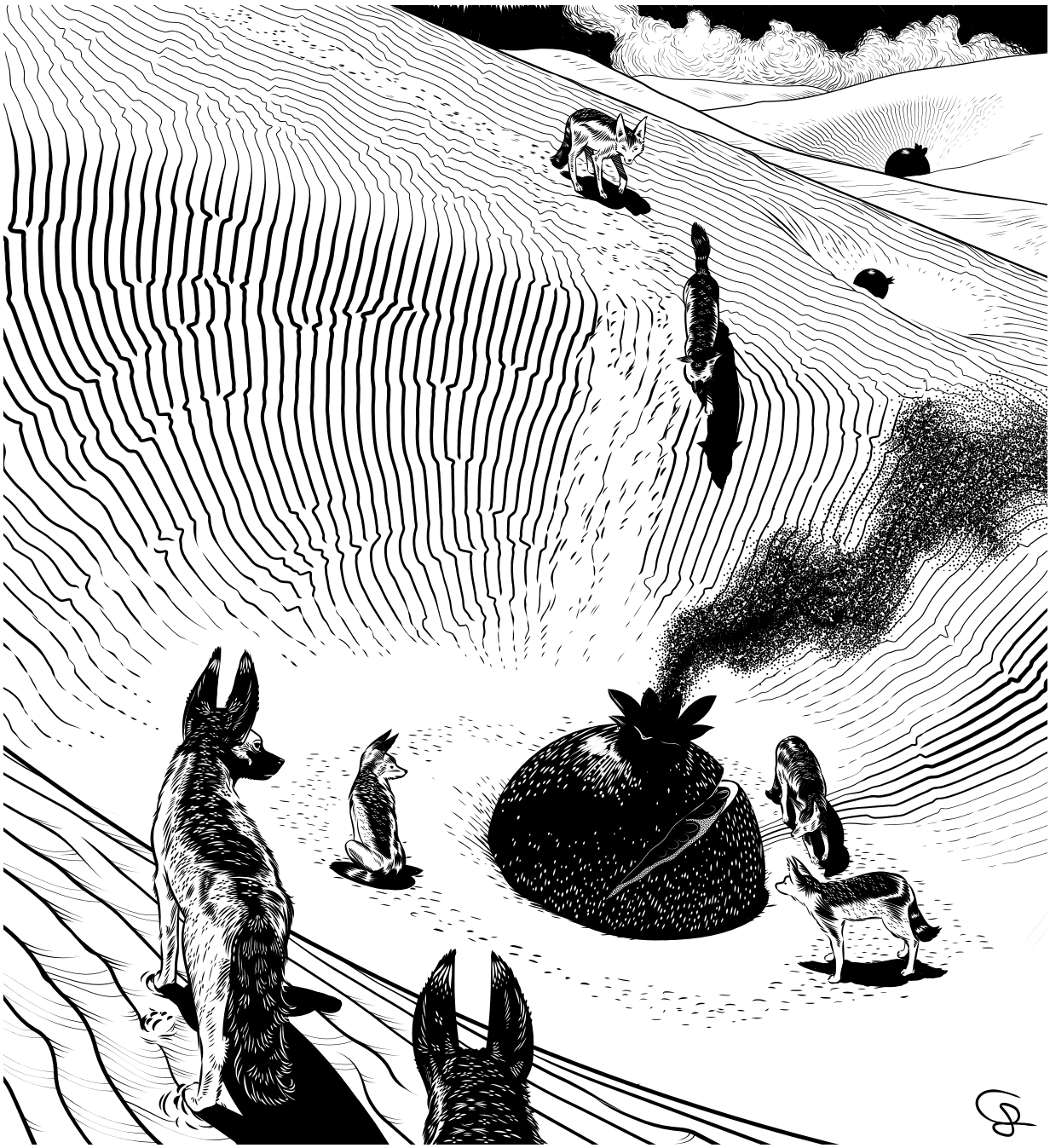


Figure 7.1: *Watch*



Figure 7.2: *Sculptor*. The hatching on the broken statue comes from a hatching shape. The hatching on the sculptor is all manual.

7.1.2 Creativity comes from encountering my research's weaknesses.

As I describe at the end of Chapter 5, I did not manage to make a hatching tool that could handle a torso comfortably (Figure 5.35). I moved away from the hatching shape to seek new primitives specifically built to render 3D forms (Section 3.4.3). When my 3D primitive attempts were unsatisfactory, I became disenchanted with focusing on traditional techniques altogether, which led to the inspiration to do something nontraditional, which led to drawing mashups.

My dissatisfaction with the hatching shape is also the reason I started playing with dot patterns. There is stippling in *Watch* simply because I had recently added the ability to procedurally generate it. Over time, dot patterns have become not just an abandoned research track, but a regular part of my picture-making. Initially, dots were an alternative to hatching, particularly to hatching generated by hatching shapes. See Figure 7.3, where I use regularly spaced dots to replace some hatching that I found eye-watering. Yet recently, I find myself attracted to dot patterns for purely artistic reasons. Take Figure 7.4, a detail from a very recent illustration I made just for myself, not to showcase any research. It has dot patterns simply because my NPR adventure gave me a knack for them. The stippling in this figure is basically manual, assisted by radial mode (Section 3.4.1).

I may give the impression that my hatching research has little value to me. True, hatching shapes are not everything I hoped for. But they did leave me in a new place artistically. *Spectacle* (Figure 7.5) summarizes where I now stand in relation to hatching. In this picture, hatching shapes do a lot of work: they cover the ocean, the terrain, and the cloud of smoke. The piece has a deliberately Op Art look, an intuitive response to hatching shapes' visual CG-ness. Considering this picture, I notice in myself a continuing desire to look in the Op Art direction, to find out how much I can lean into the computational aesthetic.

Though *Spectacle* embraces hatching shapes' CG look, there is significant manual refinement of the terrain and smoke. The hatching in these areas was originally from hatching shapes, but had to be converted to groups of strokes for by-hand improvements. This has become my standard practice whenever I introduce a hatching shape into a piece (Figure 7.6). I have become most comfortable using the hatching shape as an intermediate tool. If I need a pattern that can be constructed from an initial dense set of hatching lines, then I will use hatching shapes to place those initial lines and rely on my hand to add the details. The hatching shapes still spare significant labor. A tool with serious weaknesses may still have its uses, which leads to my third point about NPR-inspired creativity:

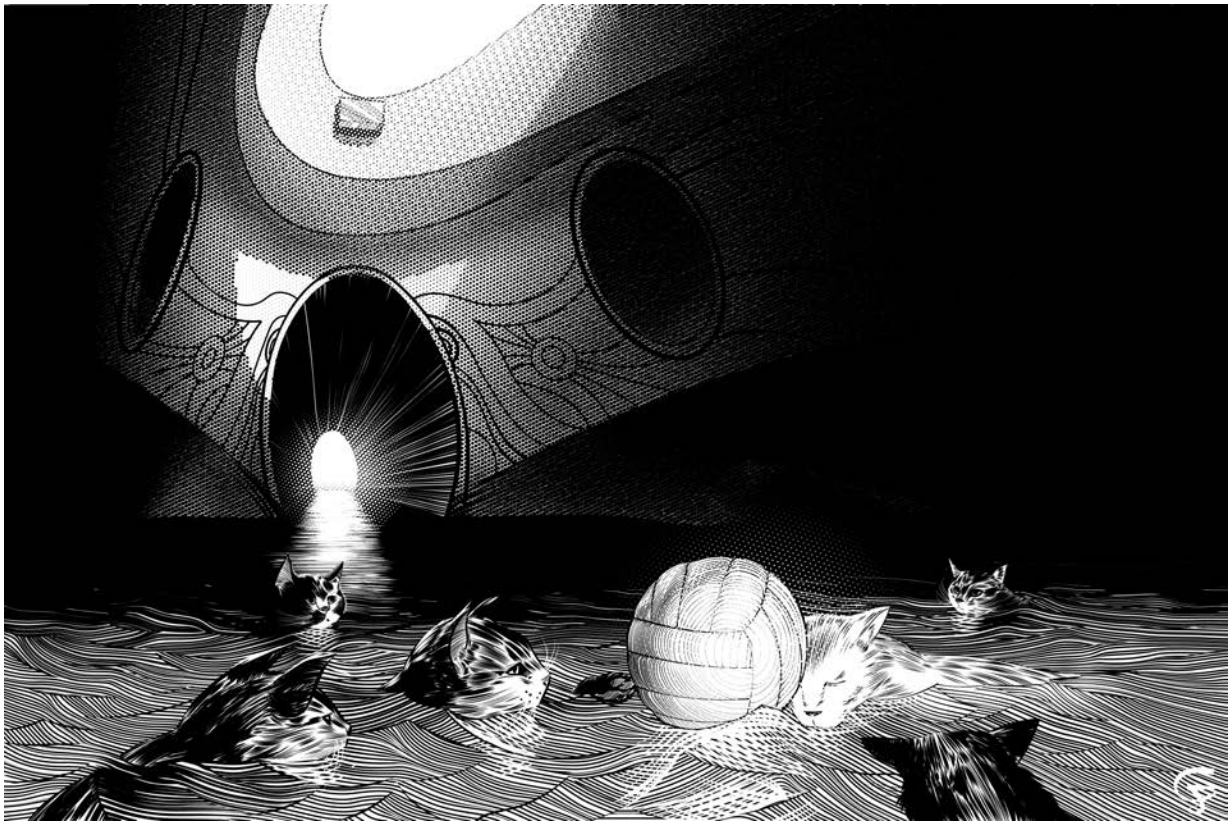


Figure 7.3: This updated version of *Twee* from Figure 5.28 uses dots instead of hatching for the background.

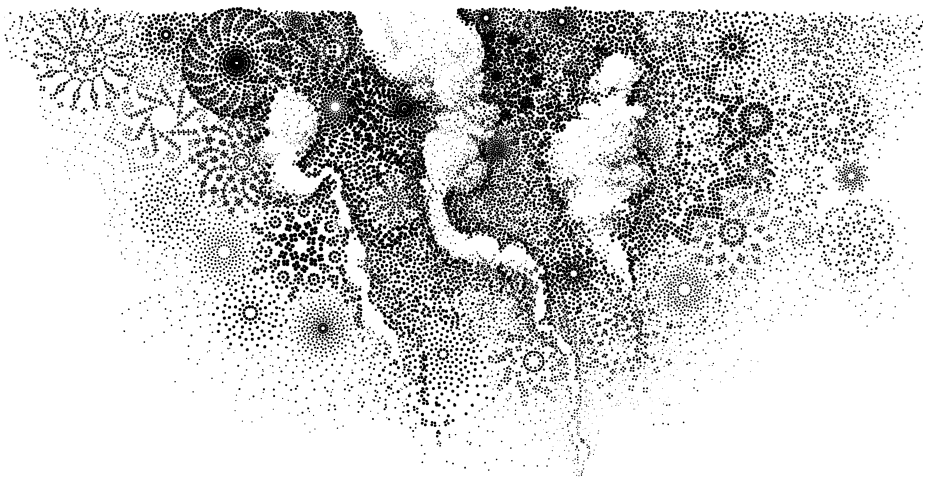


Figure 7.4: NPR research gave me a fondness for dot patterns.

7.1.3 Adaptations to my research's weaknesses can have unforeseen consequences.

In Chapter 3 I explain how my testbed application does poorly at creating beautiful individual strokes from position and pressure input. In the program, it is most convenient to draw families of what I might term *crude strokes*. I mean strokes that all have roughly the same width and taper in the same way at their ends, at least until I next change the brush settings. It is possible but very tedious to create more interesting, *refined strokes* with elegant, detailed position and width profiles; strokes that all look different from each other; strokes that are beautiful enough to hold attention individually; strokes that, to be blunt, do not look computer-generated. Ideally, my application would easily be able to create a sequence of such strokes, one per hand motion. For now, it can take numerous edit operations to produce even one individually beautiful stroke.

Because of this weakness around strokes, I mostly use my application for

1. tracing refined strokes made in other programs,
2. filling regions with solid color or procedural hatching, and
3. filling other regions manually with *patterns of crude strokes*.

The third role came about naturally once I determined to make initial drawings in a raster program like *Photoshop* or *Krita* [55] and use my application to vectorize these



Figure 7.5: *Spectacle*

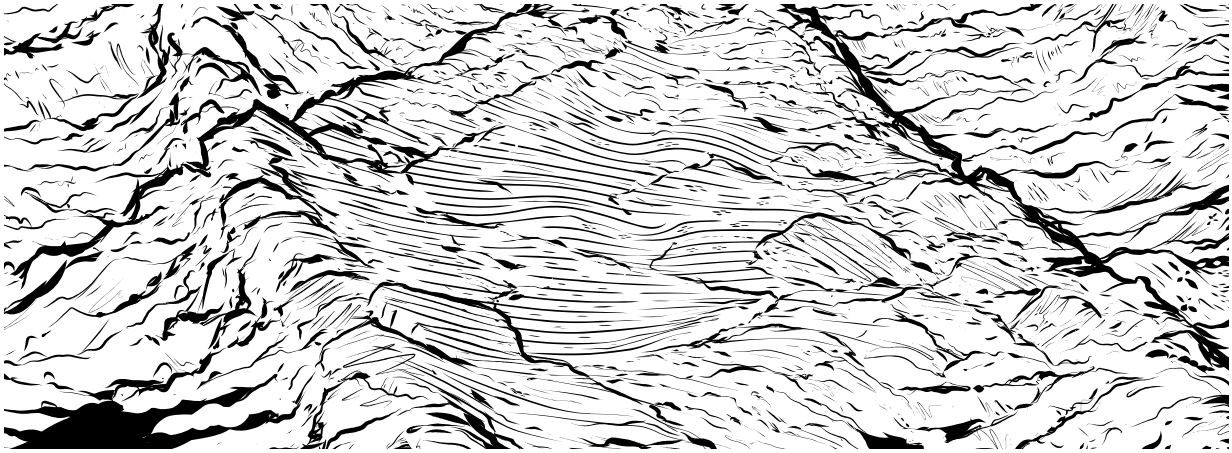


Figure 7.6: This detail from a recent illustration shows a hatching shape that I converted into strokes, which I then heavily edited by hand.

raster drawings. It was unsatisfying to view my application mainly as a manual vectorizer. The more I leaned in that direction, the more it felt like a waste of time. At the extreme, I would create a finished raster drawing and then tediously redraw every stroke of it in my program. Why use my program at all in that case? I prefer for the initial raster drawings to be finished at a compositional level, but lacking in detail. They are "comps" in illustration-speak, like the preparatory pencil layouts used in comic art, with some line work still to be done in the inking stage. When I use my application as an inker, I trace large, important strokes from the raster drawings, but I encourage myself to use my application's crude strokes whenever appropriate. As I fill regions with crude strokes, I find myself repeatedly using certain patterns. Figure 7.7 shows a few:

1. repeated star-like clusters of crude strokes (a,b);
2. crude strokes iteratively segmenting a region (c,d); and
3. heavy use of radial mode, often in concert with concentric circles (e,f).

Initially I found these patterns embarrassing, a testament to my failure as a developer. They were unprofessional in my mind. Now however, these patterns seem like *me*, for better or worse. I have used crude stroke patterns so many times that they are now a personal motif. Even if I leave behind digital and go back to traditional media, the patterns will come with me.

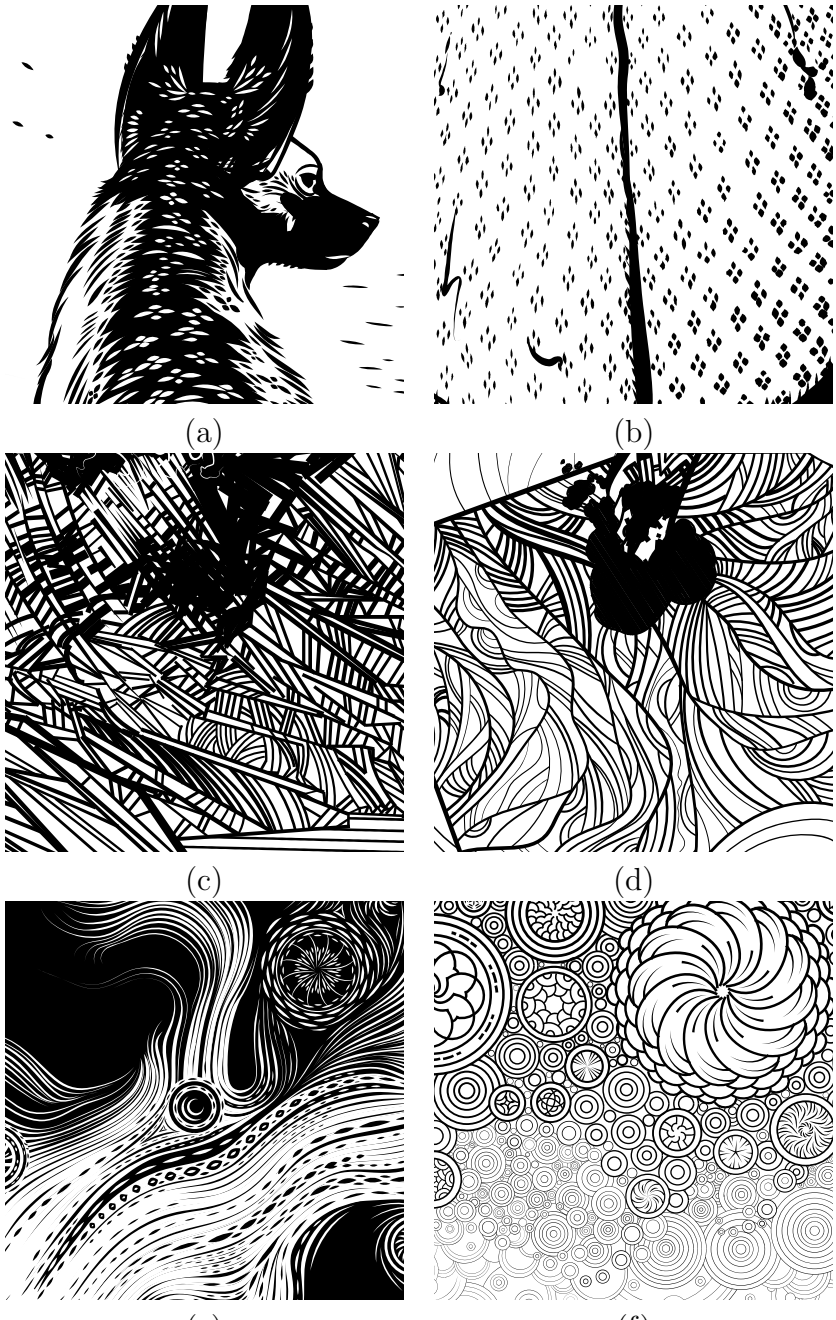


Figure 7.7: My application's mediocre stroke capturing ability encourages me to rely on patterns of *crude strokes*.



Figure 7.8: I learned illustration partly because of James Gurney. My research experience has led me to abandon the traditional mastery he exemplifies, and to rely on software instead. Detail from *Rumble and Mist*, 1992, oil on canvas mounted to panel. Used with permission.

7.2 Where I Go from Here

My gambit succeeded: I finished my doctorate as a better artist, if not in the way I expected.

My values have changed. Skills I used to consider essential now feel antiquated. Being a "real artist" feels less important. Using computers to make art no longer seems so sacrilegious. In my future artistic endeavors (I have plans), I intend to save time using software as long as I retain compositional control. I have already committed that the next time I draw a complex perspective scene like that of Figure 3.9, I will not use traditional drafting techniques. I will roughly model the scene in *Blender*, save a render of the scene, and trace the render in my program. And once AI art tools are fine-grained enough for me to say to them, "Fill this area with a crowd of figures," or "Texture the sleeve of this jacket the way I just did the other sleeve," then I will likely start using them. Why not? There are already artists like Memo Atken who face generative art head on rather than skirt around

it. When James Gurney, the illustrator who most inspired me to learn illustration, says, "I don't think I would feel fully human if I had to rely on such a system [generative AI]," [34], I do understand. He is a virtuoso, an "old-school" one (Figure 7.8). I have lost the need to be that. The art itself matters more to me now than my deftness in making it. It will be my job going forward to learn if and how the art itself suffers when its maker relies on machines.

Will I continue to be inspired artistically by NPR research? I will not rule it out. A better parting question is whether my fourth-contribution hypothesis holds, whether other artists will be as pleasantly surprised as I have in their attempts to co-opt science.

References

[1] F. Albert, D.G. Fernández-Pacheco, and N. Aleixos. New method to find corner and tangent vertices in sketches using parametric cubic curves approximation. *Pattern Recognition*, 46(5):1433–1448, 2013.

[2] Zainab AlMeraj, Brian Wyvill, Tobias Isenberg, Amy A. Gooch, and Richard Guy. Automatically mimicking unique hand-drawn pencil lines. *Computer Graphics*, 33(4):496–508, August 2009.

[3] Dmitry Apanovich. Strokes maker. http://www.strokesmaker.com, 2021.

[4] Michael Balzer. Capacity-constrained voronoi tessellations.

[5] Michael Balzer, Thomas Schlömer, and Oliver Deussen. Capacity-constrained point distributions: a variant of lloyd's method. *Trans. Graphics*, 28(3), jul 2009.

[6] Pascal Barla, Joëlle Thollot, and François X. Sillion. Geometric clustering for line drawing simplification. In *SIGGRAPH Sketches*, SIGGRAPH, pages 183–192, New York, NY, USA, 2005. ACM.

[7] Sebastiano Battiato, Gianpiero Blasi, Giovanni Farinella, and Giovanni Gallo. A survey of digital mosaic techniques. In *Proc. 4th Eurographics Italian Chapter Conference*, pages 129–135, 01 2006.

[8] Mark D. Benjamin, Stephen DiVerdi, and Adam Finkelstein. Painting with triangles. In *Proc. Workshop on Non-Photorealistic Animation and Rendering*, NPAR '14, page 13–20, New York, NY, USA, 2014. ACM.

[9] Hedlena Bezerra, Elmar Eisemann, Doug DeCarlo, and Joëlle Thollot. Diffusion constraints for vector graphics. In *Proc. Non-Photorealistic Animation and Rendering*, NPAR, page 35–42, New York, NY, USA, 2010. ACM.

[10] Alexander Browne. *Ars Pictoria*. Redmayne, Tooker, Battersby, 1669.

[11] Minh Tuan Bui, Junho Kim, and Yunjin Lee. 3D-look shading from contours and hatching strokes. *Computer Graphics*, 51(C):167–176, October 2015.

[12] Zoya Bylinskii, Laura Herman, Aaron Hertzmann, Stefanie Hutka, and Yile Zhang. Towards better user studies in computer graphics and vision. *Foundations and Trends® in Computer Graphics and Vision*, 15(3):201–252, 2023.

[13] Siddhartha Chaudhuri. Laplacian mesh processing. https://www.cse.iitb.ac.in/~cs749/spr2017/lecs/18_laplace.pdf, 2017.

[14] Yuan Chen, Jonathan Cohen, and Julian Krolik. Similarity-guided streamline placement with error evaluation. *Trans. Visualization and Computer Graphics*, 13(6):1448–1455, 2007.

[15] Hung-Kuo Chu, Wei-Hsin Hsu, Niloy J. Mitra, Daniel Cohen-Or, Tien-Tsin Wong, and Tong-Yee Lee. Camouflage images. *Trans. Graphics*, 29(4), jul 2010.

[16] Nelson S.-H. Chu and Chiew-Lan Tai. Moxi: real-time ink dispersion in absorbent paper. *Trans. Graphics*, 24(3):504–511, jul 2005.

[17] Moo K. Chung. Laplacian in regular grid. https://pages.stat.wisc.edu/~mchung/teaching/stat992/ima27.pdf, 2004.

[18] Oliver Deussen, Stefan Hiller, Cornelius Van Overveld, and Thomas Strothotte. Floating points: A method for computing stipple drawings. *Computer Graphics Forum*, 19(3):41–50, 2000.

[19] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[20] Neil A. Dodgson. Abstract depiction of human and animal figures: Examples from two centuries of art and craft. In *Proc. Expressive '18*, Expressive, pages 8:1–8:8, New York, NY, USA, 2018. ACM.

[21] Frédo Durand, Victor Ostromoukhov, Mathieu Miller, Francois Duranleau, and Julie Dorsey. Decoupling strokes and high-level attributes for interactive traditional drawing. In *Rendering Techniques 2001*, pages 71–82. Springer, 2001.

[22] G. Elber. Line art rendering via a coverage of isoparametric curves. *Trans. Visualization and Computer Graphics*, 1(3):231–239, 1995.

[23] Wayne Enstice. *Drawing: Space, Form, and Expression*. Pearson College Division, 2003.

[24] Mark Finch, John Snyder, and Hugues Hoppe. Freeform vector graphics with controlled thin-plate splines. *Trans. Graphics*, 30(6):1–10, December 2011.

[25] Jakub Fišer, Paul Asente, Stephen Schiller, and Daniel Sýkora. Advanced drawing beautification with shipshape. *Computers & Graphics*, 56:46–58, 2016.

[26] Moritz Gerl and Tobias Isenberg. Interactive example-based hatching. *Computers & Graphics*, 37(1):65–80, 2013.

[27] Domingo Martín Germán Arroyo and M. Victoria Luzón. Stochastic generation of dots for computer aided stippling. *Computer-Aided Design and Applications*, 7(4):447–463, 2010.

[28] Ahna Girshick, Victoria Interrante, Steven Haker, and Todd Lemoine. Line direction matters: An argument for the use of principal directions in 3D line drawings. In *Proc. Non-photorealistic Animation and Rendering*, NPAR, pages 43–52, New York, NY, USA, 2000. ACM.

[29] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. *Proc. ACM SIGGRAPH Conference on Computer Graphics*, 09 1999.

[30] Stephane Grabli, Fredo Durand, and Francois X Sillion. Density measure for line-drawing simplification. In *Proc. Pacific Conference on Computer Graphics and Applications*, pages 309–318. IEEE, 2004.

[31] Astute Graphics. Widthscribe. https://astutegraphics.com/plugins/widthscribe, 2021.

[32] Gaël Guennebaud, Jacob Benoît, et al. Eigen. http://eigen.tuxfamily.org, 2010.

[33] James Gurney. *Color and Light: A Guide for the Realist Painter*, volume 2. Andrews McMeel Publishing, 2010.

[34] James Gurney. What i've been thinking about generative art. Data Set, 12 2022.

[35] Peter Hall and Ann-Sophie Lehmann. *Don't Measure—Appreciate! NPR Seen Through the Prism of Art History*, pages 333–351. Springer London, London, 2013.

[36] James Herold and Thomas F. Stahovich. Speedseg: A technique for segmenting pen strokes using pen speed. *Computers & Graphics*, 35(2):250–264, 2011. Virtual Reality in Brazil Visual Computing in Biology and Medicine Semantic 3D media and content Cultural Heritage.

[37] Aaron Hertzmann. Non-photorealistic rendering and the science of art. In *Proc. Non-Photorealistic Animation and Rendering*, NPAR '10, page 147–157, New York, NY, USA, 2010. ACM.

[38] Aaron Hertzmann. Can computers create art? *Arts*, 7(2), 2018.

[39] Aaron Hertzmann. Computers do not make art, people do. *Communications of the ACM*, 63:45–48, 04 2020.

[40] Aaron Hertzmann. Toward modeling creative processes for algorithmic painting. *arXiv preprint arXiv:2205.01605*, 2022.

[41] Aaron Hertzmann. The curse of performative user studies. *IEEE Computer Graphics and Applications*, 43(06):112–116, nov 2023.

[42] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *Proc. Computer Graphics and Interactive Techniques*, SIGGRAPH, pages 517–526, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[43] Aldous Huxley. The doors of perception. *Mental*, 98:2–24, 1952.

[44] Emmanuel Iarussi, David Bommes, and Adrien Bousseau. Bendfields: Regularized curvature fields from rough concept sketches. *Trans. Graphics*, 34(3):24:1–24:16, May 2015.

[45] Tobias Isenberg. Interactive npar: What type of tools should we create? In *Proc. Computational Aesthetics, Sketch-Based Interfaces, Modeling, Non-Photorealistic Animation, and Rendering*, Expressive, page 89–96, Goslar, DEU, 2016. Eurographics Association.

[46] Bruno Jobard and Wilfrid Lefer. Creating evenly-spaced streamlines of arbitrary density. In Wilfrid Lefer and Michel Grave, editors, *Visualization in Scientific Computing*, pages 43–55, Vienna, 1997. Springer Vienna.

[47] Pierre-Marc Jodoin, Emric Epstein, Martin Granger-Piché, and Victor Ostromoukhov. Hatching by example: A statistical approach. In *Proc. Non-photorealistic Animation and Rendering*, NPAR, pages 29–36, New York, NY, USA, 2002. ACM.

[48] Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. WYSIWYG NPR: Drawing strokes directly on 3D models. *Trans. Graphics*, 21(3):755–762, July 2002.

[49] Evangelos Kalogerakis, Derek Nowrouzezahrai, Simon Breslav, and Aaron Hertzmann. Learning hatching for pen-and-ink illustration of surfaces. *Trans. Graphics*, 31(1), February 2012.

[50] Rubaiat Habib Kazi, Takeo Igarashi, Shengdong Zhao, and Richard Davis. Vignette: Interactive texture design and manipulation with freeform gestures for pen-and-ink illustration. In *Proc. SIGCHI*, CHI, pages 1727–1736, New York, NY, USA, 2012. ACM.

[51] Dongyeon Kim, Minjung Son, Yunjin Lee, Henry Kang, and Seungyong Lee. Feature-guided image stippling. *Computer Graphics Forum*, 27(4):1209–1216, 2008.

[52] Junhwan Kim and Fabio Pellacini. Jigsaw image mosaics. *ACM Trans. Graph.*, 21(3):657–664, jul 2002.

[53] Felix Knöppel, Keenan Crane, Ulrich Pinkall, and Peter Schröder. Stripe patterns on surfaces. *Trans. Graphics*, 34(4), July 2015.

[54] Hans Knutsson. Representing local structure using tensors. *Proc. Scandinavian Conference on Image Analysis*, s. 244-251, 1989.

[55] Krita Foundation, KDE. Krita.

[56] Daniel L Lau and Gonzalo R Arce. *Modern Digital Halftoning*. CRC Press, 2008.

[57] Kai Lawonn, Ivan Viola, Bernhard Preim, and Tobias Isenberg. A survey of surface-based illustrative rendering for visualization. *Computer Graphics Forum*, 37(6):205–234, 2018.

[58] Zoltán Lengyel, Tamás Umenhoffer, and László Szécsi. Realtime, coherent screen space hatching. In *Proc. Hungarian Computer Graphics and Geometry Conference*, pages 131–137, 2014.

[59] Hua Li and David Mould. Structure-preserving stippling by priority-based error diffusion. In *Proc. Graphics Interface 2011*, GI '11, pages 127–134, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2011. Canadian Human-Computer Communications Society.

[60] Liya Li, Hsien-Hsi Hsieh, and Han-Wei Shen. Illustrative streamline placement and visualization. In *IEEE Pacific Visualisation Symposium*, pages 79 – 86, 04 2008.

[61] Chenxi Liu, Enrique Rosales, and Alla Sheffer. StrokeAggregator: Consolidating raw sketches into artist-intended curve drawings. *Trans. Graphics*, 37(4):97:1–97:15, July 2018.

[62] Zhanping Liu, Robert Moorhead, and Joe Groner. An advanced evenly-spaced streamline placement algorithm. *Trans. Visualization and Computer Graphics*, 12:965–72, 10 2006.

[63] Hugo Loi, Thomas Hurtut, Romain Vergne, and Joelle Thollot. Programmable 2d arrangements for element texture design. *Trans. Graphics*, 36(4), jul 2017.

[64] A. Loomis. *Successful Drawing*. Titan Books, 2012.

[65] D. Heck M. Balzer. Capacity-constrained voronoi diagrams in finite spaces. In Kokichi Sugihara and Deok-Soo Kim, editors, *Proc. International Symposium on Voronoi Diagrams in Science and Engineering*, number 4(2) in Voronoï's Impact on Modern Science, pages 44–56, Kiev, Ukraine, sep 2008.

[66] Domingo Martín, Germán Arroyo, Alejandro Rodríguez, and Tobias Isenberg. A survey of digital stippling. *Computers & Graphics*, 67:24–44, 2017.

[67] Abdelkrim Mebarki. Adaptive distance grid based algorithm for farthest point seeding streamline placement. *Open Computer Science*, 1(open-issue), 2016.

[68] Abdelkrim Mebarki, Pierre Alliez, and Olivier Devillers. Farthest point seeding for efficient placement of streamlines. In *VIS 2005*, pages 479–486. IEEE Computer Society, 2005.

[69] Jonatas Medeiros, Mario Sousa, Luiz Velho, and Carla Freitas. Perspective contouring in illustrative visualization. In *2009 XXII Brazilian Symposium on Computer Graphics and Image Processing*, pages 48–55, Oct 2009.

[70] David Mould. Stipple placement using distance in a weighted graph. In *International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging*, 2007.

[71] David Mould. Authorial subjective evaluation of non-photorealistic images. In *Proceedings of the Workshop on Non-Photorealistic Animation and Rendering*, NPAR '14, page 49–56, New York, NY, USA, 2014. ACM.

[72] Aude Oliva, Antonio Torralba, and Philippe G. Schyns. Hybrid images. *Trans. Graphics*, 25(3):527–532, jul 2006.

[73] OpenAI. Dall-e.

[74] Günay Orbay and Levent Burak Kara. Pencil-like sketch rendering of 3D scenes using trajectory planning and dynamic tracking. *Journal of Visual Languages & Computing*, 25(4):481–493, 2014.

[75] Victor Ostromoukhov. Digital facial engraving. In *Proc. Computer Graphics and Interactive Techniques*, SIGGRAPH, pages 417–424, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[76] Yachin Pnueli and Alfred Marcel Bruckstein. DigiDürer — a digital engraving system. *The Visual Computer*, 10:277–292, 2005.

[77] Shivam Pradhan. Mid-point line generation algorithm. https://www.geeksforgeeks.org/mid-point-line-generation-algorithm/?ref=rp, 04 2020.

[78] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *Proc. Computer Graphics and Interactive Techniques*, SIGGRAPH, pages 579–584, New York, NY, USA, 2001. ACM.

[79] Bernhard Preim, Alexandra Baer, Douglas Cunningham, Tobias Isenberg, and Timo Ropinski. A survey of perceptually motivated 3d visualization of medical image data. *Computer Graphics Forum*, 35(3):501–525, 2016.

[80] Scott Robertson and Thomas Bertling. *How to Draw: drawing and sketching objects and environments from your imagination*. Design Studio Press, 2013.

[81] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen-and-ink illustration. In *Proc. Computer Graphics and Interactive Techniques*, SIGGRAPH, pages 101–108, New York, NY, USA, 1994. ACM.

[82] Michael P. Salisbury, Michael T. Wong, John F. Hughes, and David H. Salesin. Orientable textures for image-based pen-and-ink illustration. In *Proc. Computer Graphics and Interactive Techniques*, SIGGRAPH, pages 401–406, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[83] Mike Salisbury, Corin Anderson, Dani Lischinski, and David H. Salesin. Scale-dependent reproduction of pen-and-ink illustrations. In *Proc. Computer Graphics*

and Interactive Techniques, SIGGRAPH, page 461–468, New York, NY, USA, 1996. ACM.

[84] Reza Saputra, Craig Kaplan, and Paul Asente. Improved deformation-driven element packing with repulsionpak. *Trans. Visualization and Computer Graphics*, PP:1–1, 10 2019.

[85] Reza Adhitya Saputra, Craig S. Kaplan, Paul Asente, and Radomír Měch. Flowpak: Flow-based ornamental element packing. In *Proc. 43rd Graphics Interface Conference*, GI '17, page 8–15, Waterloo, CAN, 2017. Canadian Human-Computer Communications Society.

[86] Michael Schlemmer, Ingrid Hotz, Bernd Hamann, Florian Morr, and Hans Hagen. Priority streamlines: A context-based visualization of flow fields. In *Proc. Visualization*, EUROVIS, page 227–234, Goslar, DEU, 2007. Eurographics Association.

[87] Martin Schwarz, Tobias Isenberg, Katherine Mason, and Sheelagh Carpendale. Modeling with rendering primitives: An interactive non-photorealistic canvas. In *Proc. Non-Photorealistic Animation and Rendering*, NPAR '07, page 15–22, New York, NY, USA, 2007. ACM.

[88] Thomas W Sederberg. *Computer aided geometric design*. BYU Faculty Publications, September 2016.

[89] Tevfik Sezgin, Thomas Stahovich, and Randall Davis. Sketch based interfaces: Early processing for sketch understanding. *Workshop on Perceptive User Interfaces*, 01 2002.

[90] Cloud Shao, Adrien Bousseau, Alla Sheffer, and Karan Singh. CrossShade: Shading concept sketches using cross-section curves. *Trans. Graphics*, 31(4):45:1–45:11, July 2012.

[91] Mayank Singh and Scott Schaefer. Suggestive hatching. In *Proc. Computational Aesthetics in Graphics, Visualization and Imaging*, Computational Aesthetics, page 25–32, Goslar, DEU, 2010. Eurographics Association.

[92] Minjung Son, Yunjin Lee, Henry Kang, and Seungyong Lee. Structure grid for directional stippling. *Graphical Models*, 73(3):74–87, 2011.

[93] Stability AI. Stable diffusion.

[94] Thomas Strothotte and Stefan Schlechtweg. *Screening*, chapter 2.2, pages 41–59. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, San Francisco, 2002.

[95] Pjotr Svetachov, Maarten Everts, and Tobias Isenberg. Dti in context: Illustrating brain fiber tracts in situ. *Computer Graphics Forum*, 29:1023–1032, 06 2010.

[96] C.-H. Teh and R.T. Chin. On the detection of dominant points on digital curves. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 11(8):859–872, 1989.

[97] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 5.1 edition, 2020. https://doc.cgal.org/5.1/Manual/packages.html.

[98] Pekka J. Toivanen. New geodosic distance transforms for gray-scale images. *Pattern Recognition Letters*, 17(5):437 – 450, 1996.

[99] Rodolphe Vaillant. Definition: Laplacian matrix for triangle meshes, 05 2019.

[100] Vivek Verma, David Kao, and Alex Pang. A flow-guided streamline seeding strategy. In *Proceedings of the Conference on Visualization '00*, VIS, page 163–170, Washington, DC, USA, 2000. IEEE Computer Society Press.

[101] Ivan Viola, Mario Costa Sousa, David S Ebert, Bill Andrews, Bruce Gooch, and Christian Tietjen. Illustrative visualization for medicine and science. In *Eurographics (Tutorials)*, pages 1061–1200. Citeseer, 2006.

[102] Walt Disney Animation Studios. Meander.

[103] A. H. Watt. *3D Computer Graphics*. Addison-Wesley, 2000.

[104] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. In *Proc. Computer Graphics and Interactive Techniques*, SIGGRAPH, pages 91–100, New York, NY, USA, 1994. ACM.

[105] Aaron Wolin, Brian Eoff, and Tracy Hammond. Shortstraw: A simple and effective corner finder for polylines. In *Proc. Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 33–40, 01 2008.

[106] M. Woo, J. Neider, T. Davis, and OpenGL Architecture Review Board. *The Official Guide to Learning OpenGL*. Addison Wesley, 1.1 edition, 1997.